

Praktické programování v C++

Garant předmětu:

Ing. Miloslav Richter, PhD.

Autoři textu:

Ing. Miloslav Richter, PhD.

Ing. Petr Petyovský

Ing. Ilona Kalová

Ing. Karel Horák

Obsah

1	ÚVOD.....	5
2	ZAŘAZENÍ PŘEDMĚTU VE STUDIJNÍM PROGRAMU.....	6
2.1	ÚVOD DO PŘEDMĚTU.....	6
2.2	VSTUPNÍ TEST.....	7
3	JAZYK C++.....	10
3.1	STRUČNÝ ÚVOD DO C++.....	11
3.2	NEOBJEKTOVÉ VLASTNOSTI C++ A ROZDÍLY MEZI C A C++.....	17
3.2.1	<i>Komentáře, poznámky.....</i>	<i>17</i>
3.2.2	<i>Deklarace a definice proměnných.....</i>	<i>18</i>
3.2.3	<i>Předávání parametrů odkazem – reference.....</i>	<i>20</i>
3.2.4	<i>Operátor příslušnosti :: (kvalifikátor).....</i>	<i>24</i>
3.2.5	<i>Přetížení funkcí.....</i>	<i>25</i>
3.2.6	<i>Implicitní parametry.....</i>	<i>27</i>
3.2.7	<i>Přetypování.....</i>	<i>29</i>
3.2.8	<i>Modifikátor const.....</i>	<i>30</i>
3.2.9	<i>Alokace paměti – new, delete.....</i>	<i>32</i>
3.2.10	<i>Enum.....</i>	<i>35</i>
3.2.11	<i>Inline funkce.....</i>	<i>36</i>
3.2.12	<i>Prototypy funkcí.....</i>	<i>37</i>
3.2.13	<i>Funkce bez parametrů.....</i>	<i>38</i>
3.2.14	<i>Logické proměnné – bool, true, false.....</i>	<i>38</i>
3.2.15	<i>Přetížení operátorů.....</i>	<i>39</i>
3.2.16	<i>Vstupy a výstupy v C++.....</i>	<i>40</i>
3.2.17	<i>Znakové konstanty, (dlhé) literály.....</i>	<i>52</i>
3.2.18	<i>Typ ((un)signed) long long.....</i>	<i>53</i>
3.2.19	<i>Prostor jmen – namespace.....</i>	<i>53</i>
3.2.20	<i>Restrict.....</i>	<i>56</i>
3.2.21	<i>Anonymní unie.....</i>	<i>56</i>
3.2.22	<i>Shrnutí neobjektových vlastností.....</i>	<i>57</i>
3.3	OBJEKTOVÉ VLASTNOSTI C++ - ZÁKLADY PRÁCE S TŘÍDAMI.....	57
3.3.1	<i>Třída a struktura v OOP.....</i>	<i>59</i>
3.3.2	<i>Členy třídy – data a metody.....</i>	<i>61</i>
3.3.3	<i>Data a přístupová práva.....</i>	<i>64</i>
3.3.4	<i>Ukazatel this.....</i>	<i>68</i>
3.3.5	<i>Statický datový člen třídy.....</i>	<i>71</i>
3.3.6	<i>Konstruktory a destruktory.....</i>	<i>74</i>
3.3.7	<i>Hlavičkové soubory a třída, příslušnost ke třídě.....</i>	<i>85</i>
3.3.8	<i>Inline metody.....</i>	<i>86</i>
3.3.9	<i>Deklarace a definice třídy, objektů a metod.....</i>	<i>95</i>
3.3.10	<i>Operátory přístupu k prvkům *. a ->.....</i>	<i>96</i>
3.3.11	<i>Deklarace třídy uvnitř jiné třídy.....</i>	<i>97</i>
3.3.12	<i>Modifikátor const u parametrů a metod třídy.....</i>	<i>98</i>
3.3.13	<i>Friend.....</i>	<i>106</i>
3.3.14	<i>Operátory.....</i>	<i>107</i>
3.3.15	<i>Statické metody třídy.....</i>	<i>118</i>
3.3.16	<i>Modifikátor mutable.....</i>	<i>119</i>

3.3.17	<i>Třídy a prostory jmen</i>	119
3.3.18	<i>Třídy a streamy – vstupy a výstupy</i>	119
3.4	OBJEKTOVÉ VLASTNOSTI C++ - DĚDĚNÍ	121
3.4.1	<i>Dědění</i>	121
3.4.2	<i>Vícenásobné dědění</i>	124
3.4.3	<i>Virtuální metody</i>	126
3.4.4	<i>Abstraktní (bázové) třídy</i>	130
3.4.5	<i>Přetypování / RTTI</i>	135
3.4.6	<i>Šablony – generické programování</i>	136
3.4.7	<i>Výjimky</i>	138
3.4.8	<i>Informace o typu za běhu programu</i>	140
3.4.9	<i>Standardní knihovny</i>	141
3.4.10	<i>C v C++</i>	141
4	DODATKY	142
4.1	VÝSLEDKY TESTŮ	142
4.1.1	<i>Vstupní test</i>	142
4.1.2	<i>Kapitola 3.1</i>	144
4.1.3	<i>Kapitola 3.2</i>	144
4.1.4	<i>Kapitola 3.3</i>	144
4.2	SEZNAM PŘÍLOH	144
5	PŘÍLOHA 1 - JAZYK C	145
5.1	ÚVOD	145
5.1.1	<i>Předmluva</i>	145
5.1.2	<i>V čem programovat</i>	146
5.1.3	<i>Stručná charakteristika C</i>	147
5.1.4	<i>Stručná charakteristika C++</i>	148
5.1.5	<i>Událostmi řízené programování</i>	149
5.1.6	<i>Odchylky C a C++</i>	149
5.1.7	<i>Návrh programu</i>	150
5.1.8	<i>Algoritmizace</i>	150
5.1.9	<i>Ladění programů</i>	150
5.1.10	<i>Programátorský styl (kultura programování)</i>	151
5.1.11	<i>Data v paměti, volací konvence</i>	152
5.1.12	<i>Přenositelnost zdrojových textů</i>	152
5.2	C	157
5.2.1	<i>Struktura programu v C</i>	157
5.2.2	<i>Překlad a sestavení programu v jazyce C</i>	162
5.2.3	<i>Komentáře</i>	167
5.2.4	<i>Funkce main – základ</i>	167
5.2.5	<i>identifikátory, základní datové typy, konstanty</i>	169
5.2.6	<i>Typová konverze (přetypování)</i>	172
5.2.7	<i>Operace s proměnnými, operátory</i>	174
5.2.8	<i>Funkce</i>	179
5.2.9	<i>Příkazy preprocesoru, makra</i>	181
5.2.10	<i>Platnost identifikátorů, globální a lokální proměnné a funkce</i>	186
5.2.11	<i>Standardní znakový (terminálový) výstup / vstup</i>	192
5.2.12	<i>If – else, ternární operátor</i>	193
5.2.13	<i>Cykly, opuštění cyklu - for, while, do-while, continue, break</i>	194
5.2.14	<i>Switch</i>	197

5.2.15	<i>Ukazatele , typedef</i>	199
5.2.16	<i>Dynamická paměť</i>	200
5.2.17	<i>Funkce a ukazatele</i>	202
5.2.18	<i>Jednorozměrné pole, ukazatelová aritmetika</i>	206
5.2.19	<i>Řetězce</i>	208
5.2.20	<i>Funkce a pole</i>	209
5.2.21	<i>Formátovaný vstup a výstup</i>	210
5.2.22	<i>Standardní funkce pro práci s řetězci</i>	214
5.2.23	<i>Práce se soubory, standardní soubory vstupu a výstupu a err.</i>	214
5.2.24	<i>Pole – vícerozměrné, typedef</i>	217
5.2.25	<i>Funkce main – plné volání</i>	220
5.2.26	<i>Struktury</i>	221
5.2.27	<i>Union</i>	223
5.2.28	<i>Výčtový typ (enum)</i>	224
5.2.29	<i>Bitová pole</i>	224
5.2.30	<i>Funkce s proměnným počtem parametrů, "..." (výpustka)</i>	225
5.2.31	<i>Čtení komplikovaných definic</i>	226
5.2.32	<i>Příkaz goto</i>	226
5.2.33	<i>Assembler</i>	227
6	PŘÍLOHA 2 – PŘÍKLADOVÁ ČÁST	228
6.1	ÚVOD.....	228
6.1.1	<i>Jak přeložit program</i>	228
6.1.2	<i>Překládání</i>	228
6.1.3	<i>Doporučení pro tvorbu příkladů</i>	228
6.1.4	<i>Assert</i>	229
6.2	PŘÍKLADY PRO C.....	229
6.2.1	<i>Struktura programu v C</i>	229
6.2.2	<i>Překlad a sestavení programu</i>	229
6.2.3	<i>Komentáře</i>	229
6.2.4	<i>Funkce main – základ</i>	230
6.2.5	<i>Identifikátory, základní datové typy</i>	230
6.2.6	<i>Typová konverze</i>	231
6.2.7	<i>Operace s proměnnými, operátory</i>	231
6.2.8	<i>Funkce</i>	231
6.2.9	<i>Příkazy preprocesoru, makra</i>	232
6.2.10	<i>Platnost identifikátorů, globální a lokální proměnné</i>	232
6.2.11	<i>Standardní znakový (terminálový) výstup, vstup</i>	233
6.2.12	<i>If – else, ternární operátor</i>	233
6.2.13	<i>Cykly, opuštění cyklu</i>	234
6.2.14	<i>Switch</i>	234
6.2.15	<i>Ukazatele, typedef</i>	235
6.2.16	<i>Dynamická paměť</i>	235
6.2.17	<i>Funkce a ukazatele</i>	236
6.2.18	<i>Jednorozměrné pole, ukazatelová aritmetika</i>	236
6.2.19	<i>Řetězce</i>	236
6.2.20	<i>Funkce a pole</i>	237
6.2.21	<i>Formátovaný vstup a výstup</i>	237
6.2.22	<i>Funkce pro práce s řetězci</i>	237
6.2.23	<i>Práce se soubory</i>	238

6.2.24	<i>Vícerozměrné pole</i>	238
6.2.25	<i>Funkce main</i>	239
6.2.26	<i>Struktury</i>	239
6.2.27	<i>Uniony</i>	239
6.2.28	<i>Výčtový typ</i>	239
6.2.29	<i>Bitová pole</i>	240
6.2.30	<i>Funkce s proměnným počtem parametrů</i>	240
6.2.31	<i>Čtení definic</i>	240
6.2.32	<i>Příkaz goto</i>	240
6.2.33	<i>Assembler</i>	241
6.2.34	<i>Příklad C 1 – ovládání bitů podle sekvencí vstupního souboru</i>	241
6.2.35	<i>Příklad C 2 – funkce pro práci se soubory a řetězci</i>	241

7 SEZNAM POUŽITÉ LITERATURY243

Pozn.

1 Úvod

Tento text slouží jako příručka pro výuku programování v jazyce C++.

Předpokládá se absolvování kurzů programování v jazyce C, a tedy i znalost programovacího prostředí a základní programátorské dovednosti. Pro případ nejasností nebo nedokonalé znalosti jazyka C jsou k tomuto textu připojeny jako přílohy poznámky k programování a přednášky jazyka C včetně kontrolních otázek a příkladů. Doporučujeme nahlédnout do této části skript všem. Jsou zde uvedeny základy jazyka C, které lze očekávat ve všech překladačích (nejen pro PC/Linux, ale též pro různé mikroprocesory, hradlová pole...).

Tento text obsahuje společně učební texty i příklady pro cvičení na počítačích. Pro probíranou látku obsahuje jednoduché příklady řešené a neřešené příklady srovnatelné obtížnosti. Řešené příklady jsou voleny tak, aby na co nejmenší ploše demonstrovaly základní vlastnosti probírané látky. Pro smysluplnější využití dané techniky jsou voleny neřešené příklady. Ty jsou skutečně neřešené, protože ze zkušenosti plyne, že pouze samostatným programováním bez nápověd je možné si osvojit programovací návyky.

V následujícím textu jsou použity tyto styly

- pro běžný text
- pro jména souborů
- ***pro vstup a výstup konzoly***
- **pro klíčová slova a zdrojové texty**

Citace :

RICHTER, Miloslav; PETYOVSKÝ, Petr; KALOVÁ, Ilona; HORÁK, Karel. Praktické programování v C++. ET VUT BRNO. 2009. 244s.

kontakt:

Richter Miloslav ☎+420 541 141 194, ✉richter@feec.vutbr.cz

2 Zařazení předmětu ve studijním programu

Předmět je zařazen do druhého ročníku bakalářského studia a navazuje na kurzy počítačů a programování, kde se probírají základy programování v jazyce C. Tento text předpokládá znalosti získané v těchto kurzech. V tomto textu jsou znalosti rozšířeny o možnosti programovacího jazyka C++ a to neobjektové i objektové. Znalosti dosažené v tomto kurzu vyžadují následující odborné předměty, které předpokládají tvorbu projektů v tomto programovacím jazyce.

Jazyk C se využívá pro základní algoritmy, při programování mikroprocesorů, signálových procesorů ... Programovací jazyk C++ je potom vhodný pro řešení rozsáhlejších projektů. Složitější projekty kombinují oba jazyky - pro komunikaci s HW a pro zpracování dat využívají vlastností a rychlosti jazyka C, a zároveň pro prezentaci dat a komunikaci s obsluhou komfortu C++.

Pozn.: i v C++ platí, že je lepší nejprve myslet a potom teprve programovat. Pokud to děláte jinak C++ vám nepomůže.

2.1 Úvod do předmětu

V tomto kurzu se student naučí základní vlastnosti, dovednosti a možnosti důležité pro tvorbu programů v jazyce C++. Ty se skládají z neobjektových vlastností, které je možné využít pro zlepšení programování standardním způsobem, a z objektových vlastností, které mění pohled na styl programování a využívají nové programovací možnosti a techniky.

Neobjektové vlastnosti jazyka C++ umožňují stejný programovací styl jako u C ale větší komfort. Zároveň slouží k tomu aby bylo možno aplikovat objektové mechanismy.

I když je obecně jazyk C podmnožinou C++, vznikají rozdíly mezi C a C++ na neobjektové úrovni. Tyto mohou vzniknout například tím, že norma jazyka C "předběhne" normu jazyka C++ a zavede novinky, díky kterým se stává nepřenositelná do C++. Dále potom úspěšné novinky jazyka C++ se zpětně zavádějí do jazyka C. Nevýhodou je, že starší a jednodušší překladače tyto vlastnosti nepodporují a tak se jejich použití stává problémem, např. při programování nebo při nutnosti realizovat přenositelnost kódu.

Objektové vlastnosti jazyka – nový programovací styl založený na objektech, sdružujících data a funkce s nimi pracujícími do logických celků. Možnost využívat stávající objekty jako základ pro objekty složitější.

Programátorské vlastnosti jazyka – nové vlastnosti jazyka umožňují nové programátorské přístupy jako je polymorfismus, dědění, jednotný interface, vícenásobné využití kódu, generické programování (šablony) ...

Získané vlastnosti budou sloužit především pro tvorbu projektů v následujících předmětech.

Předchozí vlastnosti jsou rozděleny tak, že v kapitole 3.2 jsou popsány neobjektové vlastnosti jazyka C++ a rozdíly mezi C a C++. V kapitole 3.3 jsou základní vlastnosti jazyka C++. V kapitole 3.4 jsou potom uvedeny možnosti jazyka C++ z hlediska dědění a možnosti

generického programování, výjimek a knihovních funkcí. Programátorské techniky vyplývající z možností jazyka jsou uváděny průběžně.

2.2 Vstupní test

Vstupní test slouží pro zjištění znalosti programovacích technik a jazyka C.

a)

- co je to modul (projektu jazyka C)
- jaké přípony mají základní typy souborů
- jaký je základní proces návrhu programu
- lze přenášet zdrojové texty mezi jednotlivými prostředími, systémy
- jaká je struktura programu v C
- jak probíhá tvorba spustitelného programu v C
- co je projektový soubor

b)

- jak se píší komentáře v C
- co je funkce **main**, jaký je její prototyp
- základní celočíselné typy a jejich velikost
- co je prototyp funkce
- k čemu slouží příkazy **if - else**, kterým operátorem lze nahradit, v čem se liší

c)

- které typy cyklů znáte, za jakých podmínek se opakují
- k čemu slouží příkaz **switch**
- co je to ukazatel, jak se inicializuje, jak se s ním pracuje (dereference) a jak se získá
- předávají se parametry do funkce v jazyce C hodnotou nebo odkazem
- jak předat hodnoty z funkce do volajícího programu

d)

- co je ukazatelová aritmetika
- co je to pole, vícerozměrné pole, a jak se definují a jak se indexují
- co je to řetězec
- co je to struktura. Jak se přistupuje k jejím prvkům je-li dána jako objekt nebo jako ukazatel, jak se zjistí velikost struktury v paměti
- co je **union**, jakou má velikost

Příklad 2.1

- Přečtete si celé zadání
- Proved'te rozbor zadání

- Zkuste vypracovat nejdříve na papír – jako na písemných zkouškách
- nepoužívejte globální proměnné
- zkuste vytvořit co nejvíce funkcí samostatně (knihovní funkce používat jen výjimečně) – vlastní funkce píše do c souboru který je různý od souboru s main, exportujte funkční rozhraní (a deklarace) pomocí h souboru
- zkuste zapnout překlad v ANSI C (ne C++, v překladači nastavit tento překlad, lze-li, soubor by měl mít příponu c a ne cpp – prostředí podle přípony často střídají “na pozadí” příslušné překladače)
- navrhnete vlastní vstupní soubor(y), který(é) prověří činnost algoritmů (např. konec souboru jako konec řádku (či jinak), řádek začíná, končí oddělovačem nebo slovem, prázdný soubor ...)

Zadání příkladu:

- Program dostává jako parametr název vstupního a výstupního souboru
- zkontrolujte, zda jsou předány do main oba názvy souborů
- není-li druhý název přítomen, proveďte výstup na standardní výstupní zařízení
- otevřete soubory
- vytvořte funkci (vlastní), která načte ze souboru řádek, který vrátí jako řetězec – naalokovaný uvnitř funkce na přesnou délku (uvažte způsob předávání a odalokování)
- vrácený řetězec vložte do struktury, která bude kromě ukazatele na řetězec obsahovat délku řetězce
- z takto vytvořených struktur obsahujících řetězce vytvořte pole. Zařazení do pole proveďte pomocí funkce (ošetřete první vložení)
- napište funkci, která vytvoří kopii pole obsahujícího řádky souboru
- napište funkci, která načtené řádky seřadí podle abecedy (anglické). Řazení proveďte pomocí výměny struktur a také pomocí výměny obsahů (tj. dvě řešení. Pro pole a pro kopii – výsledek by měl být stejný).
- napište funkci, která v kopii pole vypustí duplicitní řetězce (obsahuje-li dva (a více) stejných řádků, zůstane jen jeden)
- ve funkci z předaného pole proveďte statistiku s tiskem na konzolu – tj. tiskněte : četnost znaků, počet řádků, počet slov (oddělovačem je vše co není písmeno nebo číslice, může být více oddělovačů za sebou, oddělovačem je i konec řádku, oddělovačem může ale nemusí začínat nebo končit řádek ... Pokuste se řešit pomocí stavového automatu se stavy např. začátek, konec, ve_slove, mimo_slovo, rozdělené_slovo (- před koncem řádky) ...
- Napište funkci, která najde výskyt daného řetězce v podřetězci. Vyhledejte touto funkcí výskyty řetězce (i vícenásobné v jednom řetězci) např. “abc” v načteném textu. Pomocí další funkce za tento řetězec vložte řetězec jiný např. “123” aniž by došlo k přepsání původního obsahu (je potřeba nově naalokovat, a tedy i odalokovat). Proveďte v kopii
- Vytiskněte (upravenou) kopii pole do výstupního souboru.

- odalokujte všechny naalokované řetězce (proměnné), ověřte, zda jsou odalokovány všechny naalokované
- Uzavřete soubory (pokud výstup není na standardní zařízení)
- vyřešte i bez použití struktur
- Cykly realizujte pomocí for, while, do – while, uvědomte si rozdíly (alternativní řešení zkuste např. pomocí řízeného, podmíněného, překladu #define ... # ifdef ...)

Příklad 2.2

- Zkuste co nejvíce vypracovat bez knihovních funkcí
- nepoužívejte globální proměnné
- Proved'te rozbor úlohy
- Zkuste vypracovat nejdříve na papír – jako na písemných zkouškách
- vlastní funkce pište do c souboru který je různý od souboru s main, exportujte funkční rozhraní (a deklarace) pomocí h souboru
- zkuste zapnout překlad v ANSI C (ne C++) (v překladači nastavit tento překlad, lze-li. Soubor by měl mít příponu c a ne cpp – prostředí podle přípony často střídají “na pozadí” příslušné překladače)
- navrhnete vlastní vstupní soubor(y), který(é) prověří činnost algoritmů

Zadání:

- Je dán vstupní textový soubor který je předán jako parametr funkce *main(---)*
- V tomto souboru jsou znaky **S**, **R**, **C**, **X**, **M** následované celým číslem.
- **M** je prvním znakem souboru následované hodnotou určující max. počet využívaných bitů. (např. **M55**)
- Hodnota uvedena u ostatních znaků (tzn. **S**, **R**, **C**, **X**) značí pozici bitu se kterým se pracuje.
- **S** (**Set**) je příkaz pro nastavení bitu na hodnotu 1. (např. **S24**)
- **R** (**Reset**) je příkaz pro nastavení bitu na hodnotu 0. (např. **R2**)
- **C** (**Change**) je příkaz pro změnu hodnoty bitu. (např. **C4**)
- Příkaz **X** (**eXchange**) je zpracován pouze v páru dvou takovýchto příkazů a slouží pro záměnu hodnot bitů na daných pozicích. Pár příkazů **X** nemusí následovat bezprostředně za sebou. (např. **X12 ... X56**)
- Výsledné hodnoty společně s číslem provedeného řádku vytisknete na konzolu.
- Zpracovávanou hodnotu držte v poli (vyzkoušejte různé celočíselné typy) minimální možné délky dle parametru **M**. (např. pro **M55** by měla mít zpracovávaná hodnota co nejmenší počet bitů (nejčastěji nejbližší vyšší příslušný násobek $8 * velikost_typu$))
- Pro vlastní provedení příkazů použijte makra . Proved'te srovnání s realizací pomocí funkcí.
- Využijte operátory : ~, ^, &, /, ?:, <<, >>

3 Jazyk C++.

Cílem kapitoly je seznámit s možnostmi programování v jazyce C++. Jazyk C++ navazuje na programování v jazyce C. Rozšiřuje možnosti jazyka C a programování v něm o další vlastnosti:

- rozšíření neobjektových vlastností C a rozdílů C a C++
- základních objektových vlastností – práci se třídou a strukturou
- mechanismů dědění
- generického programování (šablon), výjimek.

Programování v jazyce C++ umožňuje větší komfort programování pro neobjektové aplikace. Objektové programování potom přináší možnost nového způsobu tvorby aplikací (přemýšlení) a vlastní programování.

(Ne)výhodou jazyka C++ je nutnost lepšího a propracovanějšího návrhu programu a složitější činnost překladače. Výhodou je přehlednější, univerzálnější a znovupoužitelný kód.

Charakteristika a vlastnosti C++

- jazyk vyšší úrovně
- existuje pro něj norma
- objektově orientován
- přenositelný kód
- jazyk C je podmnožinou C++.

Jazyk C++ rozšiřuje programovací komfort, přidává nová klíčová slova, a objektový způsob programování.

Jazyk C používáme k práci s daty a pro jejich získávání na nižší úrovni, vlastnosti jazyka C++ jsou výhodné pro tvorbu složitějších datových struktur a aplikací.

Překladače rozlišují jazyky C a C++ podle nastavení přepínačů v prostředí, případně make file. Dále mohou mít samostatné překladače pro každý jazyk. U prostředí se volba typu překladače může řídit příponou souboru. Přípona pro C++ může být “cpp”, “C” (velké c na rozdíl od malého pro jazyk C – UNIX), hlavičkové soubory mají příponu “h”, “hpp”, “hxx”, nově potom bez přípony.

Pozn.: v nejnovějších verzích se ovšem C a C++ mírně rozcházejí – je nutné řešit pomocí include souborů, kde se za pomoci ifdef a define řeší rozdíly tak, že se pro rozdíly zavedou pomocné proměnné, jimž se přiřadí správná verze). Po normě C++ byla přijata nová norma C, která má oproti C++ něco navíc.

Pozn.: při programování v C++ je však nutné si uvědomit, že se jedná o “vyšší” jazyk než C. Jeho vlastnosti proto většinou nenajdeme v překladačích pro programovatelné obvody, jako jsou jednoduché procesory, hradlová pole či programovatelné automaty. Z tohoto důvodu je při programování v C++ k uvažování o struktuře programu dobré ještě připojit úvahu o tom, kde se bude daná část využívat, či zda se daná funkce nedá použít univerzálně i na jednodušších platformách. Funkce, které je možné “recyklovat” i v uvedených případech, proto vytváříme z důvodu široké přenositelnosti v ANSI C. C++ potom používáme v obslužných (vyšších) vrstvách návrhu, kde vyniknou zase jeho přednosti, a které v konečné fázi volají funkce psané

v ANSI C. Pokud je to možné snažíme se odsunout funkce závislé na použitém operačním systému či překladači až do nejvyšších – nejpozdějších částí návrhu.

Pozn.: Některé neobjektové vlastnosti vznikly jako nutnost při tvorbě vlastností objektových. Jsou proto nutné pro jejich zvládnutí. Kapitoly jsou děleny na neobjektové a objektové vlastnosti. Přitom jsou však řazeny tak, aby bylo možné studovat objektové i neobjektové vlastnosti současně. U každé kapitoly u objektů je uvedeno, které neobjektové vlastnosti je nutné zvládnout pro studium dané kapitoly.

3.1 Stručný úvod do C++.

Cílem této kapitoly je podat obecné shrnutí objektových možností C++. Tato část je určena jako základní přehled pro ty, kteří by rádi měli o objektovém programování rámcovou představu ještě předtím, než se začnou v C++ učit programovat. Tato představa je výhodná i z toho důvodu, že není možné začít programovat objekty hned, ale i pro jednoduchou třídu je nutné nastudovat větší množství látky o jednotlivých prvcích a vlastnostech, ze kterých je nutné složit celek. Tomu, že se jedná pouze o informativní text, odpovídají volněji volené formulace. “Vážný” učební text pokračuje v kapitole 3.2.

Rozšíření C na C++

Při tvorbě složitějších programů v C (velký objem dat, znovupoužití kódu, kontrola stavu dat ...) se začínají objevovat techniky a vlastnosti, o které by bylo dobré C rozšířit pro pohodlnější práci. Tyto prvky jsou přidány k C a vzniká C++ jejímž základem je objektové programování.

Z technik využívaných v C se jedná především o následující: ve větších projektech v C se hlavním pracovním prvkem stává struktura. Má možnost nést velké množství dat, čímž se ušetří práce spojená s předáváním velkého množství parametrů funkcím. Další výhodou spojenou s použitím složených typů je možnost používat stejně pojmenované proměnné v různých strukturách.

Další stále častěji využívaná vlastnost je možnost volat funkci pomocí ukazatele (který může být také prvkem struktury). Výhodou je, že při použití funkce, kterou si třída nese sebou, mohou odpadnout složité procedury pro vyhledání vhodné funkce (např. při stisku tlačítka je nutné provést nějakou akci. Např. po zjištění, které tlačítko bylo stlačeno je nutné najít příslušnou akci. Je-li tato akce přítomna přímo ve vlastnostech tlačítka (jako ukazatel na funkci), lze ji přímo spustit (a není nutno ji hledat, např. pomocí indexů). Výhodou také je, že stejně pojmenovaná funkce může vyvolávat různé činnosti. Máme-li tedy ve struktuře (stejná, či mírně nebo více odlišná) data, se kterými je nutné provést stejnou činnost s přihlédnutím ke tvaru či přítomnosti dat, voláme stejně pojmenovanou funkci (která je zapsána do struktury společně s daty). Např. pro vykreslení tlačítka zavoláme vždy funkci přes ukazatel kreslí. Zde může být uložena adresa funkce kreslící hranaté nebo kulaté tlačítko – tento výběr je udělán při vzniku datového prvku. Program, který volá funkci kreslí vlastně neví jaké tlačítko se bude kreslit – pouze předá řízení na nějakou adresu.

Dalšími vlivy, které vedly ke vzniku objektů je kontrola vzniku a zániku prvků a snaha mít možnost rozšířit základní datové typy. Ve standardním C je omezený počet základních typů, které je možné rozšířit jen pomocí typedef. Složitější (či složený) datový typ s podobnými vlastnostmi jako mají typy standardní vytvořit nelze. Toto omezení odstraňují až objekty, kdy je možné aby uživatelem definovaný typ měl možnosti (např. použití operátorů) standardních typů. Některé možnosti jsou dokonce rozšířené. Jedná se především o možnost kontrolovat vznik a zánik objektu. Při těchto okamžicích se automaticky volají uživatelem definované

funkce a lze tedy při vzniku inicializovat data a při zániku data zachránit (uložit) – což jsou základní prvky programátorské dovednosti.

Dále potom byly do C++ zařazeny mechanismy, které zpříjemní “klasické” programování či umožní lépe řešit práci s objekty.

Podstatnou vlastností je dědění, kterým se rozumí použití vlastností a kódu již napsané třídy jako základu pro tvorbu třídy jiné.

C++ je logickým shrnutím zkušeností s psaním složitějších programů v C. Zavádí nové programovací prvky a techniky umožňující zjednodušit psaní zdrojových textů a udržet kontrolu nad jejich tvorbou. Jedná se především o spojení dat a funkcí s nimi pracujících v jeden celek, s možností automatické kontroly jejich vzniku a zániku. Výhodou je možnost vytvoření vlastního typu s možností použití standardních operátorů. Pomocí dědění je možné využít stávající kód a provádět drobné úpravy činnosti bez nutnosti zásahu do původního kódu což zlepšuje správu kódu využívaného více programátory.

Základem je objekt

Často bývá problém, jak si objekt představit. Z hlediska přístupu nebo stylu práce to není až taková novinka a svým způsobem ho již znáte. Jedná se pouze o to, že se objekt stal součástí jazyka a byla definována práce s ním.

Třída, která je základem objektového programování a na základě jejíhož popisu se objekty chovají, je vlastně spojením dat a činností, které k sobě logicky přísluší. Z oblasti programování v C k ní mají nejbližší knihovny (či již zmíněné struktury). Z hlediska činnosti s proměnnými se vlastně jedná o definici nového typu (k základním typům jako int, float ...)

Standardní knihovny, nebo knihovny dodávané k ovládání HW zařízení, obsahují funkce patřící do stejné skupiny (např. matematické funkce hledáme v math.h, práci s řetězci ve string.h ...), společně s definicí standardních typů a předdefinovaných hodnot. Většinou jsou tyto knihovny rozděleny na prototypy v hlavičkovém souboru a na zdrojový text, který je někdy dodáván pouze ve zkompilovaném tvaru (např. lib – knihovna). Hlavičkový soubor nám definuje rozhraní, nebo-li říká jakým způsobem se s daty pracuje a co se s nimi dá dělat. Vlastní zdrojová část je potom konkrétní implementace, která nás však jako (pouhého) uživatele nezajímá. Pokud v příští verzi dojde ke změně zdrojové části (např. zrychlení algoritmů, opravení chyb) a ne ke změně rozhraní, naše stávající programy se zlepší a přitom budeme používat stejný kód.

I když se Vám objektově orientované programování (OOP) může ze začátku zdát složité, zjistíte, že se to dá dobře používat a že to není až tak nové. Např. vytvoříte-li nový typ, zjistíte, že většinu vlastností už znáte z C (některé typy konstruktorů, operátory – které C používá, akorát se to bere jako samozřejmost (je to jinak implementováno ale funguje to stejně)).

Objektem se rozumí proměnná dané třídy, která obsahuje data a funkce s nimi pracující. Dá se na ni dívat dvěma pohledy – pohledem uživatele, který se ptá jak se s ní pracuje – a pohledem programátora, který ji vytváří. Pohled uživatele je pouze na hlavičky funkcí, které bude používat, pohled programátora je i na kód funkcí. Uživatel ví co třída/objekt obecně dělá (základní popis třídy – například v komentářích nebo doprovodné dokumentaci) a jak s ní pracovat (rozhraní je udáváno v hlavičkovém souboru a říká, jaké funkce volat s jakými parametry, aby se provedla určitá činnost. Akce, které se dějí ve funkcích při volání, nebo způsob uložení dat ve vnitřních proměnných struktury se uživatele netýká, to je práce programátora, který ji vytváří.

Vytvoření třídy

Třída by měla být logickým celkem složeným z dat a funkcí které k nim přísluší. Jako taková by měla být přínosem. Proto není správný přístup, který používají začátečníci při návrhu prvních tříd, kdy se do struktury přesunou již vytvořené funkce z C a ty se potom pouze volají. (I když to neplatí obecně, dá se říci, že při správném návrhu by se jméno třídy mělo objevovat v hlavičkách jejích funkcí, čímž popisujeme jak spolu prvky dané třídy spolupracují a souvisejí).

Pro vytvoření třídy je nutné provést rozbor toho, s jakými daty se bude pracovat a jak se s nimi bude pracovat. Musí se tedy buďto zvolit reprezentace dat a k nim funkce popisující jejich činnost, nebo se vytvoří funkce a k nim vhodná reprezentace dat. Z hlediska použití je důležitější návrh prototypů (hlaviček) funkcí, protože jejich změna je po použití v rozsáhlejší kódu již prakticky nemožná. Změna dat či vlastního kódu funkce (pokud se její výsledná činnost nezmění) nejsou podstatné z hlediska uživatele, a tedy ani z hlediska programu, který je obsahuje.

Rozbor činnosti je možné udělat na základě “zapsaného” programu, kdy si řekneme co budeme potřebovat. Nebo provést přímo rozbor činností podle kategorií do kterých mohou funkce pro práci s objekty spadat. Tyto kategorie jsou:

- vznik a zánik – umožňující vytvoření objektu v definovaném stavu a jeho zánik tak aby “po sobě uklidil”. Dochází zde k nastavení proměnných do počátečního stavu, resp. k odalokování zdrojů popř. uložení dat.
- nastavení a vyčítání dat – umožňující kontrolovanou manipulaci s daty. Tento mechanismus souvisí s přístupovými právy, které je vhodné nastavit tak, že nelze přímo přistupovat k datům. K datům se vždy přistupuje přes metody, které kontrolují správnost a zabráňují poškození dat (např. kontrola mezí, platnosti dat, přístup k nezadané nebo neaktualizované položce, zadání nemožné či nedefinované hodnoty ...)
- manipulace s daty – ostatní práce s objektem a jeho daty. Tyto obhospodařují “běžnou” činnost. Patří mezi ně “klasické” funkce a operátory (je možné stanovit co bude proměnná daného typu dělat objeví-li se jako parametr ve výrazech s operátory (+, *, ...)).
- Vstup a výstup – funkce realizující ukládání a načítání dat objektu (např. z/do souboru, klávesnice/monitoru ...)

Pro daný objekt je nutné stanovit ve vzájemných souvislostech data a funkce s nimi pracující. Funkce patří do kategorií – vznik a zánik, nastavení a vyčítání, manipulace, vstup a výstup. Pro děděné objekty je nutné řešit ještě návaznost na základní třídu.

Manipulace s objektem

Přístup k funkcím a datům je stejný jako přístup k datům struktury. Pro objekt se používá operátor přístupu “.”, pro ukazatel operátor přístupu “->”. Tímto způsobem lze přistoupit nejen k datům, ale lze i vyvolat funkce patřící ke třídě – např. `ooo.pracuj(3,a)`; Důležitou věcí zde je, že objekt, který danou funkci vyvolal, je v ní přítomen jako jeden z parametrů (jeho “předání” provede automaticky překladač). To znamená, že naše funkce `pracuj` má tři parametry – první `ooo` je předán automaticky procesorem (a uvnitř funkce má implicitní jméno `this`, druhým předaným parametrem by byla “3” a posledním proměnná “a”. Kdyby funkci vyvolal objekt `uuu`, `this` zastupoval by tento objekt – to, aby `this` mohl ukazovat vždy na jiný ale správný prvek umožňuje to, že je realizován jako ukazatel. Pomocí tohoto implicitního předání se do dané funkce dopraví všechna data a metody, které daný objekt obsahuje. Data a metody jsou potom přístupné pomocí “`this->`” “

S funkcemi a daty objektu se pracuje jako s daty struktury v C. Objekt, který vyvolá danou funkci, je do ní předán jako implicitní parametr. Ve funkci je přítomen jako ukazatel, který má jméno this.

Vznik objektu

Vznik proměnné daného typu v jazyce C je možný pomocí definice, nebo definice s inicializací (`int i,j=0;`). Objektové vlastnosti umožňují, aby byl při definici objekt vždy inicializován. Inicializace se provádí pomocí volání tzv. konstruktorů, což jsou funkce, které volá automaticky překladač. C++ nově zavádí tzv. přetěžování funkcí, což znamená, že se může v programu vyskytovat více funkcí se stejným názvem (ale musí být rozlišitelné podle počtu nebo typu předávaných parametrů, podle kterých překladač vybere tu nejvhodnější). Proto může existovat i více konstruktorů. Konstruktor bezpečně ví, že je první funkcí, která pracuje s proměnnou a tedy všechna data vlastně nemají smysl – je to pouze “smetl” přítomné právě v paměti.

Konstruktor se v popisu třídy pozná podle jména, které je stejné jako jméno třídy. Je-li jméno třídy `Tmesto`, pak jméno konstruktoru je `Tmesto`. Jelikož je konstruktor volán automaticky, programátor ho nevolá).

Základní konstruktor je bezparametrický (ekvivalent `int i`; Pro třídu `Tmesto` je to podobné: `Tmesto a`; volá `Tmesto(void)` takže je vlastně automaticky přeloženo překladačem do tvaru `Tmesto a.Tmesto()` (!! Pozor - nepíše se to tak, je to tak pouze přeloženo (interpretováno !!)) který má za úkol nastavit data do základního tvaru.

Důležitým konstruktorem je tzv. kopykonstruktor, který slouží k vytvoření kopie z prvku stejného typu (např `int j=i`; neboli `Tmesto b=a`; je totéž co `Tmesto b(a)` a rozumí se tím zápis typu `Tmesto b.Tmesto(a)`). Všimněme si rozdílu v použití operátoru “=”. `int i = j` říká, aby vznikla proměnná `i` na základě hodnoty proměnné `j`, zatímco `i = j`; říká, že hodnota proměnné `j` se má zapsat do proměnné `i`. Zde není rozdíl tak výrazný, ale uvědomme si, že kopykonstruktor pracuje s proměnnou, která ještě nebyla nikdy použita, tj. říká zapiš data druhé proměnné do právě vytvářené, zatímco operátor `=` pracuje s proměnnou, která už delší dobu v programu pracuje a tedy operátor `=` musí nejdříve vyřešit co s daty v proměnné, která se má měnit (např. je-li naalokována paměť, je potřeba ji vrátit a potom teprve řešit alokaci a zápis pro nová data) .

Konstruktor s jednou proměnnou se nazývá konverzní konstruktor. Je tomu tak proto, že inicializuje proměnnou jednoho typu proměnnou druhého typu, tj. konvertuje jeden typ na druhý. Je možné provádět i méně obvyklé konstrukce, které základní typy neumí (např. z řetězce). (např. `int k=3.14`; `Tmesto d = 4.12`; nebo `Tmesto e = “Praha”`; které volají `Tmesto(double)` a `Tmesto (char *)` a volání je `Tmesto d.Tmesto(4.12)` a `Tmesto d.Tmesto(“Praha”)` – správnou funkci vybere překladač – rozliší, zda je parametr `int`, `double` nebo ukazatel na `char`).

Konstruktory s více (libovolně) parametry se u základních typů nevyskytují ale u tříd jsou běžné (např `Tmesto f(“Praha”, 450000, 23, 11000, “CZ”)`)

Je dobré si uvědomit, že stejný mechanismus je používán pro lokální proměnné funkcí, včetně proměnných do nich předávaných hodnotou. Hodnotou se také předává návratová hodnota funkce – zde je proto užitečný kopykonstruktor. Ke vzniku proměnné a tedy i k volání konstruktoru dochází i při dynamickém vytváření proměnné – je ovšem třeba použít C++ způsob vytváření pomocí operátoru `new`. Standardní C alokace konstruktory nevolá.

Při vzniku proměnné (po vyhrazení místa v paměti) je překladačem automaticky volán příslušný konstruktor, který umožňuje nastavit počáteční stav proměnné. Konstruktor se jmenuje stejně jako třída. Konstruktorů je možné pro jednu třídu

vytvořit několik, což umožňuje vznik daného objektu na základě různých variant vstupních dat.

Zánik objektu

Při zániku proměnné je volán destruktork. Je to poslední funkce, která je pro danou proměnnou volána. Proto se zde snažíme “uklidit”. Jedná se především o zavření otevřených souborů, vrácení naalokované paměti, nebo uložení rozpracovaných dat.

Proměnná zaniká na konci bloku, ve kterém byla definována nebo v okamžiku kdy končí dynamická proměnná – to se děje v C++ pomocí operátoru delete. Volání destruktork pro standardní C knihovnu (free) není podporováno.

Při zániku proměnné (před uvolněním paměti zpět systému) je překladačem automaticky volán destruktork, který umožňuje ošetřit stavy vznikající při zániku objektu. Má jméno stejné jako třídy, kterému předchází “~” (~Tmesto). Destruktork je pro danou třídu jediný a nemá žádný parametr.

Metody pro práci s objektem - nastavení, vyčtení, manipulace, operátory

Jedná se o práci s daným datovým typem. Daná funkce, která je členem třídy dostane vždy jako implicitní parametr objekt, který danou funkci vyvolal. Kromě něj jsou standardní formou předány ostatní parametry uvedené v hlavičce funkce. Uvnitř funkce jsou tedy k dispozici všechny proměnné objektu, který funkci vyvolal.

Speciálním typem funkcí jsou operátory, které umožňují pro nově vytvářený typ využít standardní operátory. Pro definici se používá tzv. funkční forma, která je možná i pro volání, zde se však častěji používá zkrácená forma. Funkční volání je např. aa.operator=(bb), ve zkrácené formě aa = bb. Toto volání zavolá funkci operator=(parametr), kde parametrem je příslušný typ, který chceme přiřadit.

Jelikož data, která třídy obsahují jsou většinou značná, bylo by paměťově a časově nevhodné předávat je hodnotou (musela by se při každém předání do funkce vytvořit kopie proměnné). K předávání se hodí např. ukazatel, který má konstantní (a minimální) délku pro libovolný typ. C++ dále zavádí nový způsob předávání pomocí reference, které má podobné vlastnosti z hlediska množství předávaných dat jako ukazatel a lépe se s ní pracuje.

Předávání proměnných pomocí reference či ukazatele s sebou nese jisté problémy jako je např. možnost změnit nechtěně proměnnou (tomu lze zabránit označením proměnné za konstantní pomocí klíčového slova const, kdy překladač pokus o změnu odmítne). Dále může dojít k chybě tím, že je některá z proměnných předána vícekrát (např. a.f(a)) a manipulací s jednou proměnnou vlastně měníme data i ve druhé proměnné (protože je to stejná paměťová oblast) – zde nám nepomůže ani const a tuto situaci je třeba u (každé navrhované) funkce uvažovat a případně i řešit.

Z hlediska práce rozlišujeme tedy funkce “klasické C”, které mají pouze předávané parametry. Dále na členské funkce dané třídy – nazýváme je pro odlišení metody, které mají předávané parametry, ale též implicitně předaný objekt který je vyvolal.

Mohlo by se zdát, že volání metod pro jednoduché operace je nevhodné. Z tohoto důvodu je v C++ použita obdoba maker s funkčním voláním, kdy se nevolá funkce, ale její text je pouze použit jako předpis pro rozvoj kódu. Toto se v C++ nazývá inline metody a programátor má možnost ovlivnit, které funkce budou takto realizovány. Pro jednoduché metody tak nedochází ke složitému funkčnímu volání, ale na místě jejího použití je přímo rozvinut kód s danou akcí pro příslušné parametry funkce.

Členské funkce dané třídy – metody - slouží k manipulaci s daty a objekty dané třídy. Od “klasických” funkcí se metody liší tím, že obsahují implicitně předaný objekt, který je přítomen jako ukazatel `this`. Mezi speciální metody patří operátory, které mají funkční a zkrácený způsob volání (`a.operator=(b.operator+(c))` je totéž co `a = b + c`). Operátory dodržují pravidla obecně platná pro operátory jazyka C. Pro předávání parametrů se používá předávání referencí nebo ukazatelem.

Vstup a výstup

Vstup a výstup není v C typově orientovaný. Tuto nevýhodu umožňuje odstranit již zmíněný princip přetěžování. Překladač v C++ na základě kontextu určí nejvhodnější metodu pro vstup nebo výstup daného typu. Pro vstupní a výstupní operace se využívají tzv. streamy a jsou pro ně přetíženy operátory “<<” a “>>”.

Dědění

Výše popsaná tvorba třídy je základem pro dědění. Při dědění se již vytvořená třída použije jako základ pro třídu novou. Nová třída obsahuje vždy vlastnosti (data a metody) třídy původní. Tyto vlastnosti můžeme použít (necháme to tak), nebo překrýt (tj. nadefinujeme metodu (či data) se stejným prototypem). Překrytí je možné dvěma způsoby – nadefinujeme zcela novou funkčnost (která je vyhledána přednostně, a stará je sice přítomná ale nevolá se) nebo nadefinujeme novou funkčnost (opět volána přednostně), která volá původní metodu (původní metoda udělá co je třeba, nová metoda řeší rozdíly či dodatky k původní metodě).

Z popsaného plyne, že základní třída je vždy nejjednodušší, obsahující společné jádro pro danou množinu. Zděděné třídy z ní vycházejí a rozšiřují její vlastnosti.

Z jedné bazové třídy může dědit více tříd, které tedy mají společné chování. Jednou z možností je vytvoření bazové třídy, která obsahuje pouze předpisy metod, které musí obsahovat každý další prvek a tím definuje pro dané třídy společný interface, díky kterému je se všemi prvky možno pracovat jednotným způsobem. Každá další třída dodá data, pro něž naplní povinné metody.

Je možné dědit i z více tříd najednou – v tomto okamžiku získá nová třída vlastnosti dvou tříd (a jejich předchůdců).

S děděním a “přítomností” bazových metod a dat souvisí i přístupová práva. Pro každou členskou metodu či proměnnou můžeme určit, zda ji může používat pouze vlastní třída nebo i externí uživatel. Programátor dané třídy tedy v přístupu nemá omezení a všechny proměnné se chovají stejně. Pro uživatele, který přistupuje přes definovaný objekt se však některé proměnné mohou jevit jako nepřístupné (ohlásí překladač). Přístupová práva se dědí, ale při dědění se mění podle zvoleného způsobu dědění. (Je opět nutné říci, že se jedná pouze o komfort přístupu k proměnným, ne o jejich přítomnost. Přítomné jsou vždy, nemusí být však dosažitelné přímým zápisem ale třeba jen pomocí metod pro přístup k datům.)

Při návrhu je nutné rozlišovat mezi vazbou dědění (TA dědí z TB), vlastněním (TA vlastní, obsahuje TB, neboli proměnná třídy TB je jednou z položek z dat třídy TA), a prací (TA pracuje s TB, neboli TB je metodě třídy TA předáno jako jeden z parametrů).

Dědění je způsob jak na základě dané třídy vytvořit třídu novou, která má všechny vlastnosti třídy původní a další vlastnosti je možné přidat. Vlastnosti třídy bazové se dědí na základě jejich přístupových práv a způsobu dědění.

3.2 Neobjektové vlastnosti C++ a rozdíly mezi C a C++.

Cílem kapitoly je seznámit s rozdíly mezi jazyky C a C++ z hlediska neobjektového programování. I v jazyce C++ lze psát programy podobně jako v jazyce C (tedy bez použití objektových vlastností). Navíc je oproti jazyku C možné použít některé nové vlastnosti, které zpříjemní a zlepši programování. Jsou to především reference, prostory jmen, přetěžování funkcí.

Některé vlastnosti jazyka C++ jsou přijaty i do C (tuto možnost nebudeme zdůrazňovat (=zamlčíme) z důvodu, že jednodušší překladače mohou mít pouze “původní” C). Z hlediska rozdílů je možné, že jazyk C “předběhne” normu jazyka C++ a bude obsahovat některé vlastnosti navíc (které C++ přibudou až v další verzi normy).

Neobjektové vlastnosti rozšiřují možnosti programovacího jazyka z hlediska komfortu užívání:

- jednořádkové komentáře
- reference umožňující předávání parametrů odkazem
- možnosti používat stejné názvy funkcí s různými parametry – přetěžování
- typově orientovaný vstup a výstup

3.2.1 Komentáře, poznámky.

Cílem je seznámení s novou možností zápisu komentářů v C++. Nové komentáře umožňují zapsat jednoduchou poznámku, jednodušším způsobem. Pro označení začátku je použit dvojznak //, konec je brán automaticky s koncem řádky. Doplňuje víceřádkové komentáře. Výhodou je snadnost zapsání. Nevýhodou omezená délka poznámky.

Test: Jak je možné zapisovat komentáře ve standardním C?

Jazyk C

- obsahuje pouze komentáře označené jako blok, který je ohraničený na začátku a na konci dvojznaky /* a */.
- musíme s nimi pracovat opatrně, protože nejdou vnořené komentáře
- problém – zakomentovat kód ve kterém je komentář

C++

- komentář C zůstává
- nově je zaveden řádkový komentář, který začíná dvojznakem // a končí (implicitně) koncem řádku

výhody

- uvnitř blokového komentáře z jazyka C je možný jeho výskyt (ošetření vnořených komentářů)
- jednoduchost a rychlost

Nově je možné tento typ komentáře použít i v C (nemusí umět všechny překladače C).

Příklad 3.1.1 komentáře

Napište tři definice proměnných typu `int`, a okomentujte je novým a starým typem komentáře.

```
int j; // jednořádkový komentář končí s koncem řádku
int k; // od začátku dalšího řádku opět zdrojový text.
// komentář může začínat kdekoliv, končí na konci řádku.

int i ; /* Starý typ komentáře lze opět použít
        končí ovšem až (dvoj)znakem ukončení komentáře*/
```

KO: jaké komentáře je možné použít v C++? Jaká je výhoda komentáře `//` z hlediska vnořených komentářů při komentování velkého bloku programu?

Příklad 3.1.1a: Vytvořte si jednoduchý zdrojový soubor pro testování komentářů. Pomocí komentářů na jeho začátku vyznačte data o vzniku souboru (kdy, proč, kdo ...).

Kromě stávajících komentářů pro C je zaveden jednořádkový komentář začínající dvojnáskem `//` a končící s koncem řádku.

3.2.2 Deklarace a definice proměnných.

Cílem je upozornit na rozdíly v deklaracích a definicích proměnných v C a C++. Hlavní rozdíl je, že proměnnou je možné definovat kdekoliv, nejen na začátku bloku. Důvodem je snaha omezit neinicializované proměnné. Definice se tedy provede až v místě, kde je možné současně provést inicializaci. Výhodou je snadnější vytvoření nových proměnných a možnost okamžité inicializace, nevýhodou je horší orientace při zjišťování typu proměnné (definice je “ztracená” kdekoliv v textu).

Test: Jaká pravidla platí pro deklarace a definice proměnné v C.

V jazyce C je nutné provést deklarace a definice nových proměnných na začátku programového bloku (tj. za `{`).

V jazyce C++ je možné proměnnou deklarovat nebo definovat v libovolném místě. Důvodem je lepší čitelnost programů a snaha nenechávat proměnné neinicializované. Proměnnou je tedy možné vytvořit až v okamžiku, kdy je známa hodnota, kterou se má inicializovat. Deklarace v bloku je brána jako příkaz.

U příkazu **for** je možné proměnnou definovat v prvním parametru jako lokální proměnnou cyklu. Za cyklem už není známa.

V jazyce C i C++ začíná platnost dané proměnné v místě definice či deklarace, a končí na konci bloku, ve kterém byla definována. Deklaraci musí vždy předcházet klíčové slovo **extern**.

Příklad 3.1.2 Deklarace a definice proměnných

Napište blok programu, ve kterém vyzkoušíte nové možnosti deklarace proměnných a deklaraci v cyklu for.

```
{
float c;          // Klasická C definice proměnné na začátku bloku

for (int i=0;i<10;i++)
{
//zde je i známo
c = fce( );
float b = i * c; /* definice proměnné (s inicializací) mimo
                  začátek bloku v místě, kdy ji potřebujeme */
...
} /* proměnné b a i končí platnost s koncem bloku kde byly
definovány */

c *= i ; // chyba, protože zde už i není známo
} // zde končí platnost proměnné c
```

Pozn.: deklarace proměnné nebo její definice, či inicializace, by neměla být přeskočena (skok, větve **if**, **while**...)

Pozn.: většina překladačů neumožňuje při definici inicializovat prvky pole lokálními proměnnými

```
int f(int x) { int pole[2] = {1,x};...}
```

Pozn.: jméno struktury v C++ může překrýt jméno objektu, funkce, výčtu nebo typu v nadřazeném bloku

```
int a[10];
void fce()
{
    struct a {int b;};
    sizeof(a) ; /* výsledek sizeof je
                  vztažen na pole v c a na strukturu v c++*/
}
```

KO: jaký je rozdíl mezi definicí proměnné v C a v C++.

Příklad 3.1.2a : Zkuste nový způsob definice proměnné např. pro uložení a tisk výsledku při násobení dvou čísel načtených z klávesnice.

Nástin řešení 3.1.2a :

- nadefinujte proměnné pro načtení čísel z klávesnice
- načtěte čísla z klávesnice
- nadefinujte novou proměnnou, kterou inicializujete součinem načtených čísel
- vytiskněte součin

Definici proměnné je možné provést v jazyce C++ v kterékoli části programu (kde může stát příkaz). Proměnná je viditelná od místa definice do konce bloku, ve kterém byla definována.

3.2.3 Předávání parametrů odkazem – reference.

Cílem je ukázat nový způsob práce s proměnnou. Jedná se vlastně o vytvoření zástupného jména (přezdívky) pro proměnnou. Máme tedy jedno paměťové místo, které má více jmen (přes které k němu přistupujeme) = pomocí více jmen pracujeme s jednou pamětí (změna hodnoty přes jedno jméno se projeví i u všech ostatních). Zástupného jména (reference) se používá pro vytvoření dalšího jména již existující proměnné, nejčastěji pro předávání parametrů do funkcí a z funkcí. Mechanismus se nazývá reference. Je alternativou k ukazatelům. Výhodou je jednodušší práce s proměnnými, nevýhodou může být horší přehlednost zápisu.

Test: jak je možné použít ukazatel pro práci s jinou proměnnou? Srovnajte ekvivalentní zápisy pro změnu hodnoty proměnné samostatně (přímo) a přes ukazatel?

V C++ je možné používat odkaz na proměnnou, neboli referenci. Odkaz je vlastně vytvoření zástupného jména dané proměnné. Je to tedy alias, přezdívka. Pomocí reference se pracuje zprostředkovaně přes nové jméno s původní proměnnou. Použití obou jmen má však stejné důsledky. Definice proměnné typu reference musí být spojena s inicializací (na již existující proměnnou).

<code>Typ& aa=j;</code>	definice pro proměnnou typu reference
<code>extern Typ &ref;</code>	deklarace proměnné typu reference
	(zde není nutná inicializace)

Příklad 3.1.3 reference

Zapište do proměnné typu int pomocí reference hodnotu 10. Použijte referenci k přístupu k vnořenému prvku struktury (např. čtyři úrovně vnoření struktury).

```
int j;
int &rj = j; //reference musí být inicializovaná v definici
           // int & je zápis reference na int
rj = 10; // použití reference, je to stejné jako j = 10,
        // protože rj je pouze jiné označení pro j.
        // k hodnotě původní proměnné se přistupuje
        // přímo pomocí proměnné typu reference
double &pom = p1.p2.p3.p4; /* výhodné pro práci s vnořeným
                           objektem struktury - ušetří psaní dlouhého,
                           složitého jména, které v textu často používáme */
pom = 12.23;
```

Test: jak je možné předávat parametry do funkcí v jazyce C? Jak předávat parametry z funkcí (je-li jeden, a je-li jich více)?

V C je možné předávat parametry pouze hodnotou. A to přímo hodnotou proměnné, nebo je hodnotou adresa (ukazatel). Přes adresu je potom možné přistupovat k paměti proměnných a měnit hodnoty těchto proměnných tak, že se změna projeví i vně funkce. Zvláštním typem pro předávání je předávání pole (kdy se prakticky předává adresa počátku pole hodnotou).

Odkaz lze použít pro předávání proměnných do funkcí místo ukazatele. Proměnná je opět pouze jiným názvem pro předávanou proměnnou, kterou je tedy možno změnit tak, že se změny projeví i mimo funkci.

double Real(Typ &c) {c=10; ... } využití k předání proměnné do funkce

Nevýhodou tohoto způsobu předávání je, že v těle funkce se proměnná na první pohled neliší od proměnné volané hodnotou. Tím se ztěžuje orientace, která proměnná se mění pouze ve funkci a které změny se promítnou vně, čímž může dojít ke změně proměnné, kterou nechceme. Řešením je vhodně zvolený název proměnné se zakódovaným typem (např. RefHodnota), nebo použití klíčového slova const (3.2.8).

Příklad 3.1.3b reference

Napište funkci, která vrací více než jednu hodnotu a využijte všech způsobů k vrácení hodnoty. Popište způsob volání této funkce a práci s proměnnými ve funkci.

```
int funkce(int a, int *p, int &h)
/* jedna vrácená hodnota je pomocí návratové hodnoty, druhá se
předává přes ukazatel (tedy adresu se kterou se pracuje) a třetí
pomocí reference (zástupné jméno pro proměnnou s níž se pracuje)
*/
{
    a = 20;    // změna proměnné předávané hodnotou se neprojeví
               // vně dané funkce
    h = 10;    /* měníme externí proměnnou ale vypadá to jako
               změna lokální proměnné */

    *p = 2;    /* přístup přes ukazatel dává tušit, že se pracuje
               s externí proměnnou */

    return *p * h; // výsledek je 4 protože 2 x 2
/* jelikož je použit při volání dvakrát stejný parametr je
použití h i *p rovnocenné z hlediska změn vně,
pro volání s různými parametry to neplatí */
}

// vlastní volání funkce s referencí
{
    int j,i=4;
    i = funkce (j,&i,i);
/* vrací se jedna hodnota pomocí návratové hodnoty a mění se druhý
a třetí předávaný parametr. U druhého parametru (předání adresy
pro práci s ukazatelem) je jasně vidět, že se bude měnit. Předání
odkazem vypadá při volání stejně jako hodnotou což snižuje
```

```
orientaci o tom zda se může ve funkci měnit.*/  
}
```

Při předávání proměnných je nutné si uvědomit, co může být vrácenou hodnotou:

- při vracení proměnné hodnotou `int fce()` to může být cokoli, protože pro předání se vytvoří nová proměnná
- při vracení proměnné ukazatelem `int* fce()` a referencí `int & fce()`, může dojít pouze k vrácení hodnoty, která existuje mimo tuto funkci (lokální proměnné zanikají s funkcí) a tedy v dobře napsaném programu jsou to ty proměnné, které jsou předány v hlavičce funkce jako ukazatel nebo reference (dále potom lze použít globální proměnné, které bychom měli používat minimálně)
- při vracení pomocí ukazatele je možné předat ukazatel na prvek, který dynamicky vznikl v dané funkci. Tento prvek je nutné v dalším programu odalokovat.

Pozn.: Nelze realizovat reference na referenci, reference na bitová pole, pole referencí a ukazatel na referenci.

Pozn.: referenci je možné použít i pro předání jako návratovou hodnotu. To má tu výhodu, že funkci (jejímž výsledkem je L-hodnota) je potom možno použít na pravé i levé straně “rovná se” (např. přístup k prvkům pole). Problémem je, že musí být vrácena smysluplná hodnota pro uložení (není například jednoduché stanovit odkaz na co vrátit při pokusu o přístup mimo meze pole).

Příklad 3.1.3c reference

Napište funkci, která vrátí požadovaný prvek z pole typu `int`. Prvek musí být vrácen tak, aby mohl být použit i k zápisu hodnoty.

```
Int Kanal = 0;    // sem svedu pokusy o přístup mimo pole  
Int Chyba=0;      // a tedy signalizují chybu  
  
int & VratPrvek(int který,int Pole[], int prvku)  
{  
    if ((který < 0) || (který >= prvku))  
    { // pokud došlo k pokusu o přístup mimo pole  
        Chyba = 1; // nastavím chybu  
        Kanal = 0; // zapíši "smysluplnou" hodnotu pro návrat  
        // sem mohla být zapsána "špatná" hodnota při pokusu  
        // o zápis mimo meze pole v minulosti,  
        // toto nemůže být lokální proměnná - zaniká s funkcí  
        return Kanal; // vrátím odkaz na Globální proměnnou  
        // se kterou se bude pracovat v případě chyby  
    }  
    return Pole[který]; // vracím odkaz na vybraný prvek tj.  
    // mimo funkci můžu pracovat přímo s tímto prvkem pole  
}  
  
// volání
```

```

int i,j,Pole[20];

i = VratPrvek (10,Pole,20); // do i se zapíše hodnota desátého
                             // prvku pole
VratPrvek(22,Pole,20) = 10; /* funkce VratPrvek vrátí referenci
                             na proměnnou Kanal (a nastaví chybu), vrácená
                             reference určuje proměnnou, se kterou se bude
                             pracovat, to je proměnnou do které se zapíše
                             10 */
VratPrvek(10,Pole,20) = 30; /* funkce VratPrvek vrátí
                             referenci na desátý prvek pole a s tím se bude
                             pracovat. Bude do něj zapsána hodnota 30 */

```

Pozn.: “velké” typy jako struktura, nebo třída se zásadně předávají odkazem nebo adresou a ne hodnotou (vytváří se kopie, to znamená ztráta času a paměti v případě, kdy kopii nepotřebujeme měnit).

Pozn.: pokud se nám povede dosadit při volání odkazem jiný typ, může překladač vytvořit dočasnou (temporary) proměnnou správného typu, které přes dostupné konverze dosadí hodnotu a s tou je zavolána funkce. Dočasná proměnná po ukončení funkce končí svoji platnost, a není proveden zpětný zápis hodnoty do původní proměnné. Změní-li se tedy uvnitř funkce hodnota, potom se změna neprojeví vně jak očekáváme. (Při volání float & typem int, se vytvoří dočasná proměnná typu float – změny se tedy neprojeví ven do předávaného int-u).

Pozn.: chyby při volání typu fce (x+5) - hodnotou lze, referencí ne. (Může se ovšem stát, že to bude fungovat, protože překladač opět vytvoří temporary (dočasný) objekt vhodného typu, se kterým se pracuje.

KO: co je to reference? Jak se používá k předávání parametrů do funkcí a jaké má výhody a nevýhody oproti předávání odkazem?

Příklad 3.1.3a : Vytvořte funkci, která vymění hodnotu dvou předávaných parametrů typu double za pomoci odkazu.

Nástin řešení 3.1.3a :

- napište hlavičku funkce, která bude mít dva parametry typu reference na double
- ve funkci proveďte výměnu prvků za pomoci pomocné proměnné
- v hlavním programu nadefinujte dvě proměnné typu double, které inicializujete rozdílnými hodnotami. Vytiskněte jejich stav.
- Zavolejte funkci s těmito proměnnými
- Vytiskněte stav a zjistěte zda došlo k výměně hodnot

Příklad 3.1.3d Vytvořte odkaz na prvek struktury. Ukažte práci s daným prvkem klasicky a přes referenci.

Nástin řešení 3.1.3d :

- nadefinujte strukturu A s prvkem xxx typu double
- nadefinujte strukturu B s prvkem aaa typu A
- nadefinujte strukturu C s prvkem bbb typu B
- nedefinujte proměnnou struktury C se jménem ccc.
- Vytvořte si referenci typu double tak aby byla inicializována prvkem xxx, který je obsažen v (v hloubi) ccc.
- Přiřaďte referenci hodnotu 13.14 a ověřte zda se přiřazení objevilo i v xxx.

Reference zlepšuje možnost pracovat s jednou proměnnou pomocí více názvů. Toho se zvláště využívá v případech, kdy je požadováno aby změny proměnných provedené ve funkcích se projeví i v proměnných, se kterými byla funkce volána. Reference je náročnější na orientaci, se kterou proměnnou se pracuje a která se tedy mění.

3.2.4 Operátor příslušnosti :: (kvalifikátor).

Cílem je uvést operátor příslušnosti. Pomocí tohoto mechanismu je možné rozlišit proměnné patřící do různých prostorů. Jednotlivé prostory jsou od sebe oddělené, což umožňuje, aby v nich byly stejně pojmenované proměnné (či funkce). Operátor příslušnosti, který předchází se jménem prostoru jméno proměnné potom umožňuje rozlišení proměnných patřících k jednotlivým prostorům (prostorem je i třída). Nevýhodou je složitější práce, výhodou je možnost oddělení částí projektů a možnost užívat stejně pojmenované proměnné v rozsáhlých projektech (např. tvořených více autory).

Častým problémem při spolupráci více lidí na jednom projektu bývá situace, kdy si několik autorů vybere pro svou proměnnou, funkci či datový typ stejný název. V okamžiku kdy se jejich kód potká v jednom programu, dochází ke kolizím jmen. I z tohoto důvodu byly zavedeny prostory jmen (3.2.19). Tyto umožňují ohraničit a označit jménem daný kód. Proměnné v této oblasti pak dostávají ke svému jménu i jakési “příjmení”, kterým se od sebe liší. Podobným způsobem se od sebe liší i proměnné se stejným názvem v různých strukturách. “Příjmením” je pro ně jméno (typu) struktury, ve které jsou obsaženy.

Operátor příslušnosti “::” slouží k odlišení přístupu k datům a funkcím z různých prostorů. Umožňuje přístup ke globálním proměnným. Jelikož je možné mít stejné názvy proměnných a metod v různých třídách nebo prostorech, je nutné je odlišit. To se děje pomocí operátoru příslušnosti tak, že se vlastnímu názvu proměnné či funkce předradí jméno příslušné třídy (prostoru), ke kterému patří, takže její plný název je potom **Prostor::JménoProměnné** nebo **Prostor::JménoFunkce**. Rozšíření **Prostor::** slouží k vyhledání správné funkce z více možností. Je-li zřejmé co máme na mysli (a je-li to možné) potom se **Prostor::** neuvádí.

Např. má-li lokální proměnná jméno stejné jako globální, potom k lokální se přistupuje přímo – např. **stav = 10**, zatímco ke globální se přistupuje **::stav = 10**, (kde globální prostor je vlastně beze jména). Podobně se přistupuje k funkcím (např. přetěžujeme-li funkci (např. new, delete, sqrt) a chceme volat funkci původní).

Operátor příslušnosti společně s jmennými prostory umožňuje skrytí identifikátorů a tedy zamezuje kolizím mezi stejnými jmény.

// tento příklad není úplný, ale dává představu o tom

```

// k čemu slouží operátor příslušnosti
Struct A {float aaa;};
Struct B {float aaa;};

A::aaa          // proměnná aaa ze struktury A
B::aaa          // proměnná aaa ze struktury B
// pomocí operátoru :: rozlišujeme, které aaa máme na mysli.
// aaa je skryto ve struktuře A,B neboli patří do prostoru A,B

```

KO: k čemu se používá a jak se nazývá operátor “ :: ”?

Příklad 3.1.4a: napište program, který bude mít stejně pojmenovanou lokální a globální proměnnou. Zkuste s nimi pracovat v jedné funkci současně (s globální přes operátor ::).

Nástin řešení 3.1.4a :

- nadefinujte globální proměnnou typu int a stejně pojmenovanou lokální proměnnou typu double.
- V definici inicializujte dané proměnné různými hodnotami
- vytiskněte stav obou proměnných (ke globální přistupte pomocí operátoru příslušnosti)

Operátor příslušnosti slouží k rozlišení (stejně pojmenovaných) identifikátorů, které jsou umístěny v prostorech s různými jmény. Prostorem jmen je i třída nebo struktura. Operátorem příslušnosti se rozliší ke kterému typu daná proměnná patří.

3.2.5 Přetížení funkcí.

Cílem je ukázat možnost použití více funkcí se stejným jménem – přetížení. Na rozdíl od jazyka C, kde musí být identifikátory jedinečné, je v C++ možné vytvořit více funkcí stejného jména. Funkce je nutné odlišit (počtem nebo typem parametrů) tak aby překladač mohl podle kontextu určit, kterou funkci vybrat. Tuto vlastnost je možné též označit jako překrývání názvů funkcí – tj. více funkcí může mít stejné jméno - polymorfismus. Výhodou je snadnější práce a orientace v textu. Nevýhodou jsou problémy související s volbou správné funkce v okamžiku kdy je nutné přetypovat parametry (dvě či více možností přetypování vedoucí k různým funkcím).

Test: co se stane jsou-li v programu jazyka C obsaženy dvě funkce se stejným jménem?

V jazyce C je funkce jednoznačně odlišena svým (jedinečným) názvem.

V C++ se v programu může vyskytovat více funkcí se stejným názvem. Tento mechanismus nazýváme přetěžování - polymorfismus. Přetížené funkce je však nutné od sebe odlišit počtem nebo typem parametrů.

Přetížení funkcí je možné když:

- Mají různý počet formálních parametrů (výběr správné funkce se provede na základě počtu parametrů) -
`double Fce(double d)` a `double Fce(double d,int i)`
volání: `Fce(3)` a `Fce(3,5)`
- Liší se v typu alespoň jednoho formálního parametru (výběr správné funkce se provede na základě shody typu parametrů) `double`
`Fce(double d)` a `double Fce(int d)` volání:
`Fce(3.14)` a `Fce(10)`
- Patří k různému prostoru jmen - lze přetěžovat i v rámci různých prostorů (struktur, či tříd, které zapouzdřují – oddělují funkce) kde mohou být i s totožnými parametry. Není-li možno správnou funkci určit z kontextu, uvede se kvůli rozlišení před názvem prostor jmen (struktura, nebo třída, ke které patří) jako bližší specifikace – `Prostor::Jméno`. (výběr správné funkce se vybere pro aktuální prostor (při dědění též bazové prostory) , nebo explicitně uvedený prostor
`double String::Fce(double d,int i)` a
`double Komplex::Fce(double d,int i)`

Typ návratové hodnoty není podstatný pro rozlišení. Dvě stejnojmenné funkce nelze odlišit pouze návratovou hodnotou. Návratové hodnoty u stejnojmenných funkcí však mohou být různé.

Výběr správné funkce provádí překladač.

Pozn.: pozor na přetypování. Překladač se snaží najít vhodnou funkci i za pomoci implicitních konverzí. Pokud máme např. funkci s parametrem `int` a `float`, kterou voláme s parametrem `double`, může překladač uskutečnit jak konverzi na `int` tak na `float` – jelikož jsou pro něj tyto konverze rovnocenné, zahlásí chybu.

```
double abs(float d) a int abs(int i)
volání špatně: abs(3.13) (kolize, protože 3.13 je double!),
a jak vyřešit: abs((float)3.14)
```

Pozn.: není rozdíl mezi `int f(int i)` a `int f(int &i)` protože z parametru při volání není možné rozlišit, kterou zavolat

Pozn.: z důvodu přehlednosti raději nevyužívat rozdíl mezi nekonstantním a konstantním parametrem: `int f(int i)` a `int f(const int i)`

Pozn.: pole a ukazatele jsou také nerozlišitelné

```
f(char *a) = f(char a[ ]) = f(char a[5])
```

Pozn.: `typedef` zakládá nové jméno typu a ne nový typ, proto nelze původní typ od `typedef` odlišit

Příklad 3.1.5 Přetížené funkce

Napište přetížené funkce **max** pro vyhledání maxima ze dvou a tří parametrů. Popište volání.

```
double max(double p1, double p2) // max se dvěma parametry
// používám double jako nejpřesnější typ
{
```

```

    return p1 > p2 ? p1 : p2;
}

double max(double p1, double p2, double p3)
/* max se třemi parametry - přetížení max */
{ // volám stejnojmennou fci s jiným počtem parametrů
    return max(p1,max(p2,p3));
}

{
    double p;
    p = max (33.4, 85 );           // volá max se dvěma parametry
    p = max (p    , 45,23.4);     // volá max se třemi parametry
}

```

KO: Které funkce lze označit za přetížené? Čím se musí přetížené funkce lišit?

Příklad 3.1.5a: Napište tři funkce, které vrací maximální hodnotu z předaných parametrů – první pro dva parametry **int**, druhou pro dva parametry **float** a třetí pro tři parametry **float**. Vyzkoušejte je pomocí volání. Zkuste volání s parametry (**double, double**) nebo (**int, float**) a s (**double, double, double**) a (**int, float, int**). Proč dojde k chybě pro dva parametry a ne pro tři?

Jazyk C++ zavádí možnost použít stejné jméno pro více funkcí – **polymorfizmus**, **přetížení**. Aby bylo možné určit správnou funkci, je nutné aby se od sebe stejnojmenné funkce lišily v počtu nebo typu argumentů, nebo aby patřily do jiného prostoru jmen.

3.2.6 Implicitní parametry.

Cílem je popsat možnost definice jedné funkce, kterou je možné volat s různým počtem parametrů (ale ve funkci používat vždy stejný počet parametrů). Díky tomuto mechanismu je možné neuvádět některé parametry při volání funkce. Používá se především v případě, že ve většině případů volání má parametr jednu hodnotu a výjimečně hodnotu jinou. Pokud dochází k volání s touto “nejčastější” hodnotou, potom není tuto hodnotu nutné vyplňovat (uvádět) a na základě deklarace funkce jsou překladačem doplněny implicitní hodnoty. Definice a deklarace funkce je provedena pro maximální počet parametrů s tím, že v deklaraci jsou uvedeny implicitní hodnoty parametrů pro dosazení. Funkce je potom univerzální pro různý počet parametrů. Výhodou je možnost úspory textu při volání, nevýhodou je nutnost využívat implicitní parametry od konce a bez přeskakování.

Pokud při volání funkce nabývá některý z parametrů ve většině případů stejné hodnoty, můžeme ho definovat pomocí implicitního přiřazení pro daný parametr (např. u výpočtů s komplexními čísly zadáváme čísla reálná (zadáme jeden parametr, druhý je nula) nebo komplexní(dva parametry)). To znamená, že při deklaraci uvedeme nejčastější používanou hodnotu jako implicitní a při volání ji již nezadáme explicitně.

Pozn.: v hlavičce funkce je uvedena hodnota, kterou překladač dosadí při volání funkce, v případě, že ve volání je tato hodnota vynechána.

Aby bylo možné jednoznačně určit, které parametry nejsou uvedené, platí následující pravidla:

- Implicitní parametry musí být uvedeny jako poslední v seznamu parametrů (za implicitním parametrem nesmí být parametr bez implicitní hodnoty).
- Při volání musíme vynechat všechny poslední parametry souvisle – není možné vynechat parametr uprostřed seznamu.

Implicitním parametrem může být libovolný výraz obsahující konstanty, proměnné a funkce:

`int f(float a=4, float b=random())` je dosažitelná pomocí volání `f()`, `f(22)`, `f(4,2.3)` kdy se volá `f(4,random())`, `f(22,random())`, `f(4,2.3)` a z toho plyne, že takto deklarovaná funkce slouží pro (koliduje s, zastupuje, je volána i pro) `f(void)`, `f(float)`, `f(float, float)`

Pozn.: i zde platí pravidla pro přetěžování funkcí a tedy svým způsobem je tato funkce použitelná (a tedy může dojít ke kolizi) i pro volání s parametry, které lze na float konvertovat.

Pozn.: implicitní parametry je lepší uvádět do hlavičkového než do zdrojového souboru – mohou být uvedeny pouze v jednom místě. Tedy raději do deklarace než do definice.

Pozn.: implicitní parametr může být dodán i jako výsledek volání nějaké funkce

Příklad 3.1.7 implicitní parametry

Napište funkci, která čte znak z otevřeného souboru, není-li soubor ve volání zadán, načte se z klávesnice – řešte implicitním parametrem. Ukažte možná volání.

```
#include <stdio.h >

int Cti_Znak(FILE *vstup = stdin)
{
    return getc(vstup);
}

// volání s jedním parametrem - vstup je soubor
i = Cti_Znak(fr);

// volání bez parametrů - vstup je klávesnice
i = Cti_Znak( );
```

KO: Které z funkcí `int f(int)`, `int f(char)`, `char f(int, int)`, `float f(float)`, `int f(void)` přetěžuje funkce `void f(float g, int h=4)` a které `f(int g=3, float h)`?

Příklad 3.1.7a: napište funkci pro výpočet vzdálenosti komplexního čísla od počátku. Dva parametry, jedna návratová hodnota. Za pomoci implicitního parametru ošetřete převážný výskyt reálných čísel ve výpočtech (při voláních s přímými parametry).

Pomocí implicitních parametrů je možné psát jednu funkci pro volání s různým počtem parametrů. Nejčastěji používané hodnoty jsou uvedeny v deklaraci funkce jako implicitní – jsou dosazeny v případě neuvedení jiných hodnot při volání. Implicitní parametry jsou uváděny od konce deklarace. Ve volání je opět možné vynechávat parametry pouze postupně od posledního.

3.2.7 Přetypování.

Cílem je upozornit na nový způsob explicitního přetypování, které je rozšířeno o možnost ve stylu funkčního volání. V jazyce C je možné provádět explicitní přetypování. Tato činnost je možná i v jazyce C++, přidává se ale ještě nový způsob (a dále také úplně nové způsoby přetypování určené pro třídy - kap. 3.4.5). Nové použití připomíná volání funkce, která má stejný název jako požadovaný výsledný typ. Toto umožňuje vytvářet funkce vhodné pro přetypování i u vlastních tříd.

Test: jak se provádí explicitní konverze (přetypování) v C?

Přetypování (konverze) slouží ke změně typu proměnné. V jazyce C se používá ve tvaru - **(int) f** - kdy proměnná f je přeměněna na typ int

V jazyce C++ je nově zavedená konverze pomocí funkčního volání ve tvaru - **int(c)** – což je základ uživatelského přetypování (u tříd).

Pozn.: způsob přetypování používaný v C nemusí fungovat v dalších verzích C++

Příklad 3.1.7 Přetypování

Vyzkoušejte nový typ přetypování při dělení celočíselných proměnných při požadavku na reálný výsledek.

```
{
    int i = 10 , j = 3;

    float f = float( i ) / j;
    /* přetypování pomocí funkčního volání - výsledek je float 3.33333
    */
}
```

Druhý způsob umožňuje definovat přetypování jako vlastnost vytvářených tříd a tím například umožnit jejich převod na standardní (nebo jiné definované) typy.

KO: Kdy se používá explicitní přetypování? Kdy implicitní? Jak je možné provést přetypování v C a v C++?

Příklad 3.1.7a: Napište část kódu, ve které použijete přetypování z float na int k zaokrouhlení kladných hodnot. Použijte oba dva způsoby přetypování (pro C i C++).

Jazyk C++ zavádí možnost přetypování pomocí “funkčního” volání. Přetypovávaná proměnná je na místě parametru a název funkce je shodný s požadovaným výsledným typem převodu – `int i; float f; f = float(i);`.

3.2.8 Modifikátor `const`.

Cílem je popsat možnost “tvorby” konstantních proměnných. Tato vlastnost slouží pro vytváření proměnných, které nemají měnit svoji hodnotu. Je realizována pomocí použití modifikátoru `const`. Používá se například pro tvorbu konstant, kdy na rozdíl od konstanty definované pomocí `#define` preprocesoru je možné určit přesný typ konstanty. Dále se také používá při volání funkcí, v případě že nechceme aby se proměnná ve funkci změnila. Tento způsob je použitelný i v novější normě C. Výhodou je, že přímo z deklarace proměnné (např. v hlavičce funkce) vidíme, že se proměnná nemění a toto kontroluje překladač.

Test: Jak je možné nadefinovat konstantní proměnnou v (starším) C?

Potřebujeme-li používat proměnnou, která má mít určitou neměnnou hodnotu (např. eulerovo číslo, π ...), je možné použít definici proměnné určitého typu s použitím klíčového slova **`const`**. Hodnotu v definici (s inicializací) není možné měnit. Případnou snahu o změnu kontroluje překladač a označí ji jako chybu.

Tento způsob je alternativou k definici konstantních hodnot pomocí `define`, kdy se typ určuje pomocí přípon (`0xFFu, 0xFFL` ...).

`const` umožňuje zadání konstanty s daným typem:

`const long double PI=3.1415;`

Dále se tento modifikátor používá k označení parametrů předávaných funkcím, které nemají být ve funkci měněné. **`int fce(const int *i)`**. Toho se nejčastěji využívá při “úsporném” předávání parametrů pomocí ukazatele či reference, kdy se změna může projevit mimo funkci (což nechceme). Při pokusu o změnu této proměnné ve funkci překladač ohlásí chybu.

<code>T</code>	je proměnná daného typu
<code>T *</code>	je ukazatel na daný typ
<code>T &</code>	reference na T
<code>const T</code> resp. <code>T const</code>	deklaruje konstantní T (<code>const char a='b';</code>)
<code>T const *</code> resp. <code>const T*</code>	deklaruje ukazatel na konstantní T
<code>T const &</code> resp. <code>const T&</code>	deklaruje referenci na konstantní T
<code>T * const</code>	deklaruje konstantní ukazatel na T

T const * const resp. const T* const deklaruje konstantní ukazatel na konstantní T

Pozn.: S konstantní proměnnou překladače většinou pracují jako s přímým konstantním parametrem, který dosazují jako hodnotu v době překladu (tedy obdobně jako define). Z tohoto důvodu by se tato hodnota neměla předávat jako parametr funkcí odkazem (některý překladač je toho ovšem schopen díky vytvoření dočasné proměnné s danou hodnotou, která po návratu z funkce zanikne, při přísnějších překladech není povolen tento typ předání), nebo získávat její adresu.

Pozn.: Má-li konstantní parametr platit pro více modulů, potom u:

- C je **const char a='b'**; ekvivalentní **extern const char a='b'**; a pro viditelnost pouze v jednom modulu je nutné zde uvést **static const char a**
- C++ je **const char a='b'**; ekvivalentní **static const char a='b'**; a pro viditelnost v ostatních modulech je nutné uvést **extern const char a**

Pokud tedy v definici konstantní proměnné vždy uvádíme u const static nebo extern nemůžeme nic zkazit – zvláště pak pokud má být kód přenositelný mezi C a C++.

Pozn.: použití modifikátoru const pro parametry metod třídy může mít za následek chybu překladu plynoucí z použití metod na předávané const objekty. Tyto vlastnosti jsou popsány v kapitole 3.3.12.

Příklad 3.1.8 const

Napište funkci, která bude mít dva parametry – první předaný pomocí konstantní reference a druhý pomocí reference. Pokuste se oba parametry ve funkci změnit.

```
Void funkce (double const &h1, double & h2)
{
    h1 = 10; // chyba - parametr const
    h2 = 10; // lze - není const
}

{
    double p = 10, r = 20;
    funkce (p,r);
}
```

KO: Jak nadefinovat konstantní proměnnou dané hodnoty a typu? Proč se používá const v hlavičkách funkcí, zvláště pak u ukazatelů a referencí?

Př 3.1.8a: Nadefinujte si konstantní (globální) proměnnou PI. Napište funkci pro výpočet obvodu kruhu za pomoci této proměnné. Hodnotu průměru předejte do funkce odkazem a zkuste tuto uvnitř funkce změnit. Co se stane v tomto případě? Jak se situace změní je-li v hlavičce uveden jako const?

Jazyk C++ umožňuje pomocí klíčového slova const nadefinovat proměnnou daného typu a dané hodnoty. Jako konstantní je možné též označit proměnné předávané do

funkcí. Proměnné označené jako `const` není možné změnit. Při pokusu o změnu překladač zahlásí chybu.

3.2.9 Alokace paměti – `new`, `delete`.

Cílem je uvedení nového způsobu práce s pamětí (získání a vrácení). Jelikož C++ rozšiřuje vlastnosti jazyka C o objektové vlastnosti, je nutné aby tyto byly použity i při dynamickém vzniku a zániku prvků. Bylo rozhodnuto, že původní způsob (skupina funkcí `xxalloc`, `xxfree`) zůstane zachována v původní formě a pro potřeby C++ bude použito nových funkcí pro práci s pamětí - (klíčová slova) `new` a `delete`. Tyto funkce jsou typově orientovány = dokáží zjistit typ a tedy i rozměr typu pro který alokují paměť. Existují ve verzi pro jednu proměnnou a pro pole. Výhodou kromě možnosti používat typ je navázání na objektové vlastnosti jazyka C++ (jsou volány konstruktory a destruktory při vzniku a zániku dynamického objektu).

Test: Jak je možné naalokovat dynamickou proměnnou v jazyce C?

Na rozdíl od knihovních funkcí pro práci s pamětí v jazyce C, které pracují pouze s délkou požadovaného paměťového bloku, zavádí C++ typově orientovaný způsob práce s dynamickým přidělováním paměti. Umožňuje alokovat jednotlivé proměnné, nebo pole proměnných daného typu. Společně s alokací u objektů tříd zajišťuje i jejich inicializaci pomocí konstruktorů. Nový je též způsob odalokace paměti, který u tříd zajistí “úklid” proměnné pomocí volání destrukturu.

Pro alokování paměti slouží klíčové slovo `new`, pro vrácení paměti klíčové slovo `delete`. Při alokaci se uvede požadovaný typ alokované proměnné a případně i počet prvků. Vracená hodnota je ukazatel typu `void`, který ukazuje na počátek paměťového bloku, do kterého je možné umístit požadovaný počet proměnných daného typu.

<code>char *p=(char*) new char,</code>	alokace jedné proměnné typu <code>char</code>
<code>delete p;</code>	a příslušné vrácení paměti
<code>*pp= (char *) new char[10*i];</code>	alokace pole prvků <code>char</code> o délce <code>10*i</code>
<code>delete[] lpp;</code>	a příslušné vrácení paměti

Pozn.: `new` vrací `NULL` pouze v okamžiku, kdy se volá s parametrem `nothrow`.

```
MyClass* fPtr3 = new( nothrow ) MyClass;  
připomíná “klasickou” alokaci pole (příklad z Borland Builder)  
int *pn;
```

```
// nothrow verzi používáme v případě, že upřednostňujeme  
// test na NULL ukazatel před výjimkami  
pn = new(nothrow) int[5000000]; // umístění nothrow  
if(pn != NULL) { // nyní “klasický” test  
    // v případě úspěšné alokace  
}
```

V případě chyby (nemůže-li naalokovat), volá `new_handler` – funkci, která není definovaná (je `NULL`) ale kterou je možná nadefinovat a ošetřit chybu. V případě neúspěchu této funkce je vyvolána výjimka. Nehledáme tedy návratovou hodnotu `NULL`, ale “odchytíme” výjimku (viz. kapitola o výjimkách – exception).

následující příklad je z helpu Borland Builderu – alokace 2D pole

```
long double **data;    // dvourozměrné pole

try {                  // v následujícím bloku "odchytáváme" výjimky.
    data = new long double*[m]; // alokace pro řádky
    for (int j = 0; j < m; j++)

        data[j] = new long double[n]; // alokace řádků
// obě dvě new v případě chyby alokace "hodí" výjimku chyby
} // konec bloku kde odchytáváme výjimky
catch (std::bad_alloc) // specifikace typu "chytané" výjimky
{ // v případě, že došlo k chybě výjimky pro alokaci
// vstoupíme do tohoto bloku, kde chybu (z obou new)
// můžeme ošetřit
    cout << "Could not allocate. Bye ...";
    exit(-1);
} // konec bloku ošetření výjimky
// pokračování programu
```

Pozn.: na rozdíl od funkcí v C je při alokaci paměti pomocí `new` volán na každý vznikající prvek třídy (i pro každý objekt v alokovaném poli) konstruktor. Stejně tak pro každý odalokovávaný prvek je volán destruktory. Proto je nutné rozlišovat mezi `delete` a `delete[]`. Oba odalokují paměťový blok. První však zavolá jeden destruktory, druhý potom destruktory pro všechny prvky pole. Používat pro práci s pamětí pro objekty funkce z jazyka C je tedy špatný přístup, protože zde nefungují dva důležité předpoklady života objektu – kontrolovaný vznik a zánik.

Výše uvedené použití operátorů (operátory jsou popsány blíže v 3.2.15) `new` a `delete` má tyto definice:.)

```
jedna proměnná (Je možné tyto přetížít pro každou navrhovanou třídu
void* T::operator new(size_t)      void T::delete (void *)
pole
void* T::operator new[] (size_t)   void T::delete[] (void *)
```

`[]` určuje, že se jedná o pole (volají se konstruktory (resp. destruktory) všech prvků pole)

`new T` je totéž co `new(sizeof(T))` a použije `T::operator new()` (to je. Použije se operátor alokace `new` v závislosti na právě požadovaném typu)

`new T[5]` je totéž co `new (sizeof(T) * 5 + Hlavička_alokace)` použije `T::operator new[] ()`

`new (2) T` je totéž co `new (sizeof(T),2);` - `new` může mít i více parametrů, první musí být `size_t`, ostatní parametry potom slouží jako parametry konstruktory (lze pouze pro jeden prvek, pole je vždy vytvářeno za pomoci implicitních konstruktory)

```
// definice přetížení operátoru delete pro vlastní třídu
void T::operator delete(void *ptr)
{
    if (změna)
        Save("xxx"); /* tady si uděláme co potřebujeme navíc
                        oproti "standardu" */
    if (ptr!=NULL)
        ::delete ptr; /* voláme "standardní", neboli "globální"
                        delete pro uvolnění paměti */
}
```

Pozn.: předdefinovaných new a delete se používá se pro vlastní memory management

Pozn.: Parametrem size_t nemusí být konstanta

Pozn.: při chybě by měl new volat výjimku

Pozn.: new se dědí (takže pokud alokuje paměť pro daný prvek, musí se brát v úvahu to, že třída při dědění bobtná – použít správně sizeof, předdefinovat new).

Pozn.: new je vlastně funkce, která má tři části. V první požádá funkci pro alokaci o příslušně velkou paměť, ve druhé části zajistí inicializaci objektů pro které byla paměť alokována a nakonec vrátí ukazatel na tuto paměť.

Příklad 3.1.9 new a delete

Vytvořte část kódu, kde naalokujete pole 20 intů a následně ho odalokujete. Naalokujte také jeden prvek float.

```
{
    int * pole = (int *) new int[20];    // alokace pole
    float *pf = (float *) new float;     // alokace jednoho prvku

    delete [ ] pole; // odalokování pole
    delete pf;       // odalokování jednoho prvku
                    // odalokování je možné v libovolném pořadí
}
```

KO: která klíčová slova slouží k alokaci a odalokování v C++? Je rozdíl mezi alokací jedné proměnné a pole? Proč se musí používat pro objekty?

Příklad 3.1.9a: Napište funkci a v ní naalokujte pole s počtem prvků, které je dáno jako parametr. Toto pole naplňte hodnotami. V hlavním programu po volání této funkce hodnoty vytiskněte a pole zrušte. (Nepoužívejte globální proměnné).

Jazyk C++ zavádí nový způsob dynamické práce s pamětí. Zavádí klíčová slova new a delete, která jsou z hlediska svých vlastností považována za operátory. Operátor new na základě zadaného typu a počtu prvků naalokuje příslušný paměťový blok. Operátor

delete tento blok uvolní. Existují dvě verze operátorů pro práci s pamětí – pro jeden prvek a pro pole prvků. Při alokování objektů jsou volány konstruktory a při odalokování destruktory.

3.2.10 Enum.

Cílem je upozornit na odlišnosti v přístupu k enum v jazyce C a C++. Typ enum nemá v C++ tak úzkou vazbu na typ int jako v jazyce C. V C++ vzniká nový typ, pro jehož reprezentaci je použit vhodný (jakýkoli) celočíselný typ. Liší se proto přístup při konverzích i v reprezentaci hodnot. Pro konverze (především na enum) platí přísnější pravidla.

V jazyce C je možné přiřazovat (přecházet) libovolně mezi enum a int (enum je reprezentován intem a nevzniká nový typ). Velikost typu je stejná jako velikost int.

`enum b{A};` potom `sizeof(A)` je v C stejné jako `sizeof(int)` .

V C++ je jméno výčtu jménem typu. Lze mu tedy přiřadit pouze konstantu daného typu.

`enum b{A}; sizeof(A)` je v C++ `sizeof(b)` – záleží na konkrétních hodnotách. enum bývá většinou pokryto některým standardním celočíselným typem (int, long ...), ale konkrétní velikost může být v různých místech programu zvolena překladačem jinak. Při použití není nutné psát enum.

Je-li enum součástí třídy, potom je nutné jeho položky z vnějšku používat přes přístup pomocí operátoru příslušnosti tj `T :: název`.

Pozn.:

<pre>enum { Počet=80; Int pole [Počet]; };</pre>	Tímto způsobem lze “schovat” často používaný název a přitom mít konstantu
---	---

Pozn.: enum nelze v C++ přiřadit z int, ale lze explicitně přetypovat. Např. pro enum `DEN={PO,UT,...}` a přiřazení pomocí konverze z int je `den = DEN(3)`, kde přetypovávané číslo musí být celočíselné a musí ležet uvnitř intervalu hodnot použitých v deklaraci enum (tj. teoreticky i hodnota, která není v deklaraci (explicitně) přítomná.) Hodnoty jsou přiřazovány překladačem, nebo je možné je explicitně uvést.

Pozn.: U některých překladačů existuje přepínač “`treat enum like int`”, kdy je možné zvolit reprezentaci enum typem int

Pozn.: přesněji by měly enum v C mít rozměr signed int.

KO: jakého typu nabývají hodnoty v enum?

Zatímco v jazyce C je enum reprezentováno typem int a je možné mezi int a enum volně přecházet, jazyk C++ může pro enum zvolit libovolný celočíselný typ. Pro přechod mezi int a enum je nutné explicitní přetypování.

3.2.11 Inline funkce.

Cílem je prezentovat možnost vkládání “funkcí” přímo do kódu bez funkčního volání. Inline funkce znamená, že pro implementaci není použito funkční volání, ale funkce slouží pouze jako předpis pro rozvinutí kódu. Inline funkce jsou obdobou maker s parametry v jazyce C. Výhodou je lepší ladění, snadná možnost měnit funkci a inline funkci. Nevýhodou je, že musí být uveden typ parametrů.

Test: co jsou to makra s parametry a k čemu slouží ?

Inline funkce jsou obdobou maker v C. Standardně je funkce volána pomocí skoku do funkce a návratu z funkce (call, return), s příslušným obhospodařováním zásobníku (vytvoření a rušení lokálních proměnných, úklid registrů, režijní činnosti pro volání funkcí ...)

Pro označení takovéto funkce se používá klíčové slovo inline uvedené na začátku definice funkce:

```
inline int plus2(int a) {return a+2;}
```

Při “volání” této funkce, tj. pro zápis kódu ve tvaru:

plus2(3+b*c) překladač použije definici k vytvoření kódu, který se vloží do právě překládaného místa. Parametr je zde vypočten a jeho výsledná hodnota s příslušnou konverzí je dosazena do míst, kde se používá (rozdíl od maker), podobně jako lokální proměnná z předávaného parametru u funkcí.

Jelikož se jedná pouze o předpis, umísťují se deklarace inline funkcí do hlavičkových souborů - jsou pouze předlohou pro kód ale netvoří ho.

Inline funkce šetří režii na práci se zásobníkem a volání funkce, prodlužují však kód. Proto se hodí pouze pro jednoduché funkce. Neměly by obsahovat složitější nebo delší kód, cykly ... (v tomto případě může překladač neakceptovat inline a vytvořit standardní funkci).

Pozn.: Takto definovaná funkce fyzicky neexistuje – nelze na ni skočit, nemá konkrétní adresu.

Pozn.: dají se na rozdíl od maker lépe ladit. Pokud je zapnut mód ladění, potom jsou překládány jako funkce.

Pozn.: je už i v nové normě C99

Pozn.: lépe využívat inline než makra

Příklad 3.1.11 inline funkce

napište inline funkci max pro zjištění a navrácení maxima ze dvou hodnot. Srovnajte s přístupem C (makro s parametry).

```
inline double max (double a, double b)  
{  
    return a > b ? a, b;  
}  
  
{  
    int i,c = 5,d=7;
```

```
i = max (3.15,c+d)*5; // volání
```

KO: jaké jsou výhody a nevýhody inline funkcí? Kdy se inline funkce používají?

Příklad 3.1.11a: Napište libovolnou jednoduchou funkci. Zjistěte její rychlost a změňte ji na inline. Jaká bude rychlost po této úpravě? (V případě, že je u překladače zapnuté ladění, mohou být inline funkce překládány jako opravdové funkce. Teprve v modu bez ladění se tedy může projevit vliv předpony inline).

Pro krátké a jednoduché funkce, kdy je režie na jejich volání větší než samotná činnost, je možné v C++ používat klíčové slovo inline. Takto označené funkce nejsou volány ale jsou pouze rozvinuty do kódu.

3.2.12 Prototypy funkcí.

Cílem je upozornit na odlišnosti v prototypech funkcí v C a v C++. Rozdíl spočívá v tom, že C++ striktně vyžaduje uvedení prototypu funkce před jejím prvním použitím. To umožní předejít chybám vzniklým při implicitní tvorbě prototypů funkcí.

V jazyce C, není v definici nutné uvádět návratovou hodnotu nebo seznam parametrů. Dále je možné provést volání funkce bez uvedení jejího prototypu (definice či deklarace).

v C++ musí být prototyp funkce přesně uveden (specifikovat všechny parametry i návratovou hodnotu) a musí být znám před jejím použitím - voláním. Proto se musí se natahovat standardní hlavičky (i ostatní).

Např.: **void fce()**

- je v c (neurčená) funkce. V C je rozdíl mezi funkcí bez parametrů a funkcí s parametrem void. Pokud není prototyp, pak se má za to, že existuje a vrací int (a není informace o parametrech). Parametry jsou brány jako libovolné. První (bez uvedení typu) je pro funkci s nespecifikovaným počtem parametrů, druhý (s void) pro funkci bez parametrů. Prototyp nemusí být deklarován před prvním použitím. Při neuvedení se v C jedná o implicitní definici funkce
- v C++ je to funkce bez parametrů. V C++ je při vynechání parametrů počítáno s tím, že chybí void. Funkci je nutné deklarovat před prvním voláním.

Pozn.: pokud se v C zapomene typ, má se za to, že je to int, či jiná explicitní konverze. V C++ to není možné.

Pozn.: správná programátorská technika je uvádět vždy návratovou hodnotu a vyplnit seznam parametrů. Lépe je tedy vždy uvádět návratový typ i typ v seznamu parametrů.

KO: jak se liší přístup k nedeklarovaným funkcím v C a v C++?

Pokud není u prototypu funkce uveden typ návratové hodnoty nebo parametru, má se v jazyce C++ za to, že je to void. Prototyp funkce musí být v C++ uveden před jejím použitím.

3.2.13 Funkce bez parametrů.

Cílem je upozornit na odlišnosti mezi deklarací funkce bez parametrů v C a v C++. Funkce bez parametrů musí být v C++ definována pomocí výpustky – "...", kterou již nelze vynechat jako v C (přísnější typová kontrola při předávání parametrů).

```
int fce (int a, int b, ...);
```

Funkce s proměnným počtem parametrů nebo proměnným typem parametrů – výpustka (z kategorie oddělovačů, interpunkce, znamének, " ...") se v C++ chovají jinak a vyžadují definici "...". (U C být nemusí, může být volné místo)

U parametrů volaných na místě "..." nedochází ke kontrole typů.

KO: Zjistěte si jak je výpustka "..." definována v překladačích jazyka C a jak v případě C++. (Najděte ji ve standardní knihovně).

3.2.14 Logické proměnné (bool, true, false).

Cílem je prezentovat nový typ reprezentující logické proměnné a hodnoty, kterých mohou nabývat. C++ zavádí nový typ pro logické proměnné a příslušné konstanty, které reprezentují hodnoty, kterých tento typ může nabývat. Nová klíčová slova tedy označují typ (bool) a jeho hodnoty (true, false). Nový typ řeší logické proměnné, které dříve nabývaly dvou hodnot – nula a nenula, které reprezentovaly false a true. Dále bylo možné se setkat s tím, že logické výrazy nabývají dvou hodnot – nula a jedna. Po zavedení typu bool nabývají logické výrazy hodnot – false a true. Nový a starý způsob jsou na sebe převoditelné (zpětná kompatibilita) pomocí konverzí.

Test: Jakých hodnot nabývají logické proměnné a výrazy (ve starším) C?

Jazyk C++ zavádí nový typ bool pro reprezentaci logických proměnných. Dále zavádí nová klíčová slova false a true pro hodnoty, kterých tento typ může nabývat.

V překladačích je přítomna implicitní konverze na int (je u bool : 1 a 0.) "Staré" hodnoty nula-nenula či nula-jedna je možné na místě logických proměnných stále používat.

Příklad 3.1.14 typ bool

Vyzkoušejte definice typu bool a jeho základní použití. Nadefinujte proměnné typu int a bool. Zkuste jim přiřadit hodnoty logické a int. Co se stane?

```
bool a = true; // přiřazení do typu bool
int j = true;  // totéž pro int

a = z && b==c; // logický výraz do bool
j = z && b==c; // logický výraz do int (konverze)
a = 10;       // konverze int bool
a = j;
```

```
j = a;           // konverze bool int
j = false;
if (j ) b=5;     // použití v podmínce
if (a) b=7;
```

KO: jak se konvertují hodnoty true a false? Jak byly reprezentovány v C?

Příklad 3.1.14a: Vyzkoušejte si kód programu, kdy přiřadíte proměnné typu int, float proměnným typu bool a naopak. Při trasování sledujte co se děje. Zkuste přiřazovat i výrazy.

Jazyk C++ zavádí nový typ bool s hodnotami true a false. Podmíněné příkazy jsou schopny pracovat i se starými hodnotami. Je prováděna implicitní konverze na int (tam i zpět).

3.2.15 Přetížení operátorů.

Cílem je ukázat možnosti přetížení standardních operátorů, tj. vytvoření vlastní funkčnosti pro standardní operátory jazyka C. Aby se nově vytvářené typy (třídy a struktury = objekty) mohly v objektovém návrhu chovat podobně jako standardní typy (int, double ...) je nutné zpřístupnit možnost popsat jejich chování pomocí operátorů (např. +). Jelikož operátor je speciální typ funkce, je i pro operátory použitelná vlastnost přetěžování. Důsledkem je možnost vytvářet operátory, které provádějí operace mezi novými typy, nebo mezi novými a původními typy. Výhodou tedy je "lidštější" možnost zápisu používáme-li pro nové typy operátory. Omezením je, že musí být zachovány některé původní vlastnosti operátoru (např. priorita, počet parametrů ...).

Jazyk C++ zavádí nové klíčové slovo operátor, které slouží k (pře)definování vlastností operátoru. Znak přetěžovaného operátoru následuje za slovem operator – např. operator+. Pro definici se využívá tzv. funkční volání operátoru. V běžné práci se využívá zkráceného tvaru. Např. zkrácený tvar je a = b, zatímco funkční volání je operator=(a,b).

Kromě standardních operátorů jazyka C lze přetěžovat i nové operátory jazyka C++ jako je new a delete (memory management) a operátory vstupu a výstupu (streamy).

Hlavní použití je u objektů, kterým přetížení operátorů dodává novou funkčnost a možnost použití operátorů jako pro standardní typy.

```
struct dva{float a; float b} ; // definice vlastního typu

dva operator+(dva &aa, dva &bb) /* přetížení operátoru + pro
    vlastní typ umožní používat pro něj operátor +.
    Syntaxe: návratová hodnota, název funkce,
    parametry je stejná jako u funkcí.
    Parametry mohou být libovolné (např. (dva, char*)
    Pro "součet" struktury a dat ve formě řetězce. */
{ // realizace sčítání pro vlastní typ
    dva pom;
    pom.a = aa.a + bb.a;
    pom.b = aa.b + bb.b;
    return pom ;
}
```



```

{
dva c,d,e;
e = c + d;  /* zkrácené použití operátoru - volá operátor
             sčítání podle typu parametrů*/
e = operator+( c ,d);  // funkční použití operátoru

```

KO: k čemu slouží možnost přetížení operátorů? Jak se realizuje?

Příklad 3.1.15: Zkuste přetížit operátory new a delete. Naalokujte větší paměť a z ní při požadavcích (v cyklu) předávejte bloky. Při vyčerpání paměti vraťte NULL a ukončete – odalokujte všechny částečně naalokované bloky i celek. Je-li počet odalokování a alokování různý zahlaste chybu. (Nepředpokládá se vrácení jednotlivých bloků – volná paměť je vždy od poslední naalokované do konce a vždy dojde k celkovému uvolnění paměti).

Jazyk C++ umožňuje přetížení operátorů pro nově vytvářené typy. Vlastnosti operátorů (počet parametrů, priorita, asociativita) zůstávají zachovány. K definici (i pro případné použití) slouží zápis ve funkčním volání, kdy se místo názvu funkce napíše klíčové slovo operator následované znakem přetěžovaného operátoru.

3.2.16 Vstupy a výstupy v C++.

Cílem je prezentovat nové možnosti vstupů a výstupů v C++. To znamená, že vstup a výstup je nově možné provádět jednotným způsobem pro různé typy (díky přetěžování funkcí a operátorů). C++ zavádí typově orientovaný vstup a výstup, kdy se podle typu proměnné vybere příslušný typ V/V operátoru (díky přetížení je možné mít několik stejně pojmenovaných funkcí-operátorů, mezi nimiž se vybírá na základě shody typů parametrů). Jedná se stejně jako v jazyce C o knihovní funkce (není vlastností jazyka na úrovni klíčových slov). Vlastnosti knihoven pro V/V jsou popsány v normě. Výhodou je jednotný přístup pro práci s různými typy. Výběr operátoru překladačem odstraňuje chyby způsobené nesouhlasem mezi proměnnou a oznamovaným typem (v řídicím řetězci jazyka C).

Test: jak je realizován vstup a výstup v jazyce C?

I když je tato kapitola zařazena do neobjektových vlastností, jedná se vlastně o objektovou vlastnost, protože vstupy a výstupy jsou řešeny pomocí objektů, které převádějí dodávané parametry do/ze specifikovaného zařízení (soubor, klávesnice, paměť ...). K základnímu použití pro vstup a výstup však není nutná znalost objektů, ale vlastnosti lze používat mechanicky.

Vstup a výstup v C nemá žádné klíčové slovo a je realizován pomocí knihovních funkcí. Protože neexistuje ani možnost přetěžování funkcí, které by umožnilo mít stejné funkce pro různé parametry, univerzální řešení by bylo složité. Pro vstup a výstup se tedy používá funkce s proměnným počtem parametrů, ve které jeden z parametrů udává seznam proměnných (počet a typ parametru, typ konverze nebo zobrazení).

C++ zavádí vstup a výstup, který má stejný zápis pro všechny typy – ale opět se jedná o knihovní funkce – z knihoven xxxstream.h (xxx může být nic, i,o,io, of,if,iof, ... např. práce

s diskem `fstream`, `ifstream`, `ofstream` (f je formátovaný)). Operátory vstupu a výstupu jsou značeny pro všechny typy stejně a správný operátor je vybrán podle typu parametru, se kterým pracuje. Označení `stream` je zobecněním pro vstupní či výstupní zařízení. Stream je možné “napojit” na soubor, standardní I/O zařízení, paměťový prostor ... pro vstupně výstupní operace je možno používat streamu, které jsou definovány pomocí (standardních) streamů `cin`, `cout`, `cerr`, nebo streamů vytvořených uživatelem a spojených s V/V médii

Z hlediska obtížnosti je možné rozlišit práci se streamy následovně:

- Pro vstup a výstup základních datových typů jazyka jsou již přetíženy operátory `<<` a `>>`. Jejich použití je jednoduché – je nutné naincludovat příslušnou knihovnu, která definuje vstupní a výstupní stream (`proud`, `tok` - což je zobecnění pro přístup k datům) pro standardní zařízení (klávesnice, monitor). Poté je možné pomocí operátorů “posílat” hodnoty mezi příslušnými zařízeními a proměnnými.
- Standardní vstup a výstup je prováděn v předdefinovaných obecných formátech. Pokud požadujeme formátovaný vstup či výstup v daném tvaru (šířka pole, typ výpisu, hex, oct, dec ...) použijeme modifikátory vstupu a výstupu (`hex`, `oct`, `endl`, `get`, `getl` ...) tak, že je zařadíme jako parametry do streamu, který se podle nich zařídí. Tyto vlastnosti mají i funkční obdoby. Pro tyto vlastnosti je znalost objektů výhodou.
- Vstup a výstup pro soubory - je prováděn obdobnými mechanizmy jako jsou obdobné funkce vstupu a výstupu na standardní zařízení. Rozdíl je pouze v tom, že je nutné vytvořit objekt typu stream spojený se souborem, prostřednictvím kterého dojde k výměně dat. Je možné se naučit mechanicky, znalost objektů je výhodou.
- Přetížení operátoru vstupu a výstupu pro třídy – jelikož je třída vlastně novým typem, který má možnost se chovat jako typy standardní, je možné pro ni přetížít příslušné operátory vstupu a výstupu tak, aby se při použití vstupních a výstupních operátorů chovaly standardně. Pokud operátory nepřetížíme, není možné je pro nově definované typy použít. (3.3.18)
- obecné vlastnosti vstupu a výstupu spojené s objektovým programováním – jak již bylo řečeno jedná se u vstupů a výstupů o práci s objekty. Snaha o univerzální použití streamů pro různé zařízení, a další požadavky, vedly k tomu, že se v konečném tvaru jedná o složitý systém, založený na hierarchii tříd, které postupným děděním rozšiřují své vlastnosti, a specializují se na určitý typ či způsob práce s daty. Zde už je pro zvládnutí a pochopení principů činnosti nutná znalost tříd a dědění.

Pozn.: vstupy a výstupy probereme ve výše uvedeném pořadí, i když správně by se mělo postupovat od obecného konceptu hierarchie tříd, přes základní třídy a ukončit specializovanými třídami.

Vstup a výstup základních datových typů

Vstup a výstup základních datových typů se provádí pomocí knihovních funkcí. Knihovna se nejčastěji jmenuje “`iostream`”, “`iostream.h`”, “`iostream.hpp`” . V ní jsou standardně otevřené streamy:

`cin`, který je spojen se standardním vstupem, klávesnicí

`cout` , který je spojen se standardním výstupem, monitor

`cerr` , který je spojen se standardním výstupem chyb, monitor

`clog` – podobně jako `cerr` ale má vyrovnávací buffer.

(k tomu jsou i “široké” varianty `wcin`, `wcout`, `wcerr`, `wclog` – pro rozšířenou znakovou sadu)

K vlastnímu čtení resp. zápisu se používá přetížený operátor bitového posuvu `<<` resp. `>>`. Volání se provádí:

`vstupní_proud >> proměnná1 >> proměnná2 ...;`

kde vstupní proud je v našem případě `cin`, v dalším potom jakýkoli otevřený proud spojený s konkrétním vstupním zařízením. Znak `>>` jsou znakem přetíženého operátoru (`operator>>`), tj. funkce, která je volána. Díky mechanismu C++ a jeho použití v definici operátorů, je možné je zřetěžit a tím načíst či získat více proměnných naráz. Předchozí lze chápat jako

`(vstupní_proud.operator>>(proměnná1)).operator>>(proměnná2)`

kde návratovou hodnotou prvního operátoru je objekt streamu, který se ihned použije pro další proměnnou. Pokud jsou proměnné 1 a 2 různých typů, potom i volané funkce operátor `>>` jsou různé – volají se podle typu parametru, a pro každý typ je nutné mít jiný operátor, protože výstupní tvary jsou pro každý typ jiné. Parametr “proměnná” je předáván referencí, a tak je možné do něj přímo zapisovat.

`výstupní_proud<<proměnná1<<proměnná2<<...;`

výstupním proudem je `cout`, nebo `cerr`, podle toho k čemu chceme výstup použít. Opět je možné zřetěžit výstup několika proměnných.

příklad 3.1.16 vstupy a výstupy: načtěte dvě celočíselné proměnné a vytiskněte je spolu s jejich podílem.

```
#include <iostream>
using namespace std;

int main()
{
    int i,j;
    cout << " zadejte dvě celá čísla /n"; /* tisk pomocného
                                         textu, odřádkování není automatické */
    cin>> i>> j; //postupné načtení dvou celočíselných proměnných
    cout<< "\n" << i<< " / " << j<< " = " << double(i) / j << "\n"; /*
postupně se provede odřádkování, tisk zadané hodnoty i, lomítko,
tisk zadané hodnoty j, tisk rovná se, výsledná hodnota (v pohyblivé
čárce) a odřádkování. Pro každý výstup je vybrán podle typu
příslušný operátor výstupu - znak, int, řetězec, int, řetězec,
double, znak. */
    return 1;
}
```

Operátor se definuje následovně:

`stream& operator xx (stream & s, datový typ & prom)`

Díky typu výstupní hodnoty je možné ho zřetěžit. Místo `xx` se doplní `<<` nebo `>>`. Místo `stream` je možné uvést některou z hierarchie tříd definovaných pro streamy.

Pozn.: načítá se proměnný počet znaků. Skutečný načtený počet znaků (`get`, `getline`) lze zjistit pomocí `gcount`.

Pozn.: Dále (v základním nastavení) dochází při ukládání znaků (při načítání znaků, tj do proměnné char) do bufferu k vypouštění bílých znaků (mezera, znak nový řádek, čárka...) což může být v některých případech nežádoucí. Je možné odstranit použitím funkce get objektu cin, nebo pomocí přepínačů ws. Funkce get pro zapsání do proměnné cc typu char je volána cin.get(cc).

Pozn.: Protože operátory << a >> vypouští bílé znaky lze použít funkce get. Funkce get (getline) funguje jako obdoba getchar z jazyka C – načítá se do bufferu, a teprve po stisku Enter je načítání ukončeno a vráceno vlastnímu programu.

Pro přepínání vypouštění či nevypouštění bílých znaků je možné použít přepínače:

ws vynechá bílé znaky, přeskočí mezery.

noskipws – nepřeskakuje bílé znaky (pozor , ještě je flag pro úvodní ws)

skipws – přeskakuje bílé znaky na začátku

Jelikož většina operací se streamy se provádí za pomoci bufferů (jsou u tříd, které jsou výše v hierarchii, nebo jsou součástí HW) je užitečné používat manipulátor flush. Tato metoda zajistí vyčištění bufferu a jeho přepis na zařízení. (např. pokud není ihned vidět znak vytištěný na monitor, nebo neobjeví-li se zapsaná data v souboru na disku je nutné použít flush. Bývá součástí jiných metod takže není nutné ho při nich provádět explicitně). Např. je použit u manipulátoru endl, který ukončí řádek (odřádkuje) a volá flush.

endl skok na nový řádek = \n a flush

os << "as fűlsaf " << endl; // endl zajistí okamžité vytištění a odřádkování

Pozn.: další metody pro práci se streamy jsou v oddílu o souborech.

Pro práci s řetězci se používá ends

ends nový řetězec = \0 a flush

Modifikátory vstupu a výstupu (formátování)

Slouží k definování parametrů vstupu a výstupu, jako jsou šířka pole, přesnost tisku, formát či typ.

Pozn.: pro práci s formáty tisku se používají tzv. manipulátory což jsou funkce pracující s objektem typu stream (např. fff(stream)). Je možné je také použít cout << fff;. Některé překladače pro ně ovšem nepodporují zřetězení (nevrací objekt typu stream).

manipulátory - speciální znaky a metody pro streamy – při (pře)definování je nutné aby vracely stream& (a měly stream & jako parametr), objekty s předefinovaným operátorem >>,<<

Pozn.: dříve byly součástí třídy ios, nověji jsou ve třídě ios_base. Obě jsou základem ostatních tříd, takže jsou v nich obsaženy a mohou se tedy obecně používat.

Pozn.: stejně jako pro C je i pro C++ prvotní snaha o přesný tisk. Pokud by tedy modifikátory měly zapříčinit ztrátu přesnosti, nebudou použity a vytiskne se více než je požadováno.

Pozn.: některé z vlastností jsou zařazeny v poslední normě a tak na ně některé překladače ještě nestačily reagovat. Proto se můžete setkat s různými variacemi.

Je možné je použít pro nastavení nové (verze s parametrem), nebo pro získání aktuální hodnoty (verze bez parametru). Verze s parametrem potom může vrátet hodnotu před změnou, která je výhodná v případě, že se po použití má vrátit původní hodnota. Některé působí pouze na následující tisk a po něm se vrací předchozí nastavení.

Formát pro tisk je udržován pomocí nastavení bitů v objektu třídy `ios_base` (a tedy jsou to změny, které platí až do další explicitní změny). Ty lze nastavovat např. pomocí metody `setf` s jedním parametrem (vrací aktuální nastavení všech bitů, předává se bit, který je třeba nastavit). Tato metoda slouží pouze k nastavení bitu, proto pokud je více bitů spojeno do celku a má smysl aby byl vždy nastaven pouze jeden (např. tisk může být hexa nebo dekadický ale ne oba současně), potom musíme použít `setf` se dvěma parametry, kdy první říká, který bit se nastaví a druhý parametr označí skupinu (pole) bitů, ke které patří a ta se před nastavením vynuluje.

Nulování příslušných bitů, které jsou označeny v parametru je také možné pomocí metody `unsetf`.

manipulátory mohou být uloženy v souboru “`iomanip.h`”

Obecné vlastnosti

Šířka pole - `width` - vliv pouze na následující tisk, tisk do pole zprava. `Width(n)` z `ios_base` – minimální délka pole (default 0) (k tomu `<< setw(n)`), `width()` zjistí aktuální šířku

`setw(int)` nastaví šířku výpisu – pro jeden výstupní znak – minimální počet který vyjde ven

Znaky pro vyplnění - `fill` – standardně mezera, vliv stále. `Fill()` z `basic_ios` – znak pro doplňování (default mezera) (k tomu `<< setfill()`), `fill()` zjistí aktuální výplňový znak

`setfill(int)` nastaví výplňové znaky pro jeden výstup

Tisk koncových null se dá zajistit nastavením bitu `showpos` nebo `fixed` podle (stáří) překladače (tj. je to navázáno na tisk desetinné tečky).

Zobrazí vždy znaménka, tj. i “+” – `ios_base::showpos`. Také manipulátory `showpos` a `noshowpos`.

Tisk velkých písmen (hexa a E pro exponent) nastavit `ios_base::uppercase`. Též manipulátory `uppercase` a `nouppercase`.

Zarovnání vlevo bit `ios_base::left`. vpravo `ios_base::right` a kombinace znaménko zarovnat vlevo, číslo vpravo `ios_base::internal` patří do skupiny `ios_base::adjustfield`. Je také možné použít manipulátory `left`, `right`, `internal`.

Logické proměnné

Tisk `true` a `false` pro logické hodnoty - `ios_base::boolalpha`. Totéž lze pomocí manipulátoru `boolalpha` a `noboolalpha`.

celočíselné proměnné

Dekadicky formát tisku manipulátor – `dec`

Oktalově formát tisku manipulátor – `oct`

Šetnáctkově formát tisku manipulátor – hex, pro tisk 0x na začátku nastavit ios_base::showbase. Totéž pomocí manipulátorů showbase a noshowbase.

Totéž lze pomocí ios_base::dec, ios_base::hex, ios_base::oct, které patří do skupiny bitů označené ios_base::basefield

Např. pro nastavení hexadecimálního zápisu můžeme použít metodu setf, kdy nastavíme příslušný bit. Aby nezůstal nastaven bit např. pro dekadický výpis, označíme ve druhém parametru příslušnou skupinu bitů. Lépe by zde bylo použití manipulátoru hex, který provede totéž – tj. zruší starý nastavený bit a nastaví bit hex. fmtflags je označení typu (který je definován “uvnitř” třídy ios_base) a je určen pro držení příznakových bitů.

```
ios_base::fmtflags puvodni = cout.setf(ios::hex, ios_base::basefield)
```

setbase nastaví soustavu

Pohyblivá řádová čárka

Tisk desetinné tečky, se dá zajistit nastavením bitu ios_base::showpoint pomocí setf. Také možno pomocí manipulátorů showpoint a noshowpoint.

ios_base:: fixed pro zápis s desetinnou tečkou (čárkou) bez exponentu, ios_base::scientific pro zápis v exponenciálním tvaru – oba patří do skupiny bitů ios_base::floatfield. Též manipulátory fixed a scientific.

V knihovně iomanip jsou definovány manipulátory setw pro nastavení šířky pole, pro přesnost (set) precision – počet zobrazených čísel, nebo ve vědecké notaci počet desetinných čísel, vliv stále, pro výplňový znak je definován setfill. setprecision(int) přesnost – pro jeden výstupní znak. Precision(n) z ios_base – (<<setprecision) přesnost plovoucí řádové čárce – počet významných číslic (default 6), nebo místa za desetinnou tečkou, precision() pro zjištění aktuálního nastavení

A další...

Soubory a vstupy a výstupy

Pro práci se soubory se používají knihovny fstream, kde jsou třídy (typy pro objekty zajišťující vstup či výstup pro soubory) ofstream, ifstream, iostream.

Při vzniku objektu je možné mu předat několik parametrů. Prvním parametrem je název otevíraného souboru (jedná se vlastně o konstrukci objektu třídy ofstream, ifstream, za pomoci konstruktoru s jedním parametrem, kterým je název souboru (řetězec)).

```
ofstream os("nazevsouboru.txt")
```

```
ifstream is("nazevsouboru.txt")
```

objekty os,is se dále používají (podobně jako cin, cout) s operátory vstupu a výstupu

```
os << "takhle to vystupuje ";
```

```
is >> NacitanaPromenna;
```

Otevření se provádí pomocí konstruktoru, nebo funkcí metodou. Zavření se realizuje metodou close nebo destruktoem (většinou zánik objektu na konci bloku ve kterém byl objekt definován). Kontrola správného otevření je možná např. pomocí metody is_open

```

os.close( );
os.open("jiny soubor.txt");
if (!os.is_open()) ...

```

Při otevírání souboru je možné použít i druhý parametr, který upravuje způsob otevření souboru (obdobně jako mod při otevírání souborů v C). K vytvoření tohoto parametru se využívají hodnoty typu enum báze třídy ios ("přepínače"). Z tohoto důvodu je nejprve nutné "vstoupit" do třídy ios a potom vybrat daný parametr. Parametry se mohou sdružovat. Je možné použít následující:

ios::in	Pro otevření souboru pro čtení
ios::out	Pro otevření souboru pro zápis
ios::ate	Pro nastavení na konec souboru (po otevření)
ios::app	Pro otevření (automaticky ios::out) a zápis na konec souboru
ios::binary	Práce v binárním modu
ios::trunc	Při otevření vymaže stávající obsah souboru
ios::nocreate	Otevře pouze existující soubor (nevytváří ho)
ios::noreplace	Otevře pouze soubor, který vytváří. Neotevře existující soubor

```

ofstream os("soub.dat",ios::out || ios::ate || ios::binary || ios::nocreate);
istream is("soub.txt", ios::in || ios::nocreate);
fstream iostr("soub.txt", ios::in || ios::out);

```

Po vytvoření (zvláště s nocreate a noreplace) je nutné vědět zda nedošlo k chybě při otevírání. To je možné zjistit pomocí bitů chyb, pomocí funkcí chyb či rdstate, nebo přímo z názvu streamu.

Pro stav streamu, jak zjistit zda nedošlo k chybě je možné použít zápis

```

if (!os) {řešení chyby...}

```

Použití názvu je možné proto, že objekt má pro tento tvar použití předefinovanou metodu, jejíž návratovou hodnotu lze převést na logickou proměnnou (buď přímo logickou hodnotu, nebo ukazatel) a nula se zde objeví při chybě. Tímto způsobem je např. také možné zjistit, zda došlo k načtení správné hodnoty z klávesnice. Pokud požadujeme číslo typu int a uživatel zadá pouze znaky nepatřící do čísla, potom nedojde ke správnému načtení a stream je v chybovém stavu. Chybu načítání je možné ošetřit pomocí podmínky: `if (cin >> x) {...}`. Pokud v rámci ošetření chyby nevynulujeme chybové bity, zůstává stream v chybovém stavu.

If (cin) – je možné díky konverzi istream::operator void*() tj. přetypování na ukazatel. Často se používá ke kontrole, zda nedošlo k chybě (vrací NULL). Výhoda void* je v tom, že povoluje minimum operací a tedy z jeho použití plyne minimum chyb (a dá se srovnat – používá se ke zjištění stavu proudu, tj. void * se často používá k testování stavu objektu.

Nejčastěji je cin roven !fail().

Druhým způsobem je možnost získat stav pomocí funkcí. K tomu slouží obecná funkce pro získání stavu `rdstate`, nebo konkrétní metody pro zjištění stavu – `fail`, `bad`, `good`. Je také možné použít metodu `is_open`.

```
if (os.is_open) {je otevřen}
```

```
if (is.fail( )) {došlo k chybě}
```

Posledním způsobem je přímý přístup k nastaveným bitům pro stav (flags), které jsou opět součástí `ios`

goodbit	V pořádku
badbit	Vážná chyba
failbit	Méně závažná chyba
eofbit	Dosažení konce souboru

Stav těchto bitů lze zjistit pomocí metod `good()`, `bad()`, `fail()`, `eof()` pro jednotlivé bity, nebo pomocí `rdstate()` pro všechny.

```
if (is.rdstate( ) & (ios::badbit||ios::failbit)) ...
```

Je-li možné chybu opravit, potom se dá pomocí metody `clear` daný bit přenastavit (bit který se má nastavit je dodán jako parametr). `Clear()` chybových bitů. Mazání bitů. Umožňuje např. po dosažení konce souboru (načteného z klávesnice) opět načítat (jinak se už nenačítá, protože za koncem nic není.)

```
is.clear(ios::goodbit);
```

Pozn.: S chybovými stavy souvisí i možnost vyhození výjimek (typu `basic_ios::failure`). Výjimku hodila chyba (nastaven příslušný bit), kterou je možné získat pomocí `exceptions()`. Metoda `clear` může hodit výjimku při nastavení daného bitu, který je možné nastavit pomocí `exceptions(iostate ist)`. (Pozor: `setstate` volá funkci `clear` a tím může vyvolat výjimku)

Při čtení ze souboru je nutné vědět zda jsme nedosáhli konce souboru. To je možné zjistit např. pomocí funkce `eof`, která vrací logickou hodnotu značící dosažení konce souboru. Je nutné si uvědomit, že stejně jako v jazyce C i zde se výsledek týká posledního čtení a tedy konec souboru zaregistrujeme až při načtení prvního znaku za koncem souboru. Při dosažení konce souboru je nastaven příznak konce souboru `ios::eof`.

```
if (is.eof( )) return; // odchod po dosažení konce souboru
```

Pro práci se soubory v binárním režimu je možné použít práci s typem `char`, či pomocí funkcí `put` a `get` (nezapomenout ošetřit vypouštění bílých znaků). Lépe je ovšem pracovat s funkcemi `write` a `read`, které mají jako parametr adresu a délku pole se kterým se pracuje (v bytech).

```
is.write(buffer,30);
```

```
os.read(buffer,23);
```

Při práci se soubory, je někdy nutné se v nich pohybovat (neasynchronně). Pro pohyb se používá metoda `seekg` (pro vstup) a `seekp` (pro výstup). Tyto metody slouží k nastavení nové aktuální pozice. Ve tvaru s jedním parametrem určuje parametr pozici v souboru kam se nastaví

ukazatel pro další práci. Ve tvaru se dvěma parametry, je prvním parametrem offset a druhý parametr určuje místo, od kterého se offset uplatní. Místa jsou: počátek (a pro něj symbolická konstanta) `ios::beg`, aktuální pozice – `ios::cur`, a konec souboru – `ios::end`. Pro poslední dvě může být offset i záporné číslo.

```
os.seekp(23); is.seekg(-10, ios::cur);
```

Potřebujeme-li znát pozici v souboru, na které se nacházíme, je možné použít metodu `teelp` pro výstup a `teelg` pro vstup.

Pro přesun slouží metoda `ignore`. Zde je možné uvést počet znaků, které se mají přeskočit, zároveň je možné uvést druhý parametr, který označí znak, při kterého načtení se má ignorování ukončit. Pohyb se také vždy ukončí s koncem souboru.

Pro načítání řádků je možné použít metodu `getline`, která má jako parametr adresu na kterou se čte řádek a maximální počet načítaných znaků. Dalším parametrem může být ukončovací znak po jehož načtení načítání skončí. (někdy bývá problém s tím, že vlastní konec řádku, či stisk enteru u `cin`, není touto metodou načten a musí se vyčíst přes `char`).

`getline (kam,kolik)`, `get(kam,koli)` – jako `getline` ale nechá znak konce řádku nenačten. `get` – umí číst i konce řádků – čte jeden znak (ale vždy). `cin.get(name,30).get ()` – přečte vše díky zřetězení

Načítání řetězce končí na bílém znaku – celý řádek čte `getline`

Při nechtěném načtení znaku, je možné ho vrátit zpět pomocí metody `putback`. Jelikož toto může být i vlastností HW nemusí se vždy podařit. Proto je lepší používat `peek`.

Pokud potřebuje zkontrolovat jaký bude následující znak bez jeho vyčtení ze streamu (např. podle načteného znaku rozhodneme, kterou metodu volat, ale znak by měl být zároveň prvním načteným v této metodě) potom můžeme použít metodu `peek`.

Při konci souboru se nastaví dva bity `eofbit` a `failbit` na 1. `cin.eof()`., - hodnotí poslední akci obdobně jako C. následující volání po konci souboru nemají smysl.

Pozn.: při práci s čísly v pohyblivé řádové čárce může být uplatněny národní specifikace prostředí – potom se např. místo desetinné tečky, tiskne (či vyžaduje) desetinná čárka. Dokonce se může stát, že metody (funkce), které jsou součástí C a odvozené používají standardní notaci, zatímco metody C++ lokalizovanou (a to i v rámci jednoho objektu – takže výstup jedné metody nelze použít jako vstup pro druhou – pro bližší zkuste knihovnu `locale`).

Pozn.: samozřejmě lze použít modifikátory uvedené v předchozích kapitolách (pro `cin` a `cout`).

Vstupy a výstupy pro uživatelské třídy

V /V operace jsou definovány pro základní typy a je možno je napsat (přetížít operátory) i pro jakýkoli vlastní typ. Detailněji jsou popsány v kapitole 3.3.18

Obecné vlastnosti hierarchie vstupů a výstupů

Vstupy a výstupy jsou tvořeny hierarchií tříd. Bázové třídy definují konstanty a základní vlastnosti. Odvozené třídy se postupně specializují (přidává se buffer, specializace na vstup, výstup, konkrétní zařízení ...)

Základní třída ios - pro obecné definice I/O, nově přidána (částečně přerozdělení původního ios) třída ios_base

Třída pro manipulaci s vyrovnávacím bufferem (streambuf) (lepší manipulace s bloky dat přes vyrovnávací paměti – např. přizpůsobení se způsobu dat na disku)

Existují varianty pro vstup, pro výstup a společné (kombinované)

Většina streamů si vytváří buffer pro práci s daty. Je možné ho přidat ke třídě, nebo v rámci konstruktoru zvolit vlastní buffer – např. s konkrétní velikostí. K tomu nejčastěji slouží definice proměnné streambuf(char *buf, int delka).

z této třídy dědí ostatní buffery (pro soubor - filebuf, pro paměť - strstreambuf, pro konzolu – conbuf).

třída ios

slouží jako obecný základ pro implementaci I/O operací – bázová třída nezávislá na typu. Definice pro formátování a chyby.

obsahuje ukazatel na buffer (získaný pomocí streambuf), v této úrovni ještě nevyužitý - nedefinovaný

definují se zde (enum) hodnoty pro typ modu otevření a příznaky chyb
je v ios.h

enum pro nastavení vlastností streamu

```
ios :: enum {  
in                interface vstupu  
out               interface výstupu  
ate              at end - na konci souboru  
app              append na konec  
trunc            otevření s nulovou délkou  
nocreate, noreplace, dá chybu když neexistuje, když existuje  
binary           otevření pro binární operace  
}  
  
ios::enum io_state { goodbit, badbit, eofbit, failbit, }  
pro zjištění stavu
```

eofbit – konec souboru (nebo chyba zařízení)

badbit – chyba vyrovnávací paměti proudu

failbit – chyba ve formátování

good() zda může být další operace úspěšná, po úspěšné operaci (tj. další operace může být úspěšná)

eof() zda je na konci, po úspěšné operaci

fail() upozornění na chybu v datech (či data v pořádku nic se neztratilo ale dál nemůžem), nevrací failbit!, může být i eof

bad() nikdo neví, co se děje, chyba vyrovnávací paměti

rdsate() vrací všechny stavové bity

formátovací flagy `x_flags` určují řídicí informace pro formátování a lokalizaci proudících dat. Je možné s nimi manipulovat pomocí `setf()` `unsetf()`.

přetížen **operátor !** – vrací 1 při chybě ve streamu

formátování

`x_flags` enum {skipws, left, right, internal, dec, oct, hex, showbase, showpoint, uppercase, showpos, scientific, fixed, unitbuf, stdio} nastavitelné pomocí `flags(newflags)`

`x_precision`, `x_width`, `x_fill` – nastavitelné pomocí `width`

třídy `istream`, `ostream`

doplňuje společné operace o operace výstupu 'o-' a vstupu 'i-'

stále je nutné "ručně" vytvořit buffer

objevují se zde poprvé přetížené operátory `<<` a `>>` s prvním parametrem `&stream` a druhým parametrem reference na `typ - typ&`

vznikají konstruktorem

základním prvkem proudu je `char`

jsou zde metody pro čtení `get()`, `getline()` a zápis `put()` (`put` může mít až tři prototypy – pro `char`, `unsigned char` a `signed char`, z čehož plyne, že při tisku znaku pomocí jeho reprezentace v `int` proměnné dochází k nejednoznačnosti volání a k chybě překladu)

pro binární čtení `read()` a `write()`

`ignore()` pro přeskočení znaků

`peek()` pro otestování znaku (nevyčte ho ze streamu)

pro nastavení polohy ve streamu se používá `seek()` (někdy `seekp`, `seekg` (`put` a `get` zvlášť)), nastaví se buďto pozice, nebo se určí posuv a odkud

`tell()` pro zjištění aktuální pozice ve streamu (někdy `tellp`, `tellg` (`put` a `get`))

pro vyprázdnění bufferu a vynulování streamu se používá `flush()`

eatwhite() pro přeskočení mezer
writes pro výpis řetězce
je v iostream.h

třída iostream spojuje obě předchozí

třídy ifstream, ofstream

třídy pro práci s diskovými soubory
buffer vytváří sám automaticky
je v fstream.h
textově i binárně
mode otevření viz. ios – příznaky lze združovat (přes |)
verze s i a o jsou specializovány na vstup a výstup (mají své .h soubory)

třídy istrstream, ostrstream, strstream

třídy pro práci s řetězcí
paměť pro řetězce může být parametrem konstruktoru
je to v strstream.h

třídy xxx_withassign

usnadňují přesměrovávání streamů, nejčastěji pro cin, cout, cerr (pomocí operátoru > systému), standardní třída (istream, ostream) není schopná přesměrování, proto se přidává ještě funkčnost pro přesměrování (tj. pokud by cout byl objekt základní třídy, potom by uměl pouze výstup na obrazovku a nebyl by schopen reagovat na požadavek systému přesměrovat výstup do souboru – tu umí, protože je objektem ostream_withassign, který o tuto vlastnost rozšiřuje (dědí z) ostream).

třída constream

třída pro práci s obrazovkou
metody clrscr() pro mazání obrazovky, window pro nastavení aktuálního výřezu obrazovky ...

Nové operátory vstupu a výstupu zjednodušují práci a umožňují unifikaci kódu.

KO: jaký je rozdíl mezi vstupem a výstupem v C a v C++?

Příklad 3.1.16a.: Napište funkce pro tisk (uložení) a načtení komplexního čísla do dané struktury. Funkce první, která uloží (do souboru, na obrazovku) komplexní číslo jako (12.2, 12.22). Druhá funkce ve stejném formátu číslo načte z klávesnice nebo ze souboru. Použijte operátory << a >>.

3.2.17 Znakové konstanty, (dlouhé) literály.

Cílem je upozornit na nový celočíselný (znakový) typ schopný pojmout hodnoty reprezentující znaky určené pro regionální odlišnosti. Jelikož přestaly stačit znakové sady do 256 znaků, bylo třeba vytvořit nový typ, který by byl vhodný k držení znaků. Zároveň je nutné ho odlišit od ostatních celočíselných typů z důvodu vstupů a výstupů. Jazyk C++ zavádí nový celočíselný typ pro uložení znaků širších než 8bitů. Výhodou je možnost použití rozsáhlejších znakových sad a práce s nimi. Nevýhodou je zatím nejednotná (špatná) implementace v překladačích. Původní znaková sada (char) zůstává zachována.

V jazyce C se pro práci se znakovými proměnnými používá nejmenší celočíselný typ – char. Pro práci s většími sadami znaků (UNICODE), pro které typ char nemusí vyhovovat, je v C++ zaveden nový typ w_char. Pomocí tohoto typu jsou implementovány “dlouhé” znakové konstanty využívající lépe HW vlastnosti paměti.

Znaková konstanta v C++ již také není konvertována na int ale char je “opravdový” typ – z důvodu rozlišení při přetěžování funkcí. Pro vstup a výstup (cin, cout) však není možné rozlišit char a w_char a tak je nutné použít streamy wcin a wout.

wchar_t je wide character type – drží znak z rozšířené znakové sady. V C je to přes define a macro v headru, u C++ je to klíčové slovo typu. K tomu zadání konstanty

Zápis pro konstantu tohoto typu - wchar_t b = L'a'; nebo řetězec L"abcd";

Pro “klasickou” práci pro tento typ platí stejná pravidla jako pro celočíselné typy.

Pozn.: Znakové literály jsou char v C++, v c jsou int. sizeof('a') je sizeof(int) v C, ale sizeof(char) v C++. Proto je možné např. při vstupu či výstupu načíst char.

Pozn.: to umožní rozlišit int a char při přetěžování funkcí. V C se provádí implicitní konverze při práci se stackem.

Pozn.: Víceznakové literály v C++ mohou být int

KO: proč se používá typ w_char a jaký je jeho vztah k typu char?

Příklad 3.1.18a: nadefinujte řetězec typu char a w_char a oba vytiskněte do cout.

Jelikož standardní typ char nemusí být vždy dostatečně veliký pro znaky rozšířené znakové sady (obsahující národní znaky), byl dodán nový typ w_char, který slouží k uložení textů z rozšířené znakové sady. Pro jeho standardní vstup a výstup je nutné používat streamy wcin a wout. Znakové a řetězcové konstanty se od konstant typu char odlišují předřazením znaku 'L'. Některé překladače používají typedef wchar_t, který je reprezentován delším celočíselným typem.

Pozn.: tento typ je brán jako problematický (dokonce pro umístění UNICODE znaků). Některé překladače ho nepodporují, popř. se k němu chovají rozdílně). Borland

má `wchar_t` jako klíčové slovo a `int`. Microsoft visual jako `typedef` a minimálně 16bitů `unsigned int`. Na MSDN se o `w_char` jako typu mluví jako o parametru funkcí pro rozšířenou sadu ale nikde není blíže specifikován.

3.2.18 Typ ((un)signed) long long.

Cílem je upozornit na nový celočíselný typ, s definovanou minimální přesností. Se zvyšující se náročností výpočtů a přesností výpočtů v procesorech byl zaveden nový (přesnější) celočíselný typ. Stejně jako ostatní typy v C nemá pevně stanovenou přesnost, má však stanovenou přesnost minimální. Zatím nebyl převzat do C++.

Test: které celočíselné typy znáte a jakou mají přesnost?

Nová celočíselná proměnná v pevné čárce s větší (a definovanou minimální) přesností. Musí mít přinejmenším 64 bitů délku.

```
long long int lli=-12345675LL; /* definice a inicializace
                                konstantou */
unsigned long long int ulli=1234567890LLU;
printf("%lld : ",lli);          /* označení typu proměnné ve
                                formátovacím řetězci */
```

Příklad 3.1.19a: zjistěte jakou přesnost má typ `long long` ve vašem prostředí.

Pozn.: v případě, že potřebujeme pracovat s typy s přesnou přesností (která se nemění podle použité platformy), potom je nutné použít typy - `__intn` ($n=8,16,32,64 \dots$ a značí počet bitů daného typu). Tyto typy nejsou dány normou, ale jsou definovány v .h souborech (u některých překladáčů).

Jazyk C zavádí nový celočíselný typ, u kterého je dána minimální přesnost 64bitů.

3.2.19 Prostor jmen – namespace.

Cílem je prezentovat možnost oddělení názvů proměnných a tříd do společných prostorů a tím zmenšit pravděpodobnost kolize jmen. Z důvodů kolizí jmen proměnných a funkcí při psaní rozsáhlejších projektů a z důvodu vyhledávání správných funkcí a proměnných se zavádí prostory jmen. Příslušná část kódu se “ohraničí” názvem prostoru. Proměnné ze stejného prostoru se “vidí” a mohou spolupracovat bez omezení. Proměnné z jiného prostoru je pro použití nutné “zveřejnit” (neboli “zpřístupnit”, “zviditelnit”). Toto zpřístupnění je možné pro jednotlivé (vybrané) proměnné nebo pro celý prostor (všechny proměnné v něm obsažené). Výhodou je oddělení názvů proměnných.

Prostory jmen jsou jednotky, které oddělují jména identifikátorů (dávají jim vlastně příjmení). Prostorem jmen je např. i objekt, který sdružuje metody a proměnné.

Kolize mezi stejnými názvy globálních proměnných nebo funkcí je možné ošetřit pomocí prostorů jmen. Funkce či proměnná potom dostává “příjmení” daného prostoru a nedochází tedy mezi nimi ke kolizi při překladu. Je-li při volání uvedena funkce bez “příjmení”,

je volána funkce se stejným “příjmením” jako má aktuální prostor, tj. funkce v daném prostoru jmen. U volání do jiného prostoru je nutné uvést příjmení pomocí operátoru příslušnosti.

```
prostor::identifikátor          // celý název identifikátoru
prostor::podprostor::identifikátor
```

K manipulaci s proměnnými v prostorech slouží především klíčová slova **namespace**, a **using**. Pomocí klíčového slova **namespace** se vyhrazuje prostor s daným názvem, tj. proměnné a funkce umístěné v bloku takto ohraničeném patří do tohoto prostoru a tím jsou odděleny od ostatních funkcí a proměnných. Pomocí klíčového slova **using** je možné zpřístupnit proměnné skryté v prostoru ohraničeném **namespace** tak, aby je bylo možno využívat bez uvádění sekvence “prostor::”. I když lze zpřístupnit celý prostor, doporučuje se zveřejnit pouze vybrané proměnné a funkce.

Např. plné jméno pro přístup k standardnímu výstupu je **std::cout**. Použijeme-li **using namespace jmeno_prostoru** (zde nejčastěji **using namespace std**), nemusíme při použití **prostor** uvádět – tedy můžeme rovnou používat **cout**. Kvůli jedné funkci jsme ovšem zveřejnili vše, lépe je zpřístupnit jen to co je potřeba tj. použít pouze povolení konkrétní funkce - **using std::cout**.

Using – deklarace zpřístupní identifikátor, direktiva **prostor**.

Using je platné pro blok – tj. “natažení” končí s koncem bloku { **using ...** } a tady už ne.

```
namespace X {
definice proměnných
definice funkcí např. void f2(int i);
definice objektů
}
```

Vlastní těla funkcí můžeme definovat vně aby byl přehled

potom ke jménu funkce musíme přidat s operátorem přístupu “příjmení” a to **prostor** ze kterého fce je

```
void X::f2(int i){}
```

funkce je potom vlastně **X::f2**, čímž lze odlišit od **Y::f2** (tak se dají i volat)

Zpřístupnění se provádí také pomocí **using**. Např.

```
using X::f2; ... ; f2(10);...
```

a nebo úplně **using namespace XU** přetížených funkcí lze zápisem: **using ns::fce** zpřístupnit všechny funkce se stejným jménem bez uvedení kompletních prototypů.

Pozn.: Jmenné prostory je možné vnořovat **namespace A{ namespace B { } }**. A přístup **AA::BB::c** – tj. pomocí jménoprostrou::proměnná.

Vytvoření alias – zvláště pro několikanásobně vnořené prostory je možné také vytvořit jednodušší, nové jméno (přezdívka) se kterým se potom pracuje: **namespace vloženy = původní::vnoreny::nejvnorenejší::uplnevnoreny**

V oblastech ohraničených namespace by se nemělo používat označení proměnných static. Místo static je vhodnější použít namespace { } (tj. bezejména, které se chová jako by bylo následované using takže proměnná zde uvedená je v bloku dosažitelná, ale protože nemá jméno, nelze jinde zpřístupnit v jiných blocích – nelze tedy použít proměnnou tohoto prostoru v jiném souboru než je deklarována

```
namespace {  
    /* tyto proměnné a funkce budou dostupné pouze z tohoto modulu  
    */  
}
```

Pozn.: Pravidla pro prohledávání: např. funkce se hledá pouze jako jméno, které je uvedeno, hledá se od prostoru jmen ve kterém je uvedeno a jde se ven ke globálnímu prostoru, existuje-li funkce daného jména, rozlišuje se podle parametrů, pokud se nenajde odpovídající fce, je to chyba

Pozn.: odpovídající funkce se hledá podle shody typů parametrů bez konverzí, (pole je ukazatel, funkce je ukazatel na funkci, proměnná a const proměnná jsou totéž), hledá se shoda po možné konverzi celočíselných na int a float na double, dále potom konverze int a double a třída odvozená na báze, uživatelské konverze, použití proměnného počtu parametrů.

Pozn.: Více shod na stejné úrovni konverzí vede k chybě. Pokud napíšeme metodu, která např. používá dva parametry a mohla by se tlouct s variantou konstruktor a jiná funkce, potom to řeší klíčové slovo explicit, zabráňující explicitnímu použití konstruktoru.

Pozn.: prostor jmen nelze definovat uvnitř třídy

Pozn.: definice se mohou opakovat (lze je načíst vícekrát – ve více hlavičkách)

Pozn.: jméno prostoru se nesmí shodovat se jménem funkcí, proměnné, třídy ...

Pozn.: standardní prostor jmen je nepojmenován tj. přístup k němu je ::identifikátor

Pozn.: jako prostorem jmen se chová i třída

Pozn.: pomocí using BazováTřída::g; lze zpřístupnit i prvek, který se při dědění "ztratil" (dědění private)

KO: k čemu se používají prostory jmen?

Příklad 3.1.20a: napište dvě funkce se stejným prototypem, každou v jiném prostoru jmen. Zkuste je obě zavolat z funkce main.

Pro zlepšení možností programování, zvláště pak možnosti použít v různých modulech stejných definicí jmen proměnných a funkcí je použit princip prostoru jmen. Část kódu se ohraničí a nazve jednotným názvem. Proměnné a funkce zde uvedené potom mají své jméno rozšířeno o jméno prostoru prostor::funkce() čímž se dosáhne jejich odlišení od stejně pojmenovaných funkcí ležících v jiných prostorech.

Proměnné z jiného prostoru jmen je možné používat buď s plným názvem, nebo je možné je "zveřejnit" pomocí using, kdy nadále není již nutné používat část prostor:: pro přístup k proměnné a používá se pouze její jméno.

Třída se chová jako prostor jmen.

3.2.20 Restrict.

Cílem je upozornit na nové klíčové slovo přítomné v C. Tento modifikátor proměnné je spojen s optimalizacemi prováděnými pro ukazatele. Tím, že říká, že data na která ukazatel ukazuje se nemohou změnit, umožňuje lepší optimalizaci kódu. Za správnost označení ukazatelů tímto modifikátorem (že se data opravdu nemění – např. v přerušení) ručí programátor.

Klíčové slovo `restrict` je spojeno s ukazateli a říká, že data, na která ukazatel ukazuje, nejsou přístupná jiným způsobem (přes jiný ukazatel) v tomtéž okamžiku. To umožňuje překladači provádět optimalizace, které předpokládají, že data se kterými se pracuje se nemění (pomocí jiné proměnné) – např. práci přes cache. Měl by být vytvořen rychlejší kód, než když se musí při přístupech kontrolovat změna proměnných.

`Restrict` je modifikátor typu proměnné (patří sem např. `const`, `volatile` ...). To že je použit správně (tj. opravdu nedochází k překrytí datových bloků určených ukazateli a délkou pole) je věcí programátora a není to kontrolováno.

Je v normě C99 ne v C++.

Pozn.: mohlo by se zdát, že tento problém řeší ukazatel na konstantní proměnnou. Ten však říká pouze, že data nemohou být měněna přes tento ukazatel. Např. pokud předáme tentýž ukazatel podruhé jako nekonstantní, potom přes něj měníme data v obou blocích.

Pozn.: Tedy je umožněno překladači provést “optimalizaci” typu: načíst hodnotu z adresy např. do registru, zde s ní “dlouhodobě” pracovat a na konci teprve uložit výsledek zpět na adresu.

Pozn.: v jednodušších a starších překladačích není (a asi nebude) přítomno.

Klíčové slovo `restrict` pomáhá optimalizovat práci s pamětí, protože signalizuje překladači, že paměť na kterou ukazatel ukazuje nebude v dané chvíli měněna na pozadí (jiným ukazatelem).

3.2.21 Anonymní unie.

Cílem je upozornit na možné použití unionu

Test: co je to union a jak se používá?

Anonymní – nepojmenovaný – union umožňuje používat několik proměnných, které jsou umístěny v jednom paměťovém místě (překrývají se). Není možné je použít v jeden okamžik ale podle potřeby lze pracovat s různými typy (a tím šetřit paměť, kterou by tyto (nejčastěji pomocné) proměnné zabíraly.

```
static union          // globální anonymní union musí být static
{
    int i;
    float f;

    union
```

```

{
    char c;
    unsigned char uc;
};

void function()
{
    i = 1;           // přístup k prvkům anonymní unie je přímý
    f = 3.14;
    c = 'c';
    uc = 'u';
};

```

3.2.22 Shrnutí neobjektových vlastností.

Neobjektové vlastnosti jazyka C++ se zavádí pro zvýšení programátorského komfortu a z důvodu fungování nových mechanismů objektového programování. Některé vlastnosti jsou zpětně přijímány do normy jazyka C (komentáře, const, ...). Jsou zde také uvedeny vlastnosti, kdy C předběhlo C++ (nejnovější norma C je mladší než C++ a C++ patrně v další normě bude o tyto vlastnosti rozšířeno).

Neobjektové vlastnosti umožňují snadnější práci a zároveň umožňují mechanismy práce s objekty. Jako nejvýznamnější je vstup a výstup pomocí streamů, přetížení operátorů a funkcí, alokace paměti.

3.3 Objektové vlastnosti C++ - základy práce s třídami

Cílem je rozšířit znalosti jazyka C++ o jeho objektové vlastnosti, které přinášejí možnosti sdružování dat a operací – metod s nimi pracujících. Dále je možné kontrolovat vznik a zánik objektů a přetěžovat operátory.

Objektové vlastnosti

- umožňují úplně nový způsob programování vyžadující i odlišný způsob uvažování.
- Přinášejí možnost logického sdružování dat a funkcí spolu s přístupovými právy k nim, která umožňují zpřístupnění všem (tj. uživateli) nebo pouze pro vlastní činnost (tj. autorovi) a tím umožnit kontrolu při operacích s daty .
- Zpřehledňují zápis programu.
- Výjimky – nový princip ošetření chybových stavů v probíhajícím programu
- Šablony – způsob pro vícenásobné použití kódu. Na základě vytvořeného předpisu se vytvoří kód pro zvolený typ. Jeden předpis, ze kterého se pro daný typ vytvoří kód.

Základní pojmy

třída (class) - datový celek (datová abstrakce), který obsahuje data + operace, které s nimi manipulují a bezpečnostní prvek, kterým je řízení přístupových práv k datům a metodám. Popisuje objekty – uživatelem definovaný typ.

instance - proměnná

objekt - instance nějaké třídy, je pojmenován, má určitý stav a vlastnosti

metoda - námi definovaná operace přes kterou přistupujeme k datům

zapouzdření (encapsulation) - spojení dat a metod pro manipulaci s těmito daty do jednoho celku

konstruktor a destruktork - speciální metody pro inicializaci a likvidaci objektu

rozhraní (interface) - co třída nabízí ven (k použití uživateli)

implementace - jak to dělá třída uvnitř

dědičnost (inheritance) - nově definovaná třída může mít některé vlastnosti jiné třídy, nepotřebujeme zdrojový kód třídy, po které se dědí

polymorfismus - odvození několika tříd se stejným rozhráním, ale různou implementací, umožňuje jednotný přístup k instancím

Objektové vlastnosti umožňují spojovat do jednoho celku hodnoty (data) a jejich vlastnosti (funkce, metody pracující s těmito daty) společně s tím, že je možné určit, kdo daná data či vlastnosti může používat (zda jsou skrytá pro interní potřebu objektu či veřejně přístupná). Veřejně přístupné prvky potom tvoří rozhraní pro práci s daným objektem, přes nějž máme možnost používat daný typ. K rozhraní patří uživatelské konverze, možnost kontrolovat vznik (inicializaci) a zánik objektu, operátory

Objekty umožňují tvorbu univerzálních knihoven, sdílení kódu, snadnější údržbu programů ...

O objektové vlastnosti jsou rozšířeny i na struct a union. **Až na drobné odchylky, které budou uvedeny, platí pro struct totéž co pro třídu.**

class Třída; Deklarace třídy říká, že existuje nový typ, třída daného jména

class Třída {deklarace proměnných a metod třídy...}; Definice třídy kromě jména popisuje i vlastnosti (obsažená data a metody). Stejně jako u definice struktury v tomto okamžiku nedochází k zabránění paměti – pouze předpis. (Někdy se i tomuto zápisu říká deklarace a definice třídy potom neexistuje)

extern Třída a,b; Deklarace objektů (proměnné) třídy. Nevytváří paměť, pouze oznamuje přítomnost proměnné dané třídy s daným jménem.

Třída a,b; Definice objektů (proměnných) třídy. Vytváří na základě definice třídy objekt (paměť pro data a vytvoření metod). Je možné spojit s inicializací.

Pozn.: Rozlišujeme mezi funkcí, to je typ volání v C, a metodou, což je název pro funkci, které jsou volány prostřednictvím objektu – jsou součástí třídy.

Pozn.: program je dobré členit do modulů se sobě blízkým obsahem – tvořit logické celky kódu. Vzájemně je nemíchat. K modulu definovat obslužné funkce – interface. Proto je nutné si nejdříve rozmyslet, co patří k sobě, definovat třídu a její činnost, promyslet co se zadává a co se může požadovat (výstupy, činnost) a tím stanovit interface – komunikační funkce definující rozhraní.

Pozn.: Při tvorbě programu a volbě logických celků a rozhraní je dobré dodržovat pyramidovou strukturu návaznosti tříd či využití jiných tříd jako parametrů a snažit se vyvarovat křížového propojení struktur. (Např. pro manipulaci s daty vytvořit jednu třídu, pro grafický výstup druhou třídu (tyto třídy lze potom použít univerzálně i pro jiné aplikace) a pro skutečné zobrazení dat vytvořit třídu, která používá (ať už dědí nebo pouze používá) obě dvě (při změně grafického výstupu není potom nutno měnit manipulaci s daty).

Pozn.: Následující kapitoly jsou řazeny podle zvyšování obtížnosti demonstračních příkladů a to tak, aby se postupně rozšiřovaly možnosti tvorby objektu. V počátku kapitoly je vždy odkaz na neobjektové vlastnosti, které je nutné znát pro zvládnutí dané kapitoly.

Pozn.: Jelikož se u objektového programování jedná o komplexní a vzájemně propojené vlastnosti, upozorňuji na to, že až po kapitulu 3.3.8 jsou příklady (které vycházejí pouze z do té doby probrané látky) programátorsky ne zcela v pořádku – zkuste si proto zpětně uvědomit, jak by šly tyto příklady změnit aby využily všech možností jazyka uvedených později. Některé části lze v náhledu budoucího zapsat lépe a u některých zamlčíme věci (které probereme později), ve světle kterých se může daný příklad jevit jako špatná programovací technika.

Při tvorbě a návrhu kdy vystupuje více tříd je nutné si uvědomit tzv. vazbu objektů:

- objekty o sobě ví, používají se – tj. objekty jsou schopny přijmout jiný typ (objekt) jako parametr a pracovat s ním
- objekt je prvkem jiného objektu – obsahuje ho – tj. objekt jednoho typu je parametrem (vnitřní proměnnou) objektu jiného typu
- objekt má vlastnosti jiného objektu + další rozšíření (dědění) základu – objekt zdědil vlastnosti jiné třídy, které jsou jeho součástí

Objektové vlastnosti jazyka C++ umožňují zcela nový programátorský přístup, zpřehledňují a zjednodušují tvorbu a použití kódu. Na druhou stranu k těmto vlastnostem nutno přistupovat jako ke složitému celku s vzájemnými vazbami, což zvyšuje nároky na pochopení těchto mechanismů oproti nižším jazykům (např. C).

3.3.1 Třída a struktura v OOP

Cílem je uvést základní vlastnosti Objektově Orientovaného Programování a nový složený typ - třídu. Základem pro objektové programování je objekt – struktura nebo třída. Oproti jazyku C však struktura obsahuje nejenom data ale i metody (funkce) s nimi pracující.

Test: co je to struktura a jak se používá? Jak se přistupuje k jejím prvkům?

Z neobjektových vlastností: komentáře 3.2.1

Díky klíčovému slovu **class** (třída) je možné vytvořit nový datový typ, který může obsahovat jiné typy, a který navenek působí jako jednotlivý objekt. Tímto třída navazuje na strukturu a union známé z jazyka C. Struktura a union v C++ jsou rozšířeny o stejné vlastnosti jako má třída (bude-li se v následujícím mluvit o class, mluvíme tedy i o struct, nebude-li uvedeno jinak).

Třída je uvedena klíčovým slovem `class`, následuje název třídy (tj název nového typu), parametry jsou ve složených závorkách (následuje středník)

Deklarace třídy a struktury:

```
class    jméno_třídy        { parametry, tělo třídy };  
struct  jméno_struktury    { parametry, tělo struktury };
```

Při deklaraci třídy dochází k popisu třídy a nevyhrazuje žádnou paměť (definice třídy neexistuje, pouze definice objektu třídy)

Při deklaraci jména třídy bez těla – necháme-li vlastní popis třídy na později – není možné použít objekt dané třídy, je však možné použít ukazatel či referenci na danou třídu. Proveďte se zápisem

```
class  jméno_třídy;           nebo  
struct jméno_struktury;
```

Pozn.: Rozdíl mezi třídou a strukturou je pouze v implicitním nastavení přístupových práv a způsobu dědění.

Pozn.: připomínám, že velikost (složených) typů je nutné zjišťovat pomocí klíčového slova `sizeof(Typ)`.

Příklad 3.2.1 objekty

Nadefinujte (na základě současných znalostí) strukturu pro typ `komplex` a třídu pro typ `string`. U typu `komplex` předpokládejte, že bude mít dva prvky s pohyblivou řádovou čárkou a to pro reálnou a imaginární složku. Typ `string` v sobě bude zahrnovat ukazatel na počátek řetězce a celočíselnou proměnnou pro uložení délky řetězce.

```
// ===== komplex 2201. cpp =====  
// struktura v C++ se musí z hlediska dat  
// chovat jako struktura v C  
  
struct Komplex {  
double Re, Im; // složky čísla  
};  
  
int main ()  
{  
    Komplex A, *B, C ; // chová se jako jakýkoli jiný typ  
                        // v C++ není nutné uvádět struct  
  
    A.Re = 10;  
    A.Im = 5;  
    B = & C;  
    B->Re = A.Re; // přístup k prvkům standardními způsoby  
    B->Im = A.Im;
```

```

Return 0;
}

//===== string 2201.cpp =====

class String {
char *txt;    // adresa pro místo na umístění řetězce
int Delka; /* délka řetězce - je-li to opravdu
            "null-terminating string", pak je tato informace
            nadbytečná. Zvětšuje paměťové nároky, ale
            urychluje činnost, protože se nemusí zjišťovat
            délka řetězce - cyklus s podmínkou */
};

int main ()
{
    String A, *B, C ; // chová se jako jakýkoli jiný typ
                      // není nutné uvádět class
    int d = A.Delka;

    C.Delka = 10;
    // přístup k prvkům třídy ač správný, nefunguje
    // proč je tomu tak - viz. kapitola 3.3.3
    return 0;
}

```

KO: co znamená a k čemu slouží klíčové slovo class?

Příklad 3.2.1a: Promyslete si třídu “bod v prostoru” – co by měla umět.

Typ struct je v objektovém programování rozšířen o nové vlastnosti, z hlediska uložení dat zůstává přístup nezměněn. Dále se zavádí nový typ class, který má stejné vlastnosti jako struct avšak jeho data mají implicitně chráněný přístup (tj. nelze k nim bez explicitního povolení přistupovat). Deklarace struct a class je podobná jako u struct jazyka C s tím, že se v ní mohou vyskytovat i metody.

3.3.2 Členy třídy – data a metody

Cílem je rozšířit znalosti o prvcích třídy a ukázat způsob jak do třídy přidat prvek, proměnnou nebo metodu. Prvkem třídy mohou být kromě dat i metody. Práce s metodami je obdobná jako s členskými daty, metoda se vyvolává přes definovaný objekt, přistupuje se k ní přes “.” u objektu nebo přes “->” u ukazatele.

Test: Jak se v C++ definuje a deklaruje proměnná?

Z neobjektových vlastností: probrat deklarace a definice proměnné 3.2.2

Při deklaraci třídy mohou být v jejím těle deklarovány nejen proměnné, ale i metody. Členy třídy (members) tak mohou být: datové členy - datové položky, které mohou být jakéhokoli definovaného typu, který je v daném místě znám. Dále metody - funkce definované ve třídě pro práci s daty.

Deklarace nebo definice metod je v tomto místě shodná s “klasickou” definicí funkcí.

Poslední složkou, která se účastní definice jsou přístupová práva (3.3.3) k (členským) datům a metodám. Tato práva určují, kdo může s danými daty pracovat.

```
class Jmeno_třída { // jedná se o třídu daného jména
    // nový typ se jménem Jmeno_třída (jako int, float...)
    int data1; // členské proměnné třídy - pouze deklarace
    float data2;
    Jmeno *j;
    char string[100];

    int metoda1() {...return 2;}; // členské metody - definice
    void metoda2(int a,float b) {...}; // členské metody - definice
    float metoda3( int a1, Jmeno *a2); // metoda - deklarace
};
```

S metodami třídy se pracuje stejně jako s daty. Metody jsou také prvky třídy, stejně jako data. Objekt, si tedy sebou nese nejen data ale i metody, které může použít. Jelikož jsou metody vlastností objektu dané třídy, vyvolají se tak, že se k nim přistoupí přes objekt

```
Jmeno_třída aa;
int b = aa.metoda2(34,34.54);
```

to znamená, že objekt aa vyvolá metodu metoda2 třídy Jmeno_třída, ke které patří, tak že k ní přistoupí přes operátor přístupu “.”. Metodě jsou předány parametry a vrací hodnotu (jako “klasická” funkce).

Pozn.: Typy jejichž velikost není v daném místě známa (křížové odkazy mezi třídami nebo prvek právě vytvářené třídy) je nutné používat pouze odkazem nebo adresou jejichž velikost je známa (nezáleží na typu).

Příklad 3.2.2 členy tříd

Přidejte k datovým členům třídy z minulé kapitoly metody. Pro komplex napište metodu pro určení velikosti a uhlu fázoru. Pro string napište funkci, která vrací délku uloženého řetězce.

```
// ===== komplex 2202. cpp =====
```

```

#include <math.h>

struct Komplex {
double Re, Im; // složky čísla

// definice metody je stejná jako u obecné funkce
// návratová hodnota - jméno - seznam parametrů - tělo
double Velikost(void) {return 14;} // fázor, délka, modul ...
double Uhel(double a ) {return a-2;} // úhel,
};

int main ()
{
    Komplex A, *B, C ;

    A.Re = 10;  A.Im = 5;
    double Velikost = A.Velikost(); // volání metody u objektu
    // se provádí tak, že se vyvolá vnitřní metoda prvku A
    // jméno funkce nekoliduje s proměnnou,
    // protože je "lokální" - zapouzdřená uvnitř třídy

    B = & A;
    double Uhel = B->Uhel(2.6);
    // volání metody pomocí ukazatele na objekt

    return 0;
}

//===== string 2202.cpp =====

class String {
char *txt; // vlastní řetězec
int Delka; // délka řetězce

int VratDelku(int i) {return i;}
};

int main ()
{
    String A, *B, C ; // chová se jako jakýkoli jiný typ
    // není nutné uvádět class

    A.VratDelku(10); // použití metody ve třídě vede opět k chybě
    // z důvodu přístupových práv 3.3.3
    return 0;
}

```

Součástí objektu v C++ jsou nejen data ale i metody s nimi pracující.

KO: co jsou to členská data a metody třídy?

Příklad 3.2.1: Promyslete si a navrhnete data a metody třídy “bod v prostoru”. Která data budou potřeba, jak se nastaví na počáteční hodnotu, jak je možné je inicializovat, které funkce je nutné připravit (např. vzdálenost dvou bodů, zjistit, zda leží na spojnici jiných bodů, rotace, posun ...)

Vnitřními prvky třídy mohou být nejen data ale i funkce - metody. K metodám se přistupuje obdobně jako k datům třídy - přes “.” u objektu nebo přes “->” u ukazatele.

3.3.3 Data a přístupová práva

Cílem je popsat možnosti řízení přístupových práv k členským prvkům a metodám třídy – to je určit kdo s nimi může manipulovat, buď všichni nebo pouze daná třída. Data a metody mohou být buďto privátní, nebo veřejná. Tyto vlastnosti je možné nastavit a určují “viditelnost” dat (neboli jejich použitelnost) z hlediska obecného použití (vně objektu) – to znamená, zde je bude moci používat uživatel třídy. Objekt dané třídy definován mimo třídy (uživatelem) může používat (přístupem přes “.” nebo “->”) pouze veřejné prvky. Všechny data a metody jsou přístupné uvnitř třídy, což znamená, že všechny metody příslušné ke třídě mohou používat všechny proměnné a metody třídy bez omezení.

Z důvodů kontrolované a bezpečné manipulace s objekty je výhodné, pokud jsou některé metody a všechna data skryta před běžným uživatelem. Pro běžného uživatele jsou pro manipulaci se skrytými prvky vytvořeny metody, které s nimi manipulují (i když uživatel nemá právo přístupu, metody třídy ho mají). Pomocí těchto metod má potom autor třídy možnost kontrolovat činnost uživatele a tím zaručit, že data budou vždy v pořádku (např. pokud může některé z dat nabývat pouze kladných hodnot, potom přímým přístupem je možné ji zapsat i negativní, pomocí přístupu přes metodu je možné tuto chybu zjistit a vyřešit).

Klíčová slova pro označení práva přístupu jsou public, private a protected.

public – značí členy přístupné vně – přístupné uvnitř i uživateli - interface.

private – značí členy lokální, přístupné pouze uvnitř dané třídy

protected – z hlediska jedné úrovně jako private, rozdíl se projeví až při dědění

Použijeme je tak, že se k nim přidá ‘:’. public: private : a fungují jako přepínač, to je od jejich výskytu do dalšího přepnutí platí dané přístupové právo. Lze je libovolně vkládat do definic.

```
class {
int i;      // zde je sekce proměnných s právem private (implicitně
na počátku třídy)
...
public: int fce(double a) { } // zde je sekce proměnných s právem
public
...
private: char b; void ff(int c) { } // zde je sekce s právem private
...
public: ... // atd.
};
```

Public značí veřejný přístup a znamená, že k proměnné nebo metodě lze přistupovat nejenom z dané třídy, ale i z prvků mimo ni.

```
int main()
{
class trida b;
b.funkce(3); /* je-li funkce v definici trida v sekci public,
potom je toto volání správně. Pokud se definice nachází v sekci
private (či protected), potom je překladačem označeno za chybu -
přístup z objektu definovaného mimo třídu není možný */
}
```

private a protected značí potom přístup privátní, tj. že jsou přístupné pouze uvnitř třídy a z prvků přátelských (viz friend). (na úrovni jedné třídy je jejich použití ekvivalentní, rozdíl je až ve způsobu dědění takto označených prvků – kapitola dědění)

S privátními daty lze pracovat jen uvnitř třídy (lokální použití při tvorbě třídy, pro vlastní potřebu třídy, jako její tvůrce), veřejná je potom možno používat pro manipulaci s objektem (vně třídy, uživatelsky).

```
class trida {
fce() { trida b; // uvnitř třídy se přístupová práva neuplatňují
    b.funkce(5) ; // je možné v rámci třídy přistupovat ke všem
prvkům této třídy
}
}
```

Rozdíl mezi class a struct je v tom, že u třídy je za počáteční složenou závorkou defaultně nastaveno (neuvádí se) private, zatímco u struktury public. U prvních uvedených prvků v definici, není nutné uvádět přístupová práva, jsou dána implicitně a to zápis

```
Class { int i   je ekvivalentní   class {private: int i
Struct { int i   je ekvivalentní   struct {public: int i
```

Implicitní právo na počátku public u struct umožňuje přístup k prvkům bez omezení, tak jak ho známe pro práci se strukturou z C (zpětná kompatibilita pro kódy přenesené z C).

Přístupová práva lze uplatnit na členy tříd, kterými jsou data a metody.

Pozn.: Skrytí datových položek (a jejich ochrana), nutně vede k promyšlení interface pro nastavení a zjištění hodnot. Usnadňuje však implementaci – tím, že je omezen přístup k datům, zabraňuje možnosti externě měnit jejich hodnotu. To znamená, že pro autora třídy je možné změnit datovou část aniž by bylo nutno měnit přístup k nim a tedy i napsané programy.

Přístupné funkce lze psát pro různá data, změní-li se data, změní se implementace (vnitřek, realizace) metod ale jejich činnost je stejná (reálné číslo může být např. reprezentováno jako dva parametry Im, Re x Amplituda, Uhel. Pokud jejich zadání a čtení probíhá pomocí metod, potom při změně typu zápisu změníme jejich obsah, ale hlavičky mohou zůstat stejné). Ohraničení změn implementace na jednu třídu – změna dat nepronikne ven (při těchto změnách se doporučuje staré názvy vymazat, nebo alespoň přejmenovat – nevyřešené výskyty odhalí překladač)

Pozn.: Objekt potom může nakonec např. vypadat

```

class Jméno {

Seznam dat
Předpisy pro vznik a zánik objektu – konstruktory a destruktory
Vlastnosti – metody pro práci s daty,
    - operátory pro práci s daty
přístupová práva k předchozímu
přátele – ostatní třídy nebo funkce, které mohou pracovat se
všemi daty a vlastnostmi
};

```

Někteří autoři preferují “logický” zápis v pořadí (upotřebitelnosti metod). Metody a data jsou uváděny v pořadí jak zajímají uživatele. A to nejprve veřejné a poté privátní data a metody (protože privátní “uživatele” nezajímají jelikož se jedná o vnitřní záležitost třídy):

- konstruktory – první co mě zajímá, je jak objekt založit
- destruktory – jakým způsobem objekt zanikne
- metody pro zjištění stavu – jak zjistit aktuální parametry objektu
- metody pro nastavení stavu – jak do něj dostat to co potřebuji
- metody zajišťující funkce a operátory
- členská data

Pozn.: Public data jsou špatně – lépe je všechny skrýt – umožňuje kontrolu dat (chybná data jsou pak vždy chybou třídy) a změnu dat (není nutné měnit použití ale pouze kód třídy)

Pozn.: u unionů je default přístupovým právem public a nejde předefinovat

Příklad 3.2.3 přístupová práva

Vyzkoušejte si na příkladu z minulé kapitoly jak se projeví změna přístupových práv pro jednotlivé členská data a metody. Upravte přístupová práva tak aby vznikl logický kód

```

// ===== komplex 2203. cpp =====
#include <math.h>

struct Komplex { // public: - nemusí se uvádět je implicitní
double Re, Im; // složky čísla

private: // přepínač přístupových práv

double Velikost(void) {return 14;}
// funkce skrytá pro vše mimo třídy - interní
int pom; // interní-privátní proměnná

```

```

public: // přepínač přístupových práv
double Uhel(double a ) {return a-2;} // úhel,
};

int main ()
{
    Komplex A, *B, C ;
    A.Re = 10;  A.Im = 5;
    double Velikost = A.Velikost();
    // volání private metody u objektu (z venku) dá chybu - nelze

    A.pom = 10 ; // přístup k private položce - nelze
    // jméno funkce nekoliduje s proměnnou,
    // protože je "lokální" - zapouzdřená uvnitř třídy
    B = & A;

    double Uhel = B->Uhel(2.6);
    // volání metody u ukazatele na objekt - lze je v sekci public

    B->pom = 0; // nelze k private sekci ani přes ukazatel
    return 0;
}

```

//===== string 2203.cpp =====

```

class String { // private: - dáno implicitně
char *txt; // vlastní řetězec
int Delka; // délka řetězce

public: // přepínač přístupových práv, zveřejnění metody
int VratDelku(int i) {return i;}
};

int main ()
{
    String A, *B, C ; // chová se jako jakýkoli jiný typ
                       // není nutné uvádět class

    A.VratDelku(10);
    // použití metody ve třídě již nevede k chybě
    A.Delka = 4;
    // je v sekci private a tudíž není dosažitelná z venku
    return 0;
}

```

Přístupová práva umožňují rozdělit data a metody na interní a externí (interface). Data se snažíme skrýt pokud možno všechny.

KO: jaká jsou přístupová práva k členským prvkům a metodám? Klíčová slova a význam?

Příklad 3.2.3a: Ve třídě bod v prostoru rozhodněte o přístupových právech k datům a metodám.

Pomocí přístupových práv je možné oddělit vnitřní strukturu od uživatele. Tím, že uživatel nemůže k některým položkám přistupovat přímo ale pouze zprostředkovaně, je umožněno kontrolovat tyto procesy a ošetřit chyby, které by mohly při přímém použití nastat.

Přístupová práva jsou buď veřejná – public, kdy mají přístup všichni, nebo privátní - private, kdy je přístupové právo omezeno pouze na prostředí (namespace) právě definované třídy.

3.3.4 Ukazatel this

Cílem je uvést ukazatel this, který je pomocným prvkem ve třídě a zajišťuje možnost realizace objektových vlastností. This reprezentuje odkaz na aktuální prvek, což umožňuje předat tento prvek do metody, kterou vyvolal a zároveň zde přes něj přistupovat k datům a metodám aktuálnímu prvku.

Pokud objekt použije metodu dané třídy aa.Metoda(), potom říkáme, že objekt vyvolal tuto metodu a je pro ni aktuálním prvkem. Objekt který metodu vyvolal je skrytým parametrem ve vyvolané metodě (metoda pracuje nad tímto objektem, zná jeho data a metody) a protože metodu může vyvolat každý objekt dané třídy, je aktuální objekt uvnitř metody označen univerzálním názvem this.

Test: k čemu slouží reference?

Z neobjektových vlastností: vyžaduje referenci

this je klíčovým slovem a reprezentuje jméno proměnné typu konstantní ukazatel, na objekt aktuální třídy. Je to ukazatel, který je implicitně přítomen v každém objektu, proměnné dané třídy.

Jméno_třídy * const this

Pozn.: class T {T const *this;} – takhle si to lze představit, je to však implicitní a tudíž se nepíše, je to přítomno automaticky.

Pozn.: lze si představit, že ukazuje sám na sebe - je-li prvním prvkem objektu, potom vlastně ukazuje na objekt dané třídy

Ukazatel this se využívá při volání metod třídy. Máme-li volání metody typu:

Trida aa, b; i = aa.Metoda(b);

Potom je ve třídě Trida nadefinována metoda Metoda

```
class Trida { ...
```

```
int Metoda(Trida &bb) {tělo...} ...
```

}

Tuto metodu v našem případě použije, nebo-li vyvolá objekt aa a má parametr b. Objekt aa, který metodu vyvolal se v těle metody jmenuje this a byl předán implicitně (překladačem). Objekt b je v těle reprezentován objektem (referencí) bb. Při volání b.Metoda(aa) – bude uvnitř metody this reprezentovat ukazatel na b, protože b je objekt, který metodu vyvolal a bb bude reprezentantem aa, protože tento objekt byl předán jako parametr. Ukazatel this tedy reprezentuje objekt, který metodu vyvolal, objekt, se kterým se aktuálně pracuje.

this se používá:

- Má-li být návratovou hodnotou objekt, který metodu vyvolal, nebo-li aktuální objekt vrací sám - return * this;

- je-li nutné zkontrolovat, zda-li předávaný objekt (parametr) není totožný s objektem, který metodu vyvolal if(this==&bb) {předávaný objekt je stejný jako aktuální objekt}

- pro přístup k proměnným a metodám třídy aktuálního prvku. I pro objekt, který vyvolal metodu je možné uvnitř této metody využít jeho data a metody - lze k nim přistupovat plným přístupem pomocí this: this->prvek, this->Metoda() nebo zjednodušeně, kdy je možno this vynechat a psát přímo názvy – prvek, Metoda(). V objektu se potom na proměnné aktuálního objektu odkazujeme jejich jménem, např. prvek = 5, co je ekvivalentní this->prvek = 5

this tedy je implicitně přítomen v každé definici metody – každá nestatická metoda třídy má minimálně jeden (skrytý) parametr – this. This je nastaven vždy na aktuální instanci

Příklad 3.2.4 this

Upravte metodu Velikost tak, aby skutečně počítala velikost a uhel z dat struktury. Napište metodu, která srovná dva řetězce a vrátí delší.

```
// ===== komplex 2204. cpp =====
#include <math.h>

struct Komplex
{ // public: - nemusí se uvádět je implicitní
double Re, Im; // složky čísla

double Velikost(void)
{
    this->pom = this->Re; // přístup k prvku pomocí this,
        // dostaneme se dovnitř objektu -
        // a jelikož je to z vlastní metody objektu,
        // můžeme i na data a metody ze sekce
        // private
    pom = pom * pom + Im * Im; // zkrácený zápis
        // pokud pracujeme s metodami nebo daty aktuálního prvku,
        // nemusí se this-> uvádět

    return sqrt(pom);
} // funkce skrytá pro vše mimo třídy - interní
```

```

private: // přepínač přístupových práv
double pom; // pomocna proměnná

public: // přepínač přístupových práv
double Uhel(void)
{
    // this je prvním (skrytým) parametrem volání - neuvádí se
    return atan2(this->Im,Re);
} // přístupy lze libovolně kombinovat
};

int main ()
{
    Komplex A, *B, C ;
    A.Re = 10; A.Im = 5;
    double Velikost = A.Velikost();
    // po volání se A stává v metodě *this
    Velikost = C.Velikost();
    // po volání se C stává v metodě *this
    // objekt, který metodu vyvolá, se do ní dostane jako
    // parametr pomocí this
    // metoda pracuje nad objektem, který ji vyvolal
    B = &A;
    Velikost = B->Uhel();
    A.pom = 10; // vede stále k chybě
    return 0;
}

```

//===== string 2204.cpp =====

```

class String { // private: - dáno implicitně
char *txt; // vlastní řetězec
int Delka; // délka řetězce

public: // přepínač přístupových práv, zveřejnění metody

int VratDelku(void) {return this->Delka;}
    // přístup k prvku objektu pomocí this

void NastavDelku(int i) {Delka = i;}

String & VratDelsi(String & p)
{ // v metodě máme dvě proměnné typu string
    // jedna je this-> a druhá p, k proměnným první můžeme pomocí
    // this->Delka, nebo jen Delka. U druhé musí být vždy p.Delka
    if (this == &p) // je-li to tentýž prvek, musí být stejné -
        // srovnání adres prvků lze pouze pomocí this
        return *this; // a vracím třeba aktuální prvek - pomocí this
    if (Delka > p.Delka)
        return *this;
    else return p;
}

```

```

};

int main ()
{
    String A, *B, C ; // chová se jako jakýkoli jiný typ
                      // není nutné uvádět class

    A.NastavDelku(10);
    C.NastavDelku(15);
    int d = A.VratDelku();
    // použití metody ve třídě již nevede k chybě
    // A.Delka = 4;
    // Delka je v sekci private a tudíž není dosažitelná z venku

    d = (A.VratDelsi(C)).VratDelku();
    // první metoda vrátí delší z prvků
    // jako objekt a ten vyvolá
    // metodu, která vrátí délku (tohoto objektu)
    //- zkusit odtrasovat

    // a pomocí adres prvků A,C a this určit co se volá
    d = C.VratDelsi(A).VratDelku();
    d = C.VratDelsi(C).VratDelku();
    return 0;
}

```

KO: co je ukazatel this? Jak je definován? Kdy se používá?

Ukazatel **this** je parametrem, který uvnitř metody reprezentuje objekt, který vyvolal danou metodu. Pomocí ukazatele **this** je možné přistupovat k datům a metodám tohoto objektu. Uvádění **this** je však v této souvislosti nepovinné.

3.3.5 Statický datový člen třídy

Cílem je ukázat vytvoření společné proměnné pro všechny prvky třídy. Tento datový člen je přítomen pouze jednou v celém programu.

Pokud je člen třídy označen jako **static**, potom je společný pro všechny objekty dané třídy. Z toho plyne, že daná proměnná vzniká v programu pouze jednou pro celou třídu. Proto není fyzicky součástí každého objektu (proměnné dané třídy) ale je umístěna v globální oblasti dat, kam k ní mohou všechny objekty přistupovat.

```

class string {
    static int pocet;    // deklarace
}

```


Pozn.: pokud je statická proměnná uvedena v sekci public, pak k ní mohou přistupovat všichni.

Tento datový člen třídy je vytvořen v okamžiku deklarace třídy - nemusí být žádná instance třídy a tento prvek již existuje. Tento prvek je nutné na globální úrovni programu nadefinovat a inicializovat (protože jinak by hodnota v něm mohla být nedefinována):

```
int string::pocet=0;//definice statického prvku s inicializací
```

Tento typ proměnné může sloužit např. k počítání vytvořených objektů dané třídy a to jejich celkový počet nebo aktuální počet. V případě, že se mají všechny prvky chovat v daném okamžiku stejně (např. barva okna, typ kurzoru), mohou mít ve statické proměnné aktuální nastavení (barva plochy, kurzor ...). Obecně zde tedy jsou jiná společná data, počet volání funkcí (demo verze), otevřený tok pro ukládání dat do log file pro danou třídu ...

Příklad 3.2.5 statický datový člen třídy

Rozšířte příklady z minulých kapitol tak, aby počítaly počet volání metody Velikost (bez ohledu na to, který prvek ji vyvolal). (Řešte za pomoci zavedení statické proměnné)

```
// ===== komplex 2205. cpp =====
#include <math.h>

struct Komplex {
double Re, Im;
static int PocetVolani; //deklarace - nemá místo v objektu

double Velikost(void) {
this->pom = this->Re;
pom = pom * pom + Im * Im;
// mění proměnnou aktuálního objektu
// pokud pracujeme s metodami nebo daty aktuálního prvku,
// nemusí se this-> uvádět
PocetVolani ++;
// každé volání od kteréhokoli prvku provede přičtení
// jedničky ke společné - statické proměnné
return sqrt(pom);}

private: // přepínač přístupových práv
double pom; // pomocna proměnná

public: // přepínač přístupových práv
double Uhel(void) {
return atan2(this->Im,Re);}
};

// následující je definice, která musí říci typ,
// může (měla by) provést inicializaci
// je třeba označit, že proměnná PocetVolani není obyčejná
// globální proměnná, ale že patří ke třídě Komplex.
// To provedeme rozšířením jména proměnné o jméno třídy
// s operátorem příslušnosti - Komplex::
// celý zápis tedy říká, že je proměnná typu int, patří ke
```

```

// třídě Komplex, jmenuje se PocetVolani a bude
// inicializována na hodnotu nula
int Komplex::PocetVolani = 0;

int main ()
{
// sledujte hodnotu proměnné "Komplex::PocetVolani"
// jak zní "celé jméno" statické proměnné
Komplex A, *B, C ;

A.Re = 10;  A.Im = 5;
double Velikost = A.Velikost();
Velikost = C.Velikost();
B = &A;

Komplex::PocetVolani++;
// máme volný přístup k prvku což není vždy dobré - ale byl
// definován v sekci public takže zde je to správně

Velikost = B->Velikost();

return 0;
}

```

```

//===== string 2205.cpp =====

class String { // private: - dáno implicitně
char *txt; // vlastní řetězec
int Delka; // délka řetězce
static int PocetPrvku, PocetAktivnichPrvku;

public: // přepínač přístupových práv, zveřejnění metody
int VratDelku(void)
{
PocetPrvku++; // demonstruje činnost static proměnné
// všechny objekty daného typu mění stejnou proměnnou
return this->Delka;
} // přístup k prvku objektu pomocí this

void NastavDelku(int i) {Delka = i;}

String & VratDelsi(String & p) {
if (this == &p) // je-li to tentýž prvek, musí být stejné -
return *this;
if (Delka > p.Delka)
return *this;
else return p;
}
};

```

```

// je třeba udat typ, příslušnost ke třídě,
// jméno proměnné a inicializovat
// přístupová práva se zde neuvádí
// při inicializaci je možné přistoupit i k private prvkům
int String::PocetPrvku = 0;
int String::PocetAktivnichPrvku = 0;

int main ()
{
    // sledujte co se děje se statickou proměnnou
    // String::PocetPrvku;
    String A, *B, C ;
    A.NastavDelku(10);
    C.NastavDelku(15);
    int d = A.VratDelku();
    d = (A.VratDelsi(C)).VratDelku();
    d = C.VratDelsi(A).VratDelku();
    d = C.VratDelsi(C).VratDelku();
    // String::PocetPrvku++; // dá chybu - private - chcete-li
    // ji sledovat, je nutné vytvořit metodu, která ji vrátí

    return 0;
}

```

Pozn.: Pokud uvnitř třídy uvedeme static const int a ; , potom je vytvořena proměnná, která je konstanta, nevytváří se a zároveň je “vidět” pouze uvnitř třídy. Dá se použít pro pole konstantní délky uvnitř třídy

KO: co je a k čemu se používá statický datový člen třídy? Jaké má vlastnosti?

Příklad 3.2.5a : Navrhněte využití statického členu pro počítání aktivních prvků a počtu vytvořených prvků.

Nástin řešení: Pro počet všech vytvořených prvků je nutné zavést statickou proměnnou, kterou v každém konstruktoru inkrementujeme. Pro počet všech aktivních prvků je nutné zavést proměnnou, kterou v každém konstruktoru inkrementujeme a v destruktoru dekrementujeme (tato proměnná může na konci programu sloužit ke kontrole, zda jsou správně volány konstruktory a destruktory pro všechny prvky).

Statický datový člen třídy je prvkem pro společné užití všemi objekty třídy. Vzniká na globální úrovni a je společný všem členům třídy. Pro ostatní je přístupný pouze je-li uveden v sekci public. Je dobré ho inicializovat v definici.

3.3.6 Konstruktory a destruktory

Cílem je popsat mechanismy vzniku a zániku objektů. Při vzniku a zániku jsou překladačem volány metody zvané konstruktory a destruktory. Využití těchto metod při programování je výhodné v tom, že slouží pro definovaný vznik a zánik objektu (jsou volány ihned po vzniku objektu – první volaná metoda je konstruktor, a při zániku

objektu – poslední volaná metoda je destruktory). Konstruktory slouží převážně k inicializaci proměnných. Destruktor umožňuje ošetřit data objektu před jejich zánikem.

Test: Co je to přetěžování? Jak se definují funkce s implicitními parametry? Co je to přetypování a kdy a k čemu se používá? K čemu slouží klíčové slovo const? Jak se v C++ alokuje paměť?

Pozn.: vyžaduje přetěžování, implicitní parametry, přetypování, const parametry, alokace paměti, enum

Konstruktor

Připomeňme si jak je možné definovat (a inicializovat) proměnnou v jazyce C a co se děje pokud napíšeme např.:

```
int i, j = 5, k = j;
```

V tomto okamžiku jsme nadefinovali proměnné i,j,k typu int. Proměnnou j jsme inicializovali (naplnili) hodnotou pět. Proměnnou k jsme inicializovali za pomoci proměnné j. Překladač tedy vytvoří proměnnou i tak, že jí rezervuje v paměti oblast ve které bude uložena její hodnota (v našem případě je zde cokoli co bylo v paměti od minula). Dále překladač vytvoří místo pro proměnnou j a naplní ho hodnotou pět. Dále vytvoří místo v paměti pro proměnnou k a do něj zapíše hodnotu, která je na místě proměnné k (proměnná j v tomto nemusí být stejného typu jako k a v tom případě je ještě vložena konverze z typu proměnné j na typ proměnné k).

Nyní ukážeme tento mechanismus pro třídy (předchozím jsme chtěli ukázat, že to není až taková novinka). Zde je ovšem problém v tom, že třída je typ nový, překladači neznámý a složitější než základní typy. Proto aby překladač věděl co s inicializací, je nutné vytvořit metody, pomocí nichž se inicializace provede. Tyto metody se nazývají konstruktory. Konstruktory mají jméno shodné s názvem třídy.

Uvažujme tedy, že pro svou třídu chceme zajistit totéž, co pro případ typu int:

```
Trida ii, jj = 5, kk = jj, ll(4,5,"aldff");
```

Jak si můžeme všimnout u ll, lze pro třídy dokázat více než pro základní typy. Konstruktory lze díky přetěžování napsat s libovolnými parametry a proto můžeme např. napsat i konstruktor bez parametrů, který by se volal pro proměnnou ii a tím provedl její inicializaci (není zadána zvenku, ale každá proměnná má jistý "klidový" stav, kterým je možné ji naplnit).

Pro proměnnou ii by se tedy napsal konstruktor Trida(void) { ... } který by nastavil data do základního tvaru. Pro proměnnou jj by byl volán konstruktor Trida(int) { ... } tedy konstruktor s jedním parametrem typu int, který by na základě tohoto parametru nastavil data objektu. Pro proměnnou kk by byl volán konstruktor Trida(Trida&) { ... } který vytváří objekt na základě objektu stejného typu. Zde je dobré si uvědomit, že se nevolá klasické "rovná se" pro přiřazení, ale že je nutné číst "objekt vznikne" na základě stejného typu. Pro proměnnou ll je nutné vytvořit konstruktor Trida(int, int, char *) { ... }, který na základě těchto parametrů nastaví data objektu.

Pro definované konstruktory třídy platí to, že jsou při definici proměnných volány implicitně překladačem a správný je vybrán podle seznamu parametrů. Konstruktor není možné volat jako metodu (Třída aa.Třída(2,3,2) – není možné).

Pozn.: pro případ `jj=5` by mohlo dojít i k volání konverzního konstruktoru pro 5 a následně k volání kopykonstruktoru. Toto “nedorozumění” je možné odstranit voláním `jj(5)`.

Konstruktor je první metodou, která se automaticky volá, poté co je objekt vytvořen v paměti a proto je jasné, že v tomto okamžiku žádná z jeho proměnných nebyla použita a tedy ani inicializována. Konstruktor má v definici třídy stejný název jako je jméno třídy: `Jméno_třídy(parametry) {}` a není možné volat samostatně (volá ho překladač při vytváření objektu, jak proměnné tak pro vznik pomocí `new`). Nemá návratovou hodnotu (nepíše se ani do deklarace či definice (nepoužije se ani `void`))

Konstruktor se používá k nastavení počátečních hodnot proměnných, nastavení ukazatelů, alokaci paměti, vytvoření tabulky virtuálních metod ...

Konstruktorů může být několik, a pro jejich použití a vyhledání vhodného platí pravidla pro přetěžování funkcí.

Speciálními konstruktory jsou:

- **implicitní konstruktor** – je konstruktor bez parametrů - `Jméno_třídy(void) {}`
- **Kopykonstruktor** - konstruktor jehož parametrem je odkaz na objekt stejné třídy – `Jméno_třídy(const jméno_třídy &a) {}`
- **konverzní konstruktor** - konstruktor jenž má jeden parametr, a slouží tedy k převodu proměnné jednoho typu na jiný - `Jméno_třídy(jméno_typu b) {}`

implicitní konstruktor se vytváří implicitně pokud žádný konstruktor neexistuje. Implicitní kopykonstruktor se vytváří pokud není definován. Implicitní konstruktor se nevytvoří, je-li alespoň jeden uveden.

Pozn.: komplex `a(55)` je totéž co komplex `a=55`; (druhá možnost – konstruktor (`int`) + kopykonstruktor)

Pozn.: Kopykonstruktor se používá např v definici, když prvek vzniká na základě jiného prvku téže třídy : `string a(“askdfj”), b = a;` (to = neznamená přiřazení ale kopykonstruktor) , dále se používá, vrací – li funkce prvek stejné třídy, nebo ke tvorbě předávaných (lokálních) parametrů funkcím

Pozn. konstruktor nesmí být `static` ani `virtual`

Implicitní konstruktor je volán pro prvky polí. To znamená, že při vzniku (statického nebo dynamického) pole, je pro každý z objektů volán implicitní konstruktor od nejnižšího indexu. Implicitní konstruktor musí tedy existovat, chceme-li vytvořit pole dané třídy. Tímto mechanismem se liší alokace pomocí `new` od staršího `xxalloc`.

Je-li nutno pole inicializovat různými konstruktory, pak pro kratší pole lze:

`Třída Pole [] = {Třída(8,5), Třída(4), Třída(“afdaf”), Třída(3,4,5)};` Pro delší pole je nutné provádět inicializaci nastavením prvek po prvku. (Předtím ovšem proběhnou implicitní konstruktory).

Při tvorbě programů se může vyskytnout situace, kdy se zavolá konstruktor implicitně a je potřeba tomu zabránit (např. při větším množství konverzních konstruktorů dochází k jejich kolizím ...). K zabránění implicitního volání slouží klíčové slovo `explicit`. Klíčové slovo `explicit` před názvem konverzního konstruktoru zabrání implicitnímu využití konstruktoru, takže ten v tomto případě nebude použit při konverzi, pokud tato nebude vyžádána explicitně programátorem. (Nevýhodou je, že je to vlastností třídy a platí to tedy pro všechna místa použití).

Využití je např. v případě, že máme třídu komplexních čísel a konverzní konstruktor z typu `int`, ale chceme zabránit situaci, aby v místě požadovaného komplexního čísla mohl stát typ `int`. To je např. pro případ kdy voláme funkci s prototypem `f(Komplex a)` pomocí volání s parametrem typu `int` `f(5)`. Potom se nejprve provede konverze hodnoty `5` na komplex a následně se volá funkce. Chceme-li tomuto zabránit, označíme konstruktor s parametrem `int` jako `explicit`. Pokud v této situaci vyžadujeme volání s parametrem `5`, bude nutno explicitně uvést přetypování na typ komplex `f(Komplex(5))`.

Pozn.: provádí se pouze jedna implicitní (nedefinovaná – uživatelská, konverze zabudované v systému se nepočítají) konverze v řadě. Pokud je nutno provést přetypování přes jeden objekt (např. `int` na `double` by musel jít nejprve na `float` a z `float` teprve na `double`, a obě konverze by byly uživatelské (jakože nejsou, je to příklad), potom by se přetypování neprovedlo.

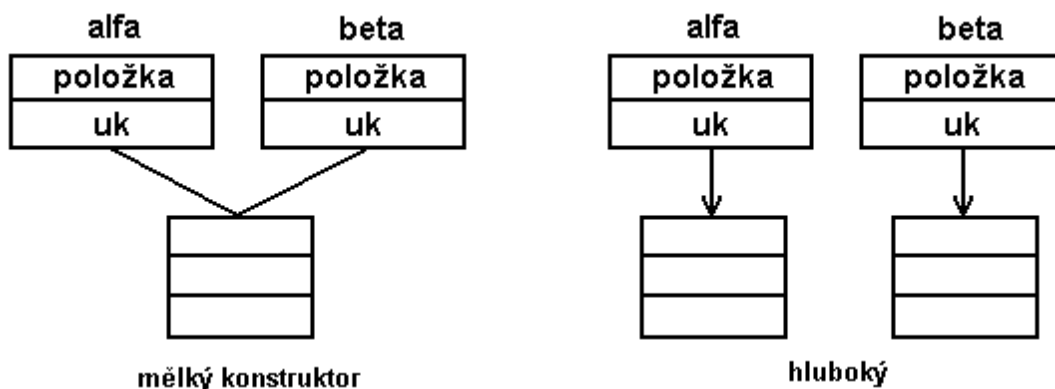
Pozn.: vytvoří-li se implicitní (`tmp`) objekt dá se říct, že se zruší ihned jak je to možné

Mělký a hluboký copy konstruktor (shallow and deep copy)

V případě, že jeden objekt má nabývat stejných hodnot jako objekt jiný, je nejčastěji použito přiřazení hodnoty hodnotě. Tato situace, která nejčastěji nastává při tvorbě kopykonstruktoru nebo operátoru “rovná se” však přináší problém při použití s ukazateli. Ukazatel totiž ukazuje na místo kde jsou data. Pokud jeho hodnotu přiřadíme jinému ukazateli, potom jsou data společná pro dva (a více) objektů. V případě, že si nepamatujeme, kolik ukazatelů na data míří, může se stát, že je jeden objekt odalokuje (např. při zániku objektu v destrukturu) a ostatní míří do prostoru, kde již nejsou alokovaná data, popř. jsou-li data změněna, změní se pro všechny objekty (i když jsou případy, kdy je to správně (úmyslně), standardní proměnná nemění svou hodnotu v okamžiku, kdy je měněna jiná ale pouze přístupem přes sebe).

Pro vytváření objektů, které obsahují dynamická data (ukazatele) nelze tedy použít prosté přiřazení hodnot, protože v tom případě, by se oba prvky dělily a stejný paměťový prostor. Při definici copy konstruktoru je tedy nutné dát pozor na dynamická data, která je nutno ošetřit. Při prosté kopii, kdy ukazatele ukazují do stejného prostoru potom mluvíme o mělkém konstrukturu, pokud je tento jev ošetřen nazýváme konstruktor hlubokým.

Ošetření je realizováno nejčastěji dvěma způsoby, prvním je udělat kopii dat pro nový objekt, druhým je způsob, který počítá odkazy na data a kopii dat se udělá pouze když se data mění.



Pozn.: vytváří-li se implicitně kopykonstruktor, pak mělký

Destruktor

Stejně jako se volá po přidělení paměti proměnné konstruktor, je před zrušením místa pro proměnnou volán destruktorem. Destruktor je poslední metoda, která se volá před ukončením existence objektu. Destruktor potom nejčastěji zajišťuje úklid (ukončení životnosti objektu), tj. odalokování paměti, vrácení handlů (prostředky systému, grafika, soubory, ovladače, HW ...), uložení dat. Pro objekty vytvořené překladačem je volán automaticky překladačem. Pro dynamicky vzniklé objekty je volán při delete (na rozdíl od xxfree).

Destruktor má název stejný jako je jméno třídy, kterému předchází ~ a nemá návratovou hodnotu (nepoužije se ani void)). Destruktor je jeden (bez parametrů).

```
~JmenoTridy(void) { ... }
```

Pokud není destruktorem definován, potom se vytváří implicitně a je prázdný.

Pozn.: destruktorem je možné zavolat explicitně (nedojde však ke zrušení paměťového místa vlastního objektu). I když se toto použití nedoporučuje (lépe je realizovat funkce např. funkci clear, která provede vyčištění a tu volat), může se stát že destruktorem bude volán jako metoda a je třeba se připravit i na situaci, kdy objekt po volání destruktorem žije dále. Je tedy dobré nastavit proměnné na počáteční podmínky pro znovu použití objektu.

Pozn.: explicitní volání destruktorem jako metody u objektů odvozených tříd (zvláště pak mají-li virtuální metody) má širší souvislosti a může měnit i chování objektu – proto se volání destruktorem používá jen v případech, kdy jsme si jisti následky této činnosti.

Pozn.: při odalokování polí se volají destruktory od nejvyššího indexu

Pozn.: destruktorem není volán (automaticky, tj. překladačem) na objekt, který vznikl pomocí new, tj. na ukazatel, je nutné prvek odalokovat pomocí delete

Alespoň jeden konstruktor a destruktorem musí být v části public, protože jinak by nebylo v uživatelské části možno objekt vytvořit nebo zrušit a tedy ani použít.

Pozn.: Implicitně vytvářené destruktory a konstruktory zděděných tříd se nevytváří prázdné, ale volají konstruktory a destruktory svých vnořených objektů.

Pokud **třída obsahuje prvky jiné třídy**, potom se jejich konstruktory volají předtím, než se provede vlastní tělo konstruktorem. Mají-li se tedy provést inicializace prvků na požadovanou

hodnotu je nevhodné toto provádět v těle konstrukturu, protože v tomto případě je volán implicitní konstruktor a následně jsou hodnoty inicializovány v těle konstrukturu.

Pokud máme definici třídy `Třída { Komplex b ;int a; ... }`, a konstruktor

`Třída(double re, double im,int aa) {b.re=re;b.im=im;a=aa;}` potom v případě použití

Třída `cc(3.1,3.2, 5)` je nejprve zavolán implicitní konstruktor na objekt `b` třídy `Komplex` a následně je v těle konstrukturu přiřazení hodnot (a tudíž je provedení implicitního konstrukturu ztrátou času, protože implicitní hodnoty jsou vzápětí přepsány konkrétními `re` a `im`).

Tento problém je možné vyřešit následujícím způsobem definice konstrukturu:

`Třída(double re, double im, int aa) : a(aa), b(re,im) { }` kterou říkáme, že proměnná `a` se má vytvořit na základě hodnoty `aa` (konstruktor `int` na základě hodnoty `int` – je tedy rozšířeno i na stávající datové typy) a proměnná `b` na základě hodnot `re` a `im` (tj. konstrukturu se dvěma parametry). Tělo konstrukturu je potom prázdné a nedochází k dvojímu zapisování hodnot.

Pořadí volání konstruktůrů je určeno jejich pozicí v definici a tak pro uvedený případ bude nejprve voláno `b(re,im)` a poté `a(aa)`, tedy v pořadí uvedeném v definici třídy a ne u konstrukturu.

Příklad 3.2.6 konstruktory a destruktory

Napište pro příklady z minulých kapitol konstruktory a destruktory.

```
//===== komplex2206.cpp =====
// trasujte a divejte se kudyma to chodi,
// tj. zobrazte *this, ...
// objekty muzete rozlisit pomoci indexu

#include <math.h>

struct Komplex {
// použití enum - tentokrát pro určení typu vstupu
enum TKomplexType {eSlozky, eUhel};
// můžeme zadat složkový tvar (Re,Im), nebo tvar amplituda,
// úhel

static int Poradi;
static int Aktivnich;

double Re,Im;
int Index;
// implicitní konstruktor - jelikož jsou i jiné,
// musí být uveden
// jinak by byl vygenerován s prázdným tělem
Komplex(void) {Re=Im=0;Index = Poradi;Poradi++;Aktivnich++; }

// konstruktor, který se použije i jako konverzní
// na celočíselné i float typy
// při povolení explicit je nutno zazávkovat
// řádek s demonstrací chyby
```



```

// explicit
Komplex(double re,double im=0, TKomplexType kt = eSlozky)
{Re=re;Im=im;Index=Poradi;Poradi++;Aktivnich++;
  if (kt == eUhel)
    // přístup k typům enum uvnitř třídy je jednoduchý
    {Re=re*cos(im);Im = Re*sin(im);}
}

// konverzní na řetězce
Komplex(const char *txt)
{ /* vlastní alg */;
  Re=Im=0;Index = Poradi;Poradi++;Aktivnich++;}

// kopy konstruktor
// nebude-li uveden, vytvoří se implicitní, tj. udělá se (s)prostá
kopie 1:1
// což nám nabourá číslování prvků a kontrolu aktivních prvků
Komplex(const Komplex &p)
  {Re=p.Re;Im=p.Im;Index=Poradi;Poradi++;Aktivnich++;}

// destruktork - kdyby nebyl bude vygenerován s prázdným tělem
// (k jeho vygenerování dojde "na pozadí",
// tj. ve zdrojích se neobjeví)
~Komplex(void) {Aktivnich--;}

// metoda, která sečte dva prvky a naplní volající
// volání hodnotou demonstruje vlastnosti volání,
// ale je to chyba
void PriradSoucet(Komplex p1,Komplex p2)
  {Re=p1.Re+p2.Re;Im=p1.Im+p2.Im;}

// správné volání odkazem u funkce, která k aktivnímu přičte
// hodnotu a vrátí výsledek. Jelikož součet nemění sčítance,
// musí se vytvořit prvek nový pro výsledek
Komplex Soucet(const Komplex &p)
  {Komplex pom(Re+p.Re,Im+p.Im);return pom;}

// přiřazení, které funguje jako klasické = musí být
// schopno zřetězeného volání
// protože volající prvek je sám výsledkem,
// může vrátit sám sebe
Komplex& Prirad(Komplex const &p)
  {Re=p.Re;Im=p.Im;return *this;}
};

// čítače pořadí vzniku a aktivních prvků
int Komplex::Poradi=0;
int Komplex::Aktivnich=0;

// pomocné řetězce - možné tvary komplexních
// čísel pro načítání

```

```

char str1[]="(73.1,24.5)";
char str2[]="23+34.2i";

int main ()
{
    Komplex a; // (implicitní) konstruktor bez parametrů
    Komplex b(5),c(4,7); // vytvoření na základě složek
    Komplex d(str1),e(str2); // vytvoření na základě řetězce
    Komplex f=c,g(c); // dva způsoby volání kopykonstruktoru
    Komplex h(12,35*3.1415/180.,Komplex::eUhel);
    // = v definici je kopykonstruktor a ne operátor =
    // pokud se vyzávorkuje kopykonstruktor, nedojde k započtení
    // do statických proměnných
    // přístup k proměnné enum je nutný pře operátor přístupu
    // a jméno třídy, uvnitř které je definován.
    // Na globální úrovni není hodnota vidět

    Komplex::TKomplexType typ = Komplex:: eUhel;
    // pokud chceme definovat proměnnou enum, potom
    // musíme použít celý název typu tj. i třídu ve které
    // je definován

    Komplex i(10,128*3.1415/180,typ);

    d.PriradSoucet(b,c);
    e.Prirad(d.Prirad(c)); // zřetězení přiřazení
    d.PriradSoucet(5,c); // pokud dáme ke konstruktoru explicit
    // musíme použít následující, které již bude fungovat

    d.PriradSoucet(Komplex(5),c);
    return 0;
    // trasujte dále - volají se destruktory vytvořených prvků
}

//===== string2206.cpp =====
// trasujte a divejte se co se deje

#include <stdio.h>
#include <string.h>

class String {
    static int Poradi;
    static int Aktivnich;
    char *txt;
    int delka;
    int Index;

public:
    // implicitní kopykonstruktor vytvoří prázdný prvek
    // Je-li vytvořen jakýkoli jiný, nevytváří se automaticky
    // (zkuste zazávorkovat)

```

```

// zároveň by při automatickém vytvoření by
// nepřičítal k indexům
String(void) { Index = Poradi;Poradi++;Aktivnich++;
               txt = NULL; delka = 0; }

// kopykonstruktor se tvoří implicitně, není - li uveden
// (je vyzávorkován)
// je ovšem vytvořen mělký konstruktor a při
// destrukci objektů dochází k chybě
String( const String &b )
{Index = Poradi;Poradi++;Aktivnich++;
  delka = b.delka;
  if (delka > 0) {txt = (char *)new char [ delka ] ;
                  int i;
                  for (i = 0; i <delka; i++)  txt [ i ] = b.txt [ i ];}
  else txt = NULL;}

// ostatní konstruktory už se implicitně nevytváří -
// chybí-li je to chyba
// konverzní konstruktor pro načtení float typů
// ( a nouzově přes konverzi, pro celočíselné typy)
// klíčové slovo explicit zakáže použití konstruktoru
/ (odzávorkovat)
// explicit
String(double d)
{
  Index = Poradi;Poradi++;Aktivnich++;
  char tmp[50] = "";sprintf ( tmp, " %lf ", d );
  delka = strlen ( tmp ) + 1;
  txt = (char *) new char [ delka ] ;int i;
  for (i = 0; i <delka-1; i++)  txt[i] = tmp[ i ];
  txt [ delka ] ='\0';}

String( const char *t )
{
  Index = Poradi;Poradi++;Aktivnich++;
  delka = strlen ( t ) + 1;
  if (delka > 1) {txt = new (char [ delka ]);
                  int i;
                  for (i = 0; i < delka-1; i++) txt [ i ] = t [ i ];
                  txt [ delka ] ='\0';}
  else { delka = 0; txt = NULL; }}

~String(void)
{
  Aktivnich--;
  if (txt != NULL) delete [ ] txt;  txt = NULL;}

// parametr by měl být odkazem -

```

```

// zbytečně se volá kopykonstruktor na vytvoření s

String Soucet(String s)
// vytvoří se lokální kopie s, a proměnná a
{
    String a ; int i , j ;
    a.delka = delka + s.delka - 1;
    if ( a. delka == 0 ) a.txt = NULL;
    else {
        a.txt = new char [ a.delka ] ;
        for ( i = 0 ; i < delka-1 ; i ++ )
            a. txt [ i ] = txt [ i ] ;
        for ( j = 0 ; j < s.delka ; j ++ , i ++ )
            a. txt [ i ] = s.txt [ j ] ;
        a.txt [ a.delka-1 ] = '\\0' ; }
// kopie výsledku, rušení lokálních proměnných
return a; }

String& Prirad (String &s) {
    if (&s == this) return (*this);
// nutno ošetřit, je-li a=a, protože, když bysme
// manipulovali jedním, měnil by se nám druhý
    if (txt) delete[] txt;
    delka = s.delka;
    if (delka > 0) {txt = (char *)new char [ delka ] ;
    int i;
    for (i = 0; i <delka; i++) txt [ i ] = s.txt [ i ];}
    else txt = NULL;
    return *this; }
// může vrátit sám sebe, není nutno tvořit nový
// návratový objekt

void PriradSpojeni(String &s1,String & s2)
{
    char *pom = (char *) new char[s1.delka+s2.delka+1];
    int i , j ;
    delka = s1.delka + s2.delka - 1;
    if ( delka == 0 ) pom = NULL;
    else {
        for ( i = 0 ; i < s1.delka-1 ; i ++ )
            pom [ i ] = s1.txt [ i ] ;
        for ( j = 0 ; j < s2.delka ; j ++ , i ++ )
            pom [ i ] = s2.txt [ j ] ;
        pom [ i ] = '\\0' ;
        if (txt) delete[] txt; txt = pom; }
    }
};

int String::Poradi = 0;
int String::Aktivnich=0;

```

```

int main() {
    String a; // volani implicitniho konstruktoru
    String b(8.3),c(3),d("sadfl");
    // konverzní konstruktory pro převod typů
    String e(b),f=d; // dva zápisy pro kopykonstruktor,
    // = v definici je kopykonstruktor
    String g="akdf ",h=5; // tento zápis volá postupně konverzní
    // konstruktory a potom kopykonstruktory
    // tento zápis je tedy podstatně náročnější na čas a tudíž je
    // to špatně
    // = tedy vždy vynutí kopykonstruktor ( a proto se pravá
    // strana snaží převést na stejný typ)
    // volání kopykonstruktoru je přebytečné
    // na což např. překladač přijde a vynechá ho (zoptimalizuje)
    // na skutečnou činnost však upozorní při použití
    // explicit (viz. výše), které
    // zakáže tuto konstrukci

    a.PriradSpojeni(b,d);
    e.Prirad(a.Soucet(c)); // soucet vytvoří tmp návratovou
        // hodnotu, která se po použití zruší
    d.PriradSpojeni(5," askdlf "); // pomocí konverzních
        // konstruktů se vytvoří tmp objekty pro volání

    return 1;
} // zde se provedou destruktory lokálních objektů

```

KO: co jsou konstruktory a destruktory? K čemu se používají a kdy se volají? Jaké konstruktory znáte?

Příklad 3.2.6a: Navrhněte konstruktory a destruktory pro třídu “bod v prostoru”.

Konstruktory a destruktory jsou základním principem kontroly vzniku a zániku objektů a tedy i kontroly získávání a vrácení použitých zdrojů.

Konstruktor jako první volaná metoda slouží k inicializaci objektu, destruktory jako poslední volaná metoda slouží k tomu, aby se uložila data, nebo vrátili naalokované zdroje. Konstruktor má stejné jméno jako třída, destruktory má stejné jméno jako je jméno třídy s tím, že je mu předřazen znak ~.

Destruktory je pouze jeden bez parametrů, konstruktory může být celá řada s rozlišením jaké platí pro přetížené funkce. Existují speciální konstruktory: implicitní, který nemá parametry, kopykonstruktor, který tvoří objekt na základě objektu stejného typu a konstruktor konverzní, který má jeden parametr a slouží tedy k převodu jednoho typu na druhý.

Při práci s konstruktory je nutné zvýšit pozornost při práci s prvky třídy typu ukazatel, kdy by mohlo dojít k vytvoření mělké kopie dat a tím i k problémům při rušení o objektů.

3.3.7 Hlavičkové soubory a třída, příslušnost ke třídě

Cílem je ukázat jak v souvislosti se třídami vytvářet hlavičkové a zdrojové soubory. Části návrhu třídy, které popisují data a metody se umísťují do hlavičkového souboru. Vlastní těla metod se uvádějí do zdrojového souboru. Definice třídy obsahuje metody a data a práva přístupu. Deklarace je uvedena v hlavičkovém souboru. Definice metod potom ve zdrojovém textu. Deklarované data a metody vznikají teprve při definici objektu dané třídy a jejich použití.

Zdrojový kód pro třídu není většinou reprezentován pouze jedním souborem, ale může jich být několik. Je možné vytvořit soubor hlavičkový, který prezentuje rozhraní třídy, dále soubor zdrojový, ve kterém je zdrojový kód k metodám volaným funkčním voláním a dále je možné vytvořit soubor inline metod, které slouží jako předpis pro rozvinutí metod do kódu (tato část je rozepsána v 3.3.8 a úzce souvisí s touto kapitolou). Tím se zpřehlední práce, zjednoduší čtení hlaviček a zároveň slouží jako předpis pro generování kódu viz. 3.2.11 inline metody.

Hlavním souborem je soubor hlavičkový. V tomto souboru se nachází vlastní popis třídy, jako je její jméno, definice členských dat a metod s plným prototypem a přístupovými právy. Dále jsou zde definice potřebné pro činnost třídy.

Do hlavičkového souboru tedy patří část - class jméno {parametry};

Vlastní těla (delších) metod se potom uvádějí do souboru zdrojového kódu. Na začátku souboru je vložen příslušný hlavičkový soubor (pomocí include), s prototypem třídy (aby metody věděly s čím pracují a zároveň se kontrolují prototypy funkcí se skutečnou implementací).

Názvy metod v definici v .cpp je potom potřebné rozšířit o název třídy ke které patří. Tj. tzv. příslušnost ke třídě. Operátor příslušnosti je :: a definice metody ve zdrojové části vypadá:

návratová_hodnota jméno_třídy::jméno_funkce (parametry) {tělo}

Pozn.: v následující kapitole je popsáno využití inline metod, pro která platí odlišná pravidla z hlediska umístění (jelikož se jedná pouze o předpis, nejsou umístěny ve zdrojovém, ale v hlavičkovém souboru)

Pozn.: do zdrojového souboru se také umísťují definice statických proměnných.

Pozn.: Při definici v hlavičkovém souboru není u členských dat a metod nutné uvádět část Jméno_třídy::, protože je jasné, že funkce patří k právě definované třídě.

Pozn.: Z důvodu možného vícenásobného načítání hlavičkových souborů je dobré vložit text hlavičkového souboru do sekce preprocesorových direktiv, které zajistí, že se souborem bude procházet pouze jednou v .h souboru.

```
#ifndef jmeno_h_defined
#define jmeno_h_defined
class jmeno_třídy {parametry};
#endif
```

zdrojová část může vypadat následovně

```
double komplex::real_part()      při definici metody
```

Pozn.: máme-li ve třídě nadefinovanou metodu, jejíž prototyp se kryje s jinou metodou, kterou chceme volat z této funkce, je nutné uvést za pomoci operátoru příslušnosti plný název metody, protože jinak by došlo k rekurentnímu volání (protože metoda třídy je ve vyhodnocování volání upřednostněna)

```
int komplex::line(int i, int j) {
```

```
    ::line(i,j)} volána funkce z grafické knihovny uvnitř metody třídy, kde je  
název překryt - bez použití operátoru “ :: ” by došlo k rekurzi
```

KO: co se u třídy píše do souboru hlavičkového a co do zdrojového a proč?

Příklad 3.2.7a: Rozdělte metody a data třídy “ bod v prostoru” do hlavičkového a zdrojového souboru.

3.3.8 Inline metody

Cílem je ukázat možnost zrychlení provádění metod třídy pomocí inline metod. Inline metody nejsou volány pomocí funkčního volání ale jsou rozvinuty do kódu. Z tohoto důvodu jsou vhodné pro jednoduché, krátké metody. Inline metody jsou umístěny do hlavičkového souboru (někdo doporučuje vytvoření inline souboru, ve kterém jsou umístěny inline metody a tento soubor je includován do hlavičkového souboru – tento přístup má zvýšit přehlednost).

Test: co je to inline funkce a k čemu slouží?

Pozn.: vyžaduje inline funkce

Pro jednoduché metody, kde je jejich vlastní činnost zanedbatelná vůči režii spojené s funkčním voláním, je možné použít jejich rozvinutí do kódu. K tomu se používají inline metody.

U třídy je tedy možné určit, které metody budou inline (vložené, rozbalené do kódu) a které budou volány jako funkce. Za inline metody jsou automaticky považovány ty, které mají dělo v definici třídy. Pokud zde tělo není uvedeno, potom jsou to normální funkce – zdrojový kód je definován v .cpp souboru. Jelikož uvádění těl v definici třídy snižuje přehlednost definice, je možné těla pro inline metody psát i vně třídy. V tomto případě ale musí být v definici i deklaraci označena taková metoda klíčovým slovem inline. Těla inline metod jsou ovšem v tomto případě uvedena v hlavičkovém souboru – je to pouze předpis, metoda nevytváří kód (dokud není použita). Část definice těl metod třídy, kterou jsme umístili do hlavičkového souboru není z hlediska metod vlastně kódem, ale předpisem pro rozvinutí

Rozsáhlé funkce, nebo funkce s cykly nemusí být jako inline přeloženy. (Některé překladače zváží vhodnost použití inline (berou ho jako doporučení).

Následující tabulka ukazuje možné rozložení mezi hlavičkový (a inline soubor) a zdrojový soubor a jak je interpretováno

.h (či inl) soubor	.cpp soubor	popis
{ Metoda() {} }	-	inline metoda, protože tělo je definováno v hlavičce. Nelze programátorskými prostředky zajistit aby nebyla inline, může ji však překladač přeložit jako neinline (v případě, že bude “složitá”)
{ metoda(); }	metoda::metoda(){} 	není inline. Tělo je definováno mimo hlavičku a v hlavičce není uvedeno inline
{ inline metoda(); } inline metoda:: metoda(){} 	-	je inline “z donucení” pomocí klíčového slova inline v definici třídy i při definici metody
{ inline metoda(); } metoda::metoda() {} 	-	Pokud překladač vztáhne inline na metodu uvedenou dále, pak v pořádku.
{ Metoda (); }	inline metoda::metoda(){} 	špatně (mělo by dát chybu) – mohlo by vést až ke vzniku dvou interpretací – někde inline a někde funkční volání
{ Inline metoda() }	metoda::metoda(){} 	špatně – mohlo by vést až ke vzniku dvou interpretací – někde inline a někde funkční volání; i když je vlastní kód metody pod hlavičkou takže se o inline ví, nedochází k chybě

Doporučuje se psát inline těla metod vně – zřehlední hlavičku, skryje implementaci.

Pozn.: pokud se překladači inline metoda jeví příliš komplikovaná, může se rozhodnout, že ji jako inline nepřeloží.

Pozn.: Při ladění se volají inline metody jako funkce, až po vypnutí ladících konstant je přeloženo jako inline. Inline se v debug modu překladače překládá jako metoda s funkčním voláním.

Příklad 3.2.7 třída a hlavičkové soubory

Rozdělte příklady z minulých kapitol na části, které je možné (a nutné) mít v hlavičkovém souboru, a na části (které musí být) ve zdrojových textech.

```
//===== komplex2207.h - hlavička třídy =====
// trasujte a divejte se kudyma to chodí, tj. zobrazte
// *this, ...
// objekty muzete rozlisit pomoci indexu

// ochrana proti vícenásobnému načtení hlavičky při
// include téhož v rámci
// různých hlaviček v modulu .cpp (míněno v jednom)
#ifndef KOMPLEX_H
#define KOMPLEX_H

#include <math.h>

// hlavička struktury Komplex je docela přehledná,
// pouze metoda konstruktoru
// z řetězce je delší a tak by neměla být inline, proto ji
// dáme do .cpp jako
// ne-inline. Trochu delší (z hlediska textu) jsou i další dva
// konstruktory
// takže je také posunem mimo

struct Komplex {
    enum TKomplexType {eSlozky, eUhel};

    static int Poradi;
    static int Aktivnich;

    double Re,Im;
    int Index;

    Komplex(void) {Re=Im=0;Index = Poradi;Poradi++;Aktivnich++; }
// inline metoda - definice těla v hlavičce

    inline Komplex(double re,double im=0, TKomplexType kt =
eSlozky); // inline metoda - klíčové slovo
    Komplex(const char *txt); // "neinline" - funkční volání
    inline Komplex(const Komplex &p);
    ~Komplex(void) {Aktivnich--;}

    void PriradSoucet(Komplex p1,Komplex p2)
        {Re=p1.Re+p2.Re;Im=p1.Re+p2.Im;} // inline

Komplex Soucet(const Komplex & p)
    {Komplex pom(Re+p.Re,Im+p.Im);return pom;} //inline
Komplex& Prirad(Komplex const &p)
    {Re=p.Re;Im=p.Im;return *this;} //inline
// složitější metody, která by se podle mínění překladače
```

```

// nemusela přeložit
// jako inline - for sám o sobě zabere tolik času a kódu,
// že se o úspoře
// času či kódu dá s úspěchem pochybovat.
double faktorial(int d)
    {double i,p=1; for (i=1;i<d;i++) p*=i;
    return p; } // překladač díky cyklu může přeložit
// pomocí funkčního volání (tj. jako kód)
};

// inline metody nepatří do kódu, protože se jedná pouze o
// předpis
// proto následují zde za hlavičkou
// implicitní parametr se definuje pouze jednou - v hlavičce

inline Komplex::Komplex(double re,double im, TKomplexType kt )
{
    // přehledný zápis kódu
    Re=re;
    Im=im;
    Index=Poradi;
    Poradi++;
    Aktivnich++;
    if (kt == eUhel)
        {Re=re*cos(im);Im = Re*sin(im);}
}

// pokud chybí inline zde není to až tak velká chyba, táhne
// se to zhora
// a překladač to (snad) ví
Komplex::Komplex(const Komplex &p)
{
    Re=p.Re;
    Im=p.Im;
    Index=Poradi;
    Poradi++;
    Aktivnich++;
}

// bla bla bla odzávorkováním vznikne chyba v dalším souboru
// pozor !!! pokud v tomto místě udělám chybu - např napíšu
// blbost či (v případě, že nejsou uvedeny inline metody)
// zapomenu středník, objeví se to jako
// chyba až v následujícím modulu

// konec define pro přeskočení definice třídy při vícenásobném
// načítání

#endif

```

```
//===== komplex2207.cpp - zdrojový kód třídy =====
// trasujte a divejte se kudyma to chodi, tj. zobrazte
// *this, ...
// objekty muzete rozlisit pomoci indexu

// bez prototypů je to k ničemu takže definici třídy
#include "komplex2207.h"

// čítače pořadí vzniku a aktivních prvků
// musí být v části zdrojového kódu aby při vícenásobném
// načtení hlavičky
// (míněno ve více modulech .cpp)

int Komplex::Poradi=0;
int Komplex::Aktivnich=0;

// toto je jediná " správná " metoda s funkčním voláním
Komplex::Komplex(const char *txt)
{
    /* vlastni alg */;
    Re=Im=0;
    Index = Poradi;
    Poradi++;
    Aktivnich++;
}

```

```
//===== komplex2207p.cpp - kód aplikace =====

#include "komplex2207.h"

char str1[]="(73.1,24.5)";
char str2[]="23+34.2i";

int main ()
{
    Komplex a;
    Komplex b(5),c(4,7);
    Komplex d(str1),e(str2);
    Komplex f=c,g(c);
    Komplex h(12,35*3.1415/180.,Komplex::eUhel);
    Komplex::TKomplexType typ = Komplex:: eUhel;
    Komplex i(10,128*3.1415/180,typ);

    d.PriradSoucet(b,c);
    e.Prirad(d.Prirad(c));
    d.PriradSoucet(5,c);
    d.PriradSoucet(Komplex(5),c);
}

```

```

    return 0;
}

//===== string2207.cpp - hlavička třídy =====
// trasujte a divejte se co se deje

#ifndef STRING_H_DEF
#define STRING_H_DEF

#include <stdio.h>
#include <string.h>

// hlavička této složitější třídy je s těly metod nepřehledná a
složitá
// některé metody nemají jako inline smysl
// ty delší, které by mohly být inline přesuneme definici za
// tělo definice třídy, ty složitější dáme do zdrojové části a
// tudíž budou volány "funkčně"

class String {
    static int Poradi;
    static int Aktivnich;
    char *txt;
    int delka;
    int Index;

public:
    inline String(void) ;
    String( const String &b );
    String(double d);
    String( const char *t );

    // destruktork bude rozvinutý do kódu
    ~String(void)
    {Aktivnich--;if (txt != NULL) delete [ ] txt;
     txt = NULL;}

    String Soucet (String & s) ;
    String& Prirad (String & s) ;
    void PriradSpojeni(String & s1,String & s2);
};

// metody rozvinuté do kódu
inline String::String(void)
{
    Index = Poradi;
    Poradi++;
    Aktivnich++;
    txt = NULL;
    delka = 0;
}

```

```

// konec definice hlavicky
#endif

//===================================================== string2207.cpp - zdrojový kód třídy
// trasujte a divejte se co se deje

// musíme "oznámit" o čem je třída
#include "string2207.h"

// definice statických proměnných musí být v oblasti kódu
int String::Poradi = 0;
int String::Aktivnich=0;

// metody s funkčním voláním
String::String( const String &b )
{
    Index = Poradi;
    Poradi++;
    Aktivnich++;
    delka = b.delka;

    if (delka > 0)
    {
        txt = (char *)new char [ delka ] ;
        int i;
        for (i = 0; i <delka; i++)
            txt [ i ] = b.txt [ i ];
    }
    else
        txt = NULL;
}

String::String(double d)
{
    Index = Poradi;
    Poradi++;Aktivnich++;
    char tmp[50] = "";
    sprintf ( tmp, " %lf ", d );
    delka = strlen ( tmp ) + 1;
    txt = (char *) new char [ delka ] ;
    int i;
    for (i = 0; i <delka; i++)
        txt[i] = tmp[ i ];
    txt [ delka -1] ='\0';
}

String::String( const char *t )
{
    Index = Poradi;
    Poradi++;

```

```

Aktivnich++;
delka = strlen ( t ) + 1;
if (delka > 1)
{
    txt = new (char [ delka ]);
    int i;
    for (i = 0; i < delka-1; i++)
        txt [ i ] = t [ i ];txt [ delka ] ='\0';
}
else
{
    delka = 0;
    txt = NULL;
}
}

String String::Soucet(String & s)
{
    String a ;
    int i , j ;
    a.delka = delka + s.delka - 1;
    if ( a. delka == 0 )
        a.txt = NULL;
    else
    {
        a.txt = new char [ a.delka ] ;
        for ( i = 0 ; i < delka-1 ; i ++ )
            a. txt [ i ] = txt [ i ] ;
        for ( j = 0 ; j < s.delka ; j ++ , i ++ )
            a. txt [ i ] = s.txt [ j ] ;
        a.txt [ a.delka -1] = '\0' ;
    }
    return a;
}

String& String::Prirad (String &s)
{
    if (&s == this) return (*this);
    if (txt) delete[] txt;
    delka = s.delka;
    if (delka > 0)
    {
        txt = (char *)new char [ delka ] ;
        int i;
        for (i = 0; i <delka; i++)
            txt [ i ] = s.txt [ i ];
    }
    else
        txt = NULL;
    return *this;
}

```

```

}

void String::PriradSpojeni(String &s1,String & s2)
{
    char *pom = (char *) new char[s1.delka+s2.delka+1];
    int i , j ;
    delka = s1.delka + s2.delka - 1;

    if ( delka == 0 )
        pom = NULL;
    else
    {
        for ( i = 0 ; i < s1.delka-1 ; i ++ )
            pom [ i ] = s1.txt [ i ] ;
        for ( j = 0 ; j < s2.delka ; j ++ , i ++ )
            pom [ i ] = s2.txt [ j ] ;
        pom [ i ] = '\\0' ;
        if (txt)
            delete[] txt;
        txt = pom;
    }
}

```

```

//===== string2207p.cpp =====
// trasujte a divejte se co se deje

#include "string2207.h"

int main() {
    String a;
    String b(8.3),c(3),d("sadfl");
    String e(b),f=d;
    String g="akdf ",h=5;

    a.PriradSpojeni(b,d);
    e.Prirad(a.Soucet(c));
    d.PriradSpojeni(5," askdlf ");

    return 1;
} // zde se provedou destruktory lokálních objektů

```

KO: jak poznáme která metoda je inline? Co to znamená?

Příklad 3.2.8a: stanovte zda bylo rozhodnutí z minulé kapitoly (o rozdělení metod) vhodné i z hlediska inline metod.

Do hlavičkového souboru patří “prototyp” třídy nebo struktury, který říká, které data a metody struktura obsahuje a jaké mají přístupová práva. Do zdrojových textů patří metody, které jsou složitější – tvoří kód.

Jednoduché metody je možné zvolit jako inline, které se rozvinou do kódu. Tyto je možné určit tak, že jejich tělo napíšeme do definice třídy, nebo když v definici třídy použijeme u metody označení inline. Inline metody neuvedené přímo v definici třídy se uvádějí za definicí třídy, nebo do souboru, který je do hlavičkového souboru naincludován.

3.3.9 Deklarace a definice třídy, objektů a metod

Cílem je shrnutí a rozšíření vlastností tříd a objektů z minulých kapitol.

Pro vytvořené objekty dané třídy platí stejná pravidla lokálnosti, globálnosti a viditelnosti jako pro jiné (standardní) typy. Pro třídu lze vytvořit :

Třída `a`, `*b`, `&c=a`, `d[10]`; - proměnná dané třídy, ukazatel na proměnnou dané třídy, reference na danou třídu a pole prvků dané třídy – platí pro ně stejná pravidla jako pro standardní typy (např. pro `int`).

Pro prvky definované v dané třídě platí pro přístup z venku pravidla přístupu podle přístupových práv. Z objektu dané třídy jsou vždy viditelné. Na prvky aktuální instance se přistupuje pomocí `this->` a nebo pouze jménem proměnné. Pokud je objekt vytvořen jako proměnná (či reference), přistupuje se k jeho prvkům pomocí `”.”`. Pokud máme ukazatel, přistupujeme k prvkům pomocí `”->”`. V tomto přístupu jsou si data i metody rovny.

Správný zápis třídy je deklarace v hlavičkovém souboru a zdrojový kód zvlášť. Ve zdrojovém kódu by měly být paměťově a časově náročné metody – ty jsou volány funkčním voláním. Metody s tělem uvedeným v definici třídy jsou inline. Delší tělo je možné zapsat mimo hlavičkový soubor (např. do `.inl` souboru) pomocí klíčového slova `inline`.

Zdrojový kód pro funkční volání vzniká z `.cpp` souborů. Inline metody jsou rozvinuty do kódu v místě uvedení.

Ukazatel na objekt můžeme získat dvěma způsoby – získat adresu již existujícího objektu, nebo použít dynamického vzniku alokování příslušné paměti – v tomto případě je nutné takto vzniklé objekty odalokovat. Každý ukazatel by měl být před použitím inicializován jedním z uvedených způsobů. U lokálních proměnných se pracuje se zásobníkem, u dynamických s (obecnou) pamětí.

Při dynamickém vytváření máme možnost vytvořit jeden nebo více prvků (pole). Z důvodu volání konstruktorů na vznikající objekty tříd se používají dva typy alokace: pomocí `new` a `new[]`, které spolu s vytvořením paměťového místa zajistí spuštění konstruktorů, ve druhé variantě pro každý prvek alokovaného pole.

Ke každému `new` by mělo být voláno `delete`, které vrátí paměť potřebnou pro uložení prvku. Zde je nutné si uvědomit, jak byl daný ukazatel vytvořen, protože pro ukazatele, inicializované na základě adresy existujícího objektu, `delete` volat nesmíme. `delete []` opět volá destruktory pro každý prvek rušeného pole.

new tedy obecně vyhradí paměť a zavolá konstruktor, delete zavolá destruktory a uvolní paměť.

Výhodou využití ukazatelů (oproti statickému poli) je vznik přesného typu objektu a možnost volby počtu objektů (dynamicky) za chodu programu.

Pozn.: u pole se volá implicitní implicitní,

Pokud má třída obsahovat prvek stejné třídy, nebo dvě třídy mají obsahovat prvky "do kříže", potom je problém při překladu. Problém je v tom, že v daném okamžiku není známa velikost objektu daného typu. Řešením je použití ukazatelů, pro které stačí i prostá deklarace názvu třídy `class Jméno_třída`; a vlastní definici třídy doplnit později.

Pokud je při vzniku objektu použit modifikátor `const`, potom se nejprve zavolá konstruktor, který může s objektem pracovat a teprve dále není možné objekt měnit

`const Třída X(1,a);` vytvoří objekt pomocí konstruktoru a dále už není možné ho měnit

Příklad 3.2.9a: Na základě úvah z minulých kapitol a současných znalostí napište (a použijte) třídu "T_BOD_3D"

3.3.10 Operátory přístupu k prvkům *. a ->.

Cílem je představit možnost přístupu k prvku daného typu uvnitř třídy tím, že si na něj vezmu referenci. Určím, se kterým prvkem v rámci třídy se bude pracovat (je možné brát jako offset). Při použití tohoto "offsetu" s konkrétním objektem dané třídy potom budu pracovat se zvoleným prvkem. Je to obdoba ukazatele na funkce pro metody a data.

Jazyk C++ zavádí nové operátory pro přístup k prvkům třídy. Jedná se o princip, kdy v rámci třídy můžu získat referenci na prvek daného typu. (Lze si představit tak, že získáme offset od počátku objektu, na místo, na kterém leží daná proměnná. Při "použití" se tento offset přičte k počáteční adrese objektu a z danou paměť se pracuje). Pokud tuto referenci použiji na konkrétní objekt dané třídy, přistoupím k prvku, nebo metodě, na kterou reference ukazovala při získání. Operátory pro přístup k určitému členu struktury nebo třídy jsou využívány např. při průchodu polem a prací pouze s jednou proměnnou struktury.

"*." (hvězdička, tečka) dereference ukazatele na člen třídy pro objekt

"->" dereference ukazatele na člen třídy pro ukazatel na objekt

Z principu plyne, že tento princip přístupu je nevhodný pro inline metody. Dále plyne, že se jedná o relativní záležitost v rámci třídy a tedy získané reference nejdou přetypovat, a to ani na `void*`.

```
int (T::*p1) (void); /* definice typu proměnné bez
                    inicializace, která říká na co
                    odkazujeme - p1 je
                    ukazatel do třídy T, na funkci
                    s parametrem void vracejícím int */
p1=&T::f1; /* vyberu konkrétní funkci ze třídy T, která má
           daný prototyp - p1 tedy ve třídě T ukazuje na metodu
           f1 - int T::f1(void) */
```

```

float T::*p2; /* p2 je ukazatel do třídy T na proměnnou
               float (tj. ukazatel na float patřící do třídy T)
               */
p2=&T::f2;    // vyberu konkrétní float - proměnnou f2
T*ut, *ut2;  // definice objektů třídy T
ut->*p2=3.14; /* přístup k vybrané proměnné (f2) v proměnné
               (ut) třídy T */
ut2->*p2=4;   /* přístup k vybrané proměnné (f2) v proměnné
               (ut2) třídy T */
p2=&T::fff;   // vyberu konkrétní float - proměnnou fff
ut2->*p2=4;   /* přístup k vybrané proměnné (fff) v proměnné
               (ut2) třídy T */
(ut->*p1)();   /* volání vybrané metody (f1) v proměnné (ut)
               třídy T- závorky pro prioritu */

```

Při použití operátoru `*`, a `->` je prvním operandem vlastní objekt třídy `T`, ze kterého chceme vybraný prvek použít. Druhým operandem je odkaz na vybraný prvek. Odkaz se získá z popisu třídy a není pro něj potřeba konkrétní objekt. Ten je nutný až pro přístup. Tyto operátory zjednodušují práci s určitým prvkem (data či metoda) třídy či struktury.

3.3.11 Deklarace třídy uvnitř jiné třídy

Cílem je prezentovat možnost ukrytí jména třídy uvnitř jiné třídy. Slouží k eliminaci kolizí jmen tříd, popř. k zabránění manipulace s interním objektem. Je to ekvivalentní samostatné deklaraci, výhodou je “skrytí” druhé třídy.

```

class A {          // první třída
    class B;       // definice vnořené třídy
    .....
}

class A::B { /* vlastní definice vnořené tříd - má tedy jména
             A::B - třída B ve třídě A */
    .....
}

```

Jméno vnořené třídy je lokální pro třídu nadřazenou. Vztahy mezi těmito třídami jsou stejné, jako kdyby byly definovány zvlášť. Jméno vnořené třídy se deklaruje uvnitř, ale samotná struktura vně. Pokud není vnořená struktura uvedena v sekci `private`, lze použít i vně, kde se objekt definuje `A::B x;`. Vnořené třídy se používají v případě pomocných objektů, které chceme skrýt před uživatelem (je výhodné řešit pomocí vlastností tříd ale nevhodné či “nebezpečné” pro samostatné užití). Vnořený objekt se využívá pro specializované objekty, které se používají pouze pro danou třídu a je rozumné je skrýt (co se týká názvu i funkčnosti).

Pozn.: tato vlastnost, je už i u `struct` v C. Tam je ovšem vnořená struktura vidět stejně jako by byla vně. Nezáleží tedy na tom zda je uvnitř či mimo strukturu. U `c++` je nutno použít pro přístup celou cestu jak je uvedeno výše.

Pozn.: práce s proměnnou se chová podobně jako enum v příkladech komplex 2206

KO: Jaké má vlastnosti a jak se pracuje s vnořenou třídou.

Je-li potřebné řešit část třídy objektově, ale zamezit samostatné použití nebo kolizi jmen, je možné vytvořit třídu jako součást jiné třídy.

3.3.12 Modifikátor const u parametrů a metod třídy

Cílem je popsat princip (a problémy při) volání metod na konstantní objekty. Při předávání objektů do metod jsou parametry modifikovány modifikátorem const, pokud na takovouto proměnnou voláme metodu, mělo by se zajistit že ji tato nezmění. Jelikož zjišťovat zda tomu tak je překladačem by bylo náročné, je toto určení na programátorovi. To že metoda nemění objekt je určeno v definici metody tím, že se za prototyp přidá klíčové slovo const.

Test: jaký typ je určen pro logické proměnné v C++ a jakých hodnot nabývá?

Pozn.: vyžaduje bool

Pro předávání parametrů do funkcí a metod se pro třídy používá přednostně reference či ukazatel. To ovšem může vést k nechtěné změně těchto tříd. Proto je možné takto předávané objekty třídy před náhodnou změnou ochránit pomocí klíčového slova const (3.2.8). Pokud se přistupuje k datům, může překladač snadno určit, zda ke změně došlo nebo ne. U volání metod na konstantní objekt už není situace tak jasná, protože zjistit zda se v ní objekt mění by mohlo být dosti pracné (a pro člověka (i překladač), který nemá zdrojový kód ale pouze lib či obj prakticky nemožné). Proto je nutné dát překladači vědět, zda je možné metodu na konstantní objekt použít (tj. že metoda objekt nemění). To, že metoda nemění parametry objektu se vyznačí v hlavičce metody uvedením const za prototyp

```
float f1(void) const {tělo metody}.
```

Pozn.: rozlišujeme tedy tímto metody na přístupové a změnové.

Např. z volání `h=o.f1()` není jasné, zda se pouze vrátí hodnota zapsaná do `h`, nebo dochází i ke změně obsahu objektu `o`. Z dané definice je potom zřejmé, že objekt, který metodu vyvolal, touto není změněn, a daná funkce může být konstantním objektem volána. Překladač tedy pro konstantní objekt volaný nekonstantní metodou může zahlásit chybu.

Tyto mechanismy vedou i k tomu, že pokud neuvedu do hlavičky k definici předávaného parametru `const`, není možno při volání metody na tomto místě použít konstantní objekt (což je omezení), protože by se jednalo o přetypování konstantního objektu na nekonstantní což je špatně. Proto je lépe uvádět `const` u parametrů metod vždy, kdy se objekt v metodě nemění.

Pozn.: tím, že nebudeme používat `const` parametry u definovaných metod a předávaných parametrů se těmito problémům nevyhneme, protože se i tak objeví - např. operátory (`=`, `...` u nových překladačů) vyžadují aby parametry byly `const` (a kvůli nim to pak musíme vše stejně udělat).

Pozn.: Někdy se doporučuje (vizuálně – blokově) oddělit `const` a `nonconst` metody.

Pozn.: `const` se prakticky použije k implicitnímu parametru (`this`).

Příklad 3.2.12 const u parametrů a metod třídy

Upravte příklady z minulých kapitol o modifikátor `const` u předávaných parametrů a upravte též hlavičky metod.

```
//===== komplex2212.h - hlavička třídy =====
// trasujte a divejte se kudyma to chodí, tj. zobrazte
// *this, ...
// objekty muzete rozlisit pomoci indexu

#ifndef KOMPLEX_H
#define KOMPLEX_H

#include <math.h>

struct Komplex {
    enum TkomplexType {eSlozky, eUhel};

    static int Poradi;
    static int Aktivnich;

    double Re,Im;
    int Index;

    Komplex(void)
        {Re=Im=0;Index = Poradi;Poradi++;Aktivnich++; }
    inline Komplex(double re,double im=0, TkomplexType kt =
eSlozky);
    Komplex(const char *txt);
    inline Komplex(const Komplex &p); /* tohle by mělo být požadováno
již z normy */

    ~Komplex(void) {Aktivnich--;}

    void PriradSoucet(Komplex const &p1,Komplex const &p2)
        {Re=p1.Re+p2.Re;Im=p1.Re+p2.Im;}

    Komplex Soucet(const Komplex &p)
        {Komplex pom(Re+p.Re,Im+p.Im);return pom;}

    Komplex& Prirad(Komplex const &p)
        {Re=p.Re;Im=p.Im;return *this;}

    double faktorial(int d)
        {double i,p=1; for (i=1;i<d;i++) p*=i; return p; }

    double Amplituda(void) const { // Re = 4;
        return sqrt(Re*Re + Im *Im);}

    // pokud nebude za deklarací funkce const, potom volání této
```

```

// funkce na const
// objekty vyvolá Warning či Error,
// podle nastavení překladače
// zároveň není-li const, umožní překladač i změnu prvku, o
// čemž volající funkce neví. Při uvedení const již
// zahlásí chybu při pokusu ho změnit
// srovnání velikosti komplexních čísel podle vzdálenosti
// od počátku - amplitudy
bool JeMensi(Komplex const &p)
    {return Amplituda() < p.Amplituda();}

// obdoba. Pouze funce není inline
double Amp(void) const;
bool JeVetsi(Komplex const &p) {return Amp() > p.Amp();}
};

inline Komplex::Komplex(double re,double im, TKomplexType kt )
{
    Re=re;
    Im=im;
    Index=Poradi;
    Poradi++;
    Aktivnich++;
    if (kt == eUhel)
        {Re=re*cos(im);Im = Re*sin(im);}
}

Komplex::Komplex(const Komplex &p)
{
    Re=p.Re;
    Im=p.Im;
    Index=Poradi;
    Poradi++;
    Aktivnich++;
}

#endif

```

```

//===== komplex2212.cpp - zdrojový kód třídy =====
// trasujte a divejte se kudyma to chodi, tj. zobrazte
// *this, ...
// objekty muzete rozlisit pomoci indexu

#include "komplex2212.h"

int Komplex::Poradi=0;
int Komplex::Aktivnich=0;

Komplex::Komplex(const char *txt)
{

```

```

    /* vlastní alg */;
    Re=Im=0;
    Index = Poradi;
    Poradi++;
    Aktivnich++;
}

double Komplex::Amp(void)const
// bez uvedení const se to překladači dokonce může jevit
// jako jiná funkce
// takže uvedení zde i v hlavičce je nutné
{
    // Re = 4; // s const opět nelze

return sqrt(Re*Re + Im *Im);
}

//===== komplex2212p.cpp - kód aplikace =====

#include "komplex2212.h"

char str1[]="(73.1,24.5)";
char str2[]="23+34.2i";

int main ()
{
    Komplex a;
    Komplex b(5),c(4,7);
    Komplex d(str1),e(str2);
    Komplex f=c,g(c);
    Komplex h(12,35*3.1415/180.,Komplex::eUhel);
    Komplex::TKomplexType typ = Komplex:: eUhel;
    Komplex i(10,128*3.1415/180,typ);

    d.PriradSoucet(b,c);
    e.Prirad(d.Prirad(c));

    d.PriradSoucet(5,c);
    d.PriradSoucet(Komplex(5),c);
    return 0;
}

//===== string2212.cpp - hlavička třídy =====
// trasujte a divejte se co se deje

#ifndef STRING_H_DEF
#define STRING_H_DEF

#include <stdio.h>
#include <string.h>

```

```

class String {
    static int Poradi;
    static int Aktivnich;
    char *txt;
    int delka;
    int Index;

public:
    inline String(void) ;
    String( const String &b );
    String(double d);
    String( const char *t );

    ~String(void)
        {Aktivnich--;if (txt != NULL) delete [ ] txt;
         txt = NULL;}

    String Soucet (String const & s) ;
    String& Prirad (String const & s) ;

    void      PriradSpojeni(String const & s1,String const& s2);

    int SrovnejDelky(String const & s) const
        {if (delka == s.delka) return 0;
         if (delka < s.delka) return -1; else return 1;}

    int StrCmp(String const &s)
        // vynechám - li const, potom se při volání const objektem
        // vytvoří (s jako) tmp objekt, který se může měnit
        {return SrovnejDelky(s);}

    bool JeMensi(String const &s)
        {if (StrCmp(s) < 0) return true;else return false;}

    bool JeVetsi(String const &s)
        {if (StrCmp(s) > 0) return true;else return false;}

    bool JeRovno(String const &s)
        {if (StrCmp(s) == 0) return true;else return false;}
};

inline String::String(void)
{
    Index = Poradi;
    Poradi++;
    Aktivnich++;
    txt = NULL;
    delka = 0;
}

```

```

#endif

//===== string2212.cpp - zdrojový kód třídy =====
// trasujte a divejte se co se deje

#include "string2212.h"

int String::Poradi = 0;
int String::Aktivnich=0;

String::String( const String &b )
{
    Index = Poradi;
    Poradi++;
    Aktivnich++;
    delka = b.delka;

    if (delka > 0)
    {
        txt = (char *)new char [ delka ] ;
        int i;
        for (i = 0; i <delka; i++)
            txt [ i ] = b.txt [ i ];
    }
    else
        txt = NULL;
}

String::String(double d)
{
    Index = Poradi;
    Poradi++;Aktivnich++;
    char tmp[50] = "";

    sprintf ( tmp, " %lf ", d );
    delka = strlen ( tmp ) + 1;
    txt = (char *) new char [ delka ] ;
    int i;
    for (i = 0; i <delka-1; i++)
        txt[i] = tmp[ i ];
    txt [ delka -1] ='\0';
}

String::String( const char *t )
{
    Index = Poradi;
    Poradi++;
    Aktivnich++;
    delka = strlen ( t ) + 1;
}

```



```

if (delka > 1)
{
    txt = new (char [ delka ]);
    int i;
    for (i = 0; i < delka; i++)
        txt [ i ] = t [ i ];txt [ delka -1] ='\0';
}
else
{
    delka = 0;
    txt = NULL;
}
}

```

String String::Soucet(String const & s)

```

{
    String a ;
    int i , j ;
    a.delka = delka + s.delka - 1;

    if ( a. delka == 0 )
        a.txt = NULL;
    else
    {
        a.txt = new char [ a.delka ] ;
        for ( i = 0 ; i < delka-1 ; i ++ )
            a. txt [ i ] = txt [ i ] ;
        for ( j = 0 ; j < s.delka ; j ++ , i ++ )
            a. txt [ i ] = s.txt [ j ] ;
        a.txt [ a.delka-1 ] = '\0' ;
    }
    return a;
}

```

String& String::Prirad (String const &s)

```

{
    if (&s == this) return (*this);
    if (txt) delete[] txt;

    delka = s.delka;

    if (delka > 0)
    {
        txt = (char *)new char [ delka ] ;
        int i;

        for (i = 0; i <delka; i++)
            txt [ i ] = s.txt [ i ];
    }
}

```

```

    else
        txt = NULL;
    return *this;
}

void String::PriradSpojeni(String const &s1,String const & s2)
{
    char *pom = (char *) new char[s1.delka+s2.delka+1];
    int i , j ;

    delka = s1.delka + s2.delka - 1;

    if ( delka == 0 )
        pom = NULL;
    else
    {
        for ( i = 0 ; i < s1.delka-1 ; i ++ )
            pom [ i ] = s1.txt [ i ] ;
        for ( j = 0 ; j < s2.delka ; j ++ , i ++ )
            pom [ i ] = s2.txt [ j ] ;
        pom [ i ] = '\0' ;
        if (txt) delete[] txt;
        txt = pom;
    }
}

```

```

//===== string2212p.cpp =====
// trasujte a divejte se co se deje

#include "string2212.h"

int main() {
    String a;
    String b(8.3),c(3),d("sadfl");

    String e(b),f=d;
    String g="akdf ",h=5;

    a.PriradSpojeni(b,d);
    e.Prirad(a.Soucet(c));
    d.PriradSpojeni(5," askdlf ");

    return 1;
}

```

KO: co znamená klíčové slovo (modifikátor) const u proměnné a co za prototypem funkce? Jak spolu souvisí?

Příklad 3.2.12a: ve třídě “T_BOD_3D” rozhodněte o tom, které parametry a funkce jsou z hlediska předávání parametrů const (dopíšte).

Pokud označíme objekt třídy jako const, nemohou být jeho data měněna. Při pokusu měnit data zahlásí překladač chybu. Při volání metody objektu může překladač zavolat pouze takovou metodu, která nemění data. To že metoda nemění data (aktuálního) objektu se označí v prototypu metody uvedením const mezi hlavičku a tělo metody. V návaznosti na vlastnosti překladače je potom nesprávný přístup kontrolován (změna const parametru, nebo změna aktuálního objektu v const metodě vede k chybě překladu).

3.3.13 Friend

Cílem je ozřejmit způsob přístupu k privátním datům třídy pro nečlenské metody a ostatní třídy. Externí metody a funkce nemohou přistupovat k privátním datům třídy. To ovšem zpomluje činnost programu. Aby bylo (ve výjimečných případech) možno přistupovat k privátním datům je možné povolit definovaným třídám a funkcím přístup pomocí friend.

Pokud provádíme návrh spolupracujících tříd, nebo funkce, které mají jako parametr objekt třídy, potom může zprostředkovaný přístup k privátním datům a metodám vést ke složitějšímu a časově náročnějšímu kódu. Proto může třída (autor) povolit vybraným třídám, nebo funkcím přístup ke svým privátním datům a metodám. Jelikož se jedná o porušení přístupových mechanismů, mělo by se toto narušení ochrany dat používat po zralém uvažování.

Pro zpřístupnění privátních (private a protected) dat a metod slouží klíčové slovo friend. Pokud je takto uvedeno v definici třídy před jménem třídy nebo funkce, potom tyto spřátelené objekty třídy a funkce mají možnost pracovat s privátními daty objektu.

```
class Třída {
...
friend complex;      /* "spřátelená" třída - objekty této třídy
                        mohou při práci s objekty typu Třída
                        přistupovat k privátním datům a metodám */
friend double f(Třída &b); /* "spřátelená" globální funkce f
                        může ve svém těle přistupovat i k private členům
                        a metodám Třída (např. objektu b, nebo u lokálních
                        proměnných tohoto typu) . */
}
```

Pozn.: kód friend funkcí je umístěn mimo hlavičku třídy (v hlavičce se uvede pouze friend a prototyp). Friend funkce tedy nemá proměnné dané třídy (nemá this).

Pozn.: vlastnosti dané friend se nedědí

KO: k čemu slouží klíčové slovo friend?

Příklad 3.2.13a: napište (friend) funkci na zjištění vzdálenosti bodu od počátku s voláním `Vzdal(bod)`. Jaký je rozdíl mezi touto funkcí a členskou metodou pro stejný výpočet?

Řešení: rozdíl je v tom, že u friend funkce je objekt dodán jako parametr (a není zde `this`) – `Vzdal(bod)`, zatímco u metody je vlastní objekt dodán přes `this` a parametr by byl `void: bod.Vzdal()`.

Klíčové slovo friend umožňuje rozšířit přístup k private položkám i pro nečlenské funkce nebo pro objekty daných tříd, a tím ulehčit, urychlit práci s danými prvky ve funkcích, které přísluší ke třídě, ale není je vhodné či možné implementovat jako metody dané třídy.

3.3.14 Operátory

Cílem je prezentovat možnost dodat objektům funkčnost využití standardních operátorů. Při zachování pravidel o prioritě a asociativitě operátorů je možné vytvořit pro vytvářenou třídu metody tak aby bylo možno použít nový objekt v zápisu s operátory (obdobně jako standardní typy), jako je přiřazení a relační operátory.

Test: vyjmenujte operátory jazyka C které znáte? Které operátory přidává jazyk C++?

Základní vlastnosti

Jazyk C++ umožňuje přetěžovat nejen funkce a metody ale i operátory. Při práci s nimi zachovává pravidla o prioritě, asociativitě a počtu parametrů. Vlastní činnost nebo parametry operátorů je možné volit libovolně.

Např. v zápise `a = b + c`; jsou použity operátory `'='` a `'+'`. V jazyce C musely být proměnné `a, b, c` proměnné základních typů. V jazyce C++ je možné, pomocí vytvoření vhodných funkcí či metod, rozšířit program tak aby proměnné `a, b, c` mohly být libovolných typů.

Pro operátory platí stejná pravidla jako pro funkce a jejich přetěžování. Správný operátor je vybrán podle seznamu parametrů (i v návaznosti na dostupné konverze). Rozlišení se provádí podle kontextu.

Pozn.: k základním operátorům jazyka C patří např.: `'+', '-', '*', '/', ...` `'&&', '&', '||', '|||', ...`

Pozn.: k novým přetížitelným operátorům patří např. `new` a `delete`

Pozn.: zjednodušený zápis zpřehledňuje tvorbu i orientaci v programech. Nové operátory by měly zachovávat stávající pravidla: např. neměnit operandy, nechovat se diametrálně odlišně (např. `'+'` by nemělo sloužit k odebírání prvků ...), vracet podobné typy.

Definice a použití

Pro definici se používá funkčního zápisu operátoru. Pro každý operátor existuje zkrácená verze použití operátoru, kterou známe z jazyka C. Plná verze operátoru se skládá z klíčového slova operátor, které je následováno symbolem příslušného operátoru. Takto zapsaný operátor se chová stejně jako funkce či metoda – patří do dané třídy, má název, návratovou hodnotu a parametry. Slovo `operator` je novým klíčovým slovem jazyka C++.

Např. zápis $a = b + c$; lze přepsat na $a.operator=(b.operator+(c))$; a pro typ `int` by definice uvnitř třídy `int` (kdybychom ji měli možnost tvořit) vypadala: `int operator+(int ii)`. Z tohoto zápisu je vidět i způsob volání. Nejprve `b` vyvolá operátor `c` s parametry `b` a `c`, a výsledek této metody je parametrem pro metodu `operator =` vyvolaný prvkem `a`.

Pozn.: Díky přetížení by mohla existovat i metoda `int operator+(float ff)`, která by se mohla chovat odlišně, či respektovat různost parametrů. Problém by nastal při volání s rozdílnými parametry než `int` a `float`, kdy by díky konverzím došlo k duplicitám v možnosti volání.

Pozn.: pokud standardní operátory nemění operandy, potom by je neměly měnit ani napsané operátory. Proto se u parametrů používá (a některé překladače vyžadují) modifikátor `const`.

Realizace operátoru

V případě, že tvoříme vlastní třídu, je vhodné realizovat operátory jako metody dané třídy. V případě, kdy je prvním operandem některý ze základních typů, nebo typ, jehož tvorbu nemůžeme ovlivnit, nemůžeme použít členskou metodu. Proto musíme vytvořit pro operátor funkci na globální úrovni.

Pozn.: Pokud bychom tedy chtěli speciální operátor sčítání typů `int` a `float`, nešlo by to realizovat způsobem uvedeným výše, ale musela by se vytvořit globální funkce s prototypem `int operator+(int ii, float ff)`; V obou případech se jedná o operátor se dvěma operandy. V prvním případě je jeden operand dodán pomocí "this" a druhý parametrem.

Pozn. I u operátorů (zvláště u operátoru `=`) je důležité řešit situace, kdy je předáno více stejných proměnných. Např. pro `a = a`; pracujeme s proměnnou `a` jako s výsledkem i parametrem. Pokud tedy zapíšeme novou hodnotu do dat výsledku, změníme tím zároveň i odpovídající hodnoty dat parametru. Při použití těchto dat u parametru tedy dojde k práci s novými (výslednými) a ne původními daty (parametru jak by se zdálo). Předpokládáme, že parametry jsou u tříd předávány vždy odkazem.

Pozn.: Při použití reference `&` se šetří paměť. Pokud by tomu tak nebylo, pak při předávání do funkce hodnotou se vytváří lokální proměnná jako kopie prvku za pomoci copykonstrukturu (obdobně při návratu je nutno použít `return *this` s voláním copy konstrukturu), který vytvoří data návratové hodnoty na zásobníku.

Konkrétní přetížení operátorů

Operátory můžeme rozlišit z různých hledisek. V následujícím jsou uvedeny:

- unární operátory – to je operátory s jedním parametrem, kam patří matematické a logické operátory
- binární operátory – to je operátory se dvěma parametry, kam patří matematické a logické operátory
- konverzní operátory – jsou operátory, díky kterým je možné používat explicitní (a implicitní) konverze, to je které převádějí jeden typ na jiný
- funkční operátor – je možné přetížít i `()`, jehož volání vypadá jako zápis funkce
- operátor indexování – přetížení operátoru `[]`, využívanému k indexování
- ...

Unární operátory

Unární operátory jsou operátory s jedním parametrem. Patří sem unární '+', '-', ++, --, !, ~, ^ ...

Jsou-li tyto definovány uvnitř třídy, potom jsou to metody bez parametrů a vlastní prvek je předán pomocí "this". Jedná-li se o "globální" funkce, potom mají jeden parametr.

```
complex& operator+(void);
```

```
complex operator-(void)
```

Na příkladu unárních operátorů + a - je vidět základní rozdíl v návratové hodnotě. Vycházíme-li ze základních vlastností těchto operátorů, potom operátor + ani - nemění hodnotu proměnné pro kterou jsou použity (její hodnota zůstává po provedení stejná). Rozdíl je v tom, že výsledkem operátoru + (který prakticky nic nedělá, výsledek je stejný jako původní prvek) může být původní prvek a proto vracíme referenci (můžeme, protože referencovaný prvek existuje i vně metody). U operátoru - je vracený prvek (výsledek činnosti operátoru) rozdílný od předávaného (který se nemění) a proto se musí vytvořit nový prvek (ztráta času i místa ale jinak to nejde).

F(+a); proměnná a vyvolá metodu operator+(void), která vrátí referenci na aktuální prvek (return *this), tj. vrátí a, které je parametrem funkce F.

F(-a); proměnná a vyvolá metodu operator-(void), která vrátí nový objekt, který vznikne při návratu z funkce pomocí konstrukturu. Parametrem funkce je tento nový (dočasný) prvek, který je po ukončení provádění funkce (až je nepotřebný) zrušen překladačem (pomocí volání destrukturu).

Unární operátory ++ a -- mají prefixovou a postfixovou verzi. Jelikož se chovají různě, rozliší se fiktivním parametrem int (jeden je s void, druhý s int, který se však nevyužívá). Pokud je nadefinován pouze jeden, zavolá se pro obě varianty. (U starších překladačů není možné nadefinovat obě varianty). Tyto operátory mění hodnotu prvku, který je vyvolal.

př. Třída & operator++(void)	je použit pro	++x
Třída operator++(int)	je použit pro	x++

Opět rozdílné návratové hodnoty. První způsob nejprve provede činnost, která mění hodnotu proměnné a tato je posléze použita. Protože používáme aktuální hodnotu proměnné, která je k dispozici, můžeme vrátit referenci. Druhá verze má vrátit původní hodnotu a poté teprve měnit prvek. Jelikož se ve skutečnosti hodnota vrací pomocí return na konci metody, kdy je aktuální prvek již změněn, je nutné vytvořit nový prvek, který obsahuje původní hodnoty a ten vrátit hodnotou.

Binární operátory

Binární operátor je operátor, který má dva parametry. Může být tedy realizován jako členská metoda třídy s jedním parametrem (druhým je implicitně přes "this" objekt, který metodu operátoru vyvolal), nebo jako globální funkce se dvěma parametry (je výhodné zařadit jako friend funkci pro lepší a rychlejší práci s proměnnými).

```
complex operator+(float c) ve třídě complex pro sečtení complex + float
```

```
complex operator +(complex & c) ve třídě complex pro sečtení complex + complex
```

```
complex operator+(float a, complex &b) mimo třídu pro sečtení float + complex
```

```
friend complex operator+(float a, complex&b) v deklaraci funkce uvnitř třídy
```

Pozn.: patří sem +, -, *, /, &, &&, relační operátory ...

Pozn.: druhý operátor a výsledek mohou být libovolného typu. Většinou se vrací hodnota, protože výsledkem binární operací bývá většinou nová hodnota.

Pozn.: Abychom umožnili součet $b = 9,7 + a$; nebo $b = \text{"text"} + a$; tak nadefinujeme operátor $+$ s více parametry. Tento operátor však musí být globální, protože se tento operátor aplikuje k levému operandu, což je číslo nebo konstantní text.

Operator=

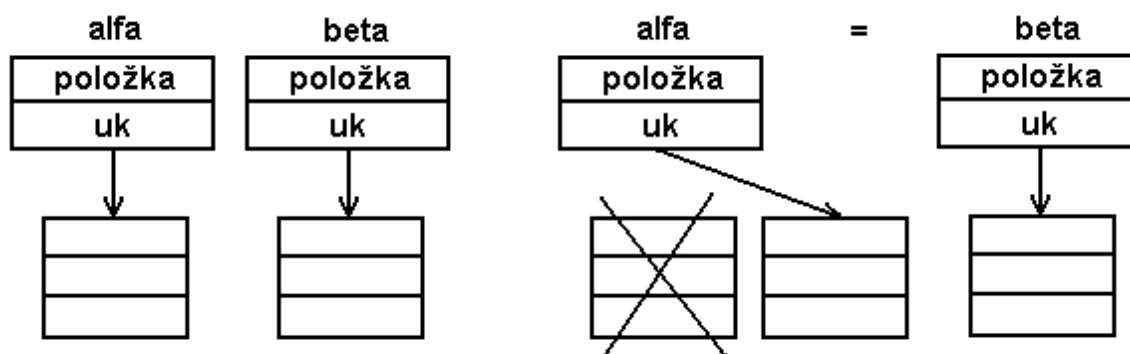
Je binární operátor, který má některé specifické vlastnosti. Díky kompatibilitě se standardními typy by měl mít možnost zřetězení, a proto musí vracet hodnotu. Jelikož vrací aktuální hodnotu prvku, může k tomu použít referenci.

Třída `&operator=(const Třída &c) { ... return *this; }`

Zřetězení: `a = b = c = d`; (stejně tak `+=`, `-=` ...)

Pokud není operátor `=` nadefinován, vytváří se implicitně. Implicitní operátor `=` však přiřazuje vytvořením pouze kopie. To stačí pro statická data. U dynamických odkazů však vzniká problém mělké kopie, kdy dva prvky odkazují na stejný paměťový prostor dat (a tato skutečnost není nikde registrována). Pro prvky s dynamickými daty je tedy vytvoření operátoru `=` nutností.

Pokud je operátor `=` uveden v sekci `private`, mohou ho použít pouze členské metody nebo funkce `friend`, nedefinujeme-li tělo, potom se projeví jako chyba při použití. Toto je výhodné, pokud není možné pro třídu realizovat přiřazení jednoduše (např. jsou nutné dva parametry) a chceme si být jisti, že se určitě nevytvoří ani implicitní operátor `=` (a tudíž se musí použít naše metoda na vytvoření kopie).



Obr. Při provedení operátoru `=` je nutné vytvořit hlubokou kopii. To znamená, že je nutno zrušit původní data, vytvořit prostor pro data nová a naplnit tento prostor aktuálními daty. Zde pozor na případ `a = a`, kdy při zrušení původních dat dojde i ke zrušení dat pro přiřazení. Tento případ je nutné ošetřit.

Např. Třída `&operator = (Třída &p) { if (this == &p) return *this; ... }`

```
string * a,* b; // a a b jsou ukazatele na objekty
a = new string; // objekty vznikají dynamicky
b = new string;
a = b;         /* pouze přiřazení ukazatelů (adres) s objekty
                 nesouvisí,
                 oba ukazatele dále ukazují na stejný objekt
                 (stejnou paměť).
delete a;      // odalokování a destruktork
```

<pre>delete b; /* tady je chyba, protože odalokováváme stejný objekt podruhé(stejná adresa, na které však již není aktivní objekt */</pre>
<pre>string {int delka; char *txt; ... }; /* v definici třídy není nadefinován operátor =. Je tedy vytvořen implicitní což je identická paměťová kopie. Třída má konstruktor, který vyhradí paměť pro txt a destruktory, který tuto paměť uvolní */ { string a , b("ahoj"); a = b; /* je použito implicitní =, to znamená, že ukazatel txt ukazuje na stejná dynamická data (má stejnou adresu) pro oba prvky. Dva objekty ale prvek typu ukazatel má stejnou hodnotu */ } // destruktory (na konci bloku) uvolní paměť na níž ukazuje // txt pro b. Opětovný pokus o odalokování paměti z adresy // txt pro a (zanikají oba objekty) vede k chybě</pre>
<pre>string {int delka; char *txt;operator =}; /* ve třídě je nadefinován operátor =, konstruktory a destruktory pro práci s pamětí pro txt */ { string a , b("ahoj"); a = b; /* je použito nadefinované = (předpokládejme že je správné), to znamená, že ukazatel txt v a ukazuje na svoji kopii dat statických i dynamických (byl vytvořen nový paměťový blok a v něm kopie originálních dat. */ } // na konci bloku zanikají objekty a volají destruktory. // Každý objekt odalokovává svoji kopii dynamických dat // které jsou rozdílné</pre>

Konverzní operátory

Slouží při implicitních a explicitních konverzích, kdy umožňují převádět objekt na jiné typy. Opačný směr jako u konverzních konstruktorů. Konverzní operátor se využívá např. když nemáme pod kontrolou třídu, ve které bychom potřebovali konverzní konstruktor. Např. pro standardní typy.

Název operátoru je dán typem na který se má převádět. Tento typ musí být i návratovou hodnotou a proto se tato v definici neuvádí.

operator typ (void) – nemá návratovou hodnotu, ta je dána požadovaným typem, nemá parametr

Např. operator int(void) {... return a;}, lze volat int(objekt), (int)objekt, nebo je volán implicitně když je potřeba.

Funkční operátor

Jedná se o přetížení operátoru `operator()`. Jelikož jeho volání připomíná volání funkce, nazýváme ho funkčním operátorem (přetížení funkčního volání).

přetížení operátoru závorek `operator () (int a, int b, int c)`

návratová_hodnota jméno_třídy::operator()(int a, int b, int c), ve verzi s referencí :

```

// tj. zobrazte *this, ...
// objekty muzete rozlisit pomoci indexu

#ifndef KOMPLEX_H
#define KOMPLEX_H

#include <math.h>

struct Komplex {
    enum TkomplexType {eSlozky, eUhel};

    static int Poradi;
    static int Aktivnich;

    double Re,Im;
    int Index;

    Komplex(void)
        {Re=Im=0;Index = Poradi;Poradi++;Aktivnich++; }

    inline Komplex(double re,double im=0,
                    TKomplexType kt = eSlozky);
    Komplex(const char *txt);
    inline Komplex(const Komplex &p);
    ~Komplex(void) {Aktivnich--;}

    void PriradSoucet(Komplex const &p1,Komplex const &p2)
        {Re=p1.Re+p2.Re;Im=p1.Re+p2.Im;}

    Komplex Soucet(const Komplex & p)
        {Komplex pom(Re+p.Re,Im+p.Im);return pom;}

    Komplex& Prirad(Komplex const &p)
        {Re=p.Re;Im=p.Im;return *this;}

    double faktorial(int d)
        {double i,p=1; for (i=1;i<d;i++) p*=i; return p; }

    double Amplituda(void)const
        { // Re = 4;
          return sqrt(Re*Re + Im *Im);}

    bool JeMensi(Komplex const &p)
        {return Amplituda() < p.Amplituda();}

    double Amp(void) const;
    bool JeVetsi(Komplex const &p) {return Amp() > p.Amp();}

    // operatory

    Komplex & operator+ (void) {return *this;}

```

```

// unární +, může vrátit sám sebe,
// vrácený prvek je totožný s prvkem, který to vyvolal

Komplex  operator- (void) {return Komplex(-Re,-Im);}

// unární -, musí vrátit jiný prvek než
// je sám (ten konstruktor v returnu je dost drastický)

Komplex & operator++(void) {Re++;Im++;return *this;}

// nejdřív přičte a pak vrátí,
// takže může vrátit sám sebe (pro komplex patrně nesmysl)

Komplex  operator++(int)  {Re--;Im--;
                          return Komplex(Re-1,Im-1);}

// vrací původní prvek, takže musí vytvořit
// jiný pro vrácení (pro komplex patrně nesmysl)

Komplex & operator= (Komplex const &p)
    {Re=p.Re;Im=p.Im;return *this;}

// bez const v hlavičce se neprelozi nektera prirazeni,
// implementováno i zřetězení

Komplex & operator+=(Komplex &p)
    {Re+=p.Re;Im+=p.Im;return *this;}

// návratový prvek je stejný jako ten, který to vyvolal,
// takže se dá vrátit sám

bool      operator==(Komplex &p)
    {if ((Re==p.Re)&&(Im==p.Im)) return true;else false;}

bool      operator> (Komplex &p)
    {if (Amp() > p.Amp()) return true;else false;}
// může být definováno i jinak

bool      operator>=(Komplex &p)
    {if (Amp() >=p.Amp()) return true;else false;}

Komplex  operator~ (/*Komplex &p*/ void)
    {return Komplex(Re,-Im);}

//komplexne sdruzeny / kdyz je bez parametru,
// musi byt bez parametru
// bylo by dobré mít takové operátory dva jeden,
// který by změnil sám prvek
// a druhý, který by prvek neměnil

```

```

Komplex& operator! ()
    {Im*=-1;return *this;};

// a tady je ten operátor
// co mění prvek. Problém je, že je to
// nestandardní pro tento operátor
// a zároveň se mohou plést.
// Takže bezpečnější je nechat jen ten první
// bool      operator&&(Komplex &p) {}

Komplex  operator+ (Komplex &p)
    {return Komplex(Re+p.Re,Im+p.Im);}

Komplex  operator+ (float f)
    {return Komplex(f+Re,Im);}

Komplex  operator* (Komplex const &p)
    {return (Re * p.Re - Im * p.Im,Re * p.Im + Im * p.Re);}

Komplex &operator*= (Komplex const &p)
    // zde je nutno použít pomocné proměnné, protože
    // je nutné použít v obou přiřazeních obě proměnné
    {double pRe=Re,pIm=Im;
     Re=pRe*p.Re-Im*p.Im;Im=pRe*p.Im+pIm*p.Re;
     return *this;}

// ale je to špatně v případě, že použijeme pro a *= a;,
// potom první přiřazení změní
// i hodnotu p.Re a tím nakopne výpočet druhého
// parametru (! I když je konst !)
// {double pRe=Re,pIm=Im,oRe=p.Re;
//  Re=pRe*p.Re-Im*p.Im;Im=pRe*p.Im+pIm*oRe;return *this;}
// verze ve které přepsání Re složky již nevadí
// friend Komplex operator+ (float f,Komplex &p);
//není nutné pokud nejsou privátní proměnné
operator int(void){return Amp();}
};

inline Komplex::Komplex(double re,double im, TKomplexType kt )
{
    Re=re;
    Im=im;
    Index=Poradi;
    Poradi++;
    Aktivnich++;
    if (kt == eUhel)
        {Re=re*cos(im);Im = Re*sin(im);}
}

Komplex::Komplex(const Komplex &p)
{

```

<pre> Re=p.Re; Im=p.Im; Index=Poradi; Poradi++; Aktivnich++; } #endif </pre>
<pre> //===== komplex2214.cpp - zdrojový kód třídy ===== // trasujte a divejte se kudyma to chodi, // tj. zobrazte *this, ... // objekty muzete rozlisit pomoci indexu #include "komplex2214.h" int Komplex::Poradi=0; int Komplex::Aktivnich=0; Komplex::Komplex(const char *txt) { /* vlastni alg */; Re=Im=0; Index = Poradi; Poradi++; Aktivnich++; } double Komplex::Amp(void)const { return sqrt(Re*Re + Im *Im); } Komplex operator+ (float f,Komplex &p) { return Komplex(f+p.Re,p.Im); } </pre>
<pre> //===== komplex2214p.cpp - kód aplikace ===== #include "komplex2214.h" char str1[]="(73.1,24.5)"; char str2[]="23+34.2i"; int main () { Komplex a; Komplex b(5),c(4,7); Komplex d(str1),e(str2); Komplex f=c,g(c); </pre>

```

Komplex h(12,35*3.1415/180.,Komplex::eUhel);
Komplex::TKomplexType typ = Komplex:: eUhel;
Komplex i(10,128*3.1415/180,typ);

d.PriradSoucet(b,c);
e.Prirad(d.Prirad(c));
d.PriradSoucet(5,c);
d.PriradSoucet(Komplex(5),c);

e = a += c = d;
a = +b;
c = -d;
d = a++;
e = ++a;
if (a == c)
    a = 5;
else a = 4;
if (a > c)
    a = 5;
else a = 4;
if (a >= c)
    a = 5;
else a = 4;
b = ~b;

//? bool      operator&&(Komplex &p) {}
c = a + b + d;
c = a + 5 + 4 + d;
c = 5 + c;
int k = int (c);
int l = d;
float m = e;
// pozor - pouzije jedinou možnou konverzi a to přes int
if (a && c)
// musi byt implementovan - neni-li konverze (např. zde se
// prohlašuje přes konverzi int, kde je && definována)
    e = 8;          // u komplex nesmysl
Komplex n(2,7),o(2,7),p(2,7);
n*=o;
p*=p; // pro první realizaci n*=n je výsledek n a p různé
// i když vstupy jsou stejné
if (n!=p)
    return 1;
return 0;
}

```

KO: co jsou to operátory a jak se definují?

Příklad 3.2.14 : Napište friend funkci realizující výstup souřadnic bodu (z třídy T_BOD_3D) na konzolu pomocí operátoru >>. Napište operátory přiřazení a rozdíl (realizující vzdálenost bodů). Trasujte a sledujte vznik a zánik objektů.

Přetížení operátorů pro vlastní třídy umožňuje lépe pracovat s objekty dané třídy pomocí standardních postupů, které jsou definovány pro základní typy. Operátory zachovávají vlastnosti jako je asociativita, priorita a počet parametrů. Lze přetížit unární i binární operátory, Lze přetížit matematické i logické operátory. Realizovat lze operátory jako metody třídy nebo jako globální funkce. Důležité jsou především operátory přiřazení a konverzní operátory.

3.3.15 Statické metody třídy

Cílem je ukázat způsob tvoření funkcí, které patří ke třídě ale nechovají se jako metody ale jako funkce – jedná se o alternativu k friend funkcím.

nepřijímají konkrétní prvek ale pracují s private členy třídy. Alternativou k friend funkcím je statická metoda třídy. Opět neobsahuje this, tedy prvek který ji vyvolá ale má přístup k privátním proměnným třídy.

Test: co je to friend funkce

Statická metoda třídy je metoda, která je vytvořena jedna pro třídu. Při jejím volání je použito standardní funční volání – nevyvolává ji tedy objekt dané třídy a neobsahuje tedy implicitní objekt reprezentovaný this. Z hlediska činnosti se chová jako friend funkce (je to obyčejná fce, která může k private datům), má přístup k private položkám. Volání se provádí se jménem třídy ale bez objektu Třída::fce(). Z metod dané třídy potom bez Třída::.

Použití metod nad třídou ve stavu, kdy není vždy k dispozici objekt – např. ša blony či přístup ke statickým proměnným.

```
class string {
    static fce(void);    // deklarace statické metody
    static int pocet;    // deklarace statické proměnné
}
```

Pozn.: Statické členy třídy slouží pro data a metody, která jsou společná pro všechny členy třídy a tedy se vytvářejí jen v jedné kopii pro program.

Pozn.: statická metoda nesmí být virtuální.

KO: jaké vlastnosti má statická metoda třídy? Čím se liší od členských metod třídy? Čím se liší od friend funkcí.

Statická metoda třídy je metoda, která se volá obdobně jako funkce. Není v ní přítomen objekt pomocí this. Má přístup k private členům a je tedy vhodnější než friend funcce.

3.3.16 Modifikátor mutable

Cílem je ukázat možnost změny vybraných proměnných v konstantním objektu. V případě nutnosti lze změnit některý z parametrů konstantního objektu. Je toto možné u prvků, které jsou označeny mutable.

Modifikátor mutable umožňuje měnit uvedené proměnné objektu třídy, u v případě, kdy je objekt označen jako const. V případě, že je objekt třídy const, potom proměnné, které jsou v deklaraci označené třídou mutable lze měnit.

```
class X {
public :
mutable int i ;
int j ;
}

class Y { public: X ; }
const Y y ;      // y je konstantní objekt
y . x . i ++ ;   //může být změněn jelikož je definován mutable
y . x . j ++ ;   //chyba - y je const
```

Pozn.: statická data nemohou být mutable.

KO: K čemu slouží operátor mutable?

Je-li nutné měnit vybrané proměnné u const objektů, potom je možné tak učinit, pokud byly v definici označeny klíčovým slovem mutable.

3.3.17 Třídy a prostory jmen

Cílem je ozřejmit souvislosti mezi prostory jmen a třídou. Třída může mít platnost v daném prostoru jmen, ale je sama prostorem jmen pro své členské data a metody.

Objekt typu třída je z hlediska vnějších prostorů jmen stejný jako jiné typy. Třída samotná se však jeví jako samostatný prostor jmen, který odděluje svoje data a metody od okolí. Pomocí using může být zveřejněna proměnná nebo metoda z jiného prostoru jmen.

3.3.18 Třídy a streamy – vstupy a výstupy

Cílem je upozornit na souvislosti mezi třídou a streamem.

Pro objekty je nutné přetížit operátory vstupu a výstupu, aby s nimi bylo možné pracovat stejně jako s ostatními standardními typy. Díky tomu, že operace vstupu a výstupu "vyvolá" objekt streamu, který nemůžeme modifikovat, používáme friend (globálních f-cí). Prvním parametrem je objekt streamu, druhým parametrem námi definovaný typ.

Funkce provede výstup jednotlivých proměnných typu (možno i s komentářem či formátem). Z důvodu zřetězení je nutné aby návratovou hodnotou funkce byl objekt streamu (reference na něj).

friend ostream & operator>>(ostream& str, trida &tt) “oznámení” operátoru ve třídě, které současně povolí nečlenské globální funkci přístup k privátním prvkům a tím urychlí práci

ostream & operator>>(ostream& str, trida &tt) {... ; return str;} vlastní definice operátoru vstupu pro typ trida. Jedná se o globální funkci, proto nemá v názvu operátor příslušnosti, a uvnitř nemá this.

příklad 3.3.18 vstupy a výstupy

napište operátory pro stream pro čtení a výpis komplexního čísla - umožňuje zřetězení příkazů

```
istream &operator >> ( istream &s, complex &a )
{
    char c = 0;

    s >> c; // levá závorka
    s >> a.re >> c; // načtení reálné složky a oddělovací čárky
    s >> a.im >> c; // načtení imaginární složky a konečné závorky
    return s;
}

ostream &operator << ( ostream &s, complex &a )
{
    s << ' (' << a.real << ',' << a.imag << ')' << '\n';
    return s;
}

{
    istream i("text.txt", ios::in);
    ostream o("vystup.txt", ios::out || ios::nocreate);
    complex a,b;

    i >> a >> b;
    o << " první proměnná " << a << " druha proměnná " << b << endl;
}
```

Přetížení operátorů vstupu a výstupu <<>> umožňuje využití definovaných tříd při vstupně-výstupních situacích stejným způsobem jako standardní typy.

Objektové vlastnosti - závěr

Objektové vlastnosti umožňují nový typ programování. Je možné sdružovat data a metody s nimi pracující do logických celků. Zároveň je možné kontrolovat vznik a zánik

objektů a tedy inicializaci objektu a ochranu dat před zánikem objektu. Přetížené operátory umožňují přistupovat k novým typům stejně jako při práci se standardními typy.

3.4 Objektové vlastnosti C++ - dědění

3.4.1 Dědění

Cílem je uvést základy a důvody pro princip dědičnosti. Dále pak prezentovat způsoby dědění a možnosti skrývání dat na základě jejich přístupových práv.

Dědění je mechanismus, jak převzít vlastnosti již vytvořené třídy a na nich vytvořit vhodnou nastavbu – rozšíření. Ke stávajícímu je možné dodělat nové proměnné a nové metody. Původní proměnné a metody zůstávají, podle typu (z)dědění je však možné řídit jejich přístupová práva – viditelnost – v odvozené třídě.

Dědičnost je

- odvození tříd z již existujících
- базové a odvozené (následnické, definované třídy)
- jednoduché, násobné dědičnosti
- sdílení kódu

class C: public A C dědí z A plus přidává vlastní položky a metody (které mohou překrýt původní z A, ale ty se neztrácí, A si je pořád na své úrovni volá a C je může valat jako “globální” s operátorem příslušnosti A:: .

Klíčové slovo určující typ dědění (za :) nemusí být uvedeno. Potom u class se předpokládá private, zatímco u struct public.

class D: public B1, protected B2, C C je děděna implicitně jako private.

Dědíme-li z jedné (bázové) struktury A do druhé (odvozené), potom při daném typu dědění se prvky s jednotlivým přístupem dědí následovně (v tabulce je базová třída A a prvky (či metody) s rozdílnými přístupovými právy. V dalších sloupcích jsou různé typy dědění a jak se při nich mění přístupová práva k prvkům):

class A	class B:private A	class C:protected A	class D:public A
public a	private a	protected a	public a
private b	-	-	-
protected c	private c	protected c	protected c

Pozn.: u datových položek se preferuje nastavení `private` – důvodem je, aby ochranné mechanismy přístupu fungovaly nejen pro externí-uživatelský přístup, ale i pro zděděné třídy. Důvodem tohoto použití tedy je, aby odvozená třída nemohla porušit kontroly zavedené ve třídě báze. U metod je vhodné použít `protected`, což vede k tomu, že interní metody báze třídy jsou přímo použitelné i ve třídách odvozených, a tím se urychluje přístup k nim.

Pokud má třída báze třídu, je konstruktor báze třídy volán dříve než konstruktor dané třídy. Postup volání :

konstruktor báze třídy

konstruktor lokálních proměnných (v pořadí jak jsou uvedeny v definici)

konstruktor dané třídy

Pořadí konstruktorů na stejné úrovni lze měnit jejich pořadím v hlavičce třídy.

Destruktory se volají v opačném pořadí (než se volaly příslušné konstruktory).

```
class Base {
    int x;
public:
    float y;
    Base( int i ) : x ( i ) { };      zavolá konstruktor třídy a
    pak proměnné, x(i) je konstruktor pro int, jinak by se zavolal
    implicitní konstruktor a následně by se provedlo v těle přiřazení
    x = i. Což by byla ztráta, protože implicitní konstruktor by byl
    k ničemu
}
```

```
class Derived : Base {
public:
    int a ;
```

```
    Derived ( int i ) : a ( i * 10 ) : Base ( a ) { }
```

// volání lokálních konstruktorů umožní konstrukci podle požadavků, ale nezmění pořadí konstruktorů

```
    Base::y; // je možné takto vytáhnout proměnnou (zděděnou zde do sekce private) na
    jiná přístupová práva (zde public)
}
```

POZOR: v tomto pořadí se do `Base (a)` dosadí neinicizované `a` . volá se totiž `Base(a)` a potom `a(i*10)`

Pozn.: lze přetížit ale nelze zrušit – fyzicky vždy obsahuje staré metody a data

Je rozdíl mezi tím, zda je proměnná přítomna a přístupná (to určuje způsob dědění a přístupová práva k dané proměnné)

Konstruktory, destruktory ani operátor = se nedědí. Ze zděděných proměnných je možné používat jen ty, které byly public nebo protected. Lze dědit i z více objektů najednou. Ukazatel na potomka je i ukazatelem na předka / konverze) protože začátek třídy je stejný (explicitně i naopak). **A* ptrA = new AB.**

```
class base {  
base (int i=10) ...  
}
```

```
class derived:base {  
complex x,y; ...
```

```
public: defived() : y(2,1) {f() ... }
```

```
volá se base::base()  
complex::complex(void) - pro x  
complex::complex(2,1) - pro y  
f() ... - vlastní tělo konstrukturu
```

jsou-li v sekci private, potom:

implicitní konstruktor zabrání dědění

destruktor zabrání dědění a vytváření instancí

copy constructor zabrání předávání (a vracení) parametru hodnotou

konstruktor zabrání vytváření instancí

operátor = nedovolí přiřazení



Proto lze použít ukazatel a výpočty nad bázovou třídou pro odvozené a ne naopak. Směr hledání vhodné metody je od odvozených směrem ke třídě bázové.

Přetěžování metod při dědění – zděděná metoda musí po svém ukončení zanechat objekt ve stavu alespoň tak jako předchozí (například bázový string odchází z metody vždy

s nastavenou proměnnou délkou na skutečnou délku – a tedy toto musí dodržet i zděděná metoda)
+ samozřejmě může spřísnit

Zděděná metoda může na svém vstupu očekávat jen to co metoda před ní (nemůže-li aktuální délku řetězce v proměnné délce očekávat bazová třída, pak na to nemůže spoléhat ani odvozená, protože by se mohla dostat na řadu po bazové, která to neřeší.)

KO: co je to dědění? Jak probíhá? Jak se mění a stanovují přístupová práva ke členům?

Příklad 3.3.1 a.: Navrhněte třídu “T_GR_BOD_3D”, který bude mít oproti původnímu bodu navíc grafické parametry a to typ bodu a barvu bodu a metody na vykreslení (uvažujme pro výstup např. textovou matici 20x20 bodů). Původní třída bude sloužit k výpočtům a držení dat o poloze, nastavbová třída zajistí vykreslení.

Příklad 3.3.1b: Navrhněte třídu “T_LINE_3D”, která vychází z původního bodu (pro krajní polohy).

3.4.2 Vícenásobné dědění

Cílem je prezentace způsobů vícenásobného dědění, které umožňuje, aby třída vznikla z více bazových tříd a přijala jejich vlastnosti.

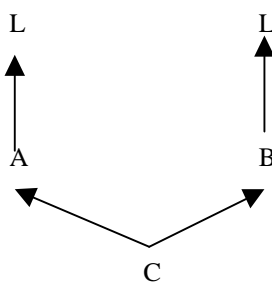
1) nelze aby se na jedné úrovni dědilo dvakrát z jedné třídy C: B,B

2)

A: public L

B: public L

C: public A, Public B



v C je A::a a B::b

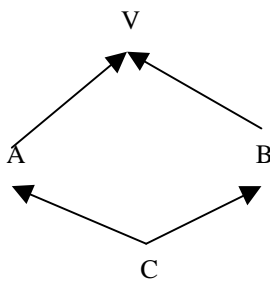
3)

A: virtual V

B: virtual V

C: public A, public B

(konstruktor V se volá pouze jedenkrát)



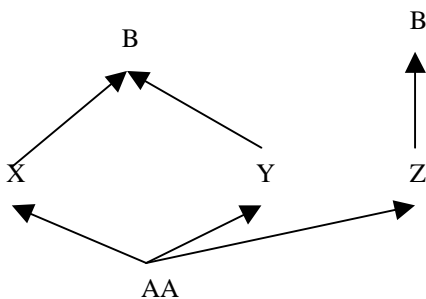
4)

X: virtual public B

Y: virtual public B

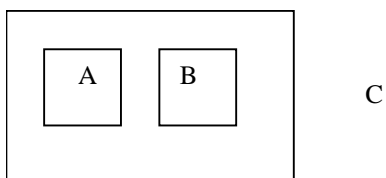
Z: public B

AA: public X, Y, Z

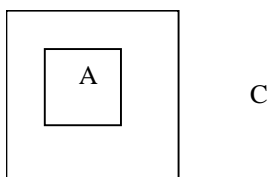


pořadí rodičovských tříd -> pořadí konstruktorů – pořadí volání konstruktorem má přednost

Problémy: dvě třídy stejné jméno proměnné, pak musíme rozlišit B:: A:: ...



c je nová třída s prvky A a B – spojení dvou "oblastí-celků" do jednoho společného bloku



c je nová třída s prvkem A – rozšíření vlastností stávající třídy (A může být společným základem pro různé prvky).

KO: K čemu slouží vícenásobné dědění? K čemu je dobré klíčové slovo virtual u způsobu dědění?

3.4.3 Virtuální metody

Cílem je ukázat možnosti pozdní vazby, tj. rozlišení metod až za běhu programu. Pro rozlišení slouží virtuální metoda a přístup k nim je realizován přes tabulku virtuálních metod. Rozlišení neprobíhá na základě v překladači definovaném typu, ale podle tabulky, která je proměnné přidělena při jejím vzniku.

zajišťují tzv. pozdní (dynamická) vazbu, tj. zjištění adresy metody až za běhu programu pomocí tabulky virtuálních metod, která se vytváří voláním konstruktoru. V "klasickém" programování je volaná metoda vybrána již při překladu překladačem na základě typu funkce či metody, která se volání účastní (statická, časná vazba). U virtuálních metod není důležité čemu je proměnná přiřazena, ale jakým způsobem vznikla – při vzniku je jí dána tabulka metod, které se mají volat. Tato tabulka je součástí prvku. Objekt má vždy statickou vazbu.

jsou-li v bázevé třídě definovány metody jako virtual, musí být v potomcích identické ve zděděných třídách není nutné uvádět virtual
metoda se stejným názvem, ale jinými parametry se stává nevirtuální, tedy statickou pokud je virtual v odvozené třídě a parametry se liší, pak se virtual ignoruje
virtuální metody fungují nad třídou, proto nesmí být ani static ani friend
i když se destruktory nedědí, může být virtuální

Využívá se v situaci kdy máme dosti příbuzné objekty, potom je možné s nimi jednat jako s jedním

Např. výkres, kresba – objekty mají parametry, metody jako posun, rotace, data ... Kromě toho i metodu kreslí na vykreslení objektu

Pozn.: Máme-li seznam objektů, pak potřebujeme zvlášť řešit čáry, body, čtverce ... Tj. mít oddělené seznamy a oddělené procházení každého typu. To není dobré. Když však dáme společný typ, schopný vyřešit vše, tj. můžeme mu přiřadit čáry čtverce .. potom se to dá zajistit v jediném cyklu. Potom potřebujeme aby b-> kreslí zavolalo vždy tu správnou funkci. Tak jak volání známe, dochází k výběru volané funkce již při překladu tzv. statická vazba (early binding). Toto bude fungovat podle toho, jaký typ proměnné překladač vidí – jakého je typu v definici. My ovšem potřebujeme rozlišovat podle toho jak vznikl, ne čemu je přiřazen. Mechanismem je, že potomek dědí všechny vlastnosti rodič, lze použít tedy ukazatel na rodiče a ukazovat tím i na potomka. Rodič má potom společnou množinu dat, která se dá použít ve společných částech kódu – najde se vždy příslušná funkce nebo data. (nelze naopak). Ví se že ukazatel je na bod a proto je vybrána funkce kreslí pro bod. Chceme ovšem možnost např. tisku pro všechny objekty ... To se zajišťuje tzv. virtuálními funkcemi.

Společné rozhraní – není třeba znát přesně třídu objektu a je zajištěno (při běhu programu) volání správných metod – protože rozhraní je povinné a plyne z bázevé třídy.

Virtuální f-ce – umožňují dynamickou vazbu (late binding) – vyhledání správné funkce až při běhu programu.

Rozdíl je v tom, že se zjistí při překladu, na jakou instanci ukazatel ukazuje a zvolí se virtuální funkce. Neexistuje-li vyhledává se v rodičovských třídách. Musí ovšem souhlasit parametry funkce. Ukazatel má vlastně dvě části – dynamickou – danou typem, pro který byl definován a statickou – která je dána typem na který v dané chvíli ukazuje.

Není-li metoda označena jako virtuální – použije se statická (tj. volá se metoda typu kterému je právě přiřazen objekt).

Pozn.: řeší přímo překladač, přesné volání je součástí kodu

je-li metoda virtuální, použije se dynamická vazba – je zařazena funkce pro zjištění až v době činnosti programu – zjistit dynamickou kvalifikaci. vazba (tj. volá se metoda typu pro který byl vytvořen objekt)

zavolat metody dynamické klasifikace – přes tabulku odkazů virtuální třídy

Pozn.: řeší se při běhu programu

Při vytvoření virtuální metody je ke třídě přidán ukazatel ukazující na tabulku virtuálních funkcí. Tento ukazatel ukazuje na tabulku se seznamem ukazatelů na virtuální metody třídy a tříd rodičovských. při volání virtuální metody je potom použit ukazatel jako bazová adresa pole adres virtuálních metod. Metoda je reprezentována indexem, ukazujícím do tabulky. Tabulka odkazů se dědí. Ve zděděné tabulce – přepíše se adresy předdefinovaných metod, doplní nové položky, žádné položky se nevypouští. Nevirtuální metoda překrývá virtuální

Máme-li virtuální metodu v bazové třídě, musí být v potomcích deklarace identické. Konstruktory nemohou být virtuální, destruktory ano.

Virtual je povinné u deklarace a u inline u definice.

ve zděděných třídách není nutn uvádět virtual

stejný název a jiné parametry – nevirtuální – statická vazba (v dalším odvození opět virtual)

pokud je virtual v odvozené třídě – a parametry se liší – virtual se ignoruje

protože virtuální f-ce fungují nad třídou, nesmí být virtual friend, ani static

i když se destruktory nedědí, destruktory v děděné třídě přetíž (zruší) virtuální

virtuální funkce se mohou lišit v návratové hodnotě, pokud tyto jsou vůči sobě v dědické relaci

Čisté virtuální metody

pokud není virtuální metoda definována, tak se jedná o čistou virtuální metodu, která je pouze deklarována

obsahuje-li objekt čistou virtuální metodu, nemůže být vytvořena jeho instance, může být ale vytvořen jeho ukazatel.

```
deklarace :      class base {
                  ....
                  virtual void fce ( int ) = 0;
                }
```


virtuální metoda nemusí být definována – v tom případě hovoříme o čistě virtuální metodě, musí být deklarována.

chceme využít jednu třídu jako Bázovou, ale chceme zamezit tomu, aby se s ní pracovalo. Můžeme v konstruktoru vypsát hlášku a exitovat. Čistější je ovšem, když na to přijde překladač – tj. použít čistě virtuální metody

deklarace vypadá: **virtual void f (int)=0** (nebo =NULL)

tato metoda se dědí jako čistě virtuální, dkud není definována

starší překladače vyžadují v odvozené třídě novou deklaraci a nebo definici

obsahuje-li objekt č.v.m. nelze vytvořit jeho instanci, může být ale ukazatel

```
class B {
public: virtual void vf1()
    {cout <<"b";}
void f()
    {cout<<"b";}
}

class C:public B{
void vf1()
    {cout << "c";}    // virtual nepovinne
void f()
    {cout <<"c";}
}

class D: public B {
void vf1()
    {cout<<"d";}
void f()
    {cout <<"d";}
}

B b; C c;D d;
```

b.f(); c.f(); d.f(); // vola normalni metody tridy / tisk b c d – protože promenne jsou typu B C D / prekladac

b.vf1();c.vf1();d.vf1(); // vola virtualni metody tridy / tisk b c d – protože promenne vznikly jako B C D/ runtime

B* bp = &c; // ukazatel na bazovou tridu (prirazeni muze byt i v ifu, a pak se neví co je dal)

Bp->f(); // vola normalni metodu tridy B / tisk b, protože promenna je typu B / prekladac

bp -> vf1(); // vola virtualni metodu / tisk c – protože promenna vznikla jako typ c / runtime

Příklad 3.3.3 virtuální metody

Co se vytykne při zpuštění následujícího programu na jednotlivých řádcích funkce main a proč.

```
#include <iostream.h>

class A {
public:
A(void)          {cout << 'a';}

virtual ~A(void) {cout << 'b';}

void f(void) {cout << 'c';}

virtual fv(void) {cout << 'd';}
};

class B:public A {
A a;

public:
B(void) {cout << 'e';}

virtual ~B(void) {cout << 'f';}

void f(void) {cout << 'g';fv();}

virtual fv(void) {cout << 'h';}
};

class C:public A {
B a;

public:
C(void) {cout << 'i';}

virtual ~C(void) {cout << 'j';}

void f(void) {cout << 'k';fv();}
};

int main ()
{
A *a;
B b;
B *c = (B*)new C;
a = &b;
```

```

a -> f();
a -> fv();
c -> f();
c -> fv();
delete c;
b.f();
b.fv();
}

```

KO: jaký je mechanismus virtuálních metod? Proč a jak se používají?

Příklad 3.3.3a: Navrhněte způsob jak by bylo možné udržovat prvky “T_BOD_3D” a “T_LINE_3D” ve společném poli. Předpokládejte, že je nutné zjistit okno pro vypsání (tj. zjistit minimální a maximální x a y souřadnice ze všech přítomných bodů (bodů a konců line) v jednom cyklu – průchodu pole) – konkrétně viz. následující kapitola.

3.4.4 Abstraktní (bázové) třídy

Cílem je ukázat princip tvorby a využití bázových tříd, které definují jednotný interface pro třídy, které z nich budou dědit.

abstraktní třída je třída, která má alespoň jednu čistou virtuální metodu

tato třída slouží jako společná výchozí třída pro potomky, neuvažuje se u ní o jejím použití

různé definice – obecně je to třída, která nemá žádnou instanci a ani se neuvažuje o jejím použití – slouží jako společná výchozí třída – objektový přístup, protože tuto definici splňují třídy mající čistě virtuální metody, hovoří se o abstraktní třídě, má-li alespoň jednu č.v.m. – C++ přístup např. tvorba rozhraní

```

class x{
public:
virtual void f()=0;
virtual void g()=0;
void h() ;
}

```

```

X b; // nelze
class Y: class x{
void f() ;
}

```

Y b; opet nelze

```

class Z: class Y{
void t();}

```

```

Z c; uz jde
c.h() z x
c.f() z y
c.g() z Z

```

Příklad 3.3.4 bázová třída

Navrhnete abstraktní bázovou třídu TGrBase pro (grafické) objekty, tak aby bylo pro různé zděděné objekty možno zjistit jejich rozložení v ose X (minimální a maximální plohu), vytisknout prvek a posunout prvek.

Vytvořte zděděné třídy Tbod, Tline tak, aby je bylo možno řadit do společných polí a pracovat s nimi jednotným způsobem. Zařaďte body a čáry do pole a zjistěte, minimální a maximální rozměr x a posuňte všechny zařazené prvky. Kromě virtuálních metod pro tisk implementujte i metody tisku nevirtuální a srovnějte možnosti jejich použití.

```

// dedeni.cpp

#include <iostream.h>

class TGrBase {
protected:
// definice spolecnych promennych
    int Index;

public:
// definice spolecneho rozhrani
    virtual double MinX(void) = 0;
    virtual double MaxX(void) = 0;

    virtual void Translace(double xx,double yy) = 0;

    virtual void TiskV(char *txt) = 0;
};

class TBod:public TGrBase {
// public kvuli prirazeni ukazatele do tgrbase - proc

    static int Celkem, Aktualni;
    double x,y;

public:
    TBod(void)
        {x=y=0;Celkem++;
        Aktualni++;Index = Celkem;
        Tisk(" konstruktor void ");}

    TBod(double xx,double yy=0)
        {x=xx;y=yy;
        Celkem++;Aktualni++;

```

```

        Index = Celkem;Tisk (" konstruktor x,y ");}

virtual ~TBod(void) {Aktualni--;Tisk(" destruktork ");}
        // pouze kvuli tisku

double GetX(void) {return x;}
double GetY(void) {return y;}

virtual double MinX(void) {Tisk("Min X");return x;}
virtual double MaxX(void) {Tisk("Max x");return x;}

virtual void Translace(double xx,double yy)
        {x += xx;y+= yy;Tisk("Translace");}

void Tisk(char *str)
        { cout << " bod " << Celkem << " " <<
          Aktualni << " " << Index << " / " <<
          x << " , " << y << str << endl;}

virtual void TiskV(char *str) {Tisk(str);}

friend ostream & operator<<(ostream &os, TBod &b);
};

ostream & operator<<(ostream &os, TBod &b)
{
    os << " ( " << b.x << " , " << b.y << " ) ";
    return os;
}

class TLine:public TGrBase {
    static int Celkem,Aktualni;

    TBod a,b;

public:
    TLine(void)
        {Celkem++;Aktualni++;Index = Celkem;
        Tisk(" konstruktor void ");}

    TLine(double x1,double y1,double x2,double y2)
        :a(x1,y1),b(x2,y2)
        {
            Celkem++;Aktualni++;
            Index = Celkem;
            Tisk(" konstruktor int,int,int,int ");}

    TLine(Tbod &aa,Tbod &bb)
        {a=aa;b=bb;Celkem++;Aktualni++;
        Index = Celkem;Tisk(" konstruktor TBod,Tbod ");}

```

```

virtual ~TLine(void ){Aktualni--;Tisk(" destruktor ");}

double MinX(void)
    {Tisk("Min X");
    return a.GetX() < b.GetX() ? a.GetX() : b.GetX(); }

double MaxX(void)
    {Tisk("Max X");
    return a.GetX() > b.GetX() ? a.GetX() : b.GetX(); }

void Tisk(char *str)
    { cout << " line " << Celkem <<
      " " << Aktualni << " " << Index << " / " <<
      a << " , " << b << str<< endl;}

virtual void TiskV(char *str)
    { cout << " line " <<
      Celkem << " " << Aktualni << " " << Index <<
      " / " << a << " , " << b << str<< endl;}

virtual void Translace(double xx,double yy)
    {a.Translace(xx,yy);b.Translace(xx,yy);
    Tisk("Translace");}
};

int TBod::Aktualni=0;
int TBod::Celkem = 0;
int Tline::Aktualni=0;
int Tline::Celkem = 0;

void f(void)
{
    TBod bod,b2(2,4);
    TLine line,l2(1,2,3,4),l3(bod,b2);
    TLine *l4;

    l4 = (Tline*) new TBod(10,10);
    // pokud pracujeme s virtual funkcemi, pak je
    // dulezite, jak prvek vznikl a ne cemu je
    // prirazen (musi respektovat dedicnost)

    #define PRVKU 6
    TGrBase *pole[PRVKU] = {&bod,&line,&b2,&l2,&l3,l4};

    int i;
    double MinX=1e32,MaxX = -1e32;

```

```

for (i=0;i<PRVKU;i++)
{
    if (pole[i]->MinX( ) < MinX)
        MinX = pole[i]->MinX();
    if (pole[i]->MaxX( ) > MaxX)
        MaxX = pole[i]->MaxX();
}

cout << " minimum x = " << MinX << "\n";
cout << " maximum x = " << MaxX << "\n";

for (i=0;i<PRVKU;i++)
    pole[i]->Translace(3.5,-4);

MinX=1e32;
MaxX = -1e32;

for (i=0;i<PRVKU;i++)
    {if (pole[i]->MinX( ) < MinX) MinX = pole[i]->MinX();
     if (pole[i]->MaxX( ) > MaxX) MaxX = pole[i]->MaxX();}

cout << " minimum x = " << MinX << "\n";
cout << " maximum x = " << MaxX << "\n";

l4->Tisk(" line jako bod ");
    // neni virtualni a proto si mysli, ze je to line

l4->TiskV(" line jako bod virtual ");
    // virtualni se vola podle vzniku tj. bod

delete l4; // neni-li destructor virtual,
    // zavola se destruktorka od line, protoze
    // prekladac vi, ze je to line.
    // U virtual respektuje ze to vzniklo jako bod
}

int main(int argc, char* argv[])
{
    char c;

f();
    cin >> c;
    return 0;
}

```

Metoda s prototypem `F(Trida &a)` je volána `f(m)` pro `m` typu `Trida` nebo pro třídy odvozené (a není možno rozlišit)

KO: co jsou abstraktní báze třídy?

Příklad 3.3.4a: Navrhněte a realizujte báze třídu pro prostorové objekty, která bude mít (abstraktní) virtuální metody pro posun a rotaci vůči počátku, vrácení maximální a minimální hodnoty x a y kterou objekt zaujímá a metodu pro tisk. Předělejte třídy pro bod a čáru v prostoru tak aby respektovaly navrženou báze třídu.

3.4.5 Přetypování / RTTI

static_část <int> (f)
static_část, const_část, reinterpret_část, dynamic_část, - C++
C++ je #define const_část(t,e) const_část <t> (e)
v C je #define const_část(t,e) ((t)(e))
v C takhle nefunguje dynamic_část.

Operátory pro přetypování (a RTTI)

Byly zavedeny nové operátory pro přetypování a pro získávání informací o typu objektu za běhu programu.

postfixový-výraz:

...původní obsah
static_cast < jméno-typu > (výraz)
reinterpret_cast < jméno-typu > (výraz)
const_cast < jméno-typu > (výraz)
dynamic_cast < jméno-typu > (výraz)
typeid (výraz)
typeid (jméno-typu)

- doplnění k explicitnímu operátoru konverze (T)výraz

Operátory static_cast, reinterpret_cast, const_cast představují specifické přetypování - může vyjádřit totéž co (T)výraz, kromě přetypování na private báze třídu (účelem těchto operátorů je explicitně zvýraznit úmysl přetypování).

const_cast<T>(e) přetypuje e na typ T přesně jako (T)e za předpokladu, že T a typ výrazu e se liší pouze v const a volatile modifikátorech, jinak dojde k chybě. Ostatní operátory respektují 'konstantnost'

- nemohou ji měnit jinak než je povoleno pro implicitní konverze.

static_cast<T>(e) přetypuje e na typ T přesně jako (T)e za předpokladu, že existuje implicitní konverze z T na typ e nebo naopak, jinak dojde k chybě. Poznamenejme, že static_cast poskytuje výsledky, které se obvykle již dále nemusí přetypovávat.

reinterpret_cast<T>(e) přetypuje e na typ T za předpokladu, že je dovoleno (T)e, jinak dojde k chybě. Ukazatele a reference uvažuje jako neúplné typy (tj. vztahy báze/odvozená třída neovlivní význam přetypování), jinak je význam tohoto operátoru stejný jako (T)e. Poznamenejme, že reinterpret_cast poskytuje výsledky, které se obvykle musí dále přetypovávat na svůj originální typ (například ukazatele na funkce - není bezpečné volat funkci přes ukazatel na funkci s jinými typy argumentů než má definice funkce). Například:

```
void f(char *p) { *p = 'x'; }  
typedef void (*FP) (const char*);
```


FP p = static_cast<FP>(&f); // ???
volání f přes p nemusí fungovat (v tomto souhlasí s ISO C)
Výsledek dynamic_cast<T>(v) je typu T. Typ T musí být ukazatel nebo reference na definovanou třídu nebo void*.

Je-li T ukazatel a v je ukazatel na třídu, pro kterou T je dostupnou базovou třídou, výsledek je ukazatel na unikátní podobjekt této třídy.

Je-li T reference a v je objekt třídy, pro kterou je T přístupná базová třída, potom výsledkem je reference na unikátní podobjekt této třídy. Jinak v musí být ukazatel nebo reference na polymorfní typ (tj. musí mít alespoň jednu virt. metodu).

Je-li T void*, potom v musí být ukazatel a výsledek je ukazatel na kompletní objekt (na který ukazuje v).

Jinak je provedena (run-time) kontrola za běhu programu, zda hodnota v může být konvertována na T. Když ne, operace skončí neúspěšně - výsledná hodnota po přetypování bude 0. Přetypování reference v takovém případě vyvolá výjimku bad_cast.

Kontrola za běhu programu probíhá takto:

Když objekt na který se odkazuje v je objekt reprezentující базovou třídu objektu na který odkazuje T, výsledkem je ukazatel na tento objekt. Jinak se hledá úplný objekt na který v odkazuje. Má-li takový objekt unikátní public базovou třídu typu T, výsledkem je ukazatel (reference) na objekt reprezentující базovou třídu. Jinak je kontrola neúspěšná.

Příklad: (musí být zapnuto RTTI)

```
class B { /* min. jedna virtuální metoda */ };  
class D : public B { /* ... */ };  
B *pb1 = new B;  
B *pb2 = new D;  
D *pd1 = dynamic_cast<D*>(pb1); // 0 (nelze přetypovat)  
D *pd2 = dynamic_cast<D*>(pb2); // ukazatel na objekt třídy D
```

3.4.6 Šablony – generické programování

Cílem je ukázat princip šablon, který je vhodný pro tvorbu předloh pro vznik metod nebo tříd pro konkrétní typ. Metoda nebo typ se deklarují pro obecný typ a podle předlohy jsou v případě použití vygenerovány příslušné metody.

vše, tj. kód i prototypy funkcí je napsáno v *.h

šablona je přepis funkce, kterou překladač vytvoří, je-li potřeba

píše se pro obecný (neznámý) typ, s danými vlastnostmi využívanými v šabloně

```
template <class T> T max ( T &h1, T &h2 )  
{  
    return (h1>h2) ? h1 : h2 ;  
}
```

Nad objektem T vytvoří funkci max s parametry.

V tomto případě může být T například typ float. Typ T musí mít nadefinovaný operátor > .

Funkce se vytvoří až tehdy, narazí-li překladač v kódu na `f = max (9.4 , 3.5)`, přičemž nikde nenašel funkci `max`.

Pokud byde zavoláno nejdřív `max(9.4 , 3.5)` a potom `max (9.4 , 3)`, bude vše v pořádku a parametr 3 se zkonvertuje na 3.0 . Pokud by však bylo volání v opačném pořadí, zahlásil by překladač chybu, protože předpis `max (float, int)` není definován.

Lze obejít šablonou `template double max (double, double);`

```
př. template    < class T, class S >
      double max ( T h1, S h2 )
```

`template <class T, int nn=10>` ..může být i konkrétní typ včetně implicitního inicializování

```
template    < class T >

      class A {
          T a , b ;
          T fce ( double, T, int )
      }

      T T<class T>:: fce (double a, T b,int c) {}
```

potom
`A <double> c, d;` bude c,d třídy A, kde a,b jsou double
`A <int> g, h;` bude g,h třídy A, kde a,b jsou int

Poznámka:

Pro specifikaci, že jde o jméno typu získaného z parametru šablony se používá klíčové slovo `typename`. Například:

```
template<class T> class X {
    typedef typename T::InnerType Ti;
        // synonymum pro typ uvnitř T
    int m(Ti o)
        { ..... }
}
```

Přednost při vyhledávání má nešablonová metoda. `Void Tiskni (Tmatrix &m)` přednost před specializovaným `template` (definovaným jako výjimka ze šablony pro “neobecný typ” (tj. není nad obecným ale nad konkrétním typem) `template <> void Tiskni <Tmatrix> (Tmatrix &m)` přednost před (obecnou) šablonou `template <class T> void Tiskni(T&m)` lze zjednodušit na `template <> void Tiskni (Tmatrix &m)`

Místo class se u šablon nověji zavádí vhodnější typename.

Šablony lze přetěžovat.

Jsou-li vícenásobné , je nutné přidat mezery mezi znaky >> aby nebyly interpretovány jako operátor posunu. Např 2D pole vektor <vektor <double> >

KO: jaký je princip šablon. K čemu slouží a jak se definují.

Příklad 3.3.5a: Navrhněte, napište a vyzkoušejte šablonu pro získání minimální hodnoty ze čtyř prvků. Srovnajte (i vzájemně) s makrem, inline funkcí a funkcí.

3.4.7 Výjimky

Cílem kapitoly je seznámit s novým principem řešení chybových stavů v programu. Slouží k tomu objekty výjimek, které umožňují řešit vzniklé chyby mimo celky ve kterých vznikly. Objekt výjimky v sobě obsahuje i typ chyby.

Výjimky – try , catch, throw

výjimku hodí výjimku a tu zachytí nějaký řešič

v knihovně dojde k chybě -> jak se zachovat, zvláště když je knihovna dodávána jako celek a má např. mnoho funkčních zanoření.

Funkce místo chyby “hodí” výjimku a doufá, že ji někdo zachytí. Funkce, která chce odchytávat výjimku to musí nějak dát najevo

catch může být použito jen po bloku začínajícím try nebo za jiným catch

Výjimka zachycení se hledá skrz předchozí funce (propadává a ukončuje funce), dokud ji někdo nezachytí – po této cestě se volají destruktory lokálních objekt

výjimku řeší pouze nejbližší catch handler, který je v cestě

není-li výjimka odchycena, je program terminated

házený není typ ale objekt

výjimka – exception

slouží k vyřešení chyb, které se dříve řešilo ukončením programu, nastavováním flagů, ignorováním, ...

try – slouží k určení začátku bloku ve kterém se budou výjimky obsluhovat

throw – slouží k vypuštění – inicializaci výjimky

catch – slouží k řešení výjimky

ve třídě (např. T), které se výjimka týká (? – může být i mimo?) se nadefinuje třída výjimky class V {...}. Při chybě se potom vytvoří objekt výjimky pomocí throw V(); Aby to mělo smysl, musí se toto stát v bloku, který

je obklíčen

```
try { ... } catch(X:V) { sem se dostanem jen když byl vytvořen objekt  
výjimky } catch (X:V2){}
```

Výjimku vyřeší první příslušné catch a tím ji zruší, další v řadě (více bloků s chytáním v sobě) již tedy nemá smysl. Může chytat i více výjimek. Nemusí chytat všechny výjimky. Lze i postupné řešení typu

```
catch (X:V) { ... catch (X:V1) {} }
```

catch může mít parametry typu T, const T, T &, const T& a zachycuje výjiku stejného typu, typu zděděného, pro T ukazatel, musí se dát zkonvertovat na T

Pomocí konstrukturu objektu výjimky s parametry je možné do objektu výjimky dostat informace o tom co chybu způsobilo a tyto informace vyhodnotit v sekci catch.

pokud uvedeme catch (...) obslouží se v následujícím bloku všechny neobsloužené výjimky

Nevýhodou při propadávání výjimek ke catch je to, že se provádí jen standardní odalokování (tj. lokální proměnné) a neprovádí se uvolnění ukazatelů (paměť, handly ...). Z toho plyne, že je lepší používat objekty lokální, které v sobě tuto činnost zahrnují. Dalším řešením je odchycení výjimky v každé další funkci, ošetření ukazatelů či jiného a opětovné hození stejné výjimky

void f() throw (v1,v3,v4) {} říká, které výjimky může funkce hodit, (což usnadní při psaní úvahy o tom, co máme vlastně chytat) ostatní potom nemůže hodit. Bez throw může hodit jakoukoli výjimku f(){}. Funkce bez parametru v throw nemůže hodit žádnou výjimku f() throw(){}

Neobsloužená výjimka nakonec zavolá funkci terminate (lze předefinovat pomocí set_terminate), která ukončí program

V standardních knihovnách je možné najít několik standardních výjimek

Pozn.: výjimky prodlužují a zpomalují kód

Doprava chyby z místa kde vznikla do místa kde se dá řešit s bezpečným průběhem.
Možná řešení

Vracení hodnot – vrací-li se číslo, nemusí existovat číslo pro chybu (není vhodné).

Ignorovat chyby – není vhodné

Chybová (globální) proměnné – není vhodné (nutno vědět, testovat) pro každou chybu

Zkončit – nešťastné

Chybová hláška (vhodné pro “úpravu” HW uživatelem (papír v tiskárně))

V výjimka – po hození se neplní návratové hodnoty

Vyvolá se objekt, zachycuje třída

Odchycení odkazem – bazová a zděděné třídy

Výjimka vyvolaná destruktorem, který ruší objekt v oblasti výjimky ukončí program – rušení pomocí destruktůrů, nepoužívat ukazatele

Použit pouze v situaci, která se předem nedá očekávat a je nutná řešit jinde

Výjimka v konstrukturu nevolá příslušný destruktör

KO: jaký je mechanismus výjimky a k čemu ?

Příklad 3.3.6a: přidejte možnost ukládat data z pole bodů a line do souboru. Pokud soubor nelze otevřít, oznamte to výjimkou. Danou výjimku ošetřete.

3.4.8 Informace o typu za běhu programu

Cílem je posat mechanismus, jak za běhu programu zjistit skutečný typ proměnné. Jazyk C++ zavádí možnosti jak zjistit typ proměnné za běhu programu.

Za běhu programu je možné zjistit typ pomocí klíčového slova typeid. Hlavní použití je v souvislosti se třídami a děděním, zvláště pak v případě virtuálních funkcí, kdy je nutné rozlišit mezi sebou třídy se společným chováním.

Operátor **typeid**

Výsledek **typeid(výraz)** je typu `const type_info&`

`type_info` reprezentuje popis typu výrazu

`výraz` je reference na polymorfní typ nebo `*ukazatel` (je-li `ukazatel 0`, pak dojde k výjimce `bad_typeid`)

Příklad 3.1.17 typ za běhu programu

Napište třídu, která bude demonstrovat zjištění typu za běhu programu

```
class X {           // polymorfní třída
    // ...

    virtual void f();
};

void g(X *p)
{
    typeid(p);      // type_info (X*)
    typeid(*p);     // type_info pro X nebo její následnickou třídu
}
```

Třída `type_info` je deklarována v `<typeinfo.h>` například takto:

```
class type_info {
    // implementačně závislá reprezentace

private:
    type_info(const type_info&);
}
```

```

    type_info & operator= (const type_info&);

public:
    virtual ~type_info();

    int operator==(const type_info&) const;
    int operator!=(const type_info&) const;
    int before(const type_info&);
    const char *name() const;
};

```

Pořadí definované metodou before je úplné a platné pouze po dobu běhu programu. Není definován vztah mezi dědičností a before. Účelem before je možnost řadit objekty type_info.

Typeid – pro správný výsledek musí mít třída alespoň jednu virtuální funkci. Explicitní identifikace (tj. nerozliší se pomocí parametrů base a zděděné třídy, které se při volání chovají stejně, nebo volají totéž) je většinou špatně tj. používat co nejméně

KO: k čemu slouží získání informace o typu za běhu programu?

3.4.9 Standardní knihovny

Cílem kapitoly je upozornit na standardní knihovny dodávané s prostředím, které realizují některé základní programátorské přístupy a zjednodušují tedy programování.

jsou knihovny dodávané s prostředím, kde jsou např. kontejnery, zásobníky, ... a funkce pro práci s nimi

KO: projděte si standardní knihovny (STL). Zjistěte které má Váš překladač dodány a podívejte se dovnitř jak jsou tvořeny.

3.4.10 C v C++

Cílem kapitoly je upozornit na rozdílné volací konvence v modulech jazyka C a C++ a tedy nutnosti ošetřit kolize při volání z jednoho jazyka do druhého.

při volání funkcí psaných v jazyce C z C++ je nutné v H souboru uvést

```

#ifdef __cplusplus
extern "C"
#endif

{
    float fce(int);
}

```

z důvodu odlišnosti způsobu volání fcí. Z toho také plyne, že při práci s ukazateli na funkce se musí pracovat odlišně.

`int (*pf) (int i) ;` je potom C++ volací konverze a pro C ukazatel musíme:

`extern "C" {typedef int (*pcf) (int) },` čímž máme C notaci volání. V programu potom musíme ovšem přiřazovat C ukazatelům C funkce a C++ ukazatelům C++ funkce. `pcf pc; pc = &cfun; (*pc)(10);`

Dědění je důležitou složkou objektového programování, která umožňuje sdílení kódu, jeho opětovné použití a mírné modifikace. Napomáhá k přehlednosti a snadné orientaci a údržbě.

4 Dodatky

Zde jsou dodatky k učebním textům

4.1 Výsledky testů

Zde jsou uvedeny výsledky testů z předchozích kapitol

4.1.1 Vstupní test

a)

- modulem se myslí dvojice souborů. První má příponu “c” a obsahuje zdrojové kódy (definice proměnných a funkcí). Druhý má (obvykle) stejné jméno ale příponu “h” a obsahuje nebo-li zveřejňuje (deklaruje) definované proměnné a funkce ze zdrojového souboru. Tím tvoří interface.
- základní příponou je “c” pro zdrojový text a “h” pro hlavičkový soubor. Po překladu se modul přeloží do objektového kódu přípona “o”, “obj”... Dále jsou zde knihovní soubory “lib”, “dll”. Výsledný soubor má příponu “exe”.
- návrh se skládá z analýzy problému (stanovení úkolu a chybových stavů), návrhu (tvorba dat a komunikace mezi nimi) a z implementace (napsání programu a jeho odladění).
- zdrojové texty lze přenášet, je však třeba striktně dodržovat normu. I přesto může dojít k rozdílům (např. architektura ukládání čísel, velikosti typů ...)
- základem programu jsou moduly (zdrojové texty a hlavičkové soubory) tvořené uživatelem. K vytvoření spustitelného programu je nutné ještě přidat knihovní funkce (které též oznamují svůj obsah – interface – pomocí hlavičkových souborů).

- tvorba spustitelného programu probíhá tak, že preprocesor prochází zdrojové texty (všechny v projektu, bez návazností tj. každý překládá od začátku), podle příkazů preprocesoru je upravuje a předává překladači. Ten kontroluje správnost jazyka a překládá do objektových souborů. Tyto soubory jsou společně s knihovními soubory slinkovány do spustitelného souboru.
- projektový soubor je soubor, který říká ze kterých souborů se skládá spustitelný soubor a zároveň nese informaci o způsobech překladu (přepínače prostředí)

b)

- komentáře jsou uvozeny dvojznakem `/*` a ukončeny dvojznakem `*/`, nesmí se vnořovat tj. mohou mít pouze jednu úroveň
- funkce `main` je základní a první uživatelskou funkcí spouštěnou překladačem. Vrací typ `int` a v seznamu parametrů dostává řetězec spouštěcí řádky a proměnné prostředí `int main(int arc, char *argv[],char *env[])`
- `char`, `short int`, `int`, `long int`. Jejich absolutní velikost je dána použitým prostředím a lze ji zjistit klíčovým slovem `sizeof`. Pro velikosti pouze platí, že menší typ se beze ztráty vleze do většího (v pořadí v jakém byly uvedeny).
- prototyp funkce je její hlavička, určuje jaká je návratová hodnota, jméno funkce a seznam parametrů, které se do ní předávají.
- slouží k větvení programu na základě podmínky uvedené za `if`

c)

- cykly `for`, `while`, `do-while`. Všechny cykly se opakují dokud je splněná podmínka cyklu.
- příkaz `switch` slouží k vícenásobnému větvení na základě přesné shody celočíselné proměnné s číslem větve.
- ukazatel je typ určený pro uložení adresy na kterém leží proměnná určitého typu. Ukazatel je možné inicializovat adresou již vytvořené proměnné, nebo dynamicky alokací paměťového prostoru. Práce s neinicializovaným ukazatelem je velkým rizikem. Získání adresy je možné operátorem `&` např. při inicializaci `int *uk=&i;`, přístup k hodnotě (dereference) je pomocí operátoru `*` např. `*uk = 4;`
- parametry se v jazyce C předávají vždy hodnotou
- hodnoty z funkce je možné dostat pomocí návratové hodnoty. Dále je možné předat hodnoty z funkce pomocí adres – ukazatelů. Funkci se předá ukazatel na místo, kam je třeba uložit výsledek.

d)

- ukazatelová aritmetika říká, že přičteme-li resp. odečteme-li k ukazateli celé číslo, posuneme se v paměti o počet prvků (daný tímto celým číslem) daného typu dopředu resp. dozadu. Rozdíl dvou ukazatelů (které musí být stejného typu a ukazovat na stejné pole) udává, kolik prvků daného typu je mezi nimi
- pole je množina po sobě jdoucích prvků stejného typu. Definuje se např. `int pole[10];` Indexuje se od nuly, takže maximální využitelný index je v našem případě devět.
- řetězec je pole znaků `char`, které obsahuje speciální ukončovací znak `'\0'`

- struktura je složený datový typ, který může obsahovat různé typy. Přístup k vnitřním složkám objektu je přes operátor “.” (tečka), přístup k prvkům struktury dané ukazatelem je přes operátor “->”. Velikost v paměti se zjistí pomocí sizeof. Obecně neplatí, že celková velikost struktury je součtem velikostí jejích položek (může být rovna nebo větší).
- union je složený datový typ (s vlastnostmi obdobnými jako má struktura), který ovšem může obsahovat současně pouze jeden prvek. Má velikost největšího obsaženého prvku.
-

4.1.2 Kapitola 3.1

Odpovědi na otázky najdete v předchozím textu. Řešení “neřešených” příkladů není k dispozici, protože je nutné programování “zažít” a to je možné pouze samostatným programováním. Předložené řešení dále nemusí být jediné, správných řešení může být několik.

4.1.3 Kapitola 3.2

Odpovědi na otázky najdete v předchozím textu. Řešení “neřešených” příkladů není k dispozici, protože je nutné programování “zažít” a to je možné pouze samostatným programováním. Předložené řešení dále nemusí být jediné, správných řešení může být několik.

4.1.4 Kapitola 3.3

Odpovědi na otázky najdete v předchozím textu. Řešení “neřešených” příkladů není k dispozici, protože je nutné programování “zažít” a to je možné pouze samostatným programováním. Předložené řešení dále nemusí být jediné, správných řešení může být několik.

4.2 Seznam příloh

Příloha 1 Jazyk C (kapitola 5)

Příloha 2 Příkladová část (kapitola 6)

5 Příloha 1 - Jazyk C

5.1 Úvod

"C je jazyk mocný, tajemný a nevyzpytatelný"

"C je síla, která se dodává bez záchranných pásů"

5.1.1 Předmluva

Tento text společně s částí obsahující příklady by měl umožnit zvládnutí základů programování v jazycích C, C++ a jejich praktické ověření a zažití při tvorbě jednoduchých programů.

Základem pro tvorbu těchto textů jsou zkušenosti nabyté při výuce jazyků C, C++ a také praktické zkušenosti z programování aplikací pro průmysl.

Při programování a zvláště pak při výuce se ukázalo, že pouze výuka jazyka k úspěchu nestačí. Je nutné mít základní přehled o dalších souvislostech a poznatcích z oblasti programování, které přímo nesouvisejí s jazykem, ale jsou důležité pro plnohodnotné využití jeho možností. Některé z těchto postřehů jsou spolu s dalšími zařazeny do této úvodní kapitoly. Další kapitoly jsou věnovány jazyku C a jazyku C++. Příklady jsou uvedeny ve zvláštním textu, který má obdobnou strukturu.

Slovo praktické v názvu kurzu vzešlo z toho, že tento kurz byl kurzem volitelným, určeným především pro studenty, kteří vypracovávali své projekty, či diplomové práce v tomto jazyce, a jejich praktické problémy byly základem práce na přednáškách i cvičeních. Cílem bylo co nejlépe využít možnosti jazyka pro daný problém. Z práce a studia těchto studentů a ze srovnání s novějším povinným kurzem vyplývá, jak velice důležitým faktorem pro studium je motivace, či možnost praktického ověření a uplatnění nabytých poznatků vycházející ze strany studujícího. Po převodu kurzu na povinný klesl počet absolventů kurzu z více než 90% na méně než 50%, a to přesto, že dosažené znalosti jsou shodné a povinný kurz navazuje na minulé semestry výuky základů jazyka C.

Z těchto zkušeností plyne doporučení pro začátek kurzu výuky jazyka – zvolte si cíl z oblasti vašich zájmů, který budete realizovat (např. slovníky, kartotéky, či hry – piškvorky, šachy apod. Pro proniknutí do tajů programování a programovacích mechanismů se jako jedno z nejlepších témat jeví naprogramování si vlastního překladače). Při této práci patrně zjistíte, že je rozumné si úkol nejdříve důkladně promyslet (stanovit si cíl minimální a maximální, včetně kroků mezi nimi – program začne brzy fungovat a s používáním jeho jednoduché verze můžete čerpat poznatky pro jeho další rozšiřování) a rozvrhnout si strukturu proměnných, funkcí ... Teprve poté co máte přehled o návaznostech jednotlivých modulů je čas začít s vlastní tvorbou programu. I přes přípravu se často zjistí, že lze vylepšovat program na základě práce s ním, na základě zkušeností při jeho využívání. U C++ vidím tuto motivaci ve vyřešení (spíše více než jen jednoho) projektu jako nutnost – C++ je nutné vyzkoušet a zažít.

Jak bylo uvedeno výše, přemýšlení a tvořivost je nedílnou součástí výuky jazyka. Pouhé naučení klíčových slov ještě nezaručuje úspěch. Klíčová slova patří k základním kamenům, ke kterým patří ovšem také jejich vlastnosti a možnosti jejich využití (rozsahy a typy proměnných,

způsoby volání funkcí, práce s pamětí ... - což jsou oblasti, o kterých se ve většině helpů nedočtete, ale které jsou bezpodmínečně nutné k úspěchu). Teprve s těmito znalostmi je možné hovořit o programování (programovací techniky, algoritmizace), které je základem pro využití jazyka k řešení prakticky nekonečné skupiny možných příkladů (aplikací). Právě široký rozsah možných příkladů je nepříjemný v tom, že se programování nejde mechanicky naučit. Jakýkoli vypracovaný vzorový příklad může být vodítkem, ale pouze samostatná práce a objevování souvislostí vede ke zvládnutí samostatného programování. Z toho plyne snaha koncipovat příklady k procvičení spíše jako rozsáhlejší celky než jako doplňování těl funkcí.

Tento text je psán spíše z pohledu praktického uživatele než ortodoxního vyznavače jazyka – v tomto případě odkazují na volně přístupnou normu jazyka. Pro ty, kterým styl tohoto textu nevyhovuje, doporučuji návštěvu knihovny či některého z knihkupectví, kde se objeví nová učebnice téměř každý druhý měsíc. A z těch dostupných je možné si vybrat způsob výkladu, který nejlépe vyhovuje. Obsah tištěných textů je vesměs srovnatelný – pokrývá látku ve stejné šíři. V neposlední řadě je možné získat velké množství textů na webu (není je možné brát bezmyšlenkovitě, protože se zde vyskytují i velice neseriózní až matoucí texty, popřípadě texty zaměřené na konkrétní překladače, které nemusí plně odpovídat normě). Patrně žádný z nich však nezaručí, že se C, C++ dá naučit snadno, rychle a pouhým jejich přečtením. Jelikož každý z autorů je ovlivněn svým oborem činnosti a svými zkušenostmi může být i jeho pohled a interpretace použití jazyka jednostranná. Z tohoto důvodu je lépe o doporučeních autora uvažovat spíše než je bezhlavě přijímat.

Jazyk C slouží v navazujících kurzech především k programování jednoduchých obvodů (mikroprocesory, signálové procesory, mikrokontrolery, průmyslové automaty). Jazyk C++ slouží k realizaci Semestrálních a Diplomových prací.

5.1.2 V čem programovat

Rozhodneme-li se pro práci s jazykem, je nutné si vybrat vývojové prostředí, ve kterém budeme pracovat. K výběru můžeme být přinuceni úlohou nebo skupinou ve které pracujeme. Dalším hlediskem může být operační systém, ve kterém se bude pracovat. Jiným požadavkem jsou vlákna (více částí jednoho programu běžícího a pracujícího současně), přesné časování, odezvy na vnější podněty v určitém časovém intervalu, snadná tvorba grafického rozhraní, návaznost na databáze atd.

Je-li již část práce hotova, je asi nejlepší volbou pokračovat v již zvoleném prostředí a měnit ho pouze po zralé úvaze. Je-li ovšem program dobře koncipován, potom jeho jádro napsané podle normy jazyka by mělo být plně přenositelné bez úprav a změny by se týkala "pouze" komunikace s uživatelem a HW.

Pracujeme-li s HW zařízením, potom (v případě, že si to nejsme schopni napsat sami) musíme brát zřetel na to v jakém prostředí (operační systém a překladač) jsou dodány knihovny, drivery nebo části kódu vhodné pro spolupráci se zařízením.

Z dostupných vývojových prostředí je možné volit jak zdarma šířené, kde nejznámější je patrně GNU C s verzí pro Microsoft platformu DJGPP, tak placené (Microsoft, Borland ...), které však pro nekomerční využití mají rozumné ceny. Hlavní starostí při volbě prostředí by měla být jeho schopnost dodržovat normu jazyka. Při výběru překladače je nutné ověřit, zda je schopen pracovat s poslední (či alespoň novější) normou, kde zvláště v C++ došlo k řadě změn.

Jelikož pro učení se programování je vhodné mít co nejméně "nepřátel", je asi nejvhodnější pro začátek zvolit tzv. konzolovou aplikaci. Je zde vidět celý zdrojový kód, knihovní funkce včetně vstupů a výstupů jsou jednoduché. Jak se ukázalo množství dnešních studentů již neví co si pod tímto názvem představit a jak s tím pracovat. Jedná se o "černou

obrazovku” známou z DOS, či LINUX. Vytváří se zde programy, které komunikují v textovém režimu – vstup z klávesnice, výstup textu na monitor. Zde bych chtěl upozornit, že Windows nejsou jediný operační systém a už vůbec ne systém ideální pro univerzální použití.

Výhodou konzolové aplikace je to, že se dá napsat program tak, aby byl jasný tok programu. Spolu s možností přesměrování vstupu a výstupu dále umožňuje reprodukování spouštění programu včetně opakování výskytu chyb a tedy dobré podmínky pro ladění i tvorbu programu.

Další výhodou konzolové aplikace je, že ji obsahuje prakticky jakékoli vývojové prostředí pro C a zdrojový text pro ni napsaný je velice dobře přenositelný (na úrovni, kterou budeme pro základy programování v tomto textu používat jsou si prostředí rovnocenná).

Novější prostředí (Borland Builder, Microsoft Visual C, ...) umožňují zvláště pro tvorbu grafického rozhraní využít předdefinované funkce nebo třídy a to dokonce stylem chytň a táhni. ”Programátor” zde dostává hodně věcí ”zadarmo”, což může vést k přehnanému uspokojení a optimismu nad jeho schopnostmi – při skutečné nutnosti použít C++ pak může přijít veliké rozčarování nad ”složitostí” jazyka. Dále se zde tvoří část kódu a hlavičkové soubory skryté na pozadí a část funkcí je skrytá úplně. Jsou totiž zobrazeny jen funkce měněné programátorem. Mimo ně je přítomna celá řada funkcí a operací, které nemají ”viditelný” kód, protože do této části zdrojového textu se nic nedoplnilo. Např. využíváme-li okno, obsahující text, vytvoří se pro něj hlavičkový soubor ale funkce pro vytvoření, vykreslení atd. se sice používají ale ve zdrojovém kódu se neobjeví. Dále zde je funkce, která sleduje pohyb kurzoru myši nad oknem, která opět není vidět ale používá se. Až v okamžiku, kdy programátor do této funkce přidá svou část kódu, objeví se ve zdrojových textech. Hlavička funkce s prázdným tělem se ve většině prostředí dá vybrat ze seznamu funkcí, nebo se najde prototyp v helpu.

Začínat s těmito prostředím učení může být příjemné, protože program rychle přibývá a i jeho funkčnost rychle narůstá. Cenou za to je, že začínající programátor netuší co se vlastně při tvorbě programu děje a při výskytu chyb ani neví kde je možné je opravit. Tyto nevýhody pramení z již zmíněné ”pomoci” na pozadí, kdy část kódu není vidět a část se tvoří automaticky, což je pro výuku a pochopení nevhodné. Pomocníky jsou komponenty, které je nutno při složitějších aplikacích upravit, či rozšířit a potom se musí zdědit a upravit, k čemuž je již nutná znalost programování v C++. Každopádně je nutné si uvědomit, že programovat v Builderu resp. Visual C++ neznamená (umět) programovat objektově. Tlačítka a jiné objekty jsou realizovány pomocí objektů, vlastní doplňování funkčnosti již nutně objektově být nemusí.

Často také dochází ke srovnávání C, C++ s ostatními jazyky. Zde bych chtěl podotknout, že základní mechanismy programů jsou totožné z valné části (cca 90%). Odchyly jsou pouze v detailech. Nejpodstatnějším rozdílem je to, že se příkazy provádějící totéž jinak jmenují, a je zde také rozdíl ve volnosti/přísnosti, s jakou je nutné srovnatelné vlastnosti/omezení jazyka dodržovat. Něco je v C jednodušší, něco možná složitější, ale řádové rozdíly oproti ostatním jazykům tohoto typu nečekejte.

5.1.3 Stručná charakteristika C

Jazyk C je programovací jazyk nízké úrovně umožňující strukturované programování s velmi efektivním a rychlým výstupním spustitelným kódem. Jeho základem je 32 klíčových slov (typy celočíselné a pracující v plovoucí řádové čárce, složený typ, podmíněné příkazy a příkazy cyklů, standardní matematické a logické operátory včetně možnosti úsporných zápisů). Důležitý je mechanismus volání funkcí, především předávání parametrů, které je u C možné

pouze hodnotou. Dalším významným prvkem je práce s adresami (ukazatele, anglicky pointery), která je mocným nástrojem pro práci, ale také výrazným zdrojem chyb.

Jazyk C nemá větší typovou kontrolu, či kontrolu mezi polí. Samostatný jazyk C neobsahuje povely pro vstup a výstup, jsou realizovány pomocí knihovních funkcí (jsou součástí normy jazyka).

Jazyk C se využívá v jednoduchých aplikacích, pro programování jednoduchých zařízení (signálové procesory, ...), pro tvorbu "recyklovatelných" funkcí, které budeme chtít využít i v těchto zařízeních a vůbec pro lepší přenositelnost mezi platformami (ANSI/ISO C). Jedná se především o low-level funkce, ty, které zpracovávají data nebo tvoří vrstvu ovladačů konkrétního HW (Hardware Abstraction Layer).

Zde bych chtěl upozornit, že většina dnešních překladačů jsou vlastně překladače dva. Překladač C a překladač C++. To, kterým se překládá, většinou závisí na příponě překládaného souboru – např. přípona ".c" vyvolá překladač C, pro C++ je potom určena přípona ".cpp". U některých překladačů je možné pomocí přepínačů říci nejenom zda překládat pomocí C či C++, ale i typ C (většinou je možné zvolit "volnější" překlad nebo překlad striktně podle normy. U překladačů C je možné ještě volit překlad K&R, což je původní norma C z roku 1978 pojmenovaná po autorech Kernighanovi a Ritchiem. Od současného C se liší především zápisem předávání parametrů funkcím a svou jednoduchostí).

5.1.4 Stručná charakteristika C++,

Jazyky C a C++ je třeba chápat odděleně. Nejedná se pouze o historické souvislosti, kdy je C předchůdcem C++. Důležitou vlastností zůstává to, že existuje celá řada aplikací, převážně v programovatelných obvodech, kde je použití jazyka C výhodnější. A tak, i když je C podmnožinou C++ a většina překladačů C++ umí, má stále smysl rozlišovat mezi C a C++ z důvodu přenositelnosti na platformy, či do programovacích prostředí, které C++ nepodporují. Proto uvádíme vlastnosti jazyka odděleně v kapitole 2 pro C a v kapitole 3 pro C++, aby bylo zřejmé, o které rozdíly se jedná.

Jazyk C++ sebou přináší rozšíření o neobjektové a objektové vlastnosti – rozšíření jazyka C. Přidává dalších 31 (na celkových 63) klíčových slov. Neobjektové vlastnosti jsou určeny ke zlepšení standardního programování, jako je možnost předávání hodnoty odkazem či překrývání názvů funkcí. Vlastnosti objektové dávají programování zcela nové možnosti stylu programování. Jedná se především o práci s datovými celky, možností ovládat přístupová práva k těmto datům a pracovat s nimi.

Objekty mají i výrazné možnosti kontroly vzniku a zániku objektu a možnost přetížení operátorů. Je rozšířena i možnost vstupu a výstupu, která je stále knihovní záležitostí realizovanou přetížením operátorů posunu, ale již je spojena s konkrétním typem.

C++ využíváme pro tvorbu prostředí, práci se složitými datovými strukturami, vizualizaci, při tvorbě kódu, který má společný základ a bude se využívat v několika (drobných) modifikacích ...

Při programování či studiu je nutné nezaměňovat C++ s objektově orientovanými překladači (BORLAND Builder, Microsoft Visual C ...), které sice značně využívají objektových vlastností, mají na objekty značnou vazbu, ale "tahání" ikonky by šlo realizovat i bez objektů, i když značně pracněji. Objektově programovat neznamená, že část aplikace "naprogramuje" překladač/prostředí na základě natahání ikonky. Objektovým programováním rozumíme návrh objektů, jejich dat, metod, interface, jejich návaznosti a

využití (což překladač/prostředí může zpříjemnit např. při grafické reprezentaci dat, ale jádro zůstává na programátorovi).

Objektové programování umožňuje logicky oddělit vlastní data a metody s nimi manipulující od zobrazení, vizualizace. K vizualizaci využíváme nadstavbové třídy určené k zobrazení dat. Známým je např. model DOC/VIEW (model / pohled), kdy dokumentem se myslí třída pro data a práci s nimi a view se myslí třídy, které zobrazují uvedená data. Těchto "pohledových" tříd může být několik pro jedna data (DOC). Např. u textu to může být pouze text, formátovaný text, text v barvách, nebo četnost slov či písmen ... U obrázků např. zobrazení barevné, černobílé, v číselných hodnotách pixelů, připravené pro černobílý tisk ... U souborů to může být jejich reprezentace v binární či textové formě, formátované zobrazení, statistika znaků nebo slov ... U řady čísel – graf, statistika, čísla jako tabulka, proložená funkce, ...

5.1.5 Událostmi řízené programování

Při "klasickém" (imperativním) stylu programování se předpokládalo, že běží základní větve programu, ze které se volají podprogramy, plnící požadovanou činnost. Většinou docházelo k jejich volání v logickém sledu a jejich vzájemné volání často předpokládalo, že předchozí část proběhla úspěšně a data jsou platná.

Návrh takového systému byl jednodušší než návrh programu řízeného událostmi, či dokonce programu, který umožňuje vícenásobné spuštění částí programu pracujících nad stejnými daty (thready - vlákna). Při programu řízeném událostmi přicházejí události, které definují požadavky co a s jakými daty se má stát. Je proto potřeba ošetřit všechny možné stavy. Při vícenásobném spuštění výkonné části programu se může stát, že s daty bude naráz pracovat více podprogramů, či dojde k vícenásobnému současnému využití stejných dat pro různé procesy a tedy je nutné koncepci návrhu propracovat ještě lépe – např. o prioritní přístup, zamykání dat, nepoužívání globálních proměnných Připravit se na takovýto provoz algoritmů již v samém počátku rozboru úlohy a návrhu funkcí a datových celků však nemůže být na škodu a svědčí o zkušenostech programátora.

Dobře napsaná funkce by měla spolehlivě pracovat nezávisle na posloupnosti volání. S tím je spojen rozbor a ošetření chyb na vstupu a výstupu funkce. Při vstupu se kontroluje stav proměnných, které musí mít platná data a nesmí být v chybových kombinacích. Při chybě na vstupu i chybě na výstupu je nutné navrhnout mechanismy oznámení chybového stavu dále. K předání chyby slouží nejčastěji návratová hodnota funkce nebo nastavení globální chybové proměnné, v C++ mechanismus výjimek. Při nutnosti používat dále proměnnou, u které se vyskytla chyba, se tato nastaví do takového stavu, aby program mohl pracovat s co nejmenším počtem chyb v další činnosti.

5.1.6 Odchyly C a C++

Z výše uvedeného (5.1.3, 5.1.4) je zřejmé, že C je podmnožinou C++. Ve většině případů to bude platit. Bohužel však norma C někdy předejde poslední úpravě normy C++ a tak se může stát, že některé novější vlastnosti nejsou z C do C++ přenositelné. Profesionální programátor by na tyto případy měl být připraven, starší standardně napsaný kód by neměl být problémem.

K "odchylkám" je nutné připočítat i platformovou závislost jazyka C a C++, kdy nejsou pevně definovány velikosti typů a též některé operace mohou být rozdílně interpretovány pro různé platformy (např. posuny u jednoduchých procesorů)

Další odchylky mohou být způsobeny "specifickými" vlastnostmi danými autory překladačů. Jedná se o překladače, které se nedrží normy (předbíhají normu zaváděním nových vlastností, které se do normy nedostanou či dostanou pozměněné) nebo jsou určeny pro jednodušší zařízení, kde není možné realizovat všechny vlastnosti jazyka. Některé překladače např. nemusí část kódu přeložit korektně a při špatném nastavení warningů ani nezahlásí, že tak učinili. Např. u jednoduchých procesorů nemajících násobení se stává, že části kódu obsahující násobení jsou při překladu ignorovány (řešením je koupit si příslušnou knihovnu funkcí nebo si násobení pomocí funkce naprogramovat sami).

5.1.7 Návrh programu

Existuje řada cest, jak dojít k fungujícímu programu. Doporučuje se, aby vlastní návrh programu obsahoval např. následující fáze v daném pořadí: analýza -> návrh -> implementace. To odpovídá postupnému zjednodušování - metoda návrhu shora dolů.

Prvním bodem návrhu by měla být analýza úlohy, kdy jsou zjištěny a definovány úkoly a podúkoly, jsou stanoveny vstupní a výstupní hodnoty a probrány chybové stavy se způsobem jejich ošetření. Následně jsou stanoveny součásti (bloky) řešení s úkoly a návaznostmi.

Druhým bodem je vlastní návrh programu, kdy se pro bloky hledá vhodná reprezentace dat, jejich vazeb a komunikace. Jedná se o návrh datových struktur se stanovením obslužných uživatelských funkcí (interface) a funkcí pro práci s těmito daty. Dále potom návrh komunikace mezi jednotlivými datovými strukturami – bloky, kdy se stanoví např. hierarchie (podřízenost, nadřízenost, vlastnění ...) a následně datové toky a řízení procesů. Zde je nutné přihlížet k možnostem zvoleného jazyka (viz. 5.1.8).

Posledním bodem je vlastní implementace, která sestává z napsání vlastního programu a jeho uvedení do funkční a bezchybné činnosti. K tomu slouží především ladění a testování programu s daty co nejvíce odpovídajícími skutečnému provozu (5.1.9).

Až poté se dá program bezpečně používat v běžné praxi.

5.1.8 Algoritmizace

K základu programátorských schopností patří kromě znalosti jazyka i logické (a někdy i selské) myšlení spočívající především v rozvržení úlohy (data a procesy) tak, aby se co nejlépe využívalo vlastností jazyka. Jazyk je pouze nástrojem, který lze využít lépe či hůře. Jeho znalost je nutná, ale sama o sobě fungování navržených algoritmů nezaručí. Základními kameny programování je jeho strukturovanost, jeho rozdělení na menší logické celky a jejich návaznosti. Z vlastního jazyka se využívá především tvorba a použití dat, funkční volání, strukturovanost programu a práce s pamětí a jinými zdroji.

5.1.9 Ladění programů

Je-li program napsán je nutné ho odladit – tj. zajistit jeho přeložení a následně bezchybnou činnost. Jedná se zde o odstranění chyb v programu a ne v jazyce. Zkušený programátor už s přihlédnutím k této etapě navrhuje vlastní algoritmy. Při ladění je nutné vyzkoušet co největší množství stavů, do kterých se program může dostat. Jelikož je nemožné pokrýt vše, snažíme se alespoň o definici tříd popisujících standardní množiny stavů. Provedeme proto základní rozbor a pro každou ze standardních možností si vytvoříme typizované vstupní hodnoty/sekvence. Tento rozbor a modelové situace nás dále vedou ve fázi návrhu, který musí být schopen pokrýt všechny možné situace. Ke zvoleným variantám vstupů si vytvoříme typizované výstupy. S pomocí dvojic vstup-správný výstup budeme dále zkoumat

chování programu nejen při ladění ale též po jednotlivých úpravách, kdy musí odezvy-výstupy k příslušným vstupům zůstat stejné.

Vyskytuje-li se v navržených algoritmech chyba, je potřebné provést její lokalizaci, tj. určit ve kterém místě vzniká. To může být někdy dosti složité díky množství dat a jejich návaznosti. Proto se např. snažíme dosáhnout stanovení minimálního kódu, ve kterém se chyba ještě vyskytuje, pro snadnější lokalizaci chyby. Příčina chyby nemusí být vždy v místě, kde se projeví.

K ladění slouží nástroje preprocesoru a debuggeru. Nástroji preprocesoru je možné vložit do kódu "bonzáčky", které nás mohou informovat o tom, že nastala situace, která by nastat neměla. Zde ovšem musíme znát situaci, která je nevhodná. Debugger je mocnějším nástrojem v tom, že můžeme provádět krokování programu. Většina programů umožňuje provést jeden příkaz či jednu funkci nebo skočit na definovaný řádek. Zároveň máme možnost sledovat hodnoty proměnných v daných částech programu.

Ladící mechanismy prodlužují kód a zpomalují chod programu. Pro správnou funkci programu je však ladění nezbytné. Současné programovací prostředky umožňují jejich vypnutí či zapnutí (nutno uvažovat, že se může jednat o předávání pomocných informací do obj kódu a do exe kódu – tj. zařídit pro překladač i linker), a to dokonce pro jednotlivé soubory. Debugger je možné používat z prostředí, kdy při správném nastavení přepínačů a po překladu jsou do obj a exe souboru vloženy informace o původní zdrojové podobě (které prodlužují kód a zpomalují běh programu), na základě kterých jsou potom možné sledovat hodnoty proměnných a volání funkcí. Některé debugery umožňují pouze procházení exe souboru ve strojovém jazyce bez návaznosti na původní kód.

Dalším pomocníkem mohou být různé "code guard" – hlídači kódu. Opět platí že nabobtná program a zpomalí se, dojde k přidání kódu kontrolních mechanismů. Podle typu se potom kontrolují různé vlastnosti. (Např. memory management, kontrola přístupů do paměti mimo nadefinované proměnné a naalokované pole). Podle komfortu prostředí se zobrazí nejen místo, kde k chybě došlo, ale i její historie (hodnoty proměnných při volání, způsob a místo definice či alokace ...)

5.1.10 Programátorský styl (kultura programování)

K programování patří i jistá kultura programování – programátorský styl zápisu programů – který by měl usnadnit orientaci při čtení zapsaného kódu. Toto se týká především zápisu kódu, volby jmen proměnných a jejich značení, používání předdefinovaných proměnných (knihovny), uvádění komentářů a popisů souborů.

Existují doporučení, která prosazují programátorské firmy (Microsoft, Intel, Symbian, ...), které se však firma od firmy liší, ale mohou sloužit k tomu, aby si programátor udělal představu o tom, jak je možné zlepšit přehlednost svých zdrojových textů.

U psaní kódu se například doporučuje uvádět pouze jeden příkaz na řádek, zanořené bloky odsazovat. Způsob odsazování se liší v počtu mezer, poloze počátečních a konečných označení bloků. Ke kódu dále patří přehledné a srozumitelné komentáře.

Klíčová slova jsou v C psána vždy malými písmeny. Pro předdefinované proměnné, makra a konstanty je zvykem používat názvy pouze z velkých písmen. Názvy ostatních proměnných je možné psát libovolně v rámci normy jazyka. Je vhodné zachovat jednotný styl, např. je-li pojmenování víceslovní, potom oddělovat např. jen velkými písmeny na začátku slov, nebo k oddělení používat podtržítka. Někdy se doporučuje do názvu proměnné

zakomponovat i její typ, např. pro celé číslo začínat cc - ccCisloPopisne, pro text txt - txtJmeno, pro adresu, na které je celé číslo acc – accRok, ...

Při práci, kdy je nutné uvažovat více prostředí, různé překladače, či kód společný pro C a C++ se často využívá předdefinovaných proměnných. Používá se zástupných jmen a předdefinovaných maker seskupených do bloků přepínaných podle aktuální situace pomocí předdefinovaných přepínačů (jejichž jména jsou definována podle zvoleného prostředí překladače, nebo volena programátorem) tak aby se pro každé prostředí přeložila (automaticky) správná varianta. Nevýhodou takovýchto řešení je jejich nepřenositelnost mezi různými skupinami, které si vytváří vlastní varianty, a také situace, kdy nejsou hlavičkové soubory, na které jsme zvyklí, k dispozici.

5.1.11 Data v paměti, volací konvence

Důležitou znalostí pro pochopení mechanismů volání funkcí a činnosti programů je uložení dat v paměti a volací konvence funkcí.

Proměnné jsou uloženy v paměti, kde proměnné s dlouhou dobou života – globální – se nacházejí v oblasti dat (datový segment), zatímco proměnné s dobou života v rámci jedné funkce či bloku – lokální – se vytvářejí na zásobníku. Z této vlastnosti plyne, že lokálními proměnnými by měly být jen proměnné s menším nárokem na paměť (protože velikost zásobníku bývá omezená – lze ji ovšem u překladačů nastavit – ale i tak nemusí stačit). Jelikož využívání globálních proměnných se nedoporučuje (z důvodů reentrantnosti, špatné čitelnosti ...), je pro zbylé případy nutné používat dynamicky vytvářené proměnné – tj. proměnné, kterým se přiděluje paměť za chodu programu, ale z neobsazené paměti mimo zásobník.

Volací konvence funkcí vypadá obecně tak, že pro volání je nutné předat parametry do volané funkce ve správném pořadí, připravit prostor pro návratovou hodnotu z funkce, upravit vrchol, připravit lokální proměnné funkce, uklidit registry, zásobník a předat řízení funkci. Na konci funkce je nutno provést opačnou činnost, přepsat registry, předat výstupní proměnnou, vrátit se do původní funkce. Tuto činnost provádí překladač, je však nutné, aby si jednotlivé moduly rozuměly. Jelikož způsobů jak toto udělat (volající nebo volaná funkce) a jak proměnné předat (pořadí proměnných) je několik, existuje pro C a C++ jeden základní způsob volání (ale pro každý jiný) a další odvozené (umožňující např. volat funkce psané pro PASCAL). Překladač lze "přinutit" pomocí klíčových slov (např. `_cdecl`, `PASCAL` ...) k tomu, aby použil volací konvenci, která je nutná, popř. mu oznámit, že daná funkce je psána v C či C++ (extern "C", extern "C++"). To je nutné respektovat i u volání cizích funkcí v obj či knihovních modulech.

5.1.12 Přenositelnost zdrojových textů

V případě, že chceme využít práci při tvorbě programových modulů vícenásobně (v rámci různých platform či v rámci jedné platformy do budoucnosti) je nutné uvažovat o přenositelnosti zdrojových textů v C na jinou platformu (portabilita). Hlubší pochopení této kapitoly již vyžaduje znalosti vybraných vlastností jazyka C, a proto doporučujeme se s ní nyní seznámit a vrátit se k ní (jako i k ostatním "moudrostem" kapitoly 1) po zvládnutí kapitoly 2.

V této kapitole se budeme snažit lokalizovat a řešit ty problémy, které vznikají při přenosu (tzv. portaci) programu v C napsaného pro jednu platformu na platformu jinou (novou). Pod pojmem jiná platforma si lze představit jednak jiný cílový HW (např. jiný procesor: Intel, Motorola, ARM, PPC, DSP), jiný operační systém (např. Windows, Dos,

Linux, MacOS a další), jiný překladač (GNU C, Microsoft C, Borland C, Watcom C, small C) nebo také jen jiný formát uložení textového řetězce (např. UNICODE).

V ideálním případě, díky tomu, že zdrojové texty programu píšeme ve vyšším programovacím jazyce (a ne přímo v platformě závislém assembleru), by měl vždy vzniknout tzv. zdrojový text nezávislý na platformě. Realita ovšem bývá na hony vzdálena tomu, co bychom si pod pojmem platformově nezávislý zdrojový text představovali. Zdrojové texty na nové platformě většinou nelze přeložit nebo výsledný program nepracuje správně. Abychom se k ideálnímu případu platformově nezávislého zdrojového textu alespoň přiblížili, je nutné při psaní programu mít na paměti tyto pravidla:

1. Velikosti standardních typů v C mohou být pro různé platformy různé.
2. Uspořádání bajtů v typu dle významnosti (tzv. Endian) může být pro různé platformy různé.
3. Řádky v textových souborech mohou být ukončeny jinými ukončovacími znaky.
4. Zarovnání proměnných v paměti (tzv. Alignment) může být pro různé platformy různé.
5. Implementace datových typů pro uložení čísel s plovoucí desetinou čárkou může být pro různé platformy různá.
6. Maximální velikost paměti určená pro uložení automatických, statických i dynamických datových typů může být pro různé platformy různá.
7. Souborový systém může využívat jiný způsob zápisu cesty k souborům (např. obrácená lomítka).

ad 1) Velikosti standardních typů v C mohou být pro různé platformy různé.

Např. typ `int` může být definován jako 16 bitový nebo jako 32 bitový. Proto používejte vždy takový typ, který bude pro daná data vždy dostačující. Další možností je otestování velikosti typu při běhu programu pomocí operátoru `sizeof()`, nebo vytvoření maker pro nové označení typů, které bude platformě nezávislé. např. `TInt16`, `TUInt16`.

```
#ifdef __MSDOS__
    typedef int TInt16;
    typedef unsigned int TUInt16;
#elif defined (__WIN32__)
    typedef short int TInt16;
    typedef unsigned short int TUInt16;
#else
    #error Unknown platform
#endif
```

ad 2) Uspořádání bajtů v typu dle významnosti (tzv. Endian) může být pro různé platformy různé.

Odlišné typy procesorů ukládají numerické hodnoty větší než jeden bajt (např. 0x01234567) do operační paměti (resp. tedy i do souboru) jako sekvence jednotlivých bajtů v různém pořadí. Tato nekompatibilita mezi jednotlivými typy procesorů se označuje jako problém endianu nebo tzv. NUXI problém. Celkem existují čtyři typy endianů přičemž prakticky se dnes již vyskytují pouze dva. Moderní procesory navíc umožňují měnit typ endianu (např. ARM, PPC).

Ukažme si na příkladu konstanty 0x01234567 její uložení v paměti pro oba používané typy endianů.

- Little-endian platforma (Intel, Zilog)
addr: byte

0000: 0x67
0001: 0x45
0002: 0x23
0003: 0x01

- Big-endian platforma (Motorola, SPARC)

addr: byte
0000: 0x01
0001: 0x23
0002: 0x45
0003: 0x67

pozn.: V literatuře se uvádí jako typický příklad Middle-endian platformy procesor PDP-11.

Řešení je několik:

- Podmíněný překlad různých částí zdrojového textu pomocí definovaných symbolů.
- Automatická detekce typu endianu za chodu programu.
- Využití postupů zabráňujících vzniku endian problému.

Jako příklad si uveďme dvě makra, jejichž použití zabezpečuje platformě nezávislé uložení 16 bitového čísla pro oba typy endianů (pro char velikosti 8bitů se berou jednotlivé osmice bitů z míst příslušného endianu a vypočte se z nich výsledné číslo INT pro danou platformu):

```
#define GET_LITTLEEND_INT16(adr) \  
    *((unsigned char *)adr) + \  
    256 * *((unsigned char *) (adr + 1))
```

```
#define GET_BIGEND_INT16(adr) \  
    *((unsigned char *) (adr+1)) + \  
    256 * *((unsigned char *) adr)
```

ad 3) Řádky v textových souborech mohou být ukončeny jinými ukončovacími znaky.

Typickým příkladem jsou textové soubory vytvořené pod operačními systémy fy. Microsoft, kde je zvykem textový řádek ukončovat vždy dvojicí bajtů 0x0D a 0x0A.

Naproti tomu v operačních systémech UNIX je textový řádek ukončen vždy pouze jedním bajtem (0x0A).

Následující program tedy vygeneruje v různých OS různě dlouhý soubor test.txt

```
int main(void)  
{  
    FILE *file;  
    file = fopen("test.txt", "wt");  
    fprintf(file, "\n\n");  
    fclose(file);  
    return(0);  
}
```

Řešení je opět několik:

- Podmíněný překlad různých částí zdrojového textu podle typu OS.
- Automatická detekce typu ukončovacích znaků za chodu programu.

ad 4) Zarovnání proměnných v paměti (tzv. Alignment) může být pro různé platformy různé.

Některé procesory (Motorola 68000, ARM) při 16 resp. 32 bitových operacích s operační pamětí mohou přistupovat pouze k operandům umístěným na adresách, které jsou dělitelné dvěma resp. čtyřmi. U jiných procesorů (x86, AMD) je přístup k operandu na jiných adresách umožněn, ale je to spojeno s větší časovou náročností operace (tzv. overhead). Proto překladač prakticky vždy optimalizuje umístění dat v paměti na takové adresy, aby k nim bylo možné procesorem přistupovat co nejrychleji.

Mějme v programu definovanou strukturu Ttest:

```
struct Ttest
{
    int a;
    char b;
    int c;
} t;
```

potom bude (podle typu překladače) platit následující podmínka.

$\text{sizeof}(t) \neq (\text{sizeof}(t.a) + \text{sizeof}(t.b) + \text{sizeof}(t.c))$

Řešení:

Tato vlastnost překladačů jazyka C se může projevit například při alokaci paměti. Je to také jeden z důvodů, proč nepoužívat ukládání celých struktur do binárních souborů, protože při přechodu na jinou platformu nebude dodržena binární kompatibilita (tj. soubory s daty budou závislé na platformě).

Jestliže tedy potřebujeme uložit do binárního souboru celou strukturu, je vždy nutné ji ukládat po jednotlivých položkách a nejlépe ještě ošetřit typ endianu ukládané položky.

ad 5) Implementace datových typů pro uložení čísel s plovoucí desetinou čárkou může být pro různé platformy různá.

Na některých platformách (např. platforma MARM u operačního systému Symbian) není dodržena IEEE norma pro uložení čísel s plovoucí desetinnou čárkou. Pokud bychom tedy například v rámci některého binárního datového souboru sdíleli data mezi stejným programem na různých platformách, dostaneme se do problémů.

Řešení:

Optimálním řešením pro uložení čísel s plovoucí desetinou čárkou je binární podobu takového čísla vůbec nepoužívat. Pokud jej chceme uložit do souboru je vždy vhodnější používat textový zápis desetinných čísel.

Pro sdílení čísel s plovoucí desetinou čárkou v reálném čase mezi různými procesory umístěnými na jedné sběrnici (typicky PC + DSP) je z hlediska rychlosti zpracování vhodnější použití konverzních knihoven mezi jednotlivými binárními podobami daného číselného typu.

ad 6) Maximální velikost paměti určená pro uložení automatických, statických i dynamických datových typů může být pro různé platformy různá.

Operační paměť (podobně jako ostatní zdroje) není nevyčerpatelná a má své limity. Tyto limity má samozřejmě nastavena každá platforma jinak. Námi vytvořené programy (pokud chceme, aby byly snadno přenositelné) by tedy měly využívat paměť hospodárně.

Pokusíme se nyní pro jednotlivé paměťové třídy lokalizovat nejčastější možné problémy dle vlivu na přenositelnost programu a zvolit vhodné řešení:

- automatické proměnné

Tyto proměnné jsou umístěny na zásobníku (stack), proto bývá nejčastěji přenositelnost programu přímo svázaná s velikostí zásobníku. Bohužel limit velikosti zásobníku bývá ze všech tří typů paměťových tříd nejpřísnější. Obecná zásada tedy zní: **omezit vznik rozsáhlých typů u automatických proměnných (pole, struktury) a jejich předávání hodnotou do volaných funkcí** (kde budou vznikat jejich kopie). U jakéhokoli typu automatické proměnné, která má velikost nad 500 bajtů je vhodné se zamyslet, zda by nebylo vhodnější ji alokovat dynamicky a pracovat pouze s odkazem.

Další problematickou programátorskou technikou z hlediska velikosti zásobníku je využívání rekurzivních algoritmů. **Každý rekurzivní algoritmus, u kterého nemá programátor představu o počtu zanoření představuje obecně hrozbu nestability programu.** Navíc pokud je volání rekurzivní funkce kombinováno s předáváním rozsáhlých typu automatických proměnných hodnotou, narůstá potřebná velikost zásobníku do většinou programátorem netušeného rozsahu.

Při psaní programů využívající rekurze by si měl programátor vždy zodpovědět otázku, zda nelze úlohu řešit jinými metodami (dekompozice, iterace), které nejsou tolik paměťově náročné. Pokud je využití rekurze nutností, měl by si programátor utvořit představu kolik zanoření bude potřebovat pro nejhorší případ a kolik paměti potřebuje pro jedno zanoření a to včetně velikosti návratových adres do rekurzivně volaných funkcí. Z těchto údajů po té určit nutnou minimální velikost zásobníku. V literatuře se uvádí doporučení tuto hodnotu ještě o 10 až 20 % "nadsadit", abychom eliminovali vliv ostatních automatických proměnných.

- statické proměnné
S omezením velikosti paměti pro statické proměnné se nejčastěji setkáte v překladačích C pro DOS, kde je celková velikost paměti určená pro statické proměnné stanovená velikostí jednoho segmentu (tj. 64Kb).
- dynamické proměnné
Limit velikosti paměti určené pro dynamické proměnné (tzv. heap) bývá ze všech výš uvedených paměťových tříd nejvyšší. Zde pravděpodobně na omezení nenarazíte, a proto je vhodné všechny rozsáhlejší datové typy ukládat dynamicky. Ale i zde limity existují (DOS cca. 600Kb, Symbian 2Mb).

ad 7) Souborový systém může využívat jiný způsob zápisu cesty k souborům. (např. MSDOS - obrácená lomítka, QDOS - podtržítka)

S tímto problémem se již patrně setkal každý uživatel Windows, který pracoval v Unix like systémech. Problém ale není na straně unixových systémů (ty totiž vznikly dávno před prvními PC). Za tuto nekompatibilitu vděčíme prvnímu MSDOSu, který zpětná lomítka zavedl. Pokud používáte Windows, DOS ,OS/2, ve vašem zdrojovém textu v jazyce C, musíte uvádět vždy dvě obrácená lomítka, neboť jedno obrácené lomítko slouží pro opis speciálních netisknutelných znaků (tj. \n \r \a apod.).

```
sprintf(str, "\\PROJECT\\SRC\\MAIN.CFG");
```

Řešení:

- Podmíněný překlad různých částí zdrojového textu pomocí definovaných symbolů.
- Definice vhodných maker zastupující oddělovací znaky a jejich využívání místo nich.

```
#ifdef __LINUX__
#define PATHSEP_ '/'
```

```

#define SPATHSEP_ "/"
#elif defined (__WIN32__)
#define PATHSEP_ '\\'
#define SPATHSEP_ "\\\"
#else
#error Unknown platform
#endif

char path[ ] = SPATHSEP_ "PROJECT"SPATHSEP_ "SRC"SPATHSEP_ "MAIN.CFG"

```

Další zásady a doporučení:

- Pište program přímo tak, aby byl přenositelný. Pokud budete upravovat části programu později tak, aby se celek stal přenositelný, pravděpodobně některou část opomenete.
- Pamatujte na to, že přenos programu s velkou pravděpodobností nebudete provádět vy, ale někdo úplně jiný, kdo se ve vašem programu bude obtížněji orientovat. Proto bývá zvykem kritická místa označit komentářem.
- Řešte všechny warningy. Správně napsaný program končí hlášením: Warning(s): 0, Error(s): 0. Každé hlášení, které je na vaší platformě "jen" warning, bude dle zákona schválnosti na nové platformě vážný error.
- Některé překladače (gcc, VC) umožňují nastavit limit důležitosti warningů, které se mají vypisovat. Snažte se dosáhnout nulového počtu warningů i u nejpřísnějšího limitu.
- Svoje programy vždy pečlivě trasujte, mnohokrát se vám stane, že teprve přenosem programu na jinou platformu se projeví fatálně chyba, která se na původní platformě projevovala velmi nenápadně.
- Pokud program prohlásíte za přenositelný, měl by mít ošetřenu nekompatibilitu endianu. (použitím makra nebo alespoň redundantními funkcemi pro oba endiany)
- Pokud program prohlásíte za přenositelný, měl by obsahovat makefile. S největší pravděpodobností na nové platformě nebude existovat vývojové prostředí, které používáte na platformě staré. Projektové soubory z vašeho oblíbeného vývojového prostředí tedy budou nepoužitelné v horším případě i nečitelné.
- Vždy oddělte část programu komunikující s uživatelem (tzv. UI) od samotných algoritmů zajišťující samotnou činnost programu (tzv. Engine). Mnohokrát totiž bude jednodušší přenést pouze engine a UI vytvořit zcela nové.
- Vždy se snažte využívat jen standardní knihovny C, nepoužívejte žádné platformě závislé speciality.
- Pokud ve zdrojovém textu programu musí být takové části, které jsou platformě závislé, umístěte je do samostatného modulu.
- Snažte se udržet binární kompatibilitu datových souborů mezi všemi platformami (tj. datové soubory jsou vždy společné pro všechny platformy)
- Nebojte se podívat (nebo se zeptat), jak váš problém řeší ostatní. U mnoho open-source projektů je dnes otázka přenositelnosti zdrojového textu nutnou součástí image.

5.2 C

5.2.1 Struktura programu v C

Jazyk C podporuje modulární programování (tj. umožňuje rozdělit zdrojový text programu do více souborů tzv. modulů). Rozdělení programu na moduly přináší mnoho výhod

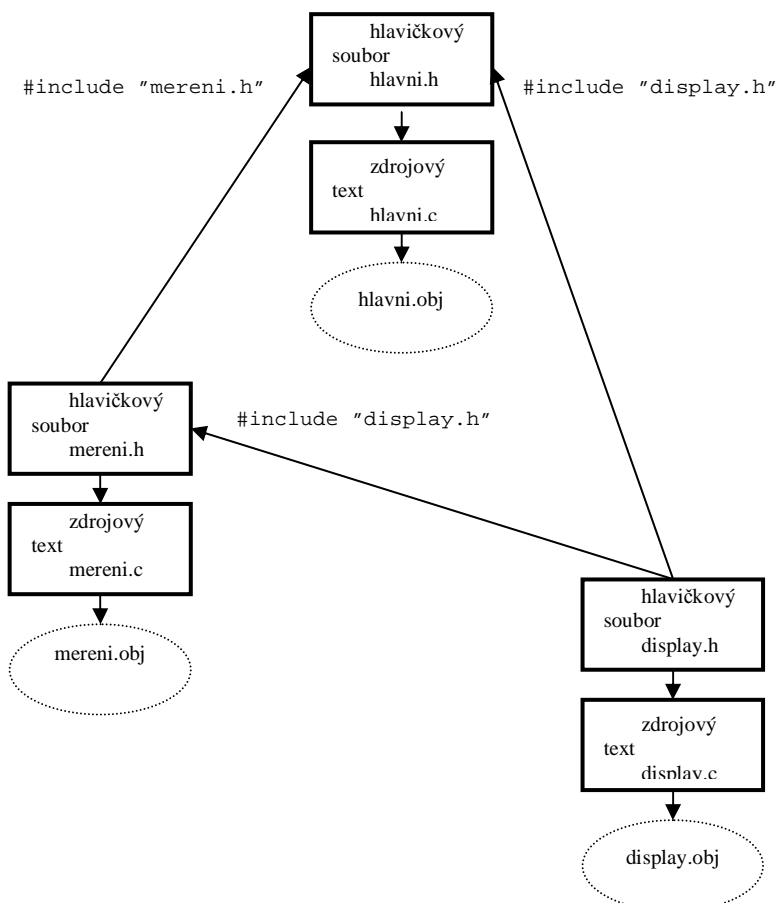
– souběžnou práci několika programátorů, jednodušší údržba textů, rychlejší opakovaný překlad, atd. Všechny uvedené výhody opakovatelného využití dříve vytvořených modulů pro nové programy lze sjednotit do jedné - **”při opětovném využívání již hotových modulů dochází k výraznému zkrácení doby potřebné k návrhu, implementaci a odladění nového programu”**. Velmi často se vám při psaní stane, že budete opětovně využívat nejen svoje (dříve) vytvořené moduly, ale i moduly, které vytvořil někdo jiný a nabídl je ostatním k využití. Aby bylo možné využívat modularitu zdrojových textů v jazyce C, vznikly určitá pravidla a zásady. Většina těchto zásad je definována normou jazyka C (norma ANSI C), a proto vám jejich dodržování mohu jen doporučit.

Každý soubor se zdrojovým textem programu psaného v jazyce C je obyčejný textový soubor, který je možné psát např. pomocí textových editorů jako je notepad nebo vi. Existují i speciální editory, které jsou součástí prostředí pro překlad, které umožňují zvýraznit, či barevně odlišit klíčová slova, proměnné, konstanty, komentáře atd. Každý zdrojový soubor je opatřen názvem a doplněn příponou `.c` nebo `.h`. Je doporučeno normou jazyka využívat v příponě pouze malá písmena (velká písmena jsou určena pro soubory se zdrojovým textem v jazyce C++).

Význam obou typů zdrojových souborů je následující:

- Základní soubory se zdrojovým kódem – typické značení `.c` (např. `hlavni.c`)
- Hlavičkové soubory s propojením (interface) – typické značení `.h` (např. `hlavni.h`)

Tyto dva soubory tvoří dohromady celek, který nazýváme modul. Celý program tedy sestává z několika modulů (`.h` a `.c` souborů), které umožňují sdílení svých funkcí a proměnných s ostatními moduly za pomoci hlavičkových souborů. Tak jak je naznačeno na obrázku:



Program uvedený na obrázku se skládá ze tří modulů (tj. šesti souborů), každému modulu bylo přiřazeno jméno, které popisuje jeho funkci a je společné pro hlavičkový a zdrojový soubor. Mohlo by se například jednat o program vypisující na připojeném LCD displeji aktuální čas. Modul `display` by obsahoval obslužné funkce pro vypsání jednoho znaku na LCD displej, smazání displeje a přesun kurzoru na danou pozici. V modulu `měření` by byly vytvořeny funkce pro výpočet aktuálního času a jeho výpis na displej. Modul `hlavní` obsahuje hlavní funkci spouštěnou při startu programu a umožňující inicializaci displeje a ruční nastavení aktuálního času při startu. Všechny tyto funkce by byly zapsány do zdrojových souborů (*.c)

Nyní se zaměříme na význam hlavičkových souborů.

Překladač jazyka C překládá každý zdrojový soubor .c samostatně a proto překladač nemá v daný okamžik informace o obsahu ostatních zdrojových souborů (*.c). Teprve poté co jsou úspěšně přeloženy všechny zdrojové soubory (*.c), je ze všech přeložených souborů sestaven jeden výsledný spustitelný soubor. Z tohoto pravidla o samostatném překladači .c souboru plyne, že **v každém zdrojovém souboru (*.c) je nutné provést "oznámení" o všech využívaných funkcích a proměnných, které nejsou součástí tohoto modulu**, tj. jsou implementovány externě (v jiném modulu). K tomuto "oznámení" slouží právě hlavičkové soubory.

Hlavičkový soubor obsahuje "oznámení" vstupů a výstupů z daného modulu (tzv. prototypy). V těchto souborech mohou být obsaženy i další povely pro překladač (viz. 5.2.9). Snahou je, aby ke každému souboru se zdrojovým textem (*.c) existoval hlavičkový soubor se stejným názvem a příponou (*.h).

Každý soubor se zdrojovým textem (*.c) obsahuje na svém začátku speciální příkaz pro vložení (`#include`) obsahu hlavičkových souborů těch modulů, jejichž funkce nebo proměnné využívá. Tím je překladač "obeznámen" s prototypem daných externích funkcí či proměnných. Díky této znalosti může provést kontrolu správnosti využívání těchto externích funkcí a proměnných v právě překládaném modulu a správně provést překlad volání těchto funkcí v celém modulu. Je třeba si uvědomit, že k překladu volání externích funkcí a využití externích proměnných není třeba znát obsah těchto funkcí ani jejich absolutní umístění v paměti, neboť tyto informace lze doplnit až při sestavování (linkování) spustitelného programu.

Hlavičkové soubory tedy obsahují pouze prototyp, který představuje tyto informace:

Pro funkci: název, seznam typů argumentů a typ návratové hodnoty funkce.

Pro proměnnou: název a typ proměnné.

Deklaraci nově vytvořených typů.

Nikdy nevkládejte do hlavičkových souborů takové části textu, které tvoří inicializaci, nebo ze kterých vznikne ve spustitelném programu kód. Proč? Ukažme si důsledek takového počínání na příkladu programu uvedeném na obrázku v této kapitole.

Vytvoříme v modulu `display` například funkci, která vrátí jako svůj výsledek větší ze dvou vstupních argumentů. Tuto funkci pojmenujeme `Maximum`. Při tvorbě ostatních modulů zjistíme, že tuto funkci budeme potřebovat i jinde, a proto vložíme celé její tělo do souboru `display.h`. Všechny zdrojové texty s příponou (*.c) obsahují na svém začátku příkaz:

```
#include "display.h"
```

Tento příkaz vloží při překladu celý obsah souboru `display.h` (tedy včetně deklarace a těla funkce `Maximum`) na začátek překládaného (*.c) souboru. Vše probíhá korektně, všechny zdrojové soubory (*.c) jsou samostatně přeloženy v pořádku, funkce `maximum` je totiž skutečně součástí každého z modulů.

Problém ale nastane v okamžiku, kdy má dojít k vytvoření výsledného spustitelného souboru ze tří přeložených zdrojových souborů (`hlavni`, `mereni`, `display`). Pomocný program překladače provádějící sestavení (tzv. linker) nalezne tři identické funkce s názvem `Maximum`, zastaví překlad a zahlásí chybu popisující situaci která nastala. V programu existuje více funkcí `Maximum`, které se stejně jmenují (tj. vícenásobně vytvořená funkce).

Řešením je umístit do hlavičkového souboru pouze "oznámení" tj. prototyp funkce `Maximum`.

A celou funkci `Maximum` umístit do souboru `display.c`.

Výsledkem bude nejen správné přeložení jednotlivých modulů, ale i bezchybné sestavení spustitelného programu. Funkce `Maximum` nyní existuje pouze v jednom exempláři a v jednom modulu (`display`). Jelikož však lze předpokládat širší využití funkce pro určení maxima, bylo by vhodné pro tuto funkci vytvořit vlastní modul a navázat ho na modul `display` (opět pomocí direktivy `include`).

Další důležitou skutečností je fakt, že při překladu zdrojového souboru `hlavní.c` je z obrázku patrné, že hlavičkový soubor je vložen opakovaně dvěma nezávislými cestami. Ošetřením vícenásobného vložení pomocí příkazů preprocesoru se zabývá kapitola 5.2.9. Příklad použití je uveden v kapitole 5.2.10.

5.2.2 Překlad a sestavení programu v jazyce C

Základní pojmy:

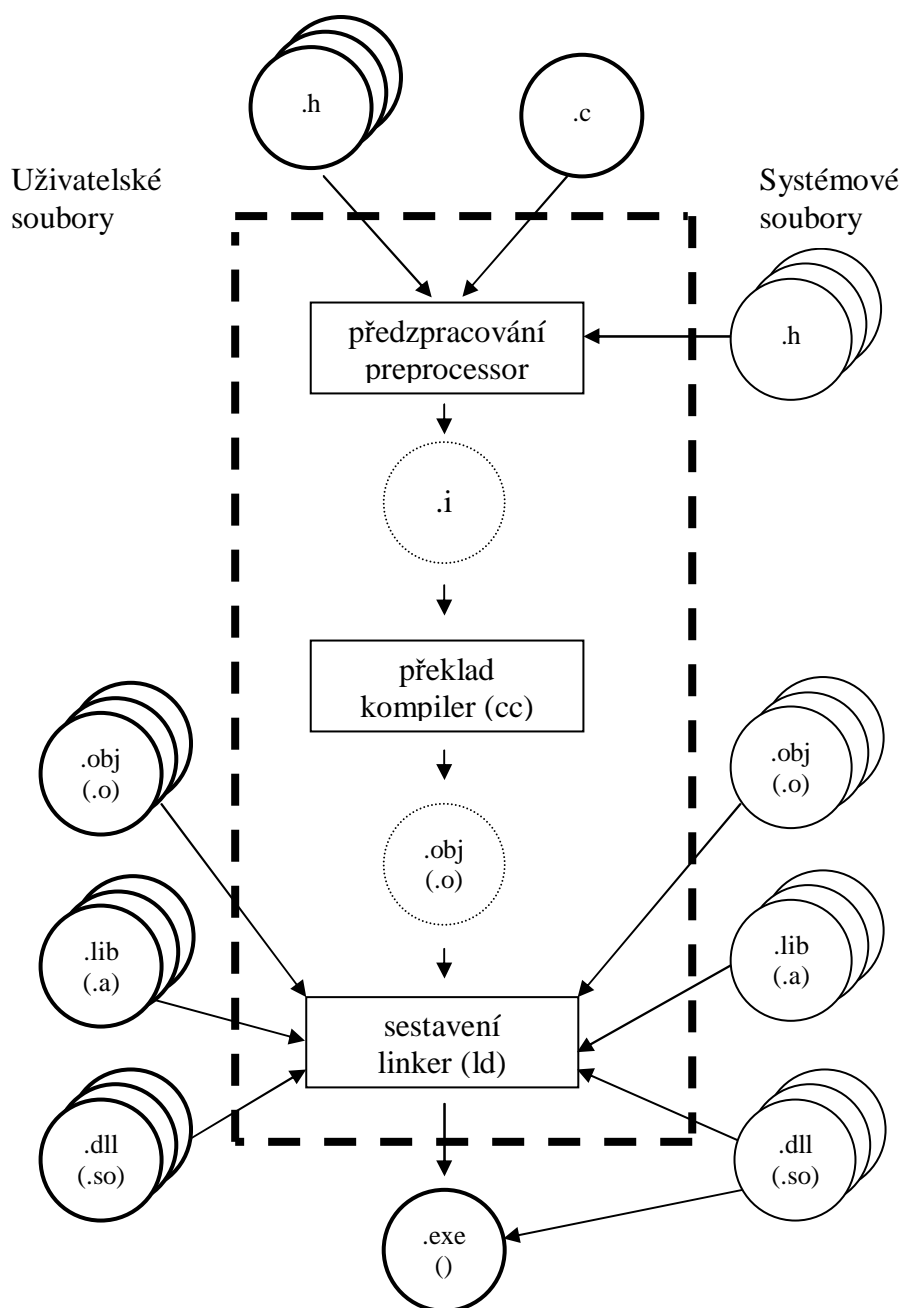
preprocessing – předzpracování

kompilace – přeložení, překlad

linkování – sestavení

building – vytvoření (tj. všechny tři předchozí procesy).

Následující obrázek zachycuje jednotlivé fáze vytvoření spustitelného souboru včetně všech typů souborů, které se mohou na překladu podílet. V závorkách jsou uvedeny přípony souborů používané na platformách jiných než MS Windows.



Vlastní vytvoření programu je většinou řízeno pomocí projektového souboru nebo souboru pro příkaz **make** (tzv. Makefile).

V prvním kroku překladu dochází k předzpracování zdrojových a hlavičkových souborů preprocesorem (5.2.9). Výsledek předzpracování je uložen do dočasného souboru (např. s koncovkou .i). Úkolem preprocesoru je ze vstupního textu odstranit komentáře, provést rozvoj maker, případně vložit další (hlavičkové) soubory, které obsahují prototypy funkcí a proměnných používané v právě překládaném zdrojovém souboru (*.c). Hlavičkové soubory (*.h) slouží k "oznámení" prototypů systémových nebo uživatelských funkcí a proměnných (umístěných v souborech *.c, *.obj, *.lib, *.dll)

V druhém kroku je výsledek z předchozího kroku (tj. předzpracování) použit jako vstup pro překladač (kompilátor). Výstupem překladače (kompilery) je tzv. object soubor obsahující již přeložený kód, ale zatím bez relokačních tabulek a dalších náležitostí nutných k tomu, abychom mohli tento soubor spustit. Object soubor mívá nejčastěji koncovku .obj nebo .o. První a druhý krok je prováděn opakovaně dokud nejsou do object souborů přeloženy všechny zdrojové soubory uvedené v projektovém nebo Makefile souboru. Tato fáze má své volby pro překlad, kde je možné nastavit požadavek na vytváření dalších doplňkových souborů s protokoly o překladu (např. protokol s chybovými hlášeními, mezivýsledek překladu, tj. zdrojových textů pro překladač assembleru (.s), atd.).

Ve třetím kroku jsou pomocí programu pro sestavení (linker), sloučeny/sestaveny všechny object soubory uvedené v projektovém nebo Makefile souboru do výsledného spustitelného programu/souboru. Hlavním úkolem linkeru je přidělit relativním vazbám v object souborech skutečné-absolutní adresy funkcí a proměnných z ostatních object souborů tak, aby bylo možné výsledný soubor spustit jako program. Do výsledného spustitelného souboru jsou navíc prakticky vždy přidány tzv. knihovní object soubory obsažené v knihovních souborech (nejčastěji s koncovkou .lib nebo .a). Knihovny obsahují základní funkce pro práci se vstupy/výstupy, se soubory atd. Je možné zvolit, zda bude výsledný program sestaven tak, že všechny potřebné knihovní funkce budou do výsledného programu celé vloženy (tzv. statické sestavení) nebo zda budou vloženy do výsledného programu pouze odkazy na funkce dynamické knihovny, kterou bude program ke své práci potřebovat (tzv. dynamické sestavení). Výhodou dynamického sestavení je výrazně menší délka výsledného spustitelného souboru. Nevýhodou je závislost na dalších souborech (dynamických knihovnách), které musí operační systém obsahovat. Další nevýhodou může být pomalejší spouštění programu dané "dosestavováním" celého spustitelného souboru s dynamickými knihovnami v okamžiku spouštění programu.

Dalším úkolem linkeru je vložení tzv. prologu a epilogu. Prolog (uvaděč) je speciální kód spouštěný ještě před začátkem programu napsaného v jazyce C. Analogicky epilog je kód spouštěný po ukončení programu v jazyce C. Obsah a funkce prologu a epilogu je nejčastěji závislá na cílovém operačním systému, pro který je spustitelný soubor určen. Zcela jistě bude jiný prolog u překladače pro Windows a jiný pro některý z mikrokontrolerů. Stejně tak, epilog může například vracet systém do stavu před spuštěním programu, ale také nemusí. Může tedy odevzdat systému programem nevrácené prostředky (např. alokovanou paměť, odblokovat myš, klávesnici nebo některý z výstupů) nebo naopak zabránit systému v jejich dalším použití.

Celý proces předzpracování, překladu a sestavení je možné řídit za pomoci velkého množství voleb a přepínačů a je možné v jeho průběhu nechat vytvářet soubory s protokoly o překladu (mapy paměti a rozložení funkcí a proměnných, ...). Bohužel tyto volby a přepínače

jsou vždy typické pro daný překladač a proto je vždy nutné nahlédnout do dokumentace k používanému překladači.

Základem tvorby programu v C je projektový soubor (nebo soubor makefile). Tento soubor v sobě sdružuje data o jednotlivých součástech, kterými jsou soubory se zdrojovým kódem, a pokyny pro překlad a sestavení výsledného programu. Součástí jsou i názvy výstupních souborů. Projekt je možné vytvořit v prostředí a zároveň lze v tomto prostředí v sekcích nastavení (options) nastavit i parametry pro překlad včetně linkování. Je také možné napsat makefile v textovém editoru. V tomto makefile je popsáno volání řádkových překladačů a linkerů včetně nastavení přepínačů pro typ překladu. Se soubory Makefile se nejčastěji setkáte u unix like operačních systémů, neboť zajišťují větší možnosti nastavení překladu a zároveň nejsou svazány s žádným programátorským prostředím, a proto jsou lépe přenositelné.

Minimální možná sestava pro spustitelný program je jeden soubor se zdrojovým kódem. I když některá prostředí jsou schopna překladu takového souboru bez vytvoření projektu, je lépe projekt vytvořit, protože součástí projektu jsou nastavení nutná pro překlad a je lépe je mít vytvořena pro jednotlivé programy a nepoužívat defaultní nastavení prostředí, které nemusí být pro zvolený kód vhodné.

K tomu, abychom pomocí programovacího jazyka C vytvořili spustitelný program, musíme udělat několik kroků:

1. Vytvořit jednotlivé moduly se zdrojovým kódem, které realizují požadované funkce.
2. Vytvořit projekt nebo makefile, do kterého zaznamenáme názvy souborů jednotlivých modulů případně další informace nutné k překladu a sestavení.
3. Spustit překlad a sestavení těchto modulů do výsledného spustitelného souboru.

Překlad zdrojového textu jazyka C je časově náročná operace. Proto byl zaveden mechanismus minimalizující čas nutný k opakované kompilaci stejného programu zajišťující vždy překlad pouze těch zdrojových textů, u kterých došlo k modifikaci. Ostatní zdrojové soubory se nepřekládají, protože v nich nedošlo k žádné změně. Všechny object soubory jsou znovu sestaveny (linkovány) a tím je zajištěn vznik aktuálního spustitelného souboru. Proto je důležité zbytečně nevkládat takové hlavičkové soubory, které nejsou v daném zdrojovém souboru potřeba. Dále se nedoporučuje vkládat zdrojové soubory, protože se zbytečně překládají dva zdrojové soubory při změně jednoho.

Vytvoření makefile souboru.

Soubor makefile představuje textový soubor s definovanými pravidly jak z daných vstupních souborů (zdrojů) za pomoci příkazů vytvořit výstupní soubor (tzv. cíl). Samotný příkaz make, který soubor Makefile zpracovává, automaticky prochází zadaná pravidla skládající se z podmínkové a příkazové části. Příkazová část pravidla je vykonána pouze pokud není splněna podmínka. Podmínka se skládá z cíle a dvoutečkou odděleného seznamu zdrojů. Podmínka není splněna jestliže:

- Cíl (výstupní soubor) neexistuje vůbec (je tedy potřeba ho vytvořit).
- Některý ze zdrojů (vstupních souborů) byl modifikován (tj. cíl je nutno aktualizovat).

Ukažme si, jak by vypadal zjednodušeně projektový soubor pro program `hlavni.exe` uvedený na obrázku v kapitole 5.2.1.

```

# Makefile hlavni.exe
# Compiler definition
CC=gcc
# Delete file definition
DEL=rm -f

hlavni.exe: hlavni.o mereni.o display.o
    (tab) $(CC) -o hlavni.exe hlavni.o mereni.o display.o
hlavni.o: hlavni.c hlavni.h mereni.h display.h
    (tab) $(CC) -c hlavni.c
mereni.o: mereni.c mereni.h
    (tab) $(CC) -c mereni.c
display.o: display.c display.h
    (tab) $(CC) -c display.c
clean:
    (tab) - $(DEL) hlavni.exe
    (tab) - $(DEL) *.o

```

Texty uvedené za znakem # až do konce řádku program make ignoruje a jsou určeny pro poznámky.

Na třetím řádku definujeme makro CC jehož hodnota bude název překladače jazyka C, v našem případě GNU C Compiler (tj. gcc). Kdykoliv bude v dalším textu uvedeno \$(CC) program make doplní místo něj text gcc.

Na dalších řádcích jsou definována jednotlivá pravidla pro příkaz make. Samotný příkaz se od podmínky odlišuje pomocí znaku tabulátoru (tab) na začátku řádku, který informuje program make o tom, že na řádku je uveden příkaz vážící se k podmínce. Jako příklad uveďme pravidlo pro cíl clean, které obsahuje dva příkazy. Pravidlo pro cíl clean je zároveň jediné pravidlo, jehož název není názvem žádného vytvářeného souboru. Cíl s názvem clean je tedy tzv. předstíraný cíl, který slouží pouze k vymazání všech souborů, které vznikly při překladu.

Pravidlo hlavni.exe slouží k sestavení spustitelného programu z jednotlivých object souborů.

Ostatní pravidla jsou pravidla pro překlad jednotlivých zdrojových textů do samostatných object souborů.

Všimněte si, že zatímco v podmínce pro překlad např. cíle hlavni.o je uvedeno mnoho zdrojů, v samotném příkazu kompilátoru je uveden pouze soubor hlavni.c (všechny ostatní jsou totiž vkládány až při překladu preprocesorem na základě příkazů #include).

Nyní by vám měl již být postup překladu pomocí příkazu make patrný. Syntaxe parametrů programu make je následující:

```
make <požadovaný-cíl>
```

Program make se pokusí automaticky vyhledat a načíst soubor s názvem Makefile nebo makefile.

Spustíme-li tedy program s parametrem hlavni.exe tj.

make hlavni.exe

program make vyhledá a načte v aktuálním adresáři soubor s názvem Makefile a začne postupně procházet jednotlivá pravidla, spouštět příkazy překladače gcc a tím překládat jednotlivé zdrojové texty, zároveň bude make kontrolovat, zda spouštěný program/příkaz korektně skončil (návratová hodnota musí být nula). Pokud některý příkaz vrátí jinou návratovou hodnotu, je vykonávání programu make ukončeno s výpisem kde k chybě došlo. Pokud chceme ignorovat návratovou hodnotu spouštěného příkazu, napíšeme před příkaz znak - , tak jak je uvedeno u příkazů smazání souborů u klamného cílu clean, kde neexistence souboru hlavni.exe neznamena ukončení mazání ostatních souborů.

Jak je patrné, program make lze použít nejen k překladu zdrojových textů jazyka C, ale jeho možnosti jsou daleko větší. V podstatě lze říci, že make je obecným program pro překlad, konverzi nebo vytváření libovolných textových souborů na základě definice pravidel a spouštění příkazů. Na druhou stranu k jeho nevýhodám patří poměrně složitý zápis a vytváření pravidel u větších projektů. Na soubory Makefile nejčastěji narazíte v unix like systémech nebo u projektů, kde je kladen velký důraz na přenositelnost programu mezi platformami, neboť program make existuje prakticky pro všechny platformy.

Vytvoření projektového souboru.

Jak bylo napsáno výše, je projektový soubor prakticky vždy svázan s konkrétním vývojovým prostředím, někdy i s jednou konkrétní verzí takového prostředí. Často se jedná o binární soubory s neznámou vnitřní strukturou. Lepší vývojová prostředí ukládají projektový soubor v textovém formátu (např. XML) nebo umožňují konverzi vlastních projektových souborů do Makefile souborů. Tyto Makefile soubory jsou ovšem velmi často špatně čitelné, nefungují vůbec nebo mají speciální syntaxi vázanou opět jen na program make dodávaný s vývojovým prostředím.

Velkou výhodou práce v některém z vývojových prostředí je prakticky absolutní izolovanost programátora od tvorby pravidel pro překlad (známých z programu make kapitola 0). Vývojová prostředí vytvářejí tato pravidla sama, což zjednodušuje tvorbu programu, ale na druhou stranu neumožňují tvorbu nových/speciálních pravidel (např. formátování dokumentace do různých formátů html/txt/tex).

Další výhodou vývojových prostředí je možnost použití různých pomocníků (wizards), které programátorovi na základě několika dotazů vytvoří projektový soubor pro daný typ aplikace (konzolová aplikace, MDI aplikace, atd.) včetně vhodných nastavení všech přepínačů překladače, takže začínající programátor je schopen velmi rychle tvořit a překládat vlastní program.

Samotný projektový soubor se ve vývojovém prostředí nejčastěji zobrazuje jako seznam zdrojových souborů (*.c), do kterých programátor přidává při tvorbě programu další zdrojové soubory a ze kterých jsou automaticky sestaveny i závislosti na hlavičkových souborech (*.h).

Využití vývojových prostředí s projektovými soubory je tedy vždy zrychlení práce, zvláště u velkých projektů, protože zbavuje programátora "otrockého" vytváření pravidel pro překlad. Mezi nevýhody patří nepřenositelnost projektových souborů mezi platformami a izolovanost programátora od samotného procesu překladu, kdy velmi často tvůrce programu ani nemá přehled o tom, kolik dynamických knihoven jeho vytvořený program potřebuje ke své činnosti.

5.2.3 Komentáře

Komentáře slouží k psaní poznámek do souborů, ze kterých je sestaven program. Poznámky slouží k lepší čitelnosti a orientaci v programu. Neměly by popisovat to, co je zřejmé z textu – např. ”násobím dvěma” - což zůstává zřejmé stále, ale vysvětlovat proč tomu tak je – např. ”je nutné uvažovat i cestu zpět” – znovu zjistit proč je to právě takhle nás, po nějakém čase může stát značnou námahu. Poznámka by měla být psána na začátku bloku, nebo vedle příkazu, ke kterému se vztahuje. Psaní poznámek uprostřed příkazů či výrazů zhoršuje čtení textů.

Vlastní komentář je možné v C napsat jediným možným způsobem a to tak, že má začátek ” /* ” a konec ” */ ”. Vše co je mezi těmito znaky je komentář, který může obsahovat i více řádků a může začínat i končit na libovolné pozici řádku (a to i vůči psanému programu).

Úvodní i konečný znak komentáře patří do kategorie dvojznaků – skládají se ze dvou znaků, které je nutné psát bez mezery mezi nimi a interpretují se jako jedna akce, v tomto případě začátek a konec komentáře.

```
/*v tomto místě je komentář*/
```

```
/*
 *          toto
 *          je
 *          víceřádkový
 *          komentář
 */
```

V textu poznámky se tedy může vyskytovat i hvězdička, nesmí však být následována lomítkem. Některé překladače umožňují tzv. vhnížděné komentáře, což jsou komentáře ve více úrovních. Překladač potom sleduje počty začátků komentáře a hledá jim příslušné konce. Nemá-li překladač tento mechanismus implementovaný, potom je problémem např. zakomentovat celou komentovanou část programu, protože první ukončení komentáře ukončí všechny předchozí komentáře (počátky dalších jsou součástí komentáře). Jelikož využití této vlastnosti, i když je k dispozici, zhoršuje přenositelnost, je lepší použít řízeného překladu pomocí direktivy #define a #if preprocesoru.

```
/* nepovoluje /*vhnížděné */ komentáře */
```

5.2.4 Funkce main – základ

Základním stavebním kamenem pro tvorbu programu jsou funkce. Funkce se skládá z hlavičky (která obsahuje typ návratové hodnoty, jméno funkce a předávané parametry) a z vlastního těla funkce – v tomto případě se jedná o definici funkce. Pokud je uvedena pouze hlavička (následovaná středníkem), potom se jedná pouze o deklaraci (”oznámení”) funkce, které slouží k tomu, aby překladač znal jméno funkce a aby věděl jakého typu jsou předávané parametry (viz. konverze 5.2.6, 5.2.7). Hlavička funkce má standardní tvar:

```
Návratová_hodnota jméno_funkce(seznam_parametrů) např. int faktorial(int ceho) { }
```

návratová_hodnota – udává typ návratové hodnoty, je pouze jedna

jméno_funkce – unikátní jméno pro volání funkce

seznam_parametrů – seznam předávaných parametrů, obsahuje typ a název pro každou z předávaných proměnných.

Výše uvedený princip volání je jediný možný v jazyce C, kde se nerozlišuje mezi procedurou a funkcí ve smyslu typu návratových či předávaných hodnot. Funkce může, ale nemusí vrátit jednu hodnotu jako návratovou. Další hodnoty je možné vrátit za pomoci seznamu parametrů. K tomuto předání však musí existovat předávané proměnné – jde o jiný mechanismus než v předchozím případě.

První funkcí, která je v C volána je funkce **main**. Tato funkce proto musí být v programu vždy obsažena. Návratovou hodnotou je celočíselná hodnota, pro kterou má C typ int (kapitola 5.2.5). Seznam parametrů umožňuje předat do programu parametry z řádky při spuštění programu (kapitola 5.2.25), ale je možné ho nechat prázdný a předávané parametry nevyužít.

Tělo funkce je tvořeno blokem programu, který je ohraničen složenými závorkami " { " a " } ", které jsou určeny pro začátek a konec bloku. Těmito závorkami může být uzavřen také složený příkaz.

Má-li funkce návratovou hodnotu, je potřebné ji předat. K tomuto předání slouží klíčové slovo **return**, následované předávanou hodnotou. Tímto příkazem tedy končí provádění funkce a výsledek i činnost programu je předána předchozí funkci. V případě funkce main se návratová hodnota vrací procesu, který spustil program (v DOS je zjistitelná např. pomocí proměnné prostředí ERRORLEVEL). Každý příkaz v jazyce C musí být ukončen středníkem. Středník je zde povinný, neslouží jako oddělovač ale opravdu jako ukončení, pokyn k provedení příkazu.

int main () ;	hlavička hlavní funkce programu jejíž název je main návratovou hodnotou je celočíselný typ int seznam předávaných parametrů nebude použit a zůstává prázdný je-li hlavička funkce ukončena středníkem, jedná se o deklaraci funkce, tj. "oznamujeme", že funkce s tímto názvem existuje a má tyto typy návratových a předávaných hodnot (deklaraci funkce main není třeba uvádět)
int main ()	při definici funkce (tj. za hlavičkou funkce následuje tělo) se středník nepíše
{	předcházela-li deklarace funkce definici, potom musí být obě v C identické začátek těla funkce (main)
	 zde je místo pro vlastní zdrojový kód
;	je-li uveden pouze středník, jedná se o prázdný příkaz (může být přeložen jako žádná operace, ale také může být ignorován (zoptimalizován))
return 3;	jelikož funkce main má vracet celočíselnou hodnotu, musí být posledním řádkem funkčního bloku její návrat. Ten se provede klíčovým slovem return následovaným předávanou hodnotou (zde hodnota tři) aby byl tento řádek příkazem, který se provede, musí být ukončen středníkem
}	konec těla funkce (main) středník se zde nepíše, protože blok není výraz, ze kterého se udělá příkaz

Počet prázdných řádků, nebo mezer nemá vliv na překlad, pokud tyto nerozdělí identifikátor (klíčové slovo, název proměnné, název funkce a jiné) na dvě části.

5.2.5 identifikátory, základní datové typy, konstanty

Identifikátory

Pro zápis programu jsou důležité názvy proměnných a funkcí, které je možné tvořit z povolených znaků. C rozlišuje malá a velká písmena. Podtržítko se povoluje užívat uprostřed, nedoporučuje se s ním název začínat (je to možné ale názvy s lichým počtem podtržítek slouží jako pomocné proměnné při překladu a mohlo by docházet ke kolizím). Délka identifikátoru není omezena, ANSI C však rozeznává pouze prvních 31 znaků (zbytek ignoruje). Názvy klíčových slov se píšou pouze malými písmeny.

Základní datové typy a příslušné konstanty

Mezi základní typy patří typy celočíselné a typy s plovoucí řádovou čárkou. Kromě těchto typů jsou zde ještě typ void (5.2.8), ukazatele (5.2.15) a složené datové typy. Typy v C nemají z normy danou přesnost (tj. velikost kterou zabírají v paměti), ale velikost typů je závislá na platformě, pro kterou je program přeložen. Jsou zde pouze stanoveny relace mezi typy. Je určeno, který typ se do kterého typu vejde, aniž by došlo ke ztrátě dat. Velikost typu lze zjistit v C pouze pomocí klíčového slova **sizeof**, které určí kolik bytů je potřeba k uložení daného typu (u složených datových typů vrací hodnotu skutečné paměti, včetně prázdných míst (mezer) na zarovnání počátečních adres jednotlivých proměnných). Parametrem **sizeof** může být název daného typu nebo proměnná daného typu.

<pre>{ unsigned long int i, ii; i = sizeof(unsigned long int); ii = sizeof(i); }</pre>	<p>definice proměnných pro výsledek, kterým je zjištění velikosti daného typu v bytech, definice musí být na začátku bloku</p> <p>zjištění velikosti typu na základě přesného uvedení typu</p> <p>zjištění velikosti typu na základě proměnné daného typu</p> <p>oba výsledky by měly být shodné</p>
---	--

Celočíselné typy mají v jazyce C tyto názvy

char

short int se zkráceným zápisem short

int

long int se zkráceným zápisem long

a pro velikosti těchto typů platí relace

sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long int)

Celočíselné typy mohou být znaménkové a bezznaménkové. Ke stanovení, zda se má jednat o bezznaménkový respektive znaménkový typ slouží klíčové slovo unsigned respektive signed předřazené danému typu např. signed char, unsigned short, signed long int. Uvádění klíčového slova signed je nepovinné (typ bez uvedení je signed). Výjimkou je typ char, který se bez uvedení signed / unsigned řídí nastavením překladače a může tedy být signed nebo unsigned v závislosti na nastavení v projektu, make file nebo překladače.

zkrácený tvar	zápis	popis
	char	nejmenší celočíselný typ, je znaménkový nebo bezznaménkový podle nastavení překladače
	unsigned char	nejmenší celočíselný typ bez znaménka
signed short	signed char	nejmenší celočíselný typ se znaménkem
unsigned short	signed short int	... další celočíselné typy (větší = přesnější) ve
int / signed	unsigned short int	znaménkové a bezznaménkové variantě
unsigned	signed int	
long/signed long/long int	unsigned int	
unsigned long	signed long int	
	unsigned long int	

{	definice je v C možná pouze na začátku bloku
char znak;	definice proměnné typu char, která se jmenuje znak. Při definici proměnné dochází k vytvoření proměnné znak a zároveň k jejímu fyzickému vytvoření, to znamená, že je pro ni rezervováno místo v paměti, které se jmenuje znak a kde bude uložena vlastní hodnota této proměnné. Definice je ukončena středníkem.
unsigned ii, jj, kk;	definice tří bezznaménkových proměnných typu int. K oddělení názvů proměnných se použije čárka. Počet proměnných není omezen.

Jazyk C nemá typ reprezentující přímo znaky (písmena, číslice, ...). Nejmenší celočíselný typ char se nejlépe hodí k ukládání znaků, a byl tedy vybrán jako typ vhodný pro manipulaci se znaky. Platí pro něj pravidla a pracuje se s ním stejně jako s ostatními celočíselnými typy a navíc slouží k ukládání znaků. Až vhodnou reprezentací celočíselného čísla zapsaného do typu char se zobrazí znak.

Celočíselné konstanty

Jejich hodnotou je ordinární číslo. Celočíselné konstanty je možné zapisovat jako znaky, které je možné psát jako alfanumerické znaky nebo jako escape sekvence, nebo jako čísla, která je možné zapisovat v osmičkové, nebo desítkové nebo šestnáctkové soustavě. Všechny tyto konstanty je možné použít pro jakýkoli celočíselný typ. Při pokusu přiřadit číslo typu, do kterého se nevejde, by mělo dojít k chybovému hlášení, každopádně však bude přiřazeno jen tolik kolik lze (to platí i pro typy v plovoucí čárce a převody mezi plovoucí čárkou a celočíselnými typy).

Znakové konstanty

Znakovou konstantou rozumíme znak, který je uveden v jednoduchých apostrofech 'a', '0', ' ', '\$' (ekvivalentní přímému zápisu v ASCII - 97, 48, 32 ...). Při překladu je takto uvedený znak převeden na ASCII hodnotu, se kterou se dále počítá. Znakovou konstantu je možné též zapsat pomocí osmičkové nebo šestnáctkové soustavy, kdy hodnota v této soustavě je opět uvedena v apostrofech a za prvním je zpětné lomítko

'\012', '\0x0B'. Zpětné lomítko představuje tzv. escape charakter. Některé často používané escape sekvence, které jsou v práci s texty často používány, mají své zástupné vyjádření

\n	0x0A	nová řádka (linefeed - LF)
\r	0x0D	na zač. řádky (carriage ret. - CR)
\t	0x09	tabulátor (tab - HT)
\a	0x07	písknutí (alert – BELL)
\0	0x00	nulový znak (NULL)
\\		zpětné lomítko (backslash)
\'		apostrof (single quote)
\"		uvozovky (double quote)

Poslední tři slouží k zápisu znaků, kterých by jinak nebylo možno dosáhnout. Důležité je nezapomínat na to, že chceme-li zapsat zpětné lomítko, je nutno jej zapsat dvakrát (např. uvádíme-li cestu k souboru ve zdrojovém textu), protože první lomítko je interpretováno jako "následující znak je znakem speciálním", a tudíž se netiskne.

vlastní ASCII tabulka

- znaková sada popsaná ordinálními čísly 0 až 255
- jazyk C nepoužívá jiný systém (např. EBCDIC)

mezera	32	lépe zapisovat ekvivalentem ' '
čísllice	48 – 57	lépe zapisovat '0' ... '9'
velká písmena	65 – 90	lépe zapisovat 'A' ... 'Z'
malá písmena	97 – 122	lépe zapisovat 'a' ... 'z'

Zápis celočíselných hodnot je možný v desítkové soustavě, v osmičkové soustavě, kdy je první číslicí povinně nula, v šestnáctkové soustavě, kdy je prvními znaky povinně *nula* a *x*. Před toto číslo je možné předřadit znaménko + nebo -. Znaménko + je nepovinné. Tyto konstanty jsou ve standardním tvaru brány jako signed int. Chceme-li je v long, musíme za poslední číslici přidat "l" (malé písmeno eL) nebo "L". Pokud chceme, aby byla konstanta unsigned, přidáme na konec znak "u" nebo "U".

neznakové celočíselné konstanty

desítkové	0, +1, 81, -123
oktalové	0, 01, 052
hexadec.	0x0, 0X1, 0xC9, 0Xab, 0xFFFFu, 0xFFFFFfI, 0xFFFFFFFfLu

Máme-li danou proměnnou je možné ji v definici inicializovat. K inicializaci může sloužit konstanta nebo jiná proměnná. K inicializaci se používá znaménko =, které je zde třeba chápat ve smyslu "proměnná je vytvořena na základě dodané hodnoty" a ne "proměnné je přiřazena hodnota" (jinými slovy: k inicializaci se používají jiné mechanismy než k přiřazení, i když se pro oboje používá stejné znaménko =).

{	definice	proměnných	včetně	definice
---	----------	------------	--------	----------

<code>char znak = 'a';</code>	s inicializací v C je možná pouze na začátku bloku definice proměnné s inicializací. Je vytvořena proměnná s místem v paměti, které je naplněno danou hodnotou
<code>char zn, ba = 'c', zn3, poslední = '\0', lomítko = '\\'; char bbb = 32;</code>	je možné libovolně střídat definice proměnných s definicemi s inicializací a různé typy inicializací
<code>unsigned long maska = 0xffffffffUL;</code>	u typu long se preferuje velké L, protože malé l vypadá jako číslice jedna
<code>int cislo=2456, maska2 = 01234, ccc = 'B';</code>	

Plovoucí řádová čárka – pro reálné proměnné v jazyce C jsou určeny typy

float

double

long double

Pro velikost opět platí relace a přesná velikost (přesnost) je závislá na prostředí

`sizeof(float) <= sizeof(double) <= sizeof(long double)`

Reálné konstanty jsou čísla s desetinnou tečkou nebo s exponentem (norma IEEE). Exponent je uveden znakem "e" nebo "E". Rozlišuje se mezi celočíselnou a reálnou nulou – reálná má u sebe desetinnou tečku. Reálné proměnné jsou reprezentovány mantisou a exponentem (ordinální typy skládající reálné číslo). Jednotlivé typy se liší velikostí mantisy a exponentu. Např. přičtení jedničky k velké hodnotě float se neprovede v případě, že mantisa neobsahuje řád jednotek – obsahuje velké řády a je krátká.

zápis reálných konstant

typu double 0., 23. , -54.3, .65, 8e4, 7E-12

typu float 1.234f, 8.45f (je jen v novější normě, starší překladače nemusí tento typ konstanty umět, pak je nutné použít konverzi (float) 3.34)

typ long double 2.45L

Minimální a maximální hodnoty pro dané typy bývají předdefinovány a jsou součástí některého z knihovnických hlavičkových souborů (values.h, float.h, cfloat - mívají jména MAX_INT, INT_MAX, MAXINT ...)

Logické hodnoty

V jazyce C jsou logické hodnoty false a true reprezentovány nulou pro false a jiné hodnoty než nula jsou true. Pro true lze tedy použít jakoukoli hodnotu různou od nuly a naopak se může stát, že výsledek logické operace bude 0 pro false a libovolné číslo pro true. Nové překladače se snaží o výsledky 0 a 1, ale jsou schopny provádět i konverze z nula a nenula.

5.2.6 Typová konverze (přetypování)

Při výpočtech je někdy nutné převést jeden typ proměnné na jiný. K tomuto převodu dochází buď automaticky – implicitní konverze, nebo je možné si převod vynutit – explicitní konverze. K automatickému přetypování dochází ze strany překladače, k vynucenému ze

strany programátora. Jazyk C je v možnostech kombinací typů ve výpočtech dosti benevolentní. Pokud existuje možnost jak převést jeden typ na druhý, potom to automaticky udělá. Obě tyto konverze jsou realizovány během překladač – již překladač ví, jaký bude výsledný typ.

Implicitní konverze (automatická, samovolná)

Dochází k ní pro samostatné operandy při předávání do funkcí, kdy typy "menší" než int jsou konvertovány na int, typ float je konvertován na double.

Dále k automatické konverzi dochází při vyčíslování operací s operandy, kdy se typy převádějí na typ s nejvyšší přesností, která je v operacích použita.

Dále je použita při přiřazení, kdy se **výsledná** hodnota na pravé straně konvertuje na typ kterému bude přiřazena – to je na typ stejný jako má proměnná na levé straně. (viz. kapitola 5.2.7)

Hierarchie priorit typů od nejnižší je:

*char => int => unsigned int => long => unsigned long =>
float => double => long double*

Explicitní konverze (požadovaná, vynucená)

Použijeme ji v situacích, kdy máme proměnnou daného typu a potřebujeme s ní pracovat jako s proměnnou jiného typu a zároveň nedojde k automatické konverzi. Explicitní konverze se provede vždy a to i v případě, že dochází ke ztrátě dat

(typ) vyraz	před výraz, který chceme konvertovat dáme požadovaný typ na který chceme konvertovat
{ char c; float f;	definice proměnných
(float) c	konverze proměnné c na typ float – dále už by se počítalo s hodnotou, kterou má c, ale reprezentovanou v pohyblivé řádové čárce
(char) f	konverze na méně přesný typ je možná, hrozí však ztráta přesnosti či dat.
(int) f	tato konverze se využívá k "zaokrouhlování". Ve skutečnosti odřeže pouze desetinnou část a nechá celou část. Kladná i záporná čísla tedy "stahuje" směrem k nule.
(int) (f + 0.5)	zaokrouhlení na celá čísla pro kladné hodnoty f
(int) (f – 0.5)	zaokrouhlení na celá čísla pro záporné hodnoty f
	Pro zaokrouhlování existují knihovní funcce. (round, ceil, floor ...)

5.2.7 Operace s proměnnými, operátory

K operacím s proměnnými patří přiřazení, unární a binární operátory, logické a matematické operátory. V jazyce C najdeme i speciální operátory pro přičítání a odečítání jedničky a rozšířené přiřazovací operátory.

Přiřazení

Přiřazení je realizováno operátorem "=" a slouží k přiřazení hodnoty, která se nachází napravo do proměnné, které se nachází nalevo. Postup přiřazení je takový, že se vezme (výsledná) hodnota na pravé straně, ta se zkonvertuje na typ výsledku (proměnné na levé straně) a přiřadí se. Navíc je možné v C operátory "=" zřetěžit, takže po přiřazení "zůstává" přiřazená hodnota k dispozici pro další použití.

Aby mohlo dojít k přiřazení, musí se na levé straně nacházet proměnná, do které je možné výsledek uložit, tj. místo v paměti kam přijde výsledek. V C se nazývá l-hodnota (l-value - tj. to co může být vlevo od = a dá se tam uložit hodnota).

{ char c, c1;	definice proměnných
float f, f1 = 5;	= při inicializaci není přiřazení ale vytvoření proměnné dané hodnoty na základě jiné
c = 42;	přiřazení konstanty
c = c1;	přiřazení hodnoty jiné proměnné
c = f;	provede se implicitní (ztrátová) konverze f a výsledná hodnota se přiřadí c
f = c;	provede se implicitní konverze c a výsledná hodnota se přiřadí
c = c1 = '4';	několikanásobné přiřazení – ASCII hodnota znaku '4' se přiřadí do c1 a následně se hodnota obsažená v c1 přiřadí do c
852 = c;	konstanta vlevo nelze použít jako l-hodnota
f + 3 = f1;	výraz vlevo nelze použít jako l-hodnota

Aritmetické operátory

Unární

Patří sem operátory plus a mínus. Plus nemění hodnotu a jeho psaní je nepovinné. Mínus mění znaménko použité proměnné (proměnnou samu nemění).

př.+56, -8.3 +c1 -f

Binární operátory

+ operátor sčítání
- operátor odečítání
* operátor násobení
/ operátor dělení

% operátor modulo (zbytek po celočíselném dělení)

Jak bylo uvedeno výše, typ, ve kterém se provádí výpočty, je nejpřesnějším typem, který do operace vstupuje. Proto např. násobíme-li dvě celočíselné proměnné, které způsobí přetečení výsledku, je výsledek chybný (bez výrazného upozornění – u některých typů chyb bývá nastavena interní chybová proměnná). Řešením této situace je buď změnit typ jednoho parametru v definici, kdy se pro výpočet provede implicitní konverze, nebo provést přímo explicitní konverzi.

K podobným problémům může dojít i u dělení. Jsou-li parametry celočíselné, jedná se o celočíselné dělení. Je-li jeden z parametrů reálný, potom je i výsledek reálný se stejnou přesností.

{ int i, j=10, k=3; long l, m, nn; float f;	
i = j * k;	násobení dvou int hodnot, výsledek je int a je přiřazen do int (možná ztráta přetečením – které se ztrácí)
f = j * k;	násobení dvou int hodnot, výsledek je int (možná ztráta přetečením) a ten je teprve konvertován a přiřazen do float
l = m * k;	je-li jeden parametr long, je druhý též převeden do long a výsledek je také long
i = m * k;	výsledek pravé strany je long, ten je (automaticky) zkonvertován na typ na levé straně (ztráta z long na int)
nn = (long) j * k;	jeden z parametrů nuceně zkonvertujeme na long a tedy výsledek pravé strany bude long
f = j / k;	výsledek na pravé straně je celočíselný a následně je konvertována (celočíselná) hodnota na typ na levé straně (float) a přiřazena – ve f je 3.0
f = (double) l / m;	pravá strana je vypočtena v přesnosti typu double, výsledek je konvertován do float – ve f je 3.33333
i = j / k;	
i = j % k;	celočíselné dělení – výsledek je 3
	zbytek po celočíselném dělení – výsledek je 1
i = j * k + m * 3 / 4;	výrazy s více operátory jsou zpracovávány na základě priorit a asociativity. Část 3/4 bývá vyčíslena předem (preprocesor, překladač) jako konstanta tj. při typech int jako nula
i = j * k - - m;	první mínus je binární operátor, druhé mínus unární operátor (tj. je odečteno záporně brané m)

Speciální unární operátory ++, --

Z důvodu zjednodušení zápisu (a patrně z důvodu výběru instrukcí) jsou v C implementovány operátory pro přičtení a odečtení jedničky (instrukce pro přičtení a odečtení jedničky je kratší a rychlejší než totéž, je-li jednička jako přímý operátor instrukce – v současné době už oba zápisy při použití optimalizací dopadnou stejně). Oba tyto operátory mají prefixovou a postfixovou notaci (mohou stát před nebo za proměnnou, se kterou pracují). Jedná se o dvojznaky takže mezi znaménky nesmí být mezera.

<pre>{ int i, j=5, k=4;</pre>	
<pre>j++; ++j;</pre>	protože se proměnná dále (v tomto příkazu) nepoužívá, je výsledkem obou činností (pouze) zvýšení j o 1 (na 7) - zde je post i prefix ekvivalentní.
<pre>k--; --k;</pre>	snížení hodnoty k dvakrát o jedna (na 2)
<pre>i = ++j;</pre>	inkrementace před použitím - nejprve se vyčíslí pravá strana, tj. nejprve se přičte jednička a potom se j použije, tj. nejdříve se j zvětší na 8 a tato hodnota se použije k výpočtu (přiřadí se do i).
<pre>i = j++;</pre>	inkrementace po použití - nejprve se j použije a následně se inkrementuje, tj. použije se hodnota 8 a s tou se počítá, následně se j inkrementuje na 9. (i = 8 a j = 9)
<pre>i = j++ + j++;</pre>	tato konstrukce je nevhodná, protože u některých překladačů je její výsledek nepředvídatelný (např. se ++j provede předem, a j++ až po provedení řádku a ne postupně, popřípadě by se výsledek mohl stát obětí optimalizačního přerovnání parametrů). Nedoporučuje se používat více než jeden tento operátor na proměnnou a výraz.

Bitové posuny (shift)

Operátory bitových posunů jsou << a >>. Slouží pro posun proměnné (celočíslného typu) o daný počet bitů doleva nebo doprava. Na prázdná místa se doplňují 0 (pokud je na daném HW tento posun implementován).

<pre>c = 0x4; d = 2;</pre>	
<pre>b = c << 3;</pre>	v b bude hodnota 0x20, proměnná c se posune o 3 bity doleva
<pre>b = c >> d;</pre>	v b bude 0x1, proměnná c se posune o 2 bity doprava
<pre>b = 1 << (c + 2);</pre>	posun 1 o 6 bitů doleva

Logické operátory

Logické operátory mají verzi matematickou (bitovou) a logickou. U bitové verze se daná operace provádí pro každý bit samostatně, zatímco u logické operace se pracuje s hodnotou jako celkem (pracuje se tedy s proměnnou jako s hodnotou false, true). Logické operátory jsou většinou zdvojené operátory bitové. Dvojznaky opět nutno psát bez mezer mezi znaky.

Při vyhodnocování logických výrazů opět dochází k implicitní konverzi tak, aby k porovnávání docházelo v maximální možné přesnosti.

operátor	činnost	zápis	výsledek pro b = 0x3 c = 0x5 (a,b,c mají 8bitů) (b,c se nezmění)
=	přiřazení	a = b	a = 0x03
==	rovnost	b == c	false (nula)
!=	nerovnost	b != c	true (nenula)
!	(logická) negace	!c	false (nula)
~	(bitová) negace	~c	0xfa
&	(bitové) and	b & c	0x01
&&	(logické) and	b && c	true (nenula)
	(bitové) or	b c	0x07
	(logické) or	b c	true (nenula)
^	bitové exclusive-or	b ^ c	0x06
<, <=, >, >=	relace	b >= c	false (nula)

c = 1 << a;	vytvoření masky pro nastavení daného bitu (bit určen hodnotou v a – 0 je bez posuvu)
d = d c;	vlastní nastavení bitu v proměnné d
c = ~(1LU << a);	postup pro tvorbu masky na nulování bitu. Použití LU zajistí, že se pracuje s největším celočíselným typem. Oproti předchozímu tedy proměnná c může být i typu long (pouhá jednička je typu int). Použitím bitové negace dosáhneme toho, že všechny bity, kromě pozice s 1 budou po negaci 1 a jediný bit bude nastaven na 0 pro nulování daného bitu) a to v maximální přesnosti a nezávisle na zvolené platformě (bude fungovat pro délku proměnné a 1 byte i pro 4byty ...).
d = d & c;	vlastní nulování bitu v proměnné d
d = d && c;	v tomto případě dochází pouze k logickému vyhodnocení a v d je výsledek – true (nenula, u nových překladačů jedna) nebo false (nula) podle výsledku výrazu na pravé straně

Rozšířené přiřazovací operátory

Slouží ke zjednodušenému zápisu přiřazení, kdy je proměnná určená pro výsledek zároveň prvním operandem. Uvede se tedy pouze jako výsledná proměnná, následovaná rozšířeným přiřazovacím operátorem, který sestává s operátoru požadované operace a znaku = a následuje druhý operand.

a op= b	je interpretováno jako a = a op (b) – důležité je, že pravá strana se nejdříve vyjádří a potom se použije jako operand
+= -=	rozšířené přiřazení pro operátory + a –
*= /= %=	pro * / %
>>= <<=	pro posuny
&= ^= =	pro & ^

{ int i = 3, j = 5; i += j; i *= j – 3; i >>= 2;	i je 8 i je 16 i je 4 (posun bývá často použit pro dělení nebo násobení mocninou 2, zde dělení 4 – rychlejší než skutečné dělení, násobení)
--	---

Vyhodnocování výrazů

Výrazy se vyhodnocují podle pravidel o prioritě a asociativitě, která jsou uvedena v tabulce preferencí tab. 1. V logických výrazech se však výpočet může zastavit (část se nemusí ani dopočítat) v případě, že je jasný výsledek false nebo true a zbylá část ho již nemůže ovlivnit. Proto se do nich nedoporučuje dávat operátory ++, --, přiřazení, které mění hodnotu použitých proměnných, a které by se nemusely v tomto případě provést.

(y != 0 && x / y < z++)	v případě, že je první část false, nemá smysl vyhodnocovat část za && a tedy se neprovede inkrementace proměnné z
-------------------------	---

Tabulka 1. – tabulka priorit a asociativity operátorů

pr.	operátory	asociativita	
1	() [] . ->	->	(prioritní) závorky, indexace, přístup k prvku struktury dané hodnotou (operátor tečka), přístup k prvku struktury dané adresou
2	! ~ ++ -- + - (typ) * & sizeof	<-	(logické) negace, bitové negace, inkrementace, dekrementace, unární + a -, explicitní konverze, přístup k prvku na adrese, získání adresy, zjištění velikosti typu
3	* / %	->	binární (matematické) operátory
4	+ -	->	binární (matematické) operátory
5	<< >>	->	posuny
6	< <= > >=	->	relační operátory
7	== !=	->	(logické) operátory pro porovnání
8	&	->	matematické (bitové, bit po bitu) and
9	^	->	matematické (bitové, bit po bitu) exclusive-or
10		->	matematické (bitové, bit po bitu) nebo

11	&&	->	logické and
12		->	logické or
13	? :	<-	ternární operátor
14	= += -= /= %= >>= <<= &= = ^=	<-	přiřazení a rozšířená přiřazení
15	,	->	operátor (sdružování) čárka

-> asociativita zleva doprava

<- asociativita zprava doleva

Asociativita říká, jak se vyčíslují operace stejné priority (při $10/4*2$ mají znaménka stejnou prioritu, ale asociativita říká, že se nejprve použije první z nich. U $8/4+4*2$ může být kterákoli část vyčíslena dříve a závisí na optimalizacích překladače.)

Pro psaní složitějších výrazů platí poučka "Máš-li pochyby, závorkuj". Je to proto, že závorky svou vysokou prioritou zajistí přednostní provedení svého obsahu. Závorky zároveň usnadní čtení a urychlí orientaci díky rozdělení na menší celky.

Pozn.

výraz	expression	$i * 2 + 3$
přiřazení	assignment	$j = i * 2 + 3$
příkaz	statement	$j = i * 2 + 3;$

5.2.8 Funkce

Funkce jsou základem programu. Tvorba funkcí a jejich volání je základem pro přehlednost programu. Základní vlastnosti funkce byly zmíněny v 5.2.4. Funkce se skládá z hlavičky a těla. V hlavičce je stanovena návratová hodnota, jméno funkce a seznam parametrů. Návratová hodnota je jedna, jméno funkce musí být unikátní (v rámci celého programu), parametry jsou uvedeny svým typem a názvem a jsou odděleny čárkami. V jazyce C dochází vždy k předávání parametrů hodnotou. Z hlediska mechanismů volání funkce z toho plyne, že proměnná uvedená v hlavičce funkce je vždy lokální hodnotou (kopií, jejíž změna nemá vliv vně funkce). V případě, že potřebujeme zajistit předání hodnot z funkce v poli parametrů, použijeme k tomu ukazatele (5.2.17).

Úplný funkční prototyp je:

```
typ_výstupní_hodnoty Název_funkce(typ1 parametr1, typ2
                                parametr2);
```

Pro případy, kdy nepotřebujeme vrátit žádnou hodnotu, nebo funkce nevyžaduje žádný parametr je k dispozici typ **void**, který nemá rozměr a proměnná tohoto typu neexistuje.

Pro předání návratové hodnoty se používá klíčové slovo *return*. Hodnota parametru tohoto klíčového slova je přepsána do návratové hodnoty funkce. Můžeme použít zápis **return (vracený_parametr)**, který je přehlednější, závorky však nejsou povinné.

<pre>void Chyba(void) { /* zde se vytiskne univerzální chybové hlášení jako třeba " nastala chyba " */ return; }</pre>	<p>této funkci není předána žádná hodnota, ani funkce žádnou hodnotu nevrací.</p> <p>její použití je tedy velice univerzální, ale bez konkrétních hodnot (nepoužijeme-li globální proměnné). Funkce přesně neví co se mimo ni děje</p> <p>u funkcí, které nevrací žádnou hodnotu nepovinné</p> <p>závorka pro ukončení těla funkce zároveň značí return (zde umístěným breakpointem je většinou možné u debuggerů "odchytit" všechna opuštění funkce).</p>
<pre>----- Chyba(); void Tiskni(int Radek, int Sloupec, float Hodnota) { /* zde proběhne tisk – např. ve formě tabulky */ return ; } { int x, y, z; Tiskni(x, y, z);</pre>	<p>volání této funkce – je-li void, potom je při volání seznam parametrů prázdný</p> <p>aby se volání provedlo (příkaz), musí končit středníkem</p> <p>typy parametrů se mohou libovolně střídát. Funkce, která nic nevrací, ale má parametry. Funkce, která ví co má dělat s daty, která se mění vně funkce, a proto je nutné je předat do funkce. Funkce, která pracuje s danými parametry, nemá však z hlediska programu návratovou hodnotu.</p> <p>volání této funkce. Proměnné x, y, z jsou proměnnými volající funkce. Hodnotami těchto proměnných jsou pomocí implicitní konverze nastaveny počáteční hodnoty lokálních proměnných (Radek, Sloupec, Hodnota) – typy proměnných ve volání tedy nemusí být stejné jako typy v hlavičce funkce</p>
<pre>----- int Nacti(void) { float Znak;</pre>	<p>funkce, která nemá žádný parametr, ale která vrací hodnotu. Funkce, která ví co má dělat, kde získat data (implicitně dané zařízení – port, klávesnice, myš...) a která vrací výsledek. Funkce, která vrací hodnotu na základě zapsaného kódu bez závislosti na parametrech.</p> <p>definice (lokálních) proměnných pouze na začátku bloku</p> <p>zde je např. načtení z (programátorem) daného zařízení</p>

<pre> /* */ return Znak; } t = Nacti(); ----- double Nasob (float a, float b) { return a*b; } a = Nasob(3.14, Nacti() * u); </pre>	<p>vrácení hodnoty – hodnota vracená funkcí je typu int, pokud skutečně vracená hodnota je jiná, dojde k naplnění návratové hodnoty pomocí implicitní konverze</p> <p>volání této funkce. Je-li parametr typu void, uvádíme závorky ale bez parametrů</p> <p>-----</p> <p>funkce, která má parametry i návratovou hodnotu. Funkce, která vrací hodnotu na základě dodaných parametrů.</p> <p>je vypočten výsledek, který je následně konvertován do požadovaného typu výstupní proměnné</p> <p>volání dané funkce. V seznamu parametrů může být i konstanta (v případě nutnosti je implicitně zkonvertovaná na správný typ) nebo volání funkce (jejíž výsledná hodnota se stane parametrem volání). Je-li parametrem výraz, potom je nejprve vyčíslen výsledek, a tím je naplněn parametr.</p>
---	--

Rekurzivní funkce

Je funkce, která volá sama sebe. V jazyce C je možná.

<pre> int faktorial(int n) { return ((n <= 0) ? 1 : n * faktorial(n - 1)); } </pre>	<p>zde je již provedena deklarace, a proto je již znám prototyp funkce a tudíž se může použít uvnitř funkce</p> <p>je možné volat funkci faktoriál – rekurzivní volání zatěžuje zásobník a proto je při tvorbě těchto funkcí třeba brát zřetel na jeho velikost (je použit ternární operátor " ? : " kapitola 5.2.12)</p>
---	---

5.2.9 Příkazy preprocesoru, makra

Preprocesor je mechanismus, kterým prochází zdrojový kód ještě předtím, než je tento zpracován překladačem. Příkazy preprocesoru by měly začínat na prvním znaku řádku znakem #. Tento znak je potom následován vlastním příkazem preprocesoru.

#define

Má několik možností použití. Slouží k definici konstant, nebo maker, nebo k definici proměnných pro řízení překladu. V případě konstant a maker se jedná o nahrazování řetězce řetězcem. Programátor vidí první řetězec, překladač zpracovává druhý řetězec.

Makra bez parametrů, umožňují nadefinovat text, kterým se nahradí jiný text. V případě, že preprocesor narazí na symbolický název (text, řetězec znaků) shodný s prvním parametrem, nahradí ho textem (řetězcem), který je uveden jako druhý parametr. Slouží převážně k definici

konstant, kdy název je jednodušší na zápis či na zapamatování a dále je možné zjednodušit zápisy dlouhých řetězců kratším zápisem. Bývá zvykem zástupný název, první parametr, psát pouze velkými písmeny. Oddělovačem prvního a druhého parametru je mezera. Nedoporučuje se pokračovat komentářem na tomto řádku.

#define MAX 1000	definuje konstantu MAX která má hodnotu 1000. Ve skutečnosti preprocesor při nalezení textu MAX (shodný s prvním parametrem), nahradí tento textem 1000 (druhý parametr).
MAX	programátor vidí zapsáno MAX, překladač již pracuje s řetězcem 1000 a o MAX neví

#define PI 3.14	zde nadefinujeme řetězec pro PI
#define PI_2 2*PI	zde řetězec pro 2_PI, který je následně doplněn již vzniklým řetězcem pro PI (definice lze i zřetěžit 2_PI se nahradí textem 2*3.14)
	následně můžeme tedy používat dvě náhrady konstant PI a 2_PI
PI	je nahrazeno textem 3.14
PI_2	je nahrazeno textem 2*3.14

Jazyk C "neumí" pokračovat v rozděleném textu příkazu na dalším řádku. I když u normálních zdrojových textů již překladače toto spojení řádků umí, pro použití víceřádkových maker a při slušném psaní zdrojových textů je nutné použít standardní postup a to znak "\ " na konci řádku, který říká, že tento řádek není ukončen a pokračuje na řádku dalším. Tento mechanismus zlepšuje čitelnost nebo zamezí psaní textu za hranici zobrazené stránky.

#define ZKRATKA 2 \	díky znaku \ se napojí druhý řádek za první a text ZKRATKA se nahrazuje správně textem 2 * PI * PI. Bez tohoto znaku by ZKRATKA byla nahrazena textem 2
* PI * PI	

Druhou možností použití příkazu **define** jsou makra s parametry – umožňují podle předpisu vygenerovat (opakující se) kód. Stejně jako v minulém případě dochází k nahrazení řetězce řetězcem, ale je možné předat jako parametry řetězce, kterými se nahradí příslušné řetězce ve druhém textu.

Např. používáme-li často součet čtverců dvou proměnných, potom k tomu můžeme použít makro definované pomocí define. Objeví-li se v textu název tohoto makra s parametry, potom makro slouží jako předpis pro rozvinutí (nahrazení) textu podle předpisu, kde parametry jsou opět nahrazeny dodanými řetězci.

#define KVADRAT(x, y)	$x * x + y * y$	definice makra s parametry
KVADRAT (aaa, bbb)		"volání" makra s parametry. Narazí-li preprocesor na známý text makra KVADRAT, použije daný předpis k rozvinutí tak, že makro rozvine druhou částí textu, ve které ovšem za parametry x a y dosazuje řetězce dodané jako parametry, to znamená, že nahrazuje x

$aaa * aaa + bbb * bbb$ $KVADRAT(aaa, bbb) * c$ $aaa * aaa + bbb * bbb * c$	řetězcem aaa a y řetězcem bbb toto tedy "vidí" překladač. jiný způsob volání. Rozvine se makro a dále se pokračuje tak, jak je psáno toto se dostane do překladače což není přesně to co chceme, protože naší snahou je, aby se KVADRAT choval jako funkce, tj. aby došlo k násobení až výsledku
---	---

<pre>#define KVADRAT (x, y) (x*x+y*y)</pre> $KVADRAT(aaa, bbb) * c$ $(aaa * aaa + bbb * bbb) * c$ $KVADRAT(aaa, bbb + ddd)$ $(aaa * aaa + bbb + ddd * bbb + ddd)$	odstranění nevýhody z minulého příkladu pomocí závorek, které mají nejvyšší prioritu, a proto dojde nejdříve k vyčíslení závorky, a tedy k tomu co jsme chtěli opět použití makra, které způsobilo chybu se nyní do překladače dostane jako tento text a výsledek je již správný další možné volání a jeho verze pro překladač opět není správně, protože jsme čekali, že druhý parametr je součet dodaných proměnných
--	--

<pre>#define KVADRAT (x, y) ((x)*(x)+(y)*(y))</pre> $KVADRAT(aaa, bbb + bbb) * c$ $((aaa) * (aaa) + (bbb + ddd) * (bbb + ddd)) * c$	nevýhodu z minulého příkladu opět odstraníme použitím závorek, kterými tentokrát zajistíme, že se parametry nejprve vypočtou a potom teprve použijí použití makra, které způsobilo chybu a verze pro překladač, která již dělá to co po ní chceme
--	---

Z uvedeného plyne, že správný přístup k makru je ozávkovat používané parametry makra a zároveň ozávkovat makro. Makro se zdá složité, ale je nutné si uvědomit, že to s čím nadále budeme pracovat je použití volání makra, které nám zjednoduší práci v tom, že tento složitý zápis nemusíme vždy psát, napíše ho za nás preprocesor na základě našeho předpisu skrytého v makru a jemu dodaných parametrů.

Třetí použití #define používáme k definování symbolů, které umožní řízený překlad. Řízení překladu znamená, že máme možnost říci, která část kódu se přeloží a která ne. Nejčastěji se tímto způsobem mohou povolit kontrolní výpisy, které oznamují kudy program chodí nebo jaké hodnoty mají hlavní proměnné, nebo zobrazování výsledků či grafické výstupy. Tyto přepínače se také používají, jsou-li části kódu odlišné pro různé překladače.

Překládá se pak vždy kód upraven pro práci v daném prostředí. Tyto části jsou trvalou součástí souboru ale volitelnou součástí programu.

`#define TEST`

Tímto příkazem se pouze nadefinuje "přítomnost" proměnné bez dané hodnoty.

`#ifdef, #ifndef, #endif, #else, #elseif, #undef`

jsou příkazy, kterými se můžeme na přítomnost či nepřítomnost definované proměnné ptát a podle jejího výskytu např. větvit překlad programu – tzv. podmíněný překlad.

<code>#define TEST</code>	zde nadefinujeme proměnnou TEST
<code>#ifdef TEST</code>	zde se ptáme, je-li nadefinována proměnná TEST pomocí direktivy preprocesoru define tato část se posílá do překladače (protože TEST je nadefinován)
<code>#else</code>	přepínač pro druhou variantu tj. tato část se posílá do překladače není-li proměnná TEST nadefinována (v našem případě bychom toho dosáhli např. zakomentováním řádku s define, nebo změnou jména definované proměnné např. na TESTX
<code>#endif</code>	ukončení sekce začínající <code>#if...</code> Od tohoto místa se již vše posílá do překladače.
<code>#ifndef TEST</code>	pokud není definována proměnná TEST potom se tato část kódu posílá do překladače
<code>#endif</code>	ukončení bloku řízeného překladu, dále se opět posílá vše
<code>#undef TEST</code>	pracuje jako opačný povel k define, tj. proměnná TEST se vymaže z tabulky definovaných proměnných. V dalším tedy již není definována. Stejně jako define platí od místa uvedení až do konce překládaného souboru. Tento příkaz používat co nejméně z důvodu nepřehlednosti jeho uvádění uprostřed kódu.

Podmíněný překlad lze i vícenásobně větvit pomocí spojení příkazu `#else` a `#if` do `#elif`.

Výše uvedený princip je možné použít pouze pro jednu proměnnou. Pokud je podmíněný překlad závislý na více proměnných, potom se použije `#if` v kombinaci s `defined`

<code>#if defined(TEST) && !defined(GRAPH)</code>	tento zápis umožňuje převést pomocí <code>defined(symbol)</code> přítomnost na logickou úroveň a tu vyhodnotit standardními logickými operátory jazyka C. Tomuto zápisu se dává přednost V našem případě je podmínka splněna a tato sekce kódu se dostane do překladače, je-li definována proměnná TEST a zároveň není
---	---

	definována proměnná GRAPH
#endif	standardní ukončení podmíněného bloku

Předdefinovaná makra jsou v systémovém hlavičkovém souboru `ctype.h`

Pozn.: častou chybou je, že makra nelze v některých konstrukcích použít, např. je zajímavé zkusit tento typ použití - `if (MAKRO() ...) MAKRO(); else Něco;` kdy může dojít k chybě. Vyrobit-li takováto makra měli bychom u nich upozornit na tato omezení.

`#include`

Direktiva `#include` umožňuje vložit preprocesorem do daného místa uvedený soubor. To znamená, že překladači se jeví, jakoby byl do tohoto místa zkopírován (vložen) soubor s daným jménem. Tímto způsobem se vkládají hlavičkové soubory. Nedoporučuje se vkládat soubory, které obsahují kód.

`#include <jmeno_souboru>`

`#include "jmeno_souboru"`

Je-li jméno souboru ohraničeno ostrými závorkami, potom se soubor hledá v adresářích překladače (systémové hlavičkové soubory). Je-li název souboru v uvozovkách, hledá se v adresářích, ve kterém je zdrojový kód, tzn. v adresáři, ze kterého je čten aktuální soubor (uživatelské hlavičkové soubory).

Z důvodu možného vícenásobného načítání, které by mohlo vést ke zpomalení překladu, nebo k jeho zacyklení v případě, že by se dva hlavičkové soubory načítaly navzájem je doporučen následující postup

	ošetření vícenásobného načtení pro hlavičkový soubor <code>hlavni.h</code>
<code>/* toto je hlavičkový soubor hlavni.h */</code>	úvodní komentář k souboru
<code>#ifndef HLAVNI_1234345554</code>	na začátku je dotaz na unikátní symbol, který je definováno dále v hlavičkovém souboru. Tento test se provede při každém <code>#include "hlavni.h"</code>
	To co je zde, se provádí v případě, že unikátní jméno ještě nebylo nadefinováno, tzn. tento soubor se prochází poprvé a je tedy nutné ho zavést. Provádí se tedy pouze pro první výskyt <code>#include "hlavni.h"</code> . V případě, že soubor již byl načten se tato část a tedy i tento soubor přeskočí. Hovoříme-li o prvním, respektive dalším načtení hlavičkového souboru, myslíme tím načtení v rámci právě překládaného zdrojového ("*.c") souboru. Při překladu dalšího zdrojového souboru se opět začíná od začátku.
<code>#define HLAVNI_1234345554</code>	součástí prováděné části musí být definice unikátního jména symbolu označující načtení, průchod tímto hlavičkovým souborem

...	zde jsou ostatní deklarace Pozor: pokud se na konci tohoto souboru vyskytne chyba, může se projevit chybovým hlášením až na začátku souboru dalšího.
#endif	na konci souboru je ukončení bloku #ifndef platnost nadefinovaného symbolu nekončí s blokem ani souborem (".h"), ve kterém byla nadefinován, ale trvá dále. Skončí až s koncem zdrojového souboru, v rámci kterého byl definován.
/* konec souboru hlavni.h */	

Pozn.: následující direktivy preprocesoru se mohou lišit v závislosti na použitém překladači. Překladače mohou mít ještě další direktivy preprocesoru.

#asm

Tato direktiva umožňuje vkládat přímo strojový kód. V současnosti se nedoporučuje. Optimalizace jazyka C je na takové úrovni, že "ruční" práce už se nevyplatí. C má jednoduchý a rychlý kód.

#error HLASENI

Umožňuje vytištění chybového hlášení. V místě, kde je uvedeno, je vytištěno hlášení včetně souboru a čísla řádku, kde se vyskytlo. Používá se např. v kombinaci s #if, kdy je uvedeno ve větvi, která by teoreticky neměla probíhat (např. grafický výstup v ladícím režimu). Pokud takováto situace nastane, jsme na ni upozorněni.

5.2.10 Platnost identifikátorů, globální a lokální proměnné a funkce

V případě, že vytvoříme identifikátor (proměnnou, funkci), musíme vědět, kde jej můžeme použít. Použití může být omezeno od části funkce až po práci ve všech zdrojových textech projektu. Pomocí klíčových slov a umístěním definice ve zdrojových textech můžeme ovlivňovat umístění proměnné v paměti, její viditelnost (možnost použití, přístupu) v programu a délku jejího trvání. Obecně se snažíme identifikátory co nejvíce skrýt, aby při velkých projektech nedocházelo k jejich kolizím.

Důležité je rozlišovat mezi deklarací a definicí. Deklarace je pouze oznámení o existenci zatímco definice je spojena s vyhrazením paměti. Definice je (pro proměnné a funkce) zároveň i deklarací. Deklarace u funkce znamená, že uvedeme dané jméno funkce společně s typem předávaných parametrů a typem návratové hodnoty. Definice navíc obsahuje tělo funkce. Deklarace proměnné říká, že proměnná daného typu existuje, ale nepřidělí se jí paměť. Při definici je také uveden typ a název, ale dochází i k přidělení paměti (pro umístění proměnné a funkce).

Zde je nutné si uvědomit, že program v C se skládá z modulů a pro jejich spolupráci je nutné mezi nimi zveřejnit společné proměnné a funkce s oznámením jejich typů (různé typy zabírají různé množství paměti a pro práci s nimi se používají rozdílné instrukce). Deklarace je určena pro překladač, který musí (v okamžiku použití) vědět, jakého typu je proměnná, se kterou má pracovat, nebo jak předat parametry do funkce a z funkce (popřípadě z důvodu případných implicitních konverzí). Překladač nezajímá, kde (funkce či proměnná) skutečně

leží, či zda fyzicky existuje. Deklarace plně dostačuje k použití identifikátoru. **Deklarací může být uvedeno několik pro stejný identifikátor, nesmějí si však odporovat.** Definice je důležitá z hlediska umístění v paměti (na úrovni jednoho souboru kontroluje kolize či nepřítomnost (identifikátorů) překladač, na úrovni projektu linker) – **pro každý identifikátor musí dojít** v rámci dané úrovně (tj. na nejvyšší úrovni v rámci celého projektu) **právě k jedné definici** – vyhrazení paměti (pro proměnnou a pro tělo funkce). Pokud definice chybí, nebo je jich několik není možné zajistit správnou práci s proměnnou a dojde k chybě při tvorbě programu.

”Viditelnost” (tj. možnost použít daný identifikátor z hlediska překladače) je vždy od místa deklarace do konce bloku, ve kterém je deklarován. Na globální úrovni je identifikátor platný do konce souboru, ve kterém je deklarován (pro nový soubor (*.c) se začíná od začátku tj. není deklarován žádný identifikátor). Ve vnořeném bloku může být definována proměnná stejného jména. Tato po dobu trvání bloku ”překryje” proměnnou z nadřazeného bloku – vybírá se vždy nejbližší proměnná daného jména.

Paměťové třídy slouží k určení umístění v paměti. Jazyk C nezná funkci definovanou uvnitř jiné funkce. Proto jsou všechny funkce na stejné úrovni v paměti. U proměnných je možné nastavit jejich umístění do datové oblasti, do oblasti zásobníku nebo do registrů. Pro volbu umístění jsou definovány paměťové třídy auto, register, static, extern.

Třída auto znamená automatickou třídu a je implicitní třídou pro lokální proměnné (tj. proměnné definované na začátku programových bloků). Proměnné tohoto typu jsou umístěny na zásobníku a jejich trvání je ukončeno s koncem bloku ve kterém jsou definovány. Po ukončení bloku proměnná zaniká. Tento typ není v jazyce C implicitně inicializován. Klíčové slovo **auto** není nutno uvádět.

Třída **register** je určena pro často používané proměnné, které umísťuje do registrů. Je to alternativa pro lokální proměnné ke třídě auto. Proměnné opět trvají od definice do konce bloku ve kterém jsou definovány. Výhodou je rychlejší přístup a práce s proměnnou. Používá se nejčastěji pro řídicí proměnné cyklů. Konkrétní registr je přidělen překladačem. Není-li možné přiřadit registr, zůstává proměnná v paměti. U některých překladačů je možné pomocí přepínačů překladu zvolit typ registrové třídy. A to buď vše do registrů, nic do registrů, nebo podle určení programátora tj. (pouze v tomto režimu) na základě klíčového slova register. Klíčové slovo register má tedy spíše charakter doporučení pro kompilátor.

Třída **static** slouží k umístění proměnných do oblasti dat. To znamená, že proměnná v průběhu programu nezaniká, existuje trvale. Třída static se používá pro lokální a globální proměnné. Static pro lokální proměnné umísťuje proměnnou do oblasti dat, její viditelnost je však omezena blokem kde je definována. Proměnná existuje stále a při návratu do bloku má stejnou hodnotu jako při jeho minulém opuštění. Je-li definice spojena s inicializací, pak se provede pouze jednou při prvním průchodu funkcí (někdy již před startem programu). Pro globální identifikátory (proměnné i funkce) omezuje třída static jejich působnost na modul, ve kterém byly definovány (v ostatních modulech nedochází ke kolizi se stejně pojmenovanými identifikátory). Někdy se tento způsob nazývá vnitřní vazba vzhledem k souboru. Klíčové slovo static musí být uvedeno.

Třída **extern** je implicitní třídou pro globální identifikátory (funkce a proměnné). U deklarací funkcí a definicí funkcí i proměnných není nutno uvádět. U deklarací proměnných je to povinné. Identifikátory této třídy jsou přístupné pro všechny moduly projektu a jsou umístěny v oblasti dat. Tento způsob se nazývá vnější vazba vzhledem k souboru. Používat globálních proměnných by se mělo co nejméně z důvodu kolize mezi moduly.

Zvláštní paměťovou třídou je dynamická paměťová třída – viz 5.2.16.

Typové modifikátory

Libovolná proměnná určitého datového typu (př. **unsigned int**) zařazená do určité paměťové třídy (např. **static**), může být navíc modifikována typovým modifikátorem

Modifikátor **const** je náhradou za symbolickou konstantu definovanou pomocí `define`. Dává se jí přednost a má výhodu, že jí může být dán přesně požadovaný typ. Po inicializaci již její hodnota nesmí být měněna. Překladač většinou v místech použití nepoužívá proměnnou, ale dosazuje (v rámci optimalizace) přímo její hodnotu. Lze získat i adresu, kde je uložena.

Modifikátor **volatile** se využívá u proměnných, které mohou být měněny asynchronní událostí např. přerušením. V praxi to znamená, že každá práce s touto proměnnou se uskuteční přes její místo v paměti (při jejím optimalizovaném umístění do registrů by při případné změně hodnoty v paměti při přerušení nebyla tato změna zaznamenána běžícím programem). Dále může nastat změna proměnné při současném spuštění více procesů – při spolupráci programů

<pre>/* soubor funkce. c – zdrojový text funkcí a definice proměnných */ #include "funkce.h" int gi; float gf = PI; static double pom; static double Pomf(double p);</pre>	<p>V následujícím příkladu máme tři soubory – zdrojový soubor s funkcemi a proměnnými, ke kterému máme hlavičkový soubor a posledním je hlavní zdrojový soubor s funkcí <code>main</code></p> <p>zde je uveden popis s názvem souboru, data o souboru ...</p> <p><code>include</code> zajistí, že do tohoto místa bude vložen soubor <code>funkce.h</code> - z hlediska překladače se bude jevit, jako by sem byl zkopírován. Uvozovky značí, že hlavičkový soubor je v adresářích programu. Toto načtení zajistí načtení např. konstant, definů, ale také díky němu dojde ke kontrole typů globálních proměnných a prototypů funkcí. Kolize jmen a typů mezi <code>h</code> a <code>c</code> souborem je hlášena jako chyba.</p> <p>globální proměnná typu <code>int</code> – definice (je pro ni vyhrazeno místo v paměti). Leží v <code>datech</code>. Přístup všichni</p> <p>globální proměnná typu <code>float</code> – definice s inicializací Leží v <code>datech</code>. Přístup všichni</p> <p>je-li uvedeno <code>static</code>, potom tato proměnná existuje od tohoto místa do konce souboru, leží v <code>datech</code>, přístup pouze tento modul</p> <p>Prototyp funkce. Tato funkce je vidět od tohoto místa do konce souboru. V ostatních modulech není možné s ní pracovat.</p>
---	---

	<p>Přístup pouze tento modul. static názvy překryjí názvy z jiných modulů, které nejdou ani naincludovat (hlásí chybu vícenásobného výskytu)</p>
int funkce (float a)	hlavička funkce s určením typu návratové hodnoty a parametrů, které jsou zároveň lokálními proměnnými funkce. Přístup všichni.
{	pokračuje tělem a proto se jedná o definici – tvoří se kód
auto float b,e ;	lokální proměnná umístěná na zásobníku. auto není třeba uvádět (a taky se to nedělá). Leží na zásobníku. Přístup pouze tato funkce
static int Pruchodu = 0;	statická proměnná je umístěna v datové oblasti, inicializace hodnotou nula se provede pouze při prvním průchodu, při dalších vstupech do funkce bude mít hodnotu stejnou jako při minulém opuštění funkce. Leží v datech, přístup pouze tato funkce
{	další vnořený blok programu
register double c, e;	na začátku každého bloku mohou být definice. Vložení proměnné do registru zrychlí výpočty. Proměnná e ve vnořeném bloku "překryje" předchozí definici e a je přednostně používána až do konce bloku
c = Pomf(a);	i když Pomf byla pouze deklarována, pro překladač stačí, že ví s jakými parametry ji zavolat
b = c + c;	je možné pracovat se všemi proměnnými z "nadřazených" bloků
}	zde končí platnost proměnné c a vnořené e zde platí původní (float) proměnná e
Pruchodu++;	statická proměnná slouží jako čítač průchodů touto funkcí
return b;	

<pre> } double dpom; static double Pomf(double p) { return a * a; } /* konec souboru funkce. c */ </pre>	<p>v tomto místě končí platnost proměnných a, b</p> <p>globální proměnnou je možné definovat kdekoli mimo funkce. Je zřejmé, že toto umístění ztěžuje orientaci.</p> <p>zde je definice dříve deklarované funkce</p>
<pre> /* soubor funkce.h – hlavičky funkcí a deklarace proměnných */ #ifndef MOJE_FUNKCE_H #define MOJE_FUNKCE_H #define PI 3.14 extern const double D_PI = 3.1415; extern int gi; extern double gd; </pre>	<p>komentář obsahu, tvorby a změn daného souboru</p> <p>standardní ošetření vícenásobného načtení hlavičkového souboru.</p> <p>definice konstanty (pomocí symbolu preprocesoru) zajistí, že bude mít stejnou hodnotu (přesnost, počet desetinných míst) ve všech modulech. Typ dán zápisem konstanty – double. Je-li nalezen preprocesorem text PI je nahrazen textem 3.14</p> <p>vhodnější "definice" konstanty. Typ dán programátorem. Je v .h souboru protože netvoří kód (překladač se k této proměnné chová obdobně jako je tomu v případě define a proto nemusí D_PI fyzicky existovat a stačí deklarace). Extern u const se doporučuje uvádět z důvodů kompatibility mezi kódy v C a C++. Const je novější vlastností C++ převzatou do C.</p> <p>existuje proměnná typu int se jménem gi – tímto je možné ji používat v jiných modulech. Pokud totéž neudělám s proměnnou gf, a pokud ji použiji, zahlásí překladač chybu, že ji nezná v ostatních modulech kromě funkce.c.</p> <p>pokud je uvedena tato deklarace, je možné proměnnou používat – překladač to přeloží,</p>

<pre>int funkce (float a); char znak; #endif /* konec souboru funkce.h */</pre>	<p>ale chybu zahlásí linker, který ji nemá bez definice fyzicky umístěnou (nemá adresu, na které leží)</p> <p>hlavička funkce ukončená středníkem – deklarace. Netvoří kód, a proto může být v h souboru. Umožňuje správné umístění parametrů pomocí explicitní konverze při volání z ostatních modulů.</p> <p>definice v h souboru je chybou. Po naincludování souboru proběhne překlad dobře, ale linker objeví několik (zde dvě protože vkládáme 2x) proměnných znak – proto zahlásí chybu</p> <p>ukončení ošetření vícenásobného načtení hlavičkového souboru</p>
<pre>/* soubor hlavní.c – využívá funkcí z druhého souboru */ #include "funkce.h" #include "funkce.h" volatile int flag = 0; int main() { float ffp; int ab = 4;</pre>	<p>zde není možné použít žádnou proměnnou</p> <p>do tohoto místa je preprocesorem vkopírován soubor funkce.h, tj. od tohoto místa jsou přítomny všechny deklarované funkce a proměnné z hlavičkového souboru</p> <p>v hlavičce se zjistí, že již byla načtena a není již podruhé zpracovávána (soubor se však musí podruhé otevřít).</p> <p>proměnná přichystaná pro práci s přerušením. Dojde-li k asynchronnímu zásahu, změní se její hodnota (řekněme, že s touto globální proměnnou umí pracovat funkce, volaná při přerušení hlavního programu, nebo, že do této proměnné má přímý přístup HW zařízení – sdílená paměť). Inicializace proběhne před spuštěním programu.</p> <p>povinná funkce, která se volá jako první</p> <p>lokální proměnná</p> <p>díky deklaraci vložené z hlavičkového</p>

<pre> ffp = funkce(ab); while (flag == 0) ; return 0; } /* konec souboru hlavni.c */ </pre>	<p>souboru ví překladač jak vložit (implicitně konvertovat) proměnné tak, aby předání do i z funkce proběhlo správně</p> <p>zde se čeká dokud je podmínka splněná. V případě uložení v rámci optimalizace proměnné flag do registru by se změna proměnné během přerušovací rutiny nebo změna paměti externím zařízením v registru nepromítla a nedošlo by k opuštění cyklu</p> <p>zde končí život lokální proměnné funkce main</p>
--	--

5.2.11 Standardní znakový (terminálový) výstup / vstup

Vstup a výstup v C není součástí jazyka, ale je možné ho realizovat pomocí knihovních funkcí. Dělíme ho na znakový, kdy se pracuje se znaky a formátovaný, který umožňuje složitější vstup a výstup podle typu proměnné a typu jejího zobrazení.

Prototypy funkcí (některé z nich jsou vlastně makra), které realizují vstup a výstup jsou v hlavičkovém souboru stdio.h.

Funkce pro vstup a výstup z terminálu jsou odvozené z funkcí pro obecnou práci se streamy, které pracují defaultně s předdefinovanými streamy pro standardní vstup, nejčastěji klávesnici, který se jmenuje stdin, pro standardní výstup, nejčastěji monitor, který se jmenuje stdout, a standardní chybový výstup, nejčastěji monitor, který se jmenuje stderr.

Pro vstup jednoho znaku ze standardního vstupu se využívá `getchar()`, `getch()`, `getche()`. Načítaný znak je návratovou hodnotou funkce. Použití `getche` je stejné, jako `getch`, ale navíc k načtení znaku provede i jeho tisk – echo – na standardní výstup. Rozdíl mezi `getchar` a `getch` je ten, že `getch` vyčte ze vstupního bufferu první znak, není-li přítomen, čeká na jeho zadání a poté ho ihned vrátí a ukončí se. Funkce `getchar` vyčítá opět jeden znak ze vstupního bufferu, je-li přítomen (načten v minulosti do bufferu). Pokud je buffer prázdný, načítá znaky do bufferu až do doby dokud není zadán bílý znak, poté vrátí první zadaný (ostatní vrátí v pořadí jak byly zadány až při dalších voláních funkce `getchar`), nejsou-li znaky zadávány čeká. Pro ukládání načítané hodnoty se využívá proměnné typu `int` – speciální znaky s informací o streamu mohou nabývat záporných hodnot, u nichž by při ukládání do (`unsigned`) `char` mohlo dojít ke ztrátě informace.

Pro výstup znaku na standardní výstup se používá funkce `putchar` jejímž argumentem je tištěný znak. K odřádkování nedochází automaticky, ale je nutné vyslat escape sekvenci pro odřádkování. Podle typu zařízení je to `\n` nebo `\n` a `\r`.

<pre> #include <stdio.h> int main() { int c; </pre>	<p>načtení hlavičkového souboru s prototypy funkcí vstupu a výstupu</p> <p>definice pomocné proměnné pro čtení a tisk znaku – nepoužívat typ <code>char</code></p>
--	--

<code>c = getch();</code>	načte jeden znak. Čeká na jeho zadání, poté jeho hodnotu vrátí.
<code>putchar(c);</code>	vytiskne jeden znak
<code>putchar(' ');</code>	vytiskne jeden konkrétní (konstantní) znak
<code>c = getche();</code>	načte znak a zároveň ho zobrazí na výstupu
<code>do</code> <code>{</code> <code> c = getchar();</code>	je-li vstupní buffer prázdný, plní ho vkládanými znaky. První zadaný znak vrátí až v okamžiku zadání bílého znaku. Jsou-li v bufferu znaky od minula, načte nejstarší
<code> putchar (c);</code> <code>}</code>	tisk znaku
<code>while (c != 'Z');</code>	k ukončení tištění zadaných znaků dojde při zadání znaku 'Z'. Zbylé znaky v bufferu by se vyčetly v následujícím čtení.
<code>return 0;</code> <code>}</code>	

5.2.12 If – else, ternární operátor

Důležitým nástrojem při tvorbě programu je možnost větvení na základě splnění nebo nesplnění určitých podmínek. V jazyce C k tomu slouží příkaz **if** pro případ vykonání činnosti pouze při splnění podmínky nebo příkazy **if – else**, kdy se sekce po **if** provede při splnění podmínky, zatímco druhá větev při nesplnění. Podmínka je splněna v případě, že logický výraz nabývá hodnoty true (nenula), nebo přítomný matematický výraz je různý od nuly.

Obecně platí, že **else** patří k nejbližšímu **if**. Tento zápis však bývá nepřehledný a proto se doporučuje raději používat programové bloky.

<code>if (výraz)</code> <code> příkaz_1;</code>	příkaz if pro provedení příkazů při splnění podmínky jeden příkaz nebo blok programu
<code>if (výraz)</code> <code> příkaz_1;</code> <code>else</code> <code> příkaz_2;</code>	příkaz if – else pro provedení nezávislých větví pro splnění a nesplnění podmínky
<code>if (a > b)</code> <code>{</code> <code> ...</code> <code>}</code>	podmínkou je jakýkoli výraz zde definice proměnných a kód programu prováděné při splnění podmínky je-li příkazem blok, nesmí být za koncem bloku – ” } ” středník (tj. prázdný, druhý příkaz, který oddělí else od if)
<code>else</code> <code>{</code>	zde definice proměnných a kód programu pro nesplnění podmínky

<pre> ... } if (1) { ... } </pre>	<p>výrazem může být i konstanta (blok se provede vždy nebo nikdy (pro nulu)), nebo jen proměnná <code>if (b)</code> – pro <code>b = 0</code> se blok neprovede jinak ano</p> <p>zde definice proměnných a kód programu prováděné při splnění podmínky</p> <p>při nesplnění podmínky se nic neprovede a pokračuje se dál</p>
--	---

Alternativou k `if - else` je ternární operátor. Dá se použít pro jednodušší rozhodnutí typu podmíněné přiřazení. Je vhodnější při použití v makrech a díky tomu, že jeho výsledkem je hodnota, dá se použít i k volbě argumentu přímo při volání funkcí.

Zápis ternárního operátoru je *podm ? výraz_1 : výraz_2* . Je-li podmínka splněna, provede se *výraz_1*, není-li splněna, provede se *výraz_2*.

<code>i = (j == 2) ? k : 3;</code>	výsledkem ternárního operátoru a tedy hodnotou přiřazenou do proměnné <code>i</code> je <code>k</code> nebo <code>3</code> podle splnění podmínky
<code>k = (i == 1) ? i++ : j++;</code>	podle výsledku podmínky dojde k inkrementaci <code>i</code> , nebo <code>j</code> a do <code>k</code> se zapíše hodnota této proměnné před inkrementací
<code>i > j ? 1 : i < j ? -1 : 0</code>	ternární operátory je možné i sdružovat
<code>f(a, b, c, active && open?1:0);</code>	ve volání funkce se podle podmínky může nastavit parametr určující činnost funkce elegantněji než za pomoci <code>if-else</code>
<code>#define max(a, b) ((a)>(b)?(a):(b))</code>	makro pro určení maxima (nezávislé na typu proměnných <code>a</code> a <code>b</code>), na rozdíl od použití <code>if</code> lze použít v širším spektru případů

5.2.13 Cykly, opuštění cyklu - **for**, **while**, **do-while**, **continue**, **break**

Cykly se používají k opakování činnosti. V C platí, že k opakování vždy dojde v případě, že je podmínka splněna. Podmínka může být na začátku – cyklus **while**, na konci – cyklus **do – while**. Dále je možné použít cyklus **for**, který má inicializační část, následuje podmínka, blok programu a závěrečná část.

Pro všechny cykly platí, že musí obsahovat činnost, která ovlivňuje podmínku. Jinak by nebylo možné cyklus opustit. Jsou zde však ještě další klíčová slova související s cykly. Cyklus se dá opustit pomocí **return**, protože ten opouští celou funkci. Klíčové slovo **continue** způsobí, že se neprovede zbytek těla cyklu (tj. skočí se před konec bloku – závorku, který patří k nevnitřnějšímu, nejbližšímu cyklu – tedy ne k závorce nejbližšího konce bloku) – a prakticky se ihned pokračuje opět podmínkou, pouze u cyklu **for** se provede před podmínkou sekce iterace. Klíčové slovo **break** způsobí ukončení (jednoho) nevnitřnějšího cyklu – tj. skáče se

ihned za konec bloku náležející k nejvnitřnějšímu cyklu. Opustit cyklus lze tedy při nesplnění podmínky, nebo příkazem **break**. Příkaz ukončení celé funkce **return** samozřejmě cyklus také přeruší.

Cyklus **while** se používá, je-li potřeba provést test na podmínku na začátku cyklu.

while (podmínka) příkaz;	
while (a > b) a -= 2;	dokud je splněna podmínka provádí se následující příkaz
while (a > b) { ... if (a > 5) { ... continue; }	pokud je splněna podmínka provádí se blok programu začíná se definicemi proměnných ... příkaz continue ukončí aktuální průběh nejvnitřnějšího cyklu, ne bloku. Cyklus neukončí a pokračuje opět podmínkou závorka ukončující blok, ne cyklus, proto nemá vliv na continue
if (a < 0) break; }	příkaz break přeruší nejvnitřnější (nejbližší) cyklus, pokračuje za cyklem před závorkou končící nejvnitřnější cyklus se jde po continue
a = 4;	zde se pokračuje po break. Jde se za závorku končící nejvnitřnější cyklus
while (1) { if (a--) return; }	nekonečný cyklus uvnitř cyklu by potom měla být podmínka na základě které se cyklus opustí příkazem break nebo return
while(getch()=='a'){ }	dva způsoby cyklů, které pouze čekají na splnění podmínky – opakují se prázdné příkazy
while (getch()=='a') ;	středník za while "oddělí" while od následujícího bloku, který se již neopakuje

Cyklus **do-while** se používá, je-li nutný test podmínky na konci cyklu, tedy v případě, kdy je nutné aby tělo cyklu proběhlo alespoň jednou.

do { příkaz } while (podmínka) ;	
do { a -= 2; } while (a > b) ;	příkaz se provádí vždy poprvé pokud je splněna podmínka provádí se blok programu mezi do a while. Za podmínkou je u do-while středník
do { ... if (a > 5) { ... continue; } if (a < 0) break; } while (a > b);	příkaz continue ukončí aktuální průběh nejvnitřnějšího cyklu, ne bloku, pokračuje se podmínkou závorka ukončující blok, ne cyklus, proto nemá vliv na continue před závorku končící nejvnitřnější cyklus se jde po continue příkaz break přeruší nejvnitřnější (nejbližší) cyklus. Pokračuje za cyklem
a = 4;	zde se pokračuje po break. Jde se za podmínku

Cyklus **for** je cyklus, který umožňuje provést inicializaci, má podmínku na začátku podobně jako while a zároveň sekci iterací. Z těchto důvodů je vhodný pro cykly s řídicími proměnnými, protože máme přehled o jejich počáteční hodnotě a jejich kroku – změně, i o testovací podmínce přímo v "hlavičce" cyklu. "Klasická" definice, že cyklus for se hodí pro cykly se známým počtem iterací platí stejně i pro while – použití cyklu for v C je podstatně komplexnější.

**for (inicializace; podmínka; iterace)
příkaz;**

Pro oddělení jednotlivých sekcí v příkazu for se používá středník. Všechny sekce jsou nepovinné a mohou zůstat prázdné. Podmínka je jakákoliv podmínka, kterou je možné použít např. u if, s tím rozdílem, že ponecháme-li toto pole volné, potom se má za to, že podmínka je splněna. V rámci sekcí se u cyklu for často používá operátor sdružování čárka, který slouží k zřetězení příkazů. Máme tak možnost v jedné sekci použít více výrazů.

for (i = 0 ; i < 10 ; i++) f(i); for (int i = 0; i < 10; i++)	v první sekci je inicializace, která se provádí jednou před vlastním cyklem, dále podmínka která se testuje v každém cyklu a při jejímž splnění se opakuje tělo a iterace, která se provádějí až po těle v každém cyklu. příkaz – tělo cyklu, který se provádí je-li podmínka splněna
---	--

<pre>{ }</pre>	dva příklady s prázdným tělem cyklu proměnnou je možné definovat v inicializační sekci, podle normy by měla končit její životnost s koncem bloku for zde by již proměnná i neměla existovat
<pre>for (int i = 0;(i = getch()) != 'z') ;</pre>	čekání na znak 'z'. Kterákoli sekce může zůstat prázdná
<pre>for (i = 0,j=100;i < 10; i++, j -= 10)</pre>	operátor čárka umožňuje inicializaci a iteraci více proměnných zároveň
<pre>{</pre>	tělo cyklu tvoří blok příkazů
<pre>if (i == 5) continue;</pre>	provede sekci iterací a skočí před závěrečnou závorku cyklu
<pre>}</pre>	před tuto závorku skočí continue, ale předtím provede sekci iterací tj. i++, j -= 10
<pre>for (;;) { ... if (i > 10) break; ... }</pre>	nekonečný cyklus, není-li podmínka uvedena je pokládána za splněnou opustit cyklus se pak musí jinak. Zde pomocí break za tuto závorku se skáče ihned po break, cyklus se ukončí
<pre>for (;i < 10 && !end; i++) { }</pre>	složitější podmínka pro ukončení cyklu
<pre>for (i=0; i < 10 ; i++) { for (j=i+1;j<10;j++) { if (j == i) break; } }</pre>	cykly lze libovolně vnořovat, kombinovat ukončí pouze nejvnitřnější cyklus, tj cyklus přes j

5.2.14 Switch

Příkaz **switch** slouží k vícenásobnému větvení a je vhodný pro konstrukci konečných stavových automatů. Jeho nevýhodou je, že pro výběr větve je nutná přesná shoda celočíselného parametru a nedá se použít např. interval. Pokud nedojde k přesné shodě, je použita implicitní větev.

Klíčové slovo switch je následováno výrazem, jehož výsledek musí být celočíselnou hodnotou. Je-li tato uvedena v těle přepínače za klíčovým slovem **case**, potom se pokračuje od tohoto místa. Není-li shoda, pokračuje se od klíčového slova **default** – tato sekce je nepovinná. Sekce přepínače začínající **case** je lépe chápat jako návěští, od kterých pokračuje tok programu,

a které neznamenaají ukončení jeho provádění. Chceme-li opustit switch, musíme to udělat příkazem break. Příkazy switch je možné vnořovat.

<pre> switch (proměnná) { case hodnota : ... break; ... default: } int i; i = a + b * c; switch (i + 3) { case 0: case 2: f (i); c = 4; i = 4; case 4: b = c + i ; break; case 1: case 3: switch (c) { case 1: c = 0; break; case 2: c = 8; break; }; break; default: i = -1; break; }; </pre>	<p>vícenásobné větvení, podmínkou je celočíselný (ordinální) výraz, následuje otevírací závorka bloku na řádku se switch</p> <p>pro hodnotu 0 se začíná zde a pokračuje se</p> <p>pro hodnotu 2 se začíná zde. Je to společný kód pro hodnotu 0 a 2</p> <p>zde mohou být libovolné příkazy</p> <p>pro hodnotu 4 se začíná zde. Další společné pro 0,2 a 4</p> <p>vše pro aktuální větve (hodnoty 0, 2 a 4) bylo uděláno. Končíme</p> <p>Ize přidat další větvení (snižuje se přehlednost)</p> <p>break ukončuje nevnitřnější switch</p> <p>konec sekce 1,3. Pokračujeme za blokem switch</p> <p>pro ostatní (nevyjmenované) hodnoty se provádí default větev.</p> <p>je dobré ukončovat každou končící větev příkazem break i když např. v tomto případě nemění funkčnost, zmenší však pravděpodobnost chyby v případě přidání další větve, do které by předchozí neměla zasahovat</p> <p>na konci switch ukončovací závorka</p>
--	--

5.2.15 Ukazatele , typedef

Významným prvkem jazyka C je využití adres – v C nazývaných ukazatel (aglicky též pointer.)

Zde je dobré říci, že chceme-li pracovat s informací, je nutné ji někam uložit. Musíme dále vědět kolik informace je kvůli rezervaci místa potřeba na jejich uložení.

Pokud použijeme definici `int iii = 4;` potom `int` svým rozsahem (od minimální do maximální hodnoty) říká kolik místa bude potřeba k jeho uložení (to je zabudováno v překladači). Umístění je dáno dalšími klíčovými slovy, popř. umístěním definice – např. globální a statické proměnné v datech, lokální proměnné na zásobníku. Hodnota 4 potom říká, že do rezervovaného místa v paměti (reprezentované adresou na které leží proměnná `iii`) bude uložena hodnota 4. Název `iii` od nynějška slouží pro práci s obsahem dané paměti – tj. hodnotou, která zde je, nebo kterou sem uložíme.

Často potřebujeme pracovat s proměnnou prostřednictvím adresy na které leží. V jazyce C jsou k tomuto určeny právě ukazatele. Protože ukazatele slouží k práci s proměnnou, jsou spojeny s daným typem. Nereprezentují tedy pouze adresu ale i daný typ, který na ní leží (soulad mezi typem ukazatele a typem hodnoty na dané adrese musí zajistit programátor).

Je dobré si uvědomit, že proměnná typu `pointer`, stejně jako proměnné ostatních typů, je umístěna v paměti na nějaké adrese, a přístup k hodnotě v tomto místě zastupuje její název. V případě ukazatele je na tomto místě uložena adresa.

To že se jedná o ukazatel se v definici zapíše pomocí `*` tj. `int *u_iii;` Hvězdička slouží jako přepínač tj. platí pouze k následující proměnné. Pro všechny proměnné lze použít `typedef` pro nový typ ukazatel.

Typedef slouží k vytvoření nového typu a většinou se používá ke zjednodušení čtení. Například při vícenásobných ukazatelích je přístup pomocí `typedef` velice výhodný. `typedef` vytváří nový typ, ale ten je ekvivalentní typu původnímu – pracuje se s nimi stejně a jsou plně zaměnitelné a přiřaditelné.

Každý ukazatel by měl být inicializován, protože práce s neinicializovaným ukazatelem znamená přístup do paměti "která nám nepatří", na které mohou být data, kód programu ... jejichž přepsání vede k chybě či zhroucení systému. Inicializace je možná přiřazením již "vlastněné" paměti, tj. existující proměnné nebo požádáním systému o paměť se kterou se bude dále pracovat (5.2.16).

Byla zavedena speciální hodnota ukazatele `NULL` (definovaná v `stdio.h` (v některých překladačích nahrazeno přímo psáním 0 (číslice nula)), který značí, že daný ukazatel je neinicializován, nebo že při minulé práci s ním došlo k chybě. Při dalších použitíh ukazatele je nutné testovat, zda nemá tuto hodnotu a pokud ano, potom s ním nepracovat (např. u některých systémů je na této adrese tabulka přerušení a její nastavení na náhodné adresy dává zajímavé a nebezpečné výsledky a fatální důsledky).

K získání adresy slouží operátor `&`. K práci s hodnotou na kterou ukazatel ukazuje potom používáme operátor `*` (všimněte si, že operátor `*` má již tři významy – operátor násobení, v definici říká, že se jedná o adresu a před ukazatelem znamená, že se jedná o přístup k proměnné). Typ ukazatele a typ proměnné, na kterou ukazuje by měl být stejný (nemá smysl přistupovat `int`-ově k proměnné `float` ...)

<code>typedef int* P_INT;</code>	vytvoření nového jména typu, který je ukazatelem na <code>int</code>
----------------------------------	--

int i = 4;	proměnná typu int, v paměti zabírá příslušné místo na uložení typu int, v němž je uložena hodnota 4
int *p_i = &i;	proměnná typu ukazatel na int (int *). V paměti zabírá příslušné místo na uložení ukazatele na int (adresy), v němž se nachází adresa, na které je uložena hodnota proměnné i
int *p_ii, ii, *p_iii, iii;	definice dvou ukazatelů na int a dvou proměnných typu int
int ** pp_ii;	ukazatel na ukazatel na int. Ukazuje na místo v paměti, kde je uložen ukazatel, který ukazuje na proměnnou typu int
*p_i = 6;	dereference - "práce s adresou"- "přístup k adrese" - pokud je v p_i adresa, na které leží i, potom je toto ekvivalentní přiřazení i = 6; a tedy hodnota i se změní z 4 na 6. Tento zápis je l-hodnotou a může do něj být přiřazeno
P_INT p_j, p_jj,p_jjj;	definice tří ukazatelů na int
p_ii = ⅈ	inicializace není nutná v definici, ale nemělo by se na ni zapomenout
p_j = NULL;	standardní způsob jak říci, že ukazatel p_j nemá smysl, nedá se s ním pracovat. Tj. je neinicializovaný, nebo při minulé práci s ním došlo k chybě.
if (p_j != NULL) *p_j = 20;	správné ošetření přiřazení s testem na správnost ukazatele
*p_iii = 10;	velice špatné přiřazení do neinicializovaného ukazatele – zápis někam do paměti
iii = * p_iii;	špatné použití hodnoty neinicializovaného ukazatele – neplatná data
p_jjj = &i; p_j = p_ii;	inicializace (přiřazení do) ukazatelů definovaných pomocí typedef
pp_ii = &p_ii; *pp_ii = &i;	dereferencí pracujeme s ukazatelem na int, na který ukazuje pp_ii (díky inicializaci je to p_ii a tedy je přiřazení ekvivalentní p_ii = &ii.)
**pp_ii = 18;	výsledkem první dereference je ukazatel, který je opět dereferencován. Postupně tedy pracujeme s ukazatelem na int a intem. (ekvivalentní *p_ii = 18 nebo ii = 18)

5.2.16 Dynamická paměť

Při práci s většími proměnnými jako je pole, nebo větší datové typy je nutné též větší množství paměti, ve které budou tyto proměnné umístěny. Problémem je, že lokální proměnné vznikají na zásobníku, jehož velikost bývá omezená a zároveň se nedoporučuje využití

globálních proměnných. Řešením je tzv. dynamická alokace proměnných, při které požádáme za chodu programu systém o přidělení potřebné paměti z oblasti dat. Velikost může být prakticky libovolná, pokud je paměť fyzicky přítomna. Tato oblast je nadále reprezentována počáteční adresou této oblasti, kterou nám systém přidělí. V případě, že již s touto pamětí nepracujeme musíme ji vrátit (pomocí přidělené adresy) systému, protože C nemá automatické mechanismy na vrácení nepotřebných zdrojů (Java – garbage collection).

K práci s přidělenou pamětí se používají ukazatele a alokace je tedy druhou možností jak dosáhnout toho, že ukazatel ukazuje na platná data (vyhrazenou paměť).

K získání paměti a jejímu vrácení slouží knihovní funkce (z knihovny alloc.h). Tyto funkce jsou závislé i na použitém systému, takže se mohou jejich názvy měnit. Některé vlastnosti však mají společné. Předně jsou to run-time procedury, tzn. že přidělují a odebírají paměť za běhu programu podle potřeb programu bez předem známé velikosti.

Pro získání paměti je nutné vědět kolik jí chceme. Alokace se provádí v bytech (základní jednotka paměti) a proto je nutné počet bytů zjišťovat pomocí `sizeof(typ) * potřebný_počet_prvků`. Některé algoritmy mají pro určení velikosti paměti dva parametry. První říká počet prvků a druhý velikost typu těchto prvků. Funkce `malloc`, `calloc`, `farmalloc`. Konkrétní jména se mohou lišit v různých systémech.

Další důležitou vlastností těchto funkcí je, že přidělená paměť je typu ukazatel na void a na příslušný typ je nutné ji přetypovat.

Návrat paměti se uskutečňuje pomocí funkcí, které mají jako parametr ukazatel, který vrátila funkce pro přidělení paměti – jejich jména mohou být `free`, `cfree`, `farfree` a většinou nemodifikují předávaný ukazatel. Funkce pro naalokování a odalokování paměti jsou párové, tj. při určité alokaci je nutné použít příslušnou funkci pro odalokování.

Pokud se alokace nezdaří, vrací se `NULL`.

<pre>#include <alloc.h> int fce(void) { double *pi, *pj, *pk; pi = (double *) malloc (sizeof(double) * 10) ; if (pi == NULL) return 0; { int sd = sizeof(double); if((pj=(double*)calloc(20,sd))= = NULL) { free(pi); return 0;} }</pre>	<p>nutno načíst knihovnu pro práci s dynamickou pamětí</p> <p>ukazatele pro uložení ukazatelů na získanou paměť</p> <p>alokace 10-ti prvků typu double (alokujeme paměť v bytech, proto musíme vzít prvky a velikost jejich typu). Návratová hodnota je ukazatel na void, který přetypujeme na požadovaný typ výsledku</p> <p>pokud dostaneme <code>NULL</code>, potom nebyla paměť přidělena a končíme</p> <p>začátek bloku, za kterým můžeme dodefinovat další proměnné</p> <p>naalokování na základě počtu prvků a velikosti typu při chybě opouštíme funkci, ale nesmíme zapomenout vrátit již získanou paměť, protože by zůstala naalokovaná a přitom by o ní program nevěděl –</p>
---	--

<pre>pk = (double *) farmalloc (sizeof(double) *10) ; if (pk == NULL) { free(pi);free(pj); return 0;} ... free(pj); free(pi); farfree(pk); } return 1; }</pre>	<p>lokální proměnná ukazatele končí svou platnost všimněte si v podmínce závorek kolem přiřazení, které jsou zde nutné z důvodu vyšší priority ==, která by se provedla před přiřazením</p> <p>má-li platforma více typů paměti, může mít i více typů příkazů na přidělování paměti z jednotlivých typů</p> <p>zde práce s pamětí</p> <p>pořadí odalokování není dáno, je však nutno odalokovat vše naalokované pomocí ukazatelů získaných při alokaci a to funkcemi, které odalokují daný typ paměti. malloc i calloc přidělují ze stejné paměti stejným mechanismem a tak mají společné odevzdání paměti</p>
---	--

5.2.17 Funkce a ukazatele

Ukazatel a funkce spolu souvisí dvěma způsoby. Zaprvé je to možnost předávání ukazatelů jako parametrů z funkce a do funkce, a za druhé funkce leží v paměti a je tedy také možno ji reprezentovat ukazatelem.

Jak již bylo uvedeno, jazyk C umí předávat parametry do funkcí pouze hodnotou. Z toho plyne, že předávaná hodnota je lokální proměnnou ve funkci a její změna se neprojeví vně funkce. Ke změně proměnné vně funkce se proto používá ukazatelů. Ukazatel je předán opět hodnotou, ale ta reprezentuje adresu, na které leží hodnota, se kterou se pracuje. Proto se změna provedená přes ukazatel ve funkci promítne i mimo funkci.

<pre>void vymen(int *p_x, int *p_y) { int pom; pom = *p_x; *p_x = *p_y; *p_y = pom; }</pre>	<p>funkce, která vymění hodnoty dvou proměnných</p> <p>není možné předat hodnoty, ale je nutné předat ukazatele na proměnné, jejichž hodnotu budeme měnit</p> <p>při dereferenci přistupujeme k proměnné vně funkce a měníme její hodnotu</p>
---	---

<pre> { int i = 5, j = 3; int *p_i = &i; vymen(&i, &j); vymen(p_i, &j); </pre>	<p>při volání pomocí ukazatele musí být shodné typy proměnných</p> <p>do funkce se předávají adresy proměnných, jejichž hodnoty je třeba vyměnit</p> <p>při předání ukazatele se vytvoří ve funkci lokální proměnná, ale ukazující na proměnnou, kterou modifikujeme</p>
<pre> int * alokuj1(int kolik) { int * pom; pom = (int *) malloc(sizeof(int) * kolik); if (pom == NULL) return pom; for (int i=0 ; i<kolik ; i++) pom[i] = 0; return pom; } int alokuj2(int **pam, int kolik) { int * pom; *pam = NULL; pom = (int *) malloc(sizeof(int) * kolik); if (pom == NULL) return 0; for (int i=0;i<kolik;i++) </pre>	<p>funkce, která naalokuje a odalokuje paměť</p> <p>funkce, která naalokuje paměť (pro typ int) o daném počtu prvků, která všechny prvky inicializuje hodnotou nula</p> <p>pomocná proměnná do níž se uloží ukazatel na alokované pole</p> <p>pokud se alokace nezdařila vrátí malloc NULL jako chybové hlášení, které pošleme dál</p> <p>zde se nuluje pole (viz 5.2.18) - inicializace</p> <p>vracíme adresu-ukazatel na naalokovaného pole, návratová hodnota je také ukazatel na stejný typ</p> <p>druhá možnost realizace alokační funkce protože hodnota, která se má měnit je ukazatel, je nutné předat ukazatel ukazující do místa, kde tento ukazatel leží (adresa na níž je měněná adresa – hodnota).</p> <p>využívá stejný postup pomocné proměnné jako v předchozím</p> <p>na pozici ukazatele zapíšeme oznámení o chybě</p> <p>kód chyby v návratové hodnotě (duplicita k NULL)</p>

<pre> pom[i] = 0; *pam = pom; return 1; } int alokuj3(int **pam, int kolik) { *pam = (int *) malloc(sizeof(int) * kolik); if (*pam == NULL) return 0; for (int i=0;i<kolik;i++) (*pam)[i] = 0; return 1; } void odalokuj1(int * pam) { free(pam); pam = NULL; } void odalokuj2(int **pam) { free (*pam); *pam = NULL; } { int *p1, *p2, *p3; p1 = alokuj1(23); alokuj2(&p2, 65); alokuj3(&p3, 120); ... odalokuj1(p2); p2 = NULL; odalokuj2(&p1); odalokuj2(&p3); </pre>	<p>ukazatel na alokovanou paměť se přepíše vně pomocí změny hodnoty v místě předaného ukazatele</p> <p>návratová hodnota "v pořádku"</p> <p>stejně jako verze 2 ale bez pomocné proměnné</p> <p>hodnoty se zapisují přímo pomocí předaného ukazatele</p> <p>složitý zápis pomocí kulatých závorek je nutný z důvodu větší priority [] před *</p> <p>funkce odalokující paměť</p> <p>změní se pouze lokální proměnná – vně se neprojeví (chyba)</p> <p>funkce, která odalokuje paměť a nastaví ukazatel na NULL znamenající, že je neinicializovaný</p> <p>volání této funkce vyžaduje přiřazení</p> <p>ve volání těchto funkcí se mění přímo parametr</p> <p>je možné volat libovolné odalokuj.</p> <p>Tato první verze je nevýhodná v tom, že nastavení neplatnosti ukazatele na NULL se musí udělat zvlášť</p> <p>zde se po odalokování nastaví ukazatele na NULL automaticky</p>
--	--

--	--

Již při alokování paměti jsme si všimli použití ukazatele na typ void. Tento ukazatel se používá v okamžiku, kdy je možné použít ukazatel na libovolný typ. Vráť se blok paměti o daném rozsahu, což je univerzální, a následně se přetypuje na požadovaný typ. Dále se používá v místech kdy, se mohou předávat různé typy ukazatelů přes společný parametr.

<pre>void vymen_p(void **p_x, void **p_y) { void *p_pom; p_pom = *p_x; *p_x = *p_y; *p_y = p_pom; } int *i1, *i2; double *f1, double *f2; void *vpom; vymen_p(&i1, &i2); vymen_p((void*) &f1, (void *)&f2); vpom = (void *) f1; vpom = (void *) i1; ... f1 = (float *) vpom;</pre>	<p>funkce, která vymění obsah dvou ukazatelů (bez ohledu na to na co ukazují – oba však na tentýž typ) předá se ukazatel na ukazatel</p> <p>pomocná proměnná typu ukazatel</p> <p>mění se hodnoty tj. ukazatele</p> <p>k hodnotám vnějších ukazatelů přistupujeme přes předané ukazatele (jejich dereferenci)</p> <p>příklad volání</p> <p>generický ukazatel pro pomocné uložení</p> <p>výměna hodnot ukazatelů (tj. ukazují nyní na paměť na kterou ukazoval před výměnou ten druhý)</p> <p>pokud na nás bude překladač přísný (měl by být), potom musíme provést přetypování na souhlas typů</p> <p>při práci s void ukazatelem je nutné přetypování. Může sloužit např. k uchování původní hodnoty ukazatele, který se bude měnit. Můžeme ho postupně použít pro různé typy.</p>
--	--

Ukazatel je též spojen s funkcí. Jméno funkce je v C vlastně ukazatel na "vstupní bod" funkce a dá se tedy použít k jejímu volání. Dá se tedy použít i k předání do jiné funkce.

<pre>double secti(double f, double g) {return (f + g); } double nasob(double f, double g) {return (f * g); }</pre>	<p>dvě funkce se stejným prototypem (stejně typy a počet parametrů a předávaných hodnot)</p>
--	--

<pre>double poc(double f, double g, \ double(*p_f)(double, double)) { ... return (*p_f)(f, g)); } int main() { double (*p_fd) (double, double); ... p_fd = (parametr) ? secti : nasob; z = (*p_fd)(x, y); z = poc(x,y,secti); z = poc(x,y,nasob); z = poc(x,y,p_fd); return 0; }</pre>	<p>funkce může být i parametrem jiné funkce. Použití např. v případě, že na základě hodnot f, g se obtížně získají hodnoty pro vlastní výpočet, který je relativně jednoduchý ale mění se</p> <p>vlastní použití – volání předané funkce</p> <p>příklad použití ukazatelů na funkce</p> <p>definice p_fd je ukazatel na funkci, která má dva parametry typu double a vrací double</p> <p>na základě načteného parametru se zvolí jedna z funkcí</p> <p>a takhle se zavolá ta vybraná</p> <p>předání funkce (secti, nasob) do funkce poc</p>
--	---

5.2.18 Jednorozměrné pole, ukazatelová aritmetika

Častým požadavkem při programování je přístup k hodnotám stejného typu pomocí indexu - pole hodnot. V jazyce C je pojem pole úzce spojen s ukazateli, na které se práce s polem konvertuje.

Pole mohou být statická – tj. vytvořená jako lokální proměnné, a dynamická – tj. alokovaná v datové paměti v době běhu programu. Protože se lokální statická pole vytvářejí na zásobníku, který má omezenou velikost, doporučuje se využívat především pole dynamická (s přihlédnutím k rozsahu pole v paměti). S oběma typy polí se pracuje podobně.

Jak již bylo ukázáno při alokaci dynamické paměti, kdy jsme alokovali místo pro více než jen jeden prvek, je možné tento blok přiřadit jedinému ukazateli.

Přístup k prvku v poli se dá chápat tak, že k ukazateli, který reprezentuje počátek, je pomocí indexu přidán offset kterým se dostáváme k požadovanému prvku. Zde se opět objevuje vazba na to, že ukazatel je svázán s typem proměnné. Díky této vlastnosti index udává počet prvků a ne fyzickou vzdálenost v paměti. Indexy mohou být kladné i záporné a jejich použití nebo práce s nimi mimo pole není v C nikterak kontrolováno. Tato kontrola je plně v kompetenci programátora. Z důvodu indexace je rozsah pole od nultého prvku, protože první (a spíše tedy nultý) prvek pole leží přímo v místě ukazatele reprezentujícího pole. Posledním prvkem pole o délce NN je tedy prvek s indexem NN-1.

S indexací ukazatelů souvisí pointerová nebo ukazatelová aritmetika. Znamená to, že k ukazateli je možné přičíst nebo odečíst celé číslo a výsledkem je to, že máme ukazatel, který

ukazuje o daný počet prvků dál (blíží). Dále pak můžeme při odečtu dvou ukazatelů (stejného typu, nad stejným polem) zjistit jaká vzdálenost – kolik prvků daného typu – je mezi nimi. Potřebujeme-li znát skutečnou vzdálenost v paměti, musíme násobit získaný počet prvků velikostí daného typu. Sčítání ukazatelů není definováno. Je však možné ukazatele porovnávat a testovat na shodu pomocí logických operací <, >, <=, >=, ==, !=.

I když je použití polí a ukazatelů úzce spojeno může, překladač mezi jednotlivými reprezentacemi rozlišovat a proto je nutné uvádět stejně definici a deklaraci (např. `int y[3][4]` nedeklarovat jako `extern int **y`).

<pre>#define MAX 100 { int Pole[MAX], Pole1[20] , i; int *Dpom, *Dpom1; int *Dpole = (int*) malloc (sizeof(int)* delka); for (i = 0; i < MAX; i++) Pole [i] = i; for (i = 0; i < delka; i++) Dpole[i] = i; for (i = 0; i < MAX; i++) *(Pole + i) = i; for (i = 0; i < delka; i++) *(Dpole + i) = i; Dpom = Dpole; for (i = 0; i < delka; i++, Dpom++) *Dpom = i; i = Dpom - Dpole; i *= sizeof (int); Dpole[0] = 1; *(Dpole + 0) = 1; *Dpole = 1; Dpom1 = &Pole[5]; Dpom1 = Pole + 5;</pre>	<p>definice pole, je nutné uvést jakého typu pole je a v hranatých závorkách počet prvků, který musí být konstanta</p> <p>definice dynamického pole. Jeho délka může být libovolná, typ pole udává typ ukazatele</p> <p>přiřazení prvkům pole je možné pomocí indexů v hranatých závorkách. Indexy probíhají od 0 a při délce pole je prvek s indexem délky pole prvním prvkem, který je za polem a tudíž se do něj nesmí zapsat. V poli budou hodnoty odpovídající indexům</p> <p>Druhá (ekvivalentní) metoda, používaná spíše pro ukazatele je přičíst k ukazateli celé číslo, což ve výsledku dává ukazatel, který ukazuje na prvek o daný počet prvků dál a následně pracovat s touto adresou.</p> <p>Pole lze naplnit i tak, že se v něm pohybuje ukazatel. Toto přiřazení však nekopíruje pole ale pouze přiřazuje ukazatele. Máme tedy dva ukazatele ukazující na totéž pole. Změny provedené pomocí jednoho ukazatele se projeví i v druhém poli</p> <p>na konci cyklu se posuneme na další prvek do aktuálního prvku zapíšeme hodnotu</p> <p>v proměnné i by měla být hodnota delka, protože mezi ukazateli je celkem delka prvků typu int</p> <p>v proměnné i je delka pole v bytech (tj. paměť) využitá pro pole</p> <p>ekvivalentní přístupy k prvnímu prvku pole</p>
--	---

<pre>Dpom1 = &*(DPole+5); free (Dpole); }</pre>	<p>Dpole[nn] je ekvivalentní *(Dpole + nn)</p> <p>možnosti zjištění ukazatele na pátý prvek pole lze i tato (horší) varianta</p> <p>dynamické pole je třeba odalokovat</p> <p>zde končí platnost polí Pole a Pole1</p>
---	--

Při použití sizeof na statické lokální pole by mělo dojít u nové implementace k vrácení délky pole v bajtech, stará implementace může vrátit velikost ukazatele. U dynamických polí je výsledek sizeof vždy rozměr ukazatele.

5.2.19 Řetězce

Speciálním případem jednorozměrného pole je tak zvaný řetězec. Jedná se o jednorozměrné pole typu char, které má tu vlastnost, že obsahuje znak '\0', který řetězec ukončuje. Pro řetězec je tedy nutné naalokovat prostor o jeho délce plus jeden bajt na ukončovací znak ('\0').

<pre>#include <stdio.h> #include <string.h> #include <alloc.h> { char *s_dyn; char s_stat[10]; char s_1[10] = "ahoj"; char s_2[] = "nazdar" char s_3[]={'a','b','\0'}; char adresar[]="c:\\tmp\\"; s_stat[0]='b'; s_stat[1]='\0';</pre>	<p>knihovna pro práci s řetězci</p> <p>statická definice pole char bez inicializace – pro řetězec nutno dodat ukončovací znak</p> <p>statická definice řetězce s inicializací. Řetězec má rezervovanou paměť 10 prvků, ze které využívá 5 (4 + 1). toto přiřazení je možné pouze v inicializaci a ukončovací znak přidá překladač</p> <p>statická definice řetězce s inicializací, stejné jako v minulém případě, až na to, že není-li uvedena velikost pole, zjistí si ji překladač z velikosti řetězce jímž chceme provést inicializaci (6+1)</p> <p>řetězec definovaný po znacích ekvivalent = "ab";</p> <p>při psaní zpětného lomítka (součást cesty k adresářům DOS) je nutné si uvědomit, že se jedná o speciální znak pro escape sekvence a je tedy nutné ho zdvojit (jinak se interpretuje jako escape sekvence nebo se ignoruje když nenásleduje escape znak)</p> <p>řetězec zůstává polem a tak je možné přistupovat standardně k jeho prvkům</p> <p>nyní už je s_stat platný řetězec</p>
--	---

<pre>s_dyn=(char *) malloc(10); strcpy(s_dyn, "Ahoj"); strcpy(s_2,"nazdarek"); for (i=0;i<10-1;i++) s_stat[i] = '0'+ i; s_stat[10-1]= '\0'; free(s_dyn); }</pre>	<p>dynamická alokace pole pro uložení řetězce</p> <p>inicializace je nutná pomocí knihovních funkcí – kopírování mezi řetězci. Knihovní funkce pro řetězce očekávají přítomnost ukončovacího znaku, podle kterého se orientují</p> <p>lze použít i pro statická pole, zde ovšem chybně, protože jsme překročili maximální povolenou délku (8+1), kterou je možné v s_2 uložit (6+1)</p> <p>naplnění řetězce ASCII hodnotami pro číslice</p> <p>poslední znak musí být zakončovací znak (takže číslice jsou jen 0 – 8)</p>
---	---

Pro práci s řetězci je možné použít standardních knihovních funkcí (viz. 5.2.22).

5.2.20 Funkce a pole

Pole jako parametry funkcí

Předáváme-li pole jako parametr funkce, potom můžeme uvést rovnocenný zápis s hranatými závorkami, nebo ukazatelem. Při použití hranatých závorek nemá smysl uvádět rozměr (necháme je prázdné), protože pro C nemá tento rozměr žádný význam. V obou případech se předává hodnota. Protože však tato hodnota reprezentuje ukazatel, je možné přistupovat k prvkům v poli i uvnitř funkce (není ovšem možné změnit parametry pole jako je jeho umístění nebo délka). Je samozřejmé, že typ předávané proměnné by měl být shodný s typem, který je uveden v deklaraci funkce.

<pre>void max(double pole[], int poc, double *p_m) { double *p_p; *p_m = pole[0]; for (p_p = pole+1; p_p < pole+poc; p_p++) { if (*p_p > *p_m) *p_m = *p_p; } }</pre>	<p>funkce, která vrátí hodnotu (pomocí ukazatele na proměnnou pro uložení výsledku) největšího prvku v poli dané délky</p> <p>hlavička funkce. Jako první parametr má pole, délka pole uvnitř [] se neuvádí (pokud ano má pouze informativní charakter pro programátora), ekvivalentní předání pole je double *pole</p> <p>přístup k prvkům pole pomocí hranatých závorek</p> <p>přístup k prvkům pole přes ukazatele, oba přístupy k prvkům jsou povoleny</p>
--	---

<pre> } } int delka (char *str) { int i; i = 0; while (str[i] != '\0') ; return i ; } { double DSpole[20], *DDpole; char CStext[190], *CDtext ; double MaxVal; int dd; ... max(DSpole,20,&MaxVal); max(DDpole, 100, &MaxVal); dd = delka(CStext); dd = delka(CDtest); </pre>	<p>funkce, která vrátí délku řetězce předává se řetězec, předání (char str[]) je ekvivalentní</p> <p>oba přístupy k prvkům pole jsou ekvivalentní</p> <p>volání se statickým a dynamickým polem</p> <p>volání se statickým a dynamickým řetězcem</p>
--	--

Pole pointerů na funkce

Prvkem pole může být jakýkoli typ a tedy i ukazatel (5.2.24) nebo i ukazatel na funkci (stejného prototypu).

<pre> typedef void (* P_FCE)(char *); P_FCE funkce[10]; void (* funkcepl[10]) (char *) ; funkce[0] = edit; funkcepl[0] = edit1; funkce[1] = file; </pre>	<p>pro zlepšení orientace nový typ, ukazatel na funkci s parametrem řetězec nyní máme pole 10 ukazatelů na daný typ funkce lze zapsat i takto (viz 5.2.31)</p> <p>použití např. při výběru funkcí podle indexu</p>
--	--

5.2.21 Formátovaný vstup a výstup

Kromě znakově orientovaného vstupu a výstupu je v C (opět jako knihovní funkce) implementován i vstup a výstup složitějších typů. Není ovšem čistě typově orientován – funkcím realizujícím vstup a výstup je nutné sdělit jakého typu jsou vstupní či výstupní proměnné.

Pro načítání se využívá funkcí s proměnným počtem parametrů. První parametr udává, kde nebo co je zdrojem, popř. výstupem dat – může to být standardní vstup/výstup, soubor, paměť ... Tento parametr je u některých funkcí volen implicitně (např. obrazovka) a proto se nepředává (není uveden v hlavičce). Druhým a jediným povinným parametrem je řídicí řetězec formátu, který udává co a jakým způsobem se bude tisknout. Dalšími parametry jsou proměnné,

kteře se budou tisknout, a o nichž se předpokládá, že jejich počet je proměnný a není znám ani jejich typ (z hlediska prototypu a návrhu funkce. Programátor při jejich použití počet a typ zná).

Funkce podle své hlavičky neví, co je jí předáno a jak z toho má vytvořit výstup. Tyto informace jsou jí dodány pomocí řídicího řetězce formátu, za jehož úplnost a shodu s předávanými parametry (počtem a typem) nese odpovědnost programátor (některé překladače umí kontrolovat při překladu).

Řídicí řetězec formátu je řetězec, který obsahuje tři typy informace. Ty mohou být použity v libovolném pořadí. Jsou to:

- tisknutelné znaky – znaky se přímo tisknou do výstupního zařízení. U vstupního zařízení se čeká na shodu (je-li uvedena čárka, čeká se na načtení čárky ze vstupu).
- escape sekvence – vybrané znaky uvozené zpětným lomítkem sloužící především k formátování a tisku "problematických" znaků jako ', ", \
- formátové specifikace – jsou uvozeny znakem % a následující znaky nesou informaci o tom, jakého typu je předávaná proměnná (typ proměnné na zásobníku) a jak ji zobrazit. Formátová specifikace uvádí typ, šířku pole pro tisk a formát tisku.

Formátová specifikace má tvar (některé sekce nemají smysl pro načítání)

%		určuje, že se jedná o formátovací specifikaci. Povinné. Programátor musí zaručit, že v seznamu parametrů bude uvedena proměnná typu, který je shodný s určením této specifikace.
příznaky	- + #	udává jak se vytiskne proměnná v poli a vyplnění pole u čísel provede zarovnání na levou část pole, zprava doplní mezery číslo bude vždy vytištěno se znaménkem (+ se normálně netiskne)
šířka	'□'	před typ o přidává 0 před typ x, X přidává 0x, 0X pro f, e, E výsledek vždy obsahuje desetinnou tečku pro g, G vždy desetinná tečka a neodstraňuje koncové nuly (mezera) pro kladná čísla se místo znaménka tiskne mezera
•	n 0n *	udává minimální počet tištěných znaků (je-li např. kvůli platným místům nebo délce čísla nutné použít více znaků, pak se použijí. Je tedy použito pokud se při skutečném tisku použije znaků méně, jinak se ignoruje). (číslo) udávající minimální počet tištěných znaků, mezery se doplňují zleva
•	n	totéž, ale doplňují se zleva nuly
přesnost	n 0 *	hvězdička udává, že číslo udávající šířku je předáno funkci v poli argumentů (na pozici, která odpovídá dané pozici v řetězci, předchází tedy parametru kterého se týká) (tečka odděluje šířku a přesnost) (číslo) udává přesnost (počet platných míst) pro celá čísla totéž co šířka pro f, e, E počet číslic za desetinnou tečkou pro g, G max. počet významových číslic pro s max. počet tištěných znaků

gcvt – pro float ve Windows
a další ...

5.2.22 Standardní funkce pro práci s řetězcí

Pro práci s řetězcí je navržena knihovna string.h. Jsou zde funkce pro práci s řetězcí přímo, pro práci s částmi řetězců (končí se buď při dosažení konce řetězce nebo limitu znaků) popř. pro práci "z opačné strany" řetězce. Většina těchto funkcí nemění naalokovaný prostor pro práci s řetězcí a proto je nutné (např. při spojování dvou řetězců) navrhnout vstupní parametry funkcí (délky řetězců) i s ohledem na předpokládaný výsledek.

#include <string.h> int strlen(char *s);	nutno vložit délka řetězce	vrátí počet znaků bez ukončovací nuly
char *strcpy(char *s1, char *s2);	kopírování řetězce	zkopíruje obsah s2 do s1, vrací pointer na s1
char *strcat(char *s1, char *s2);	spojení řetězců	připojí s2 k s1, vrací pointer na s1
char *strchr(char *s, char c);	nalezení znaku v řetězci	vrací pointer na znak c, pokud se v řetězci vyskytuje, jinak vrací NULL
int strcmp(char *s1, char *s2);	porovnání dvou řetězců	vrátí 0, jsou-li shodné, záporné číslo, je-li s1 lexikograficky menší než s2 a kladné číslo, je-li s1 větší
char *strstr(char *s1, char *s2);	nalezení podřetězce v řetězci	vrátí pointer na první výskyt s2 v s1 nebo NULL v případě neúspěchu

Práce s omezenou částí řetězce

Práce je podobná jako u funkcí předchozích., V názvu funkce se však vyskytuje písmeno **n** a předává se navíc maximální počet zpracovávaných znaků

př. char *strncpy(char *s1, char *s2, int max);
bude kopírovat po **max-tý** znak s2

Práce s řetězcem pozpátku

V názvu funkce písmeno **r** (př. **strrcpy**)

5.2.23 Práce se soubory, standardní soubory vstupu a výstupu a err

Častým případem je potřeba načítání a ukládání dat. Je možné použít funkcí pro práci se soubory přímo pomocí handlů, ale v C je častější přístup pomocí struktury FILE. S touto

strukturou už se pracovalo při vstupu a výstupu na standardní zařízení – stdin, stdout a stderr jsou tohoto typu a jsou implicitně zastoupeny v příslušných funkcích. Obecně se mluví o proudu, kterým může být paměť, soubor či jiné vstupně / výstupní zařízení. Nezávisle na typu zařízení se pracuje s proudem a je tedy možný společný přístup.

S vlastním proudem se pracuje prostřednictvím struktury FILE. Programátor však pouze předává adresu, na které jsou data o otevřeném proudu, zbytek provádí knihovní funkce. Standardní postup je pokusit se proud otevřít ve zvoleném modu. Poté se otestuje, zda došlo k otevření. Pokud ano můžeme se s proudem pracovat. Po ukončení práce je nutné proud uzavřít – není děláno automaticky.

Pro otevření souboru slouží funkce fopen, která má jako parametry název otevíraného souboru a mod otevření – oba uvedené jako řetězce. Vrací adresu struktury obsahující data o otevřeném souboru. V případě, že se otevření neprovede, je vrácena hodnota NULL, která tedy slouží i k detekci chybného otevření.

Mod otevření je dán řetězcem obsahujícím znaky pro daný mod např. "rb"

- r – čtení
- w – zápis
- a – přidání na konec souboru
- t – textový mód práce se souborem (implicitní nastavení)
- b – binární přístup k souboru

Pro práci s proudem je možné použít jak znakový tak formátovaný přístup.

práce se souborem

FILE *fr, *fw, *fr2; int c,d;	definice proměnných pro práci se souborem pro práci se opět používá int
if ((fr = fopen ("TEST.TXT", "r")) == NULL) { printf("Soubor se nepodarilo otevrit\n"); return ; }	pokus o otevření souboru pro čtení a test zda se otevření provedlo
if ((fw = fopen ("TEST.TXT", "wt")) == NULL) { printf("Soubor se nepodarilo otevrit\n"); }	pokus o otevření souboru pro zápis a test zda se otevření provedlo
fclose(fr); return; }	dříve otevřený soubor musíme uzavřít
fr2 = fr;	vytvoření kopie přístupu k souboru pro čtení. Jelikož se jedná o ukazatele, je práce s nimi ekvivalentní. Práce s jedním se promítne i do druhého. Tj. při načtení znaku z fr, se na další znak posune fr i fr2. Pro práci s jedním souborem a dvěma ukazateli je nutno přistupovat na dané pozice, nebo soubor dvakrát otevřít.

c = getc(fr);	načtení znaku z daného souboru
putc(c, fw);	uložení znaku do daného souboru
fscanf(fr, "formát", argumenty) ;	formátované čtení a zápis
fprintf(fw, "formát", argumenty) ;	
fscanf(fr, "%lf %lf ", &x, &y);	
printf("%f\n", x+y);	
while((c = getc(fr)) != EOF);	hledání konce souboru (pouze u textových souborů)
d = tell(fr);	zjištění vzdálenosti aktuální pozice od začátku souboru
fseek(fr,10l, SEEK_SET);	funkce pro nastavení polohy v souboru. Polohu lze nastavit relativně k počátku (SEEK_SET), aktuální pozici (SEEK_CUR), od konce (SEEK_END). Zde na desátý prvek od začátku. Offset je typu long.
c = getc(fr); while(feof(fr) == 0) { putc(c, fw); c = getc(fr); }	univerzální nalezení konce souboru pomocí funkce feof. Tato funkce změní svou odezvu až v okamžiku načtení prvního znaku za koncem souboru
ungetc(c, stdin);	vrácení načteného znaku do bufferu (zde použít standardní, lze i do souboru - fr). Nemusí být podporován použitým HW.
if (fclose (fr) == EOF) printf("Soubor se nepodarilo uzavřít\n"); fclose(fw);	nutno ukončit práci se soubory

Pozn: Standardní vstup a výstup

Ve stdio.h jsou definovány konstantní pointery představující standardní vstupní, výstupní a chybový proud

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

pak je ekvivalentní

getc(stdin) ~ getchar()

putc(c, stdout) ~ putchar(c)

U některých prostředí a překladačů mohou být i jiné proudy (např. pro vstupní/výstupní zařízení jako seriový či paralelní port ...)

Pozn.: Řádkově orientovaná práce s řetězcí

```
char *gets(char *str);
```

oproti scanf přečte celý řádek do str, včetně bílých znaků až do znaku '\n', který už nečte.

Řetězec ukončí znakem '\0'

```
char *fgets(char *str, int max, FILE *fr);
```

podobně pro čtení ze souboru fr až do konce řádky, nejvýše však max znaků

```
int puts(char *str);
```

vytiskne řetězec a sama odřádkuje

```
int fputs(char *s, FILE *fw);
```

zapiše řetězec do souboru, neodřádkuje ani jej neukončí nulou, vrací nezáporné číslo, když neuspěje, vrací EOF

Pozn: Binární soubory

- v binárním souboru je přesně to, co do něj zapišeme; do textového souboru jsou na konce řádků automaticky přidávány znaky '\r' (MS DOS Windows)
- pro čtení a zápis se používají funkce fread() a fwrite(), pracující s bloky paměti, lze přistupovat i znakově
- pro pohyb v binárním souboru slouží funkce fseek() a ftell()

5.2.24 Pole – vícerozměrné, typedef

V minulých kapitolách jsme se již seznámili s jednorozměrnými poli. V jazyce C je možné realizovat i pole vícerozměrná. Základním principem práce s vícerozměrným polem je postupná indexace. To znamená, že začínáme s ukazatelem, jehož indexací se dostáváme na pozici, na které leží opět ukazatel, který lze samozřejmě opět indexovat. To můžeme opakovat několikrát, až do posledního kroku, kdy se dostaneme k výsledné hodnotě uložené v tomto poli.

<pre>int ** data; data = (int**)malloc(100*sizeof(int *)); if (data == NULL) return; for (i=0;i<100;i++) { data[i]= (int*)malloc(200*sizeof(int)); if (data[i] == NULL) { /* odalokuj dosud naalokované;*/ return;} for (j=0;j<200;j++) data[i][j] = 0; }</pre>	<p>definice dvojrozměrného (dynamického) pole</p> <p>Říkáme, že proměnná data je ukazatel, který ukazuje do místa, kde je ukazatel na int.</p> <p>tedy pole data naalokujeme</p> <p>data tedy ukazuje na pole 100 ukazatelů na int</p> <p>pokud se nepodařilo naalokovat</p> <p>data tedy ukazují na pole ukazatelů na int. Každý z těchto ukazatelů ukazuje na pole intů</p> <p>první indexací proměnné data se tedy dostáváme k ukazateli na int. S tímto výsledkem opět provedeme indexaci (ukazatele na int) a přistoupíme tím k proměnné typu int. To je důvod, proč každý další index musí být v C samostatně - [][]... a nelze použít pouze</p>
---	--

	jedny závorky s více parametry
--	--------------------------------

Pole je možné definovat několika způsoby. Podle toho jak zkombinujeme statickou a dynamickou (ukazatelovou) složku.

a) statické pole `int xa[2][3];`

je definováno při překladu. xa je konstantní ukazatel a nedá se měnit. Lze vytvořit pouze pravoúhlé pole, které nelze za běhu programu měnit. Vznikne jako celek za pomoci překladače.

b) pole ukazatelů `int *xb[2];`

naalokujeme pole dvou ukazatelů, které ukazují na pole int. Z toho plyne, že pevně je dán počet 2 (první rozměr) a druhý rozměr je libovolný a měnitelný. Druhé parametry je nutno dynamicky naalokovat (postupně). Nemusí fyzicky zaujímat jeden blok v paměti.

```
xb[0] = (int *)malloc(3 * sizeof(int));
xb[1] = (int *)malloc(10 * sizeof(int));
```

c) ukazatel na pole `int (*xc)[3];`
`xc = (int *)malloc(2 * 3 * sizeof(int));`

první rozměr je libovolný, druhý rozměr určuje délku řádku (krok pro přechod na další řádek). Místo pro celé pole je nutné naalokovat (v celku)

d) ukazatel na ukazatel

```
int **xd;
xd = (int **)malloc(2 * sizeof(int *));
xd[0] = (int *)malloc(3 * sizeof(int));
xd[1] = (int *)malloc(4 * sizeof(int));
```

Oba rozměry je možné modifikovat za běhu programu. Každý řádek / sloupec může být jinak velký. Nemusí v paměti zaujímat celistvý blok.

Počátky řádků lze zjistit např. `x[1]` (ukazatel na počátek prvního řádku), `&x[2][0]` (ukazatel na prvek o souřadnicích 2,0 tj. třetí řádek první prvek).

Inicializace polí všech rozměrů

Statické pole je možné také definovat s inicializací.

<code>double f[3] = {1.5, 3.0, 7.6};</code>	pro standardní inicializaci se používá seznam prvků oddělený čárkami uzavřený ve složených závorkách
<code>double f[] = {1.5, 3.0, 7.6};</code>	není-li uveden (poslední) parametr počtu prvků, potom si ho překladač zjistí ze skutečného zadaného počtu
<code>double f[3]= {1.5, 3.0};</code>	prvky, které chybí jsou do konce pole doplněny hodnotou nula
<code>typedef int (*P_FCE)(char *, int);</code> <code>P_FCE funkce[] = {file, edit,</code>	obdobně třeba u ukazatele na funkce

search};	
char s1[10] = "ahoj"; char s1[] = "nazdar";	u řetězců je možné zadat přímo řetězec (nemusí být jednotlivé prvky), který překladač automaticky ukončí '\0'
double f[][3] = { {1.5, 3.0,3.4}, {7.6, 9.2,7.2} };	vícerozměrné pole. Rozměry musí být uvedeny až na ten poslední (přidaný – tj. první (levý) index. Tento index se vyčísluje jako první, u ostatních musíme znát krok), který se zjistí z počtu parametrů (zde řádky).
char *p_text[4];	pole (4) řetězců
p_text[0] = "prvni"; p_text[1] = "druhy";	v paměti (statických dat) je umístěn příslušný řetězec a ukazatel na něj je uložen do pole
p_text[2] = (char *)malloc(6);	zde se paměť získá dynamicky
char *p_pole[] = {"prvni", "druhy", "treti","ctvrty"};	definice s inicializací (ukazatele na řetězce ve statických datech jsou přiřazeny ukazatelům v poli)
strcpy(p_text[2], "treti"); p_text[3] = "ctvrty";	a data (ležící opět ve statických datech) se zkopírují
puts(p_text[1]);	přístup k řetězci (resp. jeho počátku)
p_text[2][3] = 'A';	nastavení čtvrtého znaku třetího řetězce

Uvedené inicializace přiřazením jsou možné pouze při definici a ne později. Kromě toho můžeme pole naplnit v příslušném cyklu např. daty ze souboru. Tento způsob je nutný u dynamických polí.

Při deklaraci externích polí se přidává klíčové slovo extern. I když, jak bylo uvedeno výše, existuje více zápisů s ekvivalentním přístupem k poli, je nutné aby deklarace byla stejná jako definice. (Souvisí to s umístěním lokálních a statických či dynamických proměnných a rozdílností práce s nimi u některých překladačů).

k float y[2][3]; uvádět extern float y[2][3] a ne float **y;

Práci s vícerozměrnými poli může zpříjemnit operátor typedef, který vytvoří nové jméno pro daný typ (je to spíše alias - přezdívka). Úvodní příklad by potom vypadal takto

typedef int * P_RADKY; P_RADKY *data;	řádkem nazveme ukazatel na int, tedy pole intů data jsou potom ukazatelem na řádky, tedy na pole řádků
--	---

<pre> data=(P_RADKY*)malloc(100*sizeof(P_RA DKY)); if (data == NULL) return; for (i=0;i<100;i++) { data[i]= (P_RADKY)malloc(200*sizeof(int)); if (data[i] == NULL) { /* odalokuj dosud naalokované;*/ return;} for (j=0;j<200;j++) data[i][j] = 0; } </pre>	<p>data naalokujeme jako jednorozměrné pole řádků kontrola zda se povedlo</p> <p>ukazatele řádků naplníme pomocí alokace intů (řádek je pole intů podle definice),</p> <p>opět test na úspěšnost</p> <p>přístup k prvkům je shodný. První index vybírá příslušný řádek a druhý index prvek v daném řádku</p>
---	--

5.2.25 Funkce main – plné volání

U funkce main jsme dosud nechávali část pro parametry prázdnou. Ve skutečnosti i funkce main může přijímat parametry od volajícího procesu (nejčastěji systému). Pokud parametry neuvedeme, znamená to, že je nepoužíváme. Předávanými parametry je systémová řádka, která způsobila spuštění, ve formátu počet slov na řádku a jednotlivá slova.

Návratová hodnota funkce main je podle nové normy povinná a je možné ji použít v systému (v DOS např. proměnná ERRORLEVEL)

<pre> test.exe /b /c all "text s mezerou" int main(int argc, char *argv [], char *envp[]) { </pre>	<p>takto se spouští program. Na řádku je jméno souboru a následují parametry příkazové ("dosovské") řádky</p> <p>"plná" hlavička funkce main. Do proměnné argc se předá počet předávaných řetězců. V poli řetězců argv jsou potom jednotlivé řetězce příkazové řádky. Rozdělení do řetězců je provedeno ještě před spuštěním main. Jako dělicí znak se bere mezera.</p> <p>V poli řetězců envp jsou předány programu proměnné nadefinované v prostředí. Počet řetězců je určen tak, že za posledním existujícím je vložen ukazatel NULL (envp[konec] == NULL)</p> <p>hodnota argc je 5 argv[0] je "test.exe". (k přidání přípony může dojít systémem i při volání bez přípony) argv[1] je "/b" argv[2] je "/c" argv[3] je "all"</p>
--	---

}	<p>argv[4] je "text s mezerou"</p> <p>Při použití wild char (zápisy typu *.*, ?ab.c??) může být předán originální zápis nebo seznam souborů, které odpovídají zápisu, v závislosti na prostředí</p>
---	---

Některé překladače umožňují vložit parametry až "v programu", kdy se po provedení `argc = ccommand(&argv)`; zobrazí dialogové okno pro zadání textu příkazového řádku.

5.2.26 Struktury

Jazyk C kromě základních typů podporuje i typy složené, které mohou obsahovat typy jednoduché i další typy složené. V případě jazyka C mluvíme nejčastěji o struktuře. Se strukturou pracujeme jako s jedním prvkem, ale máme možnost přistupovat k jednotlivým jejím složkám – prvkům.

Pro práci se strukturami je velice výhodný přístup pomocí ukazatelů, zvláště pak předávání do funkcí je při předání ukazatele podstatně rychlejší a méně paměťově náročné než předání hodnotou. Při předání hodnotou je totiž na zásobníku vytvořena kopie prvku. Stejně zákonitost platí pro předání parametru typu struktura jako návratové hodnoty – i zde se vytváří kopie prvku. Přístup k prvkům struktury vyjádřené jako objekt je pomocí tečky. Bohužel při použití ukazatele má tečka větší prioritu než přístup k adrese a proto musíme použít závorky. To je nešikovné a proto je pro přístup k prvkům struktury vyjádřené ukazatelem zvolen nový operátor `->`.

Jazyk C nedovoluje použití typu nedefinované struktury (pro její použití potřebuje znát velikost typu). Je možné ovšem použít ukazatel na strukturu, který má rozměr obecný a je možné ho získat již při pouhé deklaraci struktury, která spočívá v tom, že nemá "tělo".

Struktura se stává dalším typem a je možné ji používat jako jiné typy. Z operátorů je pro ni definován operátor přiřazení `=`. Při přiřazení se vytváří přesná kopie. To je výhodné, pokud je prvkem statické pole, které samostatně nelze zkopírovat, jako součást struktury při jejich přiřazení však dojde ke kopii obsahu statického pole. Problémy jsou ovšem s proměnnými dynamickými – ukazateli. Každý ukazatel ukazuje do nějakého místa v paměti, které se většinou musí odalokovat. V případě, že provedeme přiřazení struktur, míří do stejného místa několik ukazatelů a z toho plynou chyby vícenásobného odalokování nebo v případě odalokování prostoru jedním prvkem je paměť stále využívána prvkem jiným. Proto pro přiřazování struktur s ukazateli volíme raději přiřazení/konstrukci přes volání vhodné funkce než pomocí `=`.

Definici struktury (bez definic konkrétních proměnných – objektů) je výhodné umístit do hlavičkových souborů, protože chceme-li ji použít ve více souborech, musíme jim dodat plnou definici. Definice struktury netvoří kód ani objekt daného typu.

<pre>struct miry { int vyska; float vaha; } pavel, honza;</pre>	<p>definice struktury začíná klíčovým slovem <code>struct</code> a názvem struktury. Při definici struktury nedochází k vyhrazení místa v paměti</p> <p>následuje seznam parametrů již definovaných typů</p> <p>definice proměnných dané struktury (až zde dojde k vyhrazení</p>
---	--

struct miry karel;	místa v paměti) definice dalších proměnných dané struktury
typedef struct miry { int vyska; float vaha; } MIRY;	definice struktury jako nového typu jméno nového typu, které budeme dále používat pro proměnné dané struktury
MIRY fff (MIRY a, MIRY *b) { MIRY c;	funkce přijímá dva parametry – první se vytvoří jako lokální kopie ve funkci (stojí to místo a čas pro kopírování) lokální proměnná
c.vyska=(a.vyska+b->vyska)/2; c.vaha (a.vaha+b->vaha)/2; return c; }	vytvoření "průměrné" míry. přístup k prvkům objektu je přes "tečku", k prvkům ukazatele je přístup přes " -> " při vrácení hodnoty se provede kopie c do návratové proměnné
{ MIRY karel, pavel, jan; MIRY lide[20]; struct miry *u_osoba;	definice proměnných dané struktury lze vytvořit i pole struktur definice ukazatele na strukturu. struct miry je ekvivalentní jako MIRY.
u_osoba = (MIRY*) malloc (sizeof(MIRY)); pavel.vyska = 167; u_osoba -> vyska = 189; (*u_osoba).vaha = 100;	u struktur je nutné zásadně používat pro zjištění zabrané paměti operátor sizeof, protože díky vnitřní interpretaci či zarovnání proměnných v paměti, může struktura zabírat více místa než je prostý součet jejích prvků přístup k prvkům objektu struktury pomocí tečky přístup k prvkům ukazatele na strukturu pomocí -> přístup k prvkům ukazatele pomocí tečky – závorky jsou nutné kvůli prioritě
lide[3].vyska = 88; lide[4] = fff(karel, &pavel);	přístup k prvkům struktury v poli první proměnná se předává hodnotou, druhá ukazatelem. Výsledná hodnota funkce se zapíše do prvku určenému pro návratovou hodnotu a odtud je operátorem = přepsána do prvku v poli struktur

free(u_osoba); }	
struct Data { float h; ... }; struct polozka { struct polozka *p_polozka; int hodnota; struct Data dd; }; struct polozka pp, &u_pp; ... pp.dd.h = 8; u_pp->dd.h = 5; pp.p_polozka->dd.h *= 22;	je-li nutné aby objekt obsahoval objekt stejného typu (např. vázané seznamy), je nutné aby to byl ukazatel tento řádek je deklarací struktury struktura je již deklarována, lze použít jako ukazatel na daný typ prvkem struktury může být i jiná struktura přístup k prvku vhnížděné struktury přístup k prvku vhnížděné struktury je-li znám ukazatel na strukturu přístup k prvku vhnížděné struktury následujícího prvku. Při dlouhých názvech se hodí rozšířená přiřazení

5.2.27 Union

Podobný struktuře je typ union. Rozdílem ovšem je, že jeho prvky se v paměti překrývají a tedy může v každém okamžiku existovat pouze jeden vnitřní prvek, položka, unionu. K ostatním položkám je možné libovolně přistupovat, ale smysl má pouze přístup k prvku naposledy uloženému. Zároveň neexistuje standardní (v jazyce C) mechanismus, jak zjistit, která položka je zrovna uložena. Tuto informaci je nutné mít mimo union a je v režii programátora. Union má v paměti rozměr největšího z vnitřních prvků, opět je ale bezpečnější stanovit jeho velikost pomocí sizeof (t_union).

Používá se zřídka, nejčastěji pro šetření paměti nebo je-li nutno do funkce předávat různé parametry na jednom místě definice funkce (např. funkce může mít mnoho typů parametrů, ale nikdy ne vždy více jak 2 současně. Proto volíme funkci se třemi parametry, první určuje aktuální typy parametrů na dalších místech a tyto parametry jsou uloženy v unionu.) Další možné použití je při přechodech mezi starými a novými verzemi – jedna proměnná pak může podle verze obsahovat různý typ.

typedef union { char c; int i; float f; } POLOZKA;	definice a deklarace unionu je stejná jako u struktury
--	--

<pre>int pom; POLOZKA a, *p_a = &a; a.c = '#'; p_a->i = 1; a.f = 2.3; pom = a.i ;</pre>	<p>přístup k prvkům je shodný se strukturou</p> <p>do unionu uložíme char vložení 1 přemažeme dříve vložený char vložení 2.3 přemažeme dříve vložený int čtení int nemá smysl protože posledně uložený je prvek float</p>
---	---

5.2.28 Výčtový typ (enum)

Umožňuje zpřehlednit program definicí souvisejících symbolických konstant, které jsou pro programátora textové, vnitřní reprezentace je celočíselná, kdy je každé symbolické konstantě přiřazena unikátní hodnota (0, 1, 2 ...). Tuto hodnotu lze i explicitně určit (ovšem pouze v definici, dále by se mělo pracovat pouze se symbolickými konstantami).

K definici je výhodné použít typedef.

<pre>typedef enum { MODRA, CERVENA, ZLUTA } BARVY;</pre>	definice výčtového typu barev
<pre>typedef enum { FALSE, TRUE } BOOLEAN;</pre>	definice výčtového typu pro logické hodnoty
BARVY barva;	definice proměnné výčtového typu
int pom;	
barva = ZLUTA;	přiřazovat pouze symbolické konstanty
pom = CERVENA;	lze přiřadit do int (implicitní konverze), ale ne naopak
<pre>switch (barva) { case MODRA : printf("Byla to modrá"); break; }</pre>	barva je reprezentována celočíselným typem a proto je možné se symbolickými konstantami pracovat, ale tisknou se (jako řetězec) přímo nedají

5.2.29 Bitová pole

Bitové pole je pole bitů v paměti, nejčastěji využívané k práci s příznaky nebo stavy dvouhodnotové logiky. Pro bitové pole můžeme použít zápis přímo pomocí bitového pole implementovaného v jazyku nebo si vytvořit vlastní bitové pole z (pole) prvků celočíselného typu.

Bitové pole v jazyce C

Je definováno jako struktura, s tím rozdílem, že každá definovaná položka má určen počet bitů. Používá se při potřebě přistupovat k jednotlivým bitům pomocí identifikátoru nebo z důvodu úspory paměti. Není možné přistupovat k jednotlivým proměnným pomocí ukazatele ani získat jejich adresy. Položky jsou ukládány od LSB k MSB. Využíváno např. v návaznosti na HW, na "konverzi" na vnitřní bitovou reprezentaci (strukturu) registrů HW zařízení. Zde např. datum a čas:

<pre>typedef struct { unsigned den : 5; unsigned mesic : 4; unsigned rok : 7; } DATUM; DATUM dnes, zitra; dnes.den = 6; dnes.mesic = 1; dnes.rok = 1995 - 1980; zitra.den = dnes.den + 1;</pre>	<p>pro 0-31 hodnotu dne stačí 5 bitů - 0-tý až 4-tý bit 5-tý až 8-mý bit 9-tý až 15-tý bit</p> <p>definice proměnných typu datum pracuje se stejně jako s jiným celočíselným typem</p>
--	---

Druhým případem je realizace polem celočíselného typu. Potřebujeme-li např. 1000 bitů, potom zvolíme jako základní typ typ long a pomocí počtu bitů v bajtu a bajtů v typu long zjistíme jak veliké pole potřebujeme. Při práci s bitem určíme ve kterém longu leží a zbývající hodnota určí bit uvnitř tohoto longu. K nastavení bitu používáme matematické or s jedničkou, k nulování matematické and s bitovým doplňkem jedničky, ke změně exkluzive or (XOR) s jedničkou v místě daného bitu.

5.2.30 Funkce s proměnným počtem parametrů, "...” (výpustka)

Používá se v situacích, kdy chceme napsat funkci, o které předem nevíme, kolik bude používat parametrů. U prototypu funkce se sekce parametrů skládá z první části, kde jsou uvedeny "přesné" parametry (tj. definované typem a názvem), a z druhé části, kde je možné uvést parametry libovolně (ty jsou v deklaraci zastoupeny výpustkou). Parametry v první části musíme při volání funkce uvádět povinně, alespoň jeden z nich většinou nese informaci o parametrech volitelných. V druhé části mohou být libovolné parametry. Příkladem těchto funkcí jsou knihovní funkce xprintf, xscanf.

Při práci musíme dávat pozor na implicitní konverze (předávaný char se konvertuje na (tj. je umístěn na zásobník jako) int (a je tedy nutno přijímat int), float se konvertuje na double ...)

<pre>void Print(int pocet, ...)</pre>	funkce s proměnným počtem parametrů. Očekáváme dvojice parametrů int a float ale počet dvojic může být libovolný proměnná pocet je poslední známá proměnná, přístupná
---	--

<pre> { float f; va_list val; int i,j; va_start(val, pocet); for(i = 0; i<pocet; ++i) { j = va_arg(val, int); f = va_arg(val, double); } va_end(val); } </pre>	<p>přímo, k ostatním je nutné přistupovat pomocí maker (tyto ostatní proměnné jsou uloženy na zásobník a makra je zpřístupní – umožní načtení do proměnných)</p> <p>proměnná pro přístup k parametrům</p> <p>pro dosažení dat se musíme nastavit za poslední "legální" proměnnou</p> <p>proměnné se načítají pomocí makra, kterému se předá proměnná pro přístup k parametrům a očekávaný typ. Při nesouhlasu typů dojde k rozsynchronizování dat na zásobníku a tedy k nepředvídatelným výsledkům</p> <p>při načítání typu float musíme napsat očekávaný typ double protože na ten je při volání float konvertován. Výsledek je přetypován na správný typ</p> <p>přístup k proměnným je ukončen makrem va_end</p>
---	--

5.2.31 Čtení komplikovaných definic

1. Najdeme identifikátor a od něho čteme doprava
2. Narazíme-li na pravou samostatnou závorku **)**, vracíme se na odpovídající levou závorku **(**, od níž čteme opět doprava vyjma již přečteného
3. Narazíme-li na ukončující středník, vracíme se na nejlevější dosud zpracované místo a od něj čteme doleva

*	<i>čteme</i>	pointer na
()	<i>čteme</i>	funkce(i) vracející
[]	<i>čteme</i>	pole prvků typu

př. double *z(); ***z** je funkce vracející pointer na double*

př. int *(*v)(); ***v** je pointer na funkci vracející pointer na int*

př. double (* f())[]; ***f** je funkce vracející pointer na pole prvků typu double*

př. double (* f [])(); ***f** je pole pointerů na funkce vracející typ double*

5.2.32 Příkaz goto

Je příkazem skoku, který se používá v rámci jedné funkce. Není však povoleno přeskočit definice proměnných. Používat tento příkaz by se mělo jen v nejnútnejších případech. Nejčastější využití je skok z vícenásobně vnořených cyklů ven.

Skáče se na návěští, za kterým musí následovat kód (jehož adresa je potom adresou skoku).

<pre> { if (a==3) goto navesti1; { int aaa = 19; navesti1: } for ... { ... for ... { ... for {... if (...) goto navesti2; }}} navesti2: UzavriSoubory(...) } </pre>	<p>správné použití příkazu skoku, překladač zahlásí chybu, protože se přeskočí definice proměnné aaa</p> <p>návěští je jméno následované dvojtečkou</p> <p>akceptovatelné použití. Opustit více cyklů pomocí proměnných ukončení by bylo programátorsky náročné (a testy by braly i výpočetní čas)</p> <p>před ukončením funkce je nutné uzavřít soubory, vrátit paměť ... (nelze opustit pomocí return (uzavírací sekce by se musela opakovat u všech) Řešením by bylo otevřít soubory ve funkci, která by zavolala funkci, která by pracovala již s otevřenými soubory. Jakýkoli návrat by vedl k uzavření souborů ve funkci, která je otevřela.</p>
--	---

5.2.33 Assembler

V některých případech je nutné či vhodné vložit do kódu jazyka C přímo strojový kód. K tomu slouží sekce předznačená asm a kód je dále uveden jako tělo. Je možné uvést i instrukční sadu která se bude používat. Některé překladače mají pro tento případ jiný překladač, který se znovu spustí v okamžiku, kdy se dojde na tento příkaz. Proto je vhodné na přítomnost příkazu asm upozornit příkazem pragma asm preprocesoru na začátku souboru.

Používání assembleru by mělo být poslední možností a ne standardem. U novějších překladačů s optimalizací je výsledný kód většinou lepší z hlediska místa i rychlosti než při optimalizaci člověkem a pro volání přerušení a dalších low-level funkcí existují knihovní obdoby.

Některé překladače v asm sekci nezvládají skoky na návěští, kterými ještě neprošly (tj. skoky dopředu). Umějí však skákat dopředu na návěští v C.

6 Příloha 2 – Příkladová část

6.1 Úvod

6.1.1 Jak přeložit program

Tvorba projektu – projekt je výhoda, všechna nastavení nese v sobě, propojení pomocí include, souboru v příslušných adresářích, překlad, link.

U překladačů je možné určit, které Warningy se zobrazují – nejlepší je zobrazovat všechny, protože jsou to potenciální skryté chyby – a zda tyto warningy nehlásit jako Error (pak není zbylí a musí se opravit).

6.1.2 Překládání

compile – přeloží zdrojové soubory do objektového kódu

link – spojí objektové soubory a knihovní soubory do spustitelného kódu

make, build – provede compile a link, pouze pro soubory změněné (nebo při změně souborů, které načítají) od minulé kompilace či link (při některých změnách zdrojového kódu (v některých překladačích) může dojít ke změnám v objektovém kódu nových souborů, takže si přestanou se starými ostatními rozumět)

build all – provede se compile a link pro všechny soubory (začíná se od počátku pro všechny soubory – trvá podstatně déle než make, build, ale celý překlad je proveden za stejných podmínek – nastavení přepínačů)

run, execute – spuštění programu (někdy vyvolá i make, build v případě, že je toho potřeba)

Každý warning je skrytá chyba.

Odstraňujte vždy první chybu.

Při odstraňování chyby se podívejte i na předcházející řádky, které mohou být i v includovaných souborech.

6.1.3 Doporučení pro tvorbu příkladů

Před odpovědí si důkladně pročtěte zadání a promyslete si řešení.

Zkuste přeložit v C a C++, uvědomte si rozdíly.

Trasujte po jednotlivých krocích a sledujte tok programu a hodnoty proměnných.

6.1.4 Assert

Při ladění programů je možné využít makro `assert` z knihovny `assert.h`, které slouží k odladění situací, které "nemohou nastat". V případě, že známe situaci, kdy například číslo musí být kladné (například výraz $z = x * x$ na první pohled tuto podmínku splňuje, ale díky přetečení může nastat situace, kdy je jeho výsledek záporný, což je chyba) a chceme zjistit, zda tomu tak opravdu je, pak použitím makra `assert` s danou podmínkou `assert (z >= 0)` dojde při **nesplnění** podmínky k tisku této podmínky, jména souboru a číslo řádku na kterém nastala. Vypnutí lze zajistit přepínačem, nebo definováním proměnné `NDEBUG`.

6.2 Příklady pro C

6.2.1 Struktura programu v C

Otázky:

- jaká je struktura programování v jazyce C
- co je to modul
- co jsou a jaký název se volí pro zdrojové a hlavičkové soubory
- co dělá překladač
- jak překladač zachází se zdrojovými a hlavičkovými soubory
- co dělá preprocesor
- proč se do hlavičkového souboru nesmí ukládat části tvořící kód
- k čemu slouží hlavičkové soubory, co mohou a co nesmí obsahovat.
- co se píše do zdrojových souborů

6.2.2 Překlad a sestavení programu

Otázky:

- co dělá linker
- k čemu jsou knihovny
- jaké jsou fáze vytvoření spustitelného programu
- co je to trasování a jaké má možnosti
- k čemu je tvorba projektu, co obsahuje, jaké má výhody jeho použití
- co je a k čemu slouží projektový soubor
- co je a k čemu slouží makefile
- co dělá program make
- jak se tvoří projektový soubor

Příklad: Založte si projekt, který bude mít jeden soubor se zdrojovým kódem. Projekt nazvěte `fc`, soubor pro zdrojový kód `mainfc.c`. Příklad zkuste přeložit. K jakým došlo chybám a proč.

6.2.3 Komentáře

Otázky:

- k čemu slouží a co by měly obsahovat komentáře
- jak se zapíše komentář do textu a jaká pro jeho psaní platí pravidla
- co jsou to vnořené komentáře, jak je nahradit
- co se stane, je-li v těle poznámky dvojice znaků ” /* ”

Příklad: Vložte do zdrojových a hlavičkových souborů na úvod komentář, který uvede jméno souboru, datum jeho založení a poslední změny, číslo verze, pro jaký překladač je odladěn, kdo je autorem, později stručnou charakteristiku k čemu slouží funkce a proměnné v tomto souboru, popř. jejich přehled a jednořádkový popis ... Zkuste přeložit – neměly by se objevit nové chyby.

6.2.4 Funkce main – základ

Otázky:

- k čemu slouží funkce
- jaké náležitosti musí mít hlavička funkce
- co je to prototyp funkce, předved'te na funkci main
- jak se pozná a k čemu slouží definice a deklarace funkce
- která je první funkce volaná v C
- co je a jaké parametry má funkce main
- k čemu slouží návratová hodnota funkce main a jakého je typu
- k čemu slouží klíčové slovo return
- které klíčové slovo slouží k předání návratové hodnoty funkce
- jakou funkci má v C středník a kdy se používá
- jak se v C realizuje prázdný příkaz, uveďte příklad
- co je ohraničením bloku programu

Příklad: Doplňte do projektu funkci main, která v případě svého úspěšného průběhu vrátí hodnotu 8.

6.2.5 Identifikátory, základní datové typy

Otázky:

- jaká pravidla platí pro identifikátory (jména proměnných a funkcí)
- jak se píší klíčová slova v C
- které základní datové typy znáte
- jaké jsou celočíselné datové typy a jejich varianty
- jaké jsou reálné datové typy
- jakou přesnost mají datové typy v C, co platí pro velikosti jednotlivých typů
- jak zjistíme velikost datového typu v paměti
- k čemu slouží klíčové slovo sizeof a jak se používá
- co znamenají klíčová slova signed a unsigned, jak a k čemu se používají
- jaký je rozdíl mezi definicí, deklarací a definicí s inicializací (uveďte příklady)
- který typ se nejčastěji používá pro práci se znaky a proč
- jak se zapisují znakové konstanty, escape sekvence, escape znaky
- jak se zapisují celočíselné konstanty
- jak se zapisují reálné konstanty
- logické hodnoty
- co je bráno jako false a co jako true

6.2.6 Typová konverze

Otázky:

- co je to typová konverze a k čemu se používá
- implicitní typová konverze – co je to, kdy se používá
- explicitní typová konverze – co je to, kdy se používá, jak se provádí

6.2.7 Operace s proměnnými, operátory

Otázky:

- jaké operátory v C znáte, popište jejich činnost, co o jednotlivých víte
- do jakých skupin lze rozdělit operátory v C
- jak funguje operátor přiřazení
- co je to l-hodnota
- jak se pozná kdy je výsledek dělení celočíselný a kdy reálný
- jak ovlivňuje typ proměnné na levé straně = přesnost výpočtu výrazu na pravé straně
- jak fungují operátory ++, --
- jaký je rozdíl mezi použitím výrazů typu ++j a j++
- jakými (třemi různými) způsoby lze přičíst jedničku v C
- co jsou a k čemu slouží operátory posuvu
- jaký je výsledek $a * = b + c$
- jaké jsou logické operátory, jaký je rozdíl mezi bitovými a logickými verzemi
- jaký je rozdíl v činnosti operátorů ! a ~
- které operátory slouží k porovnávání velikosti proměnných
- jak se vyhodnocují složité matematické a logické výrazy
- co je to priorita a asociativita operátorů
- proč se ve složitých výrazech doporučuje používat závorky

Příklady:

- vyzkoušejte si přiřazení char do float a float do char pro různé hodnoty – jak probíhá konverze
- vyzkoušejte vícenásobná přiřazení včetně použití rozšířeného přiřazení
- vyzkoušejte dělení int / int, float / float, int / float, float / int s přiřazením do int a float. Jaké výsledky dostáváte a proč. Popište mechanismus výpočtu. Zkuste (pomocí explicitní konverze) opravit dělení tak aby výsledek přiřazený float byl vždy správný (neceločíselný).
- použijte operátor posuvu k násobení 8 a dělení 16
- zjistěte (načtete do proměnné) ASCII hodnotu pro 'P', tuto proměnnou srovnajte s proměnnou jejíž hodnota je 59. Výsledek srovnání zapište do další proměnné.
- převed'te int s hodnotou 0-9 do proměnné char aby obsahovala znak, který je reprezentuje

6.2.8 Funkce

Otázky:

- jak vypadá funkční prototyp funkce
- jaký je rozdíl mezi definicí a deklarací funkce
- k čemu slouží typ void
- který typ použijeme v případě, že nepotřebujeme předávat parametry
- k čemu slouží klíčové slovo return.

- jaký je mechanismus předání parametrů do a z funkce
- co je to rekurzivní funkce a lze ji v C realizovat

Příklady:

- napište funkci, která provede zaokrouhlení na celá čísla
- napište funkci, která provede zaokrouhlení na zvolený řád
- vytvořte funkci, která nastaví, vynuluje, změní hodnotu, zjistí hodnotu daného bitu v proměnné. Uvažujte použití pro různé celočíselné typy

6.2.9 Příkazy preprocesoru, makra

Otázky:

- co je a k čemu slouží preprocesor
- které hlavní příkazy preprocesoru znáte
- jak poznáte příkazy preprocesoru
- jaký je vztah preprocesoru a překladače
- co dělá preprocesor se soubory (zdrojovými, hlavičkovými)
- k čemu slouží direktiva define
- co jsou a k čemu se používají makra bez parametrů
- jaký je mechanismus makra bez parametrů
- nadefinujte si pro výpočet eulerovo číslo EULER ...
- kolik řádků může mít makro ve zdrojovém kódu, v případě že více než jeden jak se dostat na další řádek
- co jsou a k čemu slouží makra s parametry
- jaký mechanismus se využívá pro interpretaci maker s parametry, k jakým chybám může vést a jak tyto chyby odstranit
- co je to řízený překlad, uveďte příklad a vysvětlete použité direktivy
- k čemu slouží #ifdef, #ifndef, #if, #else, #elif, #undef, defined. Uveďte definice a příklady s popisem
- co je a k čemu slouží direktiva include
- jak zabránit vícenásobnému načtení vkládaných souborů

Příklady:

- napište makro pro nulování, nastavení, změnu a zjištění stavu bitu v celočíselné proměnné. Uvažujte možnost použití pro všechny celočíselné typy. Jaký je rozdíl mezi funkcí a makrem?
- využijte makro k podmíněnému překladu části kódu – např. při ladění vrací main 0 jinak 1

6.2.10 Platnost identifikátorů, globální a lokální proměnné

Otázky:

- co platí pro platnost identifikátoru
- liší se z hlediska platnosti identifikátoru definice od deklarace
- co znamená klíčové slovo static při použití s funkcí, globální a lokální proměnnou
- k čemu slouží klíčové slovo extern
- co způsobí definice v hlavičkovém souboru
- proč dáváme deklarace do hlavičkového souboru
- co (ne)umístujeme do hlavičkového souboru
- kde můžeme definovat funkce a proměnné
- co jsou paměťové třídy

- co značí paměťová třída auto, register, extern – co platí pro tyto proměnné
- co jsou typové modifikátory a k čemu slouží
- k čemu slouží typový modifikátor const
- jaký je rozdíl mezi použitím const a define pro konstantní proměnné
- k čemu slouží modifikátor volatile

Příklad: Vytvořte modul s funkcí main a modul kvadraty.c. modul funkce.c bude obsahovat funkci, která vypočte obvod kruhu. Vypočtenou hodnotu vrátí dvěma způsoby, jako návratovou hodnotu funkce a přes globální proměnnou. Pro PI použijte předdefinovanou konstantu. Srovnajte tyto vrácené hodnoty. Oznámení do hlavního modulu proveďte pomocí hlavičkové funkce. Co se stane, nebudou-li typy proměnných použitých při volání stejné jako typy v prototypu?

6.2.11 Standardní znakový (terminálový) výstup, vstup

Otázky:

- ve kterém hlavičkovém souboru jsou prototypy funkcí pro standardní znakový vstup a výstup
- která klíčová slova slouží v C pro vstup a výstup znaku
- jak je v C realizován vstup a výstup znaku
- co je to stdin, stdout, stderr
- k čemu slouží getch, getchar, putchar.
- jaký typ se využívá pro práci se standardními znakovými vstupy a výstupy
- jak se provede odřádkování (co je nutné poslat na výstup, nebo jakou funkci použít)

Příklad: Napište funkci, která načte znak z klávesnice a vytiskne ho jako znak a dále vytiskne jeho dekadickou hodnotu. Zavolejte funkci 5x a pozorujte změny použijete-li pro načítání getch, getche, getchar.

6.2.12 If – else, ternární operátor

Otázky:

- k čemu slouží ternární operátor, popište jeho činnost
- jaký je rozdíl mezi ternárním operátorem a if – else
- k čemu slouží příkaz if – else, popište možné varianty použití
- které příkazy se používají pro větvení programu

Příklady:

- napište makro se třemi parametry, které vrátí prostřední prvek, jsou-li prvky seřazeny podle velikosti. Zkuste pomocí if-else a pomocí ternárního operátoru. Zkuste např. volání if (stredni(2,3,1) > 1) i++; nebo volání if (a > b) c = stredni (a,b,0); else c = 4;
- napište obdoby knihovních funkcí isdigit (vrátí 1 pokud je načtený znak číslice), isalpha (vrátí 1 je-li načtený znak písmeno), isupper (vrátí 1 pokud je načtený znak velké písmeno). Načtěte znak z klávesnice a zavolejte pro něj všechny funkce. Výsledky vytiskněte.
- načtěte tři znaky, pokud je třetím znakem a (malé nebo velké), vytiskněte menší (dříve v abecedě) ze znaků, jinak větší (bez ohledu na velikost).

6.2.13 Cykly, opuštění cyklu

Otázky:

- které příkazy cyklu znáte
- co musí platit pro podmínku u jednotlivých cyklů, aby došlo k opakování těla cyklu
- jak lze opustit tělo cyklu
- k čemu slouží příkaz continue, co dělá
- popište cyklus while, do-while, for včetně příkazů break a continue
- jak realizovat nekonečnou smyčku pomocí různých cyklů
- uveďte příklady cyklů s prázdným tělem
- k čemu slouží operátor čárka

Příklady:

- napište funkci, která načte 20 znaků. Po načtení vytiskne malé písmeno 3x, velké 5x. Malé a velké a se netiskne. Při stisku mezery se odřádkuje. V případě, že je znakem velké nebo malé 'z', potom se cyklus ukončí. Po skončení cyklu se odřádkuje. Realizujte všemi třemi příkazy cyklu.
- vytiskněte na obrazovku 10 řádků po deseti znacích (např. 0-9, a-j) tak, aby každý další řádek byl o jeden znak posunut
- vytvořte funkci, která čeká na stisk klávesy, odpovídající znaku, který dostane funkce jako parametr
- zkuste přepis for pro makro s while, tak aby fungovaly i příkazy continue, break. Jedná se o procvičení mozku ne o standardní přístup k cyklu for. Jedno z možných řešení je načrtnuto – zkuste dokončit.

define FOR (init, podmínka, iterace, telo)

```
init ;
stav = PRVNÍ
while ( 1 )
{
    if (stav != PRVNÍ) iterace ;
    else stav = DALŠÍ
    if (!(podmínka+0)) break; // řeší (?) i situaci, je-li podmínka prázdná
// řešení např pomocí fortest(void) a fortest(int) – fortest(podmínka) volá jednu z nich
// na základě textu podmínky a dává výsledek. Nelze pro C. Jak pro C?
    telo ;
}
```

6.2.14 Switch

Otázky:

- k čemu slouží příkaz switch, popište ho
- co musí platit pro podmínku v cyklu switch
- jak je svázána podmínka a příkazy case
- kdy se ukončí větev v příkazu switch
- k čemu slouží příkaz default v příkazu switch

Příklady

- Pomocí kláves šipek se pohybujte vodorovně a svisle v rastru šachovnice s rozměry a-h, 1-8 a vypisujte aktuální pozici. klávesy qwas použijte pro pohyb diagonální. Nedovolte pohyb mimo šachovnici. K vyhodnocení klávesy na pohyb použijte příkaz switch. Při stisku klávesy k přepínejte (střídavě zapínejte a vypínejte) pohyb o jeden krok a pohyb k okraji (tj při stisku se vezme daný směr a dojde se daným směrem až k okraji šachovnice. Velká a malá písmena jsou rovnocenná.
- Napište pomocí stavového automatu test řetězce na přípustnost celého čísla (dec, oct, hex) a reálného čísla.

6.2.15 Ukazatele, typedef

Otázky:

- co je to ukazatel
- jakou má ukazatel vazbu na typ
- jak se definuje ukazatel
- co znamená zápis `int ** a;`
- k čemu slouží typedef
- jak se chová neinicializovaný ukazatel, jak se s ním pracuje
- co je a k čemu slouží NULL
- jak se inicializuje ukazatel
- jak k přistupovat k hodnotě na niž ukazuje ukazatel
- jaké významy mají operátory `&` a `*`
- co se stane když: `double *a; char c; a = &c; ...`

Příklady:

- nadefinujte si dvě proměnné typu float a vyměňte jejich hodnoty pomocí ukazatelů na ně (a pomocné proměnné)
- proveďte totéž co v předchozím, akorát typ bude `int *` - tj vymění se dvě hodnoty reprezentující adresu, avšak ne přímo ale pomocí adres na kterých leží. Tj. nadefinujte si dvě proměnné typu `int *` a vyměňte jejich hodnoty pomocí ukazatelů na ně (a pomocné proměnné). Pro ladění bude výhodné `int *` inicializovat, aby bylo vidět zda k výměně došlo.
- nadefinujte si dvě proměnné typu double a vytvořte ukazatele, které na ně ukazují. Vyměňte hodnoty ukazatelů mezi sebou (tak aby ukazovaly na druhou proměnnou než původně – tj. při přiřazení pomocí ukazatele, který původně ukazoval na první proměnnou, dojde ke změně druhé a naopak).

6.2.16 Dynamická paměť

Otázky:

- kdy je vhodné použít dynamickou paměť
- jak se dynamická paměť získá
- jak se vrací dynamicky získaná paměť
- jak zjistíme, zda byla paměť přidělena
- v jakých jednotkách se alokuje paměť (uvedte příklad s double)

Příklad: Naalokujte paměť do které bude možné umístit 50 prvků typu double. Zkontrolujte, zda je vše v pořádku. vraťte získanou paměť.

6.2.17 Funkce a ukazatele

Otázky:

- jakým způsobem je možné změnit hodnotu pomocí volané funkce
- jaký typ volání má C (jaký je rozdíl mezi voláním hodnotou a odkazem)
- k čemu se používá ukazatel na typ void
- jaká je vlastnost funkce na ukazatel
- co je to ukazatel na funkci
- k čemu se používá ukazatel na funkci

Příklady:

- vytvořte funkci, která vrátí při jednom volání tři znaky, které načetla.
- vytvořte funkci, která naalokuje paměť do které bude možné umístit 50 prvků typu double. Zkontrolujte, zda je vše v pořádku. Hodnotu pro uložení vraťte hlavnímu programu. Pro vrácení paměti použijte opět funkci.
- program má dvě proměnné double a funkce pro sčítání, odečítání, násobení a dělení. Pomocí stisku příslušného znaménka vyberte funkci, ukazatel na kterou předáte s proměnnými funkci, která vrátí výsledek, který uložte do další proměnné.

6.2.18 Jednorozměrné pole, ukazatelová aritmetika

Otázky:

- jak je v C reprezentováno jednorozměrné pole
- jaká je definice jednorozměrného pole
- jakým způsobem se přistupuje k prvkům pole
- jak se definuje a jaký je rozdíl mezi statickým a dynamickým polem
- kdy dáváme přednost statickým a kdy dynamickým polím
- co reprezentuje název pole
- jaké indexy mají prvky pole dlouhého 5 prvků
- mohou se používat záporné indexy, kdy, uveďte příklad
- co je to ukazatelová aritmetika
- kdy má smysl odečítat ukazatele a co je výsledkem

Příklady:

- vytvořte pole, do kterého zapíšete prvních 10 násobků daného čísla (např. 7)
- vytvořte pole délky 20 prvků, vynulujte ho pomocí for tak, aby se nepoužívaly indexy ale pouze ukazatele (vytvořte ukazatel na / za konec pole a plňte přes pomocný ukazatel postupně od začátku s inkrementací až dokud nedosáhne ke konečnému ukazateli)

6.2.19 Řetězce

Otázky:

- jaká je definice řetězce
- k čemu se používá znak `\0` při práci s řetězcí
- jak se inicializují statické a dynamické řetězce, jak se volí jejich délka

Příklady:

- načtěte řádek z klávesnice do řetězce. Maximální délku načítaného řetězce zvolte 80 znaků. Následně řádek vytiskněte.
- spojte dva řetězce do třetího

- spojte dva řetězce do jednoho – prvního z nich. Jeden statický s inicializací, druhý dynamický načtený z klávesnice. Zkuste naprogramovat tak aby jednou byl první statický a jednou dynamický.

6.2.20 Funkce a pole

Otázky:

- jak se předává pole do funkce
- jaký je rozdíl mezi předáváním do funkce pole a proměnné
- kdy se projeví a kdy se neprojeví změny prvků pole předaného funkci
- jaký je rozdíl mezi předáváním statického a dynamického pole

Příklady:

- vytvořte funkci, která do pole uloží prvních n mocnin o daném základu. Funkci vytvořte tak aby šla volat se statickým i dynamickým polem. Vytvořte druhou funkci, která pracuje stejně ale pole není v seznamu parametrů. Vyzkoušejte všechna možná volání funkcí (statická i dynamická pole) a ošetřete chybové stavy. Odalokujte všechny dynamicky vzniklé proměnné.
- vytvořte pole funkcí pro sčítání, odečítání, násobení a dělení čísel double. Vytvořte funkci, které se předají dva parametry, index funkce a pole funkcí. Jejím výsledkem je aplikování zvolené funkce na předané parametry.

6.2.21 Formátovaný vstup a výstup

Otázky:

- co a k čemu jsou funkce `xprintf`, `xscanf`. Jaké mají parametry.
- jak lze vytisknout proměnná daného typu v jazyce C
- jak lze načíst proměnnou daného typu v jazyce C
- co je to řídicí řetězec, jaké znaky se v něm mohou objevit
- k čemu v řídicím řetězci slouží znak `.`. Jak tento znak vytisknout?
- k čemu v řídicím řetězci slouží znak `%`. Co za ním následuje a co to řídí.
- co se stane dostane-li funkce `printf` více nebo méně parametrů než je uvedeno v řídicím řetězci
- jak realizovat formátový vstup a výstup z řetězce nebo souboru

Příklady:

- vytiskněte tabulku malé násobilky (jako první řádek a první sloupec se tisknou operandy). Snažte se aby byla tabulka ve sloupcích zarovnána.
- vytiskněte tabulku (svisle) ve tvaru `| x | A.sin(x) | A.cos(x) |`. Před tiskem načtěte krok pro proměnnou x a velikost amplitudy. Tabulku vytiskněte tak aby byla zarovnána (pro různé amplitudy).

6.2.22 Funkce pro práce s řetězcí

Otázky

- které funkce pro práce s řetězcí znáte
- co musí platit pro řetězce používané ve funkcích pro práci s řetězcí

Příklady:

- nastudujte si vlastnosti funkcí pro práci s řetězci a napište si vlastní se stejnými parametry a činností bez použití knihovních funkcí (může být jiný název funkce)
- vytvořte dynamické (jednorozměrné) pole, do kterého vložte úvodní konstantní text, ke kterému přidejte deset řetězců načtených ze stdin
- načtěte dva řetězce a hledejte místa výskytu prvního řetězce ve druhém. Uvažujte i vícenásobný výskyt. Použijte vlastní funkce pro vyhledávání podřetězce.

6.2.23 Práce se soubory

Otázky:

- jak se pracuje se strukturou FILE
- jak otevřít soubor a jak ověřit jeho správnost. Jak a kdy soubor uzavřít
- jak načítat a zapisovat do souboru
- které jsou standardně otevřené soubory (proudy)
- co obsahuje řetězec pro mod otevření souboru. Jaké možnosti má. Jak zapsat cestu k otevíranému souboru.
- jak se pohybuje uvnitř souboru

Příklady:

- upravte příklad pro tabulku násobilky a sinu a cosinu tak, aby byl výsledek uložen do souboru
- upravte příklad s vyhledáváním řetězců tak aby parametr načtl i výsledek uložil do souboru
- vytvořte program, který zjistí počet řádků, slov a četnosti znaků ve zvoleném souboru. Pro tisk četnosti znaků napište funkci umožňující vytisknout pouze znaky s četností vyšší než je zadaná.
- vytvořte program pro srovnání dvou souborů (zkuste textové i binární)
- vytvořte program pro kopírování souborů (zkuste textové i binární)

6.2.24 Vícerozměrné pole

Otázky:

- co je to vícerozměrné pole
- jak nadefinovat vícerozměrné pole. Uveďte různé způsoby a jejich vlastnosti.
- jak naalokovat a odalokovat vícerozměrné pole
- předávání vícerozměrných polí do funkcí
- co je prvkem pole `int ***ppp_i`; a jak je možné k tomuto prvku přistoupit
- co je prvkem pole `int ***ppp_i` jedná-li se o dvourozměrné pole
- jak se inicializují jednorozměrná pole
- jak se inicializují řetězce
- jak se inicializují vícerozměrná pole

Příklady:

- vytvořte funkci na vytvoření (alokace) a zrušení (odalokování) dvourozměrného pole. Vytvořte funkci na vytvoření a naplnění matice z daného souboru. Vytvořte funkci pro uložení matice do souboru. Vytvořte funkci pro sečtení a násobení dvou matic (parametry zůstanou po operaci stejné). Uvažujte vhodnost navržených funkcí pro statické a dynamické pole. Předved'te volání vytvořených funkcí.

- vytvořte funkci, která načítá z klávesnice řetězce – věty ukončené tečkou. Načtené věty ukládá do pole řetězců. Po načtení zvoleného počtu řetězců seřaďte načtené podle abecedy (stačí anglické, tj ASCII hodnot znaků).

6.2.25 Funkce main

Otázky:

- jaký je plný prototyp funkce main, co jsou jeho parametry a jak s nimi pracovat
- jaké parametry dostává a vrací funkce main

Příklady:

- vytiskněte všechny proměnné prostředí
- upravte příklady z tak, aby se jména souborů načítala z příkazové řádky. Ošetřete chybná zadání.

6.2.26 Struktury

Otázky:

- co je a co umožňuje struktura
- jak se definují prvky struktury
- jak se přistupuje k prvkům struktury je-li dána objektem
- jak se přistupuje k prvkům struktury je-li dána ukazatelem
- proč se struktura nemá předávat hodnotou do funkcí
- kdy a jaké problémy jsou s dynamickými složkami struktury (mělké a hluboké kopírování)

Příklad: Vytvořte strukturu pro adresu, která má tyto položky – číslo popisné, město, ulice, jméno – to je struktura skládající se ze jména a příjmení. Řetězce ukládejte do struktur dynamicky. Načtěte 10 struktur, seřaďte je abecedně podle jmen a vytiskněte.

6.2.27 Uniony

Otázky:

- co je to union
- jak se přistupuje k položkám unionu
- čím se liší struktura a union

Příklad: Vytvořte union, který bude součástí struktury Auto, která bude mít položku SPZ, typ (nákladní, osobní). Podle typu bude v unionu uloženo číslo float tonáž pro nákladní a int míst k sezení pro osobní. Strukturu použijte.

6.2.28 Výčtový typ

Otázky:

- co je to výčtový typ
- jaká je vazba mezi výčtovým typem a typem int

Příklad: Napište funkci, která vypustí komentáře ze zdrojových textů jazyka C. Funkce má dva vstupy – FILE pro vstupní a výstupní soubor. Funkci implementujte jako konečný

stavový automat pomocí switch, kdy stavy definujete pomocí enum (KOD, POZNAMKA, LOMITKO, HVEZDICKA, NIC, KONEC ...).

6.2.29 Bitová pole

Otázky:

- co je bitové pole
- jak lze realizovat bitové pole v jazyce C
- jak přistupovat k prvkům bitového pole.
- jak měnit hodnoty v bitovém poli

Příklad: Napište funkce (a totéž pomocí maker) pro inicializaci a ukončení bitového pole libovolné délky, pro nastavování bitů v poli (na nulu i jedničku), funkce pro změnu bitu a zjišťování hodnoty daného bitu. Jaký je rozdíl mezi funkcí a makrem. Funkce použijte.

6.2.30 Funkce s proměnným počtem parametrů

Otázky:

- co je to výpustka
- jak je definována hlavička funkce s proměnným počtem parametrů
- které standardní knihovní funkce jsou funkcemi s proměnným počtem parametrů
- jak přistupovat k předávaným parametrům funkcí s proměnným počtem parametrů

Příklad: Vytvořte funkci maximum, která vrátí maximální prvek z přítomných prvků jako double. Počet a typ parametrů může být libovolný a je dán pomocí řídicího řetězce, který je prvním a povinným parametrem funkce, ve kterém je kódován typ (a tedy i počet) předávaných proměnných.

6.2.31 Čtení definic

Otázky:

- jaký je postup při čtení komplikovaných definic
- jak zjednodušit čtení komplikovaných definic

6.2.32 Příkaz goto

Otázky:

- k čemu slouží příkaz goto. Jak funguje
- popište příkaz a mechanismus skoku – goto
- co je a jak se zapíše návěští – jak se použije

Příklad: Máme dvě dynamicky vytvořená dvourozměrná pole (100x100) s prvky 0,1 (do nulových polí náhodně umístíme 100 jedniček). Vyhledejme dva nejbližší prvky (jeden z prvního a druhý z druhého pole) z hlediska vzdáleností součtu indexů. Při nalezení prvků vzdálených méně než 5 ukončete cykly pomocí goto odalokujte a ukončete funkci. Hledejte alternativu pro goto, např. i s přihlédnutím k rozšíření pole o další rozměr (např 100x100x100) či dva. Uvažujte časovou náročnost zvoleného řešení.

6.2.33 Assembler

Otázka: Jak vložit do kódu jazyka C úsek v assembleru

6.2.34 Příklad C 1 – ovládání bitů podle sekvencí vstupního souboru

- zkuste co nejvíce vypracovat bez knihovních funkcí
- nepoužívejte globální proměnné
- proveďte rozbor úlohy
- zkuste vypracovat nejdříve na papír – jako na písemných zkouškách
- vlastní funkce pište do c souboru který je různý od souboru s main, exportujte funkční rozhraní (a deklarace) pomocí h souboru
- zkuste zapnout překlad v ANSI C (ne C++) (v překladači nastavit tento překlad, lze-li. Soubor by měl mít extension c a ne cpp – prostředí podle přípony často střídají ”na pozadí” příslušné překladače)
- navrhnete vlastní vstupní soubor(y), který(é) prověří činnost algoritmu
-
- je dán vstupní textový soubor který je předán jako parametr funkce *main(---)*
- v tomto souboru jsou znaky **S**, **R**, **C**, **X**, **M** následované celým číslem
- **M** je prvním znakem souboru následované hodnotou určující max. počet využívaných bitů. (např. **M55**)
- hodnota uvedena u ostatních znaků (tzn. **S**, **R**, **C**, **X**) značí pozici bitu se kterým se pracuje
- **S** (**S**et) je příkaz pro nastavení bitu na hodnotu 1 (např. **S24**)
- **R** (**R**eset) je příkaz pro nastavení bitu na hodnotu 0 (např. **R2**)
- **C** (**C**hange) je příkaz pro změnu hodnoty bitu (např. **C4**)
- příkaz **X** (**eX**change) je zpracován pouze v páru dvou takovýchto příkazů a slouží pro záměnu hodnot bitů na daných pozicích. Pár příkazů **X** nemusí následovat bezprostředně za sebou. (např. **X12 ... X56**).
- výsledné hodnoty společně s číslem provedeného řádku vytisknete na konzolu.
- zpracovávanou hodnotu držte v poli (vyzkoušejte různé celočíselné typy) minimální možné délky dle parametru **M** (např. pro **M55** by měla mít zpracovávaná hodnota co nejmenší počet bitů (nejčastěji nejbližší vyšší příslušný násobek $8 * velikost_typu$))
- pro vlastní provedení příkazů použijte makra. Proveďte srovnání s realizací pomocí funkcí.
- využijte operátory : ~, ^, &, /, ?:, <<, >>

6.2.35 Příklad C 2 – funkce pro práci se soubory a řetězci

- přečtete si celé zadání
- proveďte rozbor zadání
- zkuste vypracovat nejdříve na papír – jako na písemných zkouškách
- nepoužívejte globální proměnné
- zkuste vytvořit co nejvíce funkcí samostatně (knihovní funkce používat jen výjimečně) – vlastní funkce pište do c souboru který je různý od souboru s main, exportujte funkční rozhraní (a deklarace) pomocí h souboru
- zkuste zapnout překlad v ANSI C (ne C++) (v překladači nastavit tento překlad, lze-li. Soubor by měl mít extension c a ne cpp – prostředí podle přípony často střídají ”na pozadí” příslušné překladače).

- navrhnete vlastní vstupní soubor(y), který(é) prověří činnost algoritmů (např. konec souboru jako konec řádku (či jinak), řádek začíná, končí oddělovačem nebo slovem, prázdný soubor ...)
-
- program dostává jako parametr název vstupního a výstupního souboru
- zkontrolujte, zda jsou předány do main oba názvy souborů
- není-li druhý název přítomen, proved'te výstup na standardní výstupní zařízení
- otevřete soubory
- vytvořte funkci (vlastní), která načte ze souboru řádek, který vrátí jako řetězec – naalokovaný uvnitř funkce na přesnou délku (uvažte způsob předávání a odalokování)
- vrácený řetězec vložte do struktury, která bude kromě ukazatele na řetězec obsahovat délku řetězce
- každý načtený řádek = řetězec vložte do struktury a z nich vytvořte pole. Zařazení do pole proved'te pomocí funkce (ošetřete první vložení).
- napište funkci, která načtené řádky seřadí podle abecedy (anglické). Řazení proved'te pomocí výměny struktur a také pomocí výměny obsahů (tj. dvě řešení).
- napište funkci, která vytvoří kopii pole obsahujících řádky souboru
- napište funkci, která v kopii pole vypustí duplicitní řetězce (jsou-li (dva a) více stejných řádků, zůstane jen jeden)
- za využití pole proved'te ve funkci s tiskem na konzolu statistiku – četnost znaků, počet řádků, počet slov (oddělovačem je vše co není písmeno nebo číslice, může být více oddělovačů za sebou, oddělovačem je i konec řádku, oddělovačem může ale nemusí začínat nebo končit řádek ... Pokuste se řešit pomocí stavového automatu se stavy např. začátek, konec, ve_slove, mimo_slovo, rozdělené_slovo (- před koncem řádky) ...
- napište funkci, která najde výskyt daného řetězce v podřetězci. Vyhledejte touto funkcí výskyty řetězce (i vícenásobné v jednom řetězci) např. "abc" v načteném textu. Pomocí další funkce vložte za tento řetězec řetězec jiný např. "123" aniž by došlo k přepsání původního obsahu (je potřeba nově naalokovat, a tedy i odalokovat). Proved'te v kopii.
- vytiskněte (upravenou) kopii pole do výstupního souboru
- odalokujte všechny naalokované řetězce (proměnné), ověřte, zda jsou odalokovány všechny naalokované
- uzavřete soubory (pokud výstup není na standardní zařízení)
- vyřešte i bez použití struktur
- cykly realizujte pomocí for, while, do – while, uvědomte si rozdíly (alternativní řešení zkuste např. pomocí řízeného, podmíněného, překladu #define ... #ifdef ...)

7 Seznam použité literatury

- [1] Prata S.: Mistrovství v C++, Computer Press, Brno, 2004, ISBN 80-251-0098-7
- [2] www.research.att.com/~bs – webová stránka “otce” C++ Bjarne Stroustrupa
- [3] Horstmann S. Cay,: Vyšší škola objektového návrhu C++, SCIENCE, Veletiny, 1997.
ISBN – 80 –901475-9-3
- [4] norma jazyka
- [5] Racek. S., Kvoch. M, Třídy a objekty v C++, Kopp 1998, České Budějovice (www.kopp.cz)

Studenty používané odkazy (jsou v nich chyby ale celkem to jde)

www.builder.cz – převážně pro pracující v Borland Builderu ale jsou zde i výukové kurzy a odkazy

www.cplusplus.com – zdroje a výukové programy, odkazy

<http://www.eternal.cz/index.php?nLevel=21> – kurz programování v C++ (relativně stručný)

<http://www.fi.muni.cz/usr/jkucera/pb161/index.htm> – stránky MU

A něco pro pobavení:

<http://www.kiv.zcu.cz/~brada/desatero.html>

originál je na

<http://www.lysator.liu.se/c/ten-commandments.html>