Struktura (union)

- struktura a union jsou složené typy, které "v sobě" mohou obsahovat více proměnných
- struktura obsahuje v každém okamžiku všechny své proměnné, union obsahuje (=je "aktivní") pouze jednu proměnnou. Ve struktuře jsou proměnné "za sebou" a tudíž má rozměr rovnající se (minimálně) součtu rozměrů jednotlivých proměnných (někdy se vkládají mezi proměnné mezery, aby byly proměnné zarovnány na hranici – například adresa dělitelná 4,8,16…). V unionu jsou proměnné "přes sebe" a má rozměr největší z nich. Je možné přistupovat ke všem proměnným, pouze jedna má však smysl pro čtení (ta naposled naplněná. Která proměnná to právě je se musí pamatovat jinde).

struktura

adresa	typ proměnné	název proměnné
200	int	ii
204	char [16]	cc
214-21c	double	dd

union

adresa / proměnná	int ii	char [16] cc	double dd
200-204	xxx	xxx	xxx
205-208	N	xxx	xxx
208-216	N	xxx	N

N = nevyužito

Definice struktury

- dva kroky: popis struktury a definice proměnné
- popis struktury je možný dvěma způsoby
  – pouhé oznámení jména struktury (nelze použít proměnnou ale pouze ukazatel)
  - a kompletní popis jména struktury a jejích složek
- popis struktury netvoří kód -> umísťujeme do hlavičkového souboru
- kód se tvoří až při definice proměnné podle tohoto popisu struktury
- celý/používaný název typu se skládá z klíčového slova struct a jména typu

```
extern struct Jmeno; // oznámení jména, lze použít pouze ukazatel na tento typ

struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err;
};              // konec definice (nezapomínat na středník)
//  zde lze definovat proměnnou
struct SKomplex aa, *paa; // definice proměnné (vyhradí paměť minimálně na
// 2x double a 1x int); definice ukazatele (vyhradí paměť na adresu)
struct Jmeno *pjj; // pouze ukazatel, proměnná není možná – není známa velikost
```

Definice s inicializací

je možné provést definici s inicializací
struct SKomplex aa = {12.1,13.1};
struct SText bb = {“toto je text“,20,50};

v C99 lze i odkázat na proměnnou
struct SKomplex cc = {.Im = 14};

Operátory přístupu k prvkům struktury „.“(tečka) a „ ->“

- pro přístup k vnitřním proměnným se používá operátor tečka
- pokud máme ukazatel na strukturu, používáme k přístupu operátor -> což je kratší zápis než využití dvojice „hvězdička+tečka“ „(*).“

```
struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err;
};

struct SKomplex aa, *paa=NULL; // definice proměnné a ukazatele

paa = &aa; // ukazatel je nutné inicializovat

if (!aa.err) // není-li proměnná v chybovém stavu
    aa.Re = aa.Im * aa.Im;
if (! (*paa).err)  // díky nižší prioritě „.“ než má „*“ je nutné použít ( )
    paa-> Im = paa->Re; // přístup k prvkům pomocí ukazatele na strukturu
```

V komplexní proměnné podle předchozí definice zaměňte Re a Im složky.
Předveďte pro proměnnou i ukazatel na ni.

```
struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err; };

struct SKomplex aa, *paa=NULL; // definice proměnné a ukazatele
double pom;
paa = &aa; // ukazatel je nutné inicializovat
if (!aa.err) // výměna pro proměnnou typu struktura
 {
   pom = aa.Re;
    aa.Re = aa.Im;
   aa.Im = pom;
 }
if (! (*paa).err)  // výměna pro ukazatel
{
   pom = paa->Re;
    aa->Re = aa->Im;
   aa->Im = pom;
 }
```

Mechanizmus přiřazení struktur a definice s inicializací struktury strukturou

- pro přiřazení struktury do struktury se používá mechanizmus kopie paměťového bloku. Toto je výhodné, pokud nejsou vnitřní proměnné typu ukazatel. Pro ukazatel dojde k tomu, že nyní dvě struktury mají stejný ukazatel a tedy „vlastní“ společnou paměť (aniž by o tom měly nějaký signál).
- tento typ se nazývá mělká kopie, protože nejde do hloubky (nedělá kopii hodnot, na které se ukazuje)
- při přiřazení dvou ukazatelů se kopíruje adresa a ukazují na stejný objekt

```
struct SKomplex aa, *paa=NULL,*pbb=NULL;
struct SKomplex bb = aa; // provede se kopie paměťového bloku aa do místa bb
paa = (struct SKomplex*) malloc(sizeof(struct SKomplex));
pbb = &pb;
aa = bb; // provede se kopie paměťového bloku bb do místa aa

paa = pbb;  // kopíruje se hodnota, kterou je ukazatel, struktura se nekopíruje
// naalokovaná proměnná se „ztratila“ a již na ni není odkaz
free(paa); // neruší alokovanou proměnnou, ale proměnnou pb => chyba
```

Struktura jako návratová hodnota a parametr fce

- platí stejná pravidla jako pro ostatní typy
- buď se předávají hodnotou (nevhodné – velká paměťová náročnost) nebo ukazatelem -> pokud můžeme, dáváme přednost ukazateli. Nejde-li ukazatel, použijeme hodnotu.
- struktura jako návratová hodnota je nutná když se předává nová proměnná. Pokud předáváme zpět proměnnou, která byla parametrem, můžeme předat ukazatel.

Napište funkci pro součet dvou komplexních čísel.
Napište funkci pro návrat komplexního čísla s větší vzdáleností od počátku.

```
// součet dvou komplexních čísel – vzniká nová proměnná – návratem je  hodnota
struct SKomplex Soucet(struct SKomplex *p1, struct  SKomplex const *const p2)
//ukazatele u argumentů místo hodnot šetří místo (rozměr struct proti ukazateli)
{ // aby nedošlo k nechtěné změně struct ve funkci -> modifikátor const
 struct SKomplex ret; // vytvoří se pomocná proměnná
ret.Re  = p1->Re + p2->Re;
ret.Im  = p1->Im + p2->Im;
 return ret; // na základě hodnoty pomocné proměnné se naplní návratová hodnota
}

// vrácení „většího“ – vrací se jeden z parametrů => ukazatel
struct SKomplex * Delsi(struct SKomplex *p1, struct SKomplex *p2)
{double d1,d2;
  struct SKomplex *pret; // pomocná proměnná, hodnotu lze vrátit i bez ní
  d1 = sqrt(p1->Re* p1->Re+ p1->Im* p1->Im); //  výpočet délky
  d2 = sqrt(p2->Re* p2->Re+ p2->Im* p2->Im);
  pret = d1>d2 ? p1:p2; // zápis příslušné adresy do výsledné hodnoty
  return pret; // vrácení adresy většího prvku
}
```

Kopie pole pomocí struktury

Pomocí struktury lze vytvářet ve funkcích lokální kopie pole (pomocí mechanizmů překladače). Používat opatrně a pouze pro malá pole.

```
struct SVektor {
double iVektor[10];
int delka;
}
```

Při předávání do funkce hodnotou se vytvoří nová proměnná a do ní se překopíruje obsah původní proměnné – vytvoří se kopie pole. Nakreslete paměťovou mapu.

```
int funkce(struct SVektor vect, struct SVektor *v2) { … }
// upřednostňovat předávání pomocí ukazatele
```

volání

```
struct SVektor a,b;
funce(a,&b);
```

nedynamické a dynamické proměnné typu struktura

- proměnnou typu struktura je možné získat i dynamicky (musí se chovat jako standardní typy)
- Je možné také „požádat“ o pole struktur

```
struct SKomplex aa,*paa=NULL, *pap=NULL;

paa = (struct SKomplex*) malloc(sizeof(struct SKomplex)); // jedna struktura

pap = (struct SKomplex*) malloc(20 * sizeof(struct SKomplex)); // pole 20-ti struktur
// struktury jsou lineárně za sebou v paměti

paa->Re = aa.Im; // „klasický“ přístup k vnitřním proměnným u jednoho prvku
pap[10].Re = 8; // přístup k prvku pole –
// výsledek pap[X] není ukazatel, ale prvek <=> *(pap+X)
```

Práce se strukturou obsahující dynamickou proměnnou.
Hlavičkový soubor:

```
struct SText {char *txt;int delka;}; // delka neni nutna, ale zrychlí program
// txt je ukazatel, po definici proměnné neinicializovaný

#define INIT_TEXT(struc, text,del) struc.txt = malloc((del+1)*sizeof(char)); \
strncpy(struc.txt,del+1,text);struc.delka = del+1;
#define DESTR_TEXT(struc) if (struc.txt) free(struc.text); struc.text=NULL; \
struc.delka=0;

#define INIT_UKTEXT(pstruc, text,del) pstruc= malloc(sizeof(SText)); \
INIT_TEXT(*pstruc,text,del);
#define DESTR_UKTEXT(pstruc) if (pstruc) { DESTR_TEXT(*pstruc); \
free(pstruc); }; pstruc = NULL;
```

Zdrojový soubor:

```
struct SText ee,*pee=NULL;
INIT_TEXT(ee,10,“ahoj“);          INIT_UKTEXT(pee,10,“zdar“);
DESTR_TEXT(ee);                   DESTR_UKTEX (pee);
```

Využití struktur

- s výhodou se využívají při tvorbě lineárních seznamů a stromů (a typů odvozených)
- základní vlastností těchto typů je, že kromě dat obsahují i ukazatel(e) na stejný typ jako jsou sami.
- lineární seznam obsahuje ukazatel na další prvek (poslední prvek má NULL). Modifikacemi jsou cyklické seznamy, které nemají poslední prvek. Obousměrně vázaný seznam má vazbu nejen na následující, ale i na minulý prvek. S jejich pomocí lze realizovat Frontu (FIFO), Zásobník (stack, LIFO), pole ...
- při realizaci stromu struktura s daty tvoří uzel a obsahuje dva (nebo více) ukazatele, které ukazují na uzly následující (binární (vážené) stromy, příkladem může být třeba strom pro morseovku – v uzlu je aktuálně dosažený znak, ukazatele ukazují na znak, na který se přesuneme, přišla-li tečka resp. čárka).