Rychlokurs jazyka C++

tento článek má sloužit jako stručný přehled jazyka C++, a má umožnit pohled na problematiku před jejím důkladným studováním aby při něm byly zřejmé souvislosti. Jedná se pouze o schematické uvedení nejvýznamnějších vlastností, tak aby byl při podrobném studiu zřejmější kontext učené látky. Rozsah je zhruba to co zůstane po prvním letmém přečtení tlustých knih. Jsou uvedeny pouze základní vlastnosti a vazby (ve skutečnosti je to složitější). Je použita lehčí (lidštější a proto ne zcela přesná) terminologie.

„hlavně stručně“

– je zaveden nový typ jednořádkového komentáře od dvojznaku „//“ do konce řádky

// tady je komentář

„neinicializovaná proměnná je špatná (nebezpečná) proměnná“
– snaha o to aby proměnná měla vždy smysluplnou hodnotu vede k tomu, aby bylo možno proměnnou nadefinovat až v okamžiku, kdy je možné jí přiřadit smysluplnou hodnotu – to je nadefinovat kdekoli. (na rozdíl od C, kdy je možné definovat pouze na začátku bloku).

```
{
 int i = 0;
i ++;
float j = Pracuj(i);  // definice proměnné uprostřed bloku
}
```

„říkáme mu jinak ale je to ten samý (starý)“

– je zaveden nový způsob pro předávání proměnných – reference. Pomocí reference získáme nové (druhé, další) pojmenování stávající proměnné.

```
int a, b, c;
// definice proměnných v základní funkci – příprava pro volání
funkce(a,&b,c); // volání funkce
// a prototyp funkce vypadá tak
void funkce (int aa, int * bb, int &cc) // předání hodnotou –
proměnná aa má stejnou hodnotu jako a ale je to
// lokální proměnná, změna aa se na originálním a neprojeví
// předání hodnotou ukazatele (adresou) – proměnná bb je
adresa na které se nachází hodnota b, manipulací s
// hodnotou na této adrese pracujeme s proměnnou b
// předání odkazem (referencí) – hodnota cc je reference
(přezdívka, jiné jméno) pro proměnnou c.  manipulace
// s proměnnou cc je totéž jako práce s původní hodnotou (tj.
s proměnnou c)
{
```

```
  aa = 10; // mění se pouze hodnota proměnné aa, hodnota
proměnné a v původní funkci se nemění
  *bb = 10; // pomocí práce s adresou je možné pracovat
s hodnotou proměnné, která zde leží – proměnnou b
       // adresu je nutné dereferencovat pomocí operátoru pro
přístup k hodnotě na adrese „*“
  cc = 10; // jelikož je cc pouze jiné jméno pro c, projeví se
změna i na původní proměnné, je to stejné jako c=10
}
// v případě volání funkce (d,&e,f) je nutno v komentářích ve
funkci vyměnit a za d, b za e, c za f

double &rd = pole[i].data[j].hodnota[k];
// reference pro zjednodušení složitého přístupu k proměnné
// reference musí být inicializována
```

„neexistují dvě stejně pojmenované funkce“ platí v C
„neexistují dvě stejné funkce“ platí v C++

– Nyní je možné stejně pojmenovat dvě a více funkcí, pokud mezi nimi lze rozlišit na základě typu nebo počtu parametrů – tento „jev“ se nazývá přetížení funkcí. Následující funkce lze odlišit při volání „přesnými“ parametry:

```
funkce-(int)
funkce-(int, float)
funkce(char)
funkce(float, char)
```

nelze však odlišit při volání

```
funkce (6.25); // parametr je double. Ten překladač umí
// konvertovat na int i char a proto se nemůže rozhodnout
funkce ((int)6.25); // nyní už to „sedí“ – přesný typ
```

„nebudu to vyplňovat když je to furt stejný“

– pokud se dá očekávat, že některý z parametrů funkce bude nabývat stejných hodnot, a pouze výjimečně jiných, potom je možné použít – implicitní parametry ve volání funkce. Pokud v hlavičce funkce (v deklaraci) uvedeme u předávané proměnné hodnotu, potom se tato hodnota do proměnné uloží v případě, že ji při volání nepoužijeme. Např. většinou se ve funkci nic netiskne, ale někdy to potřebujeme i s tiskem daného řetězce.

```
fce(int op1,int op2, char *txt=NULL) // poslední položka je
řetězec. Pokud není řetězec zadán, potom je
//v proměnné NULL a text se netiskne
 fce(4,5,“ vysledek testu“); // jsou zadány všechny parametry,
včetně proměnné txt. Protože je txt různá od
// NULL text se vytiskne
```

```
fce(5,6); // poslední parametr není programátorem uveden, ale
ve funkci existuje – je překladačem nastaven
// na hodnotu NULL
```

„ať mi do toho nevrtají“

– pokud chceme, aby proměnná nebyla měněna, potom ji označíme klíčovým slovem const. Toho se používá při předávání pomocí referencí, které je rychlé (pro rozsáhlé struktury) ale nebezpečné v tom, že se ve funkci mohou nekontrolovaně změnit. Možnost změny se potlačí uvedením const před proměnnou. Zavedeme-li si pravidlo, že pokud se proměnné ve funkci nemění, pak je označena const, potom z pohledu na hlavičku funkce vždy víme, zda se proměnná uvnitř změní a můžeme se podle toho zachovat.

```
fce (TRIDA const &promenna) // promenna se dá ve funkci
použít, ale nejde měnit její parametry. Nelze zavolat
// ani funkci, která by proměnnou mohla změnit
```

„potřebuji hodně dat (paměti)“

– dynamická alokace je možná pomocí operátorů new a delete. Nový typ alokace umožňuje naalokovat paměť pro daný typ, nebo pole daného typu. Jelikož je znám typ, pro který je paměť vytvářena a také počet proměnných, lze provést inicializaci a zrušení (konstruktory a destruktory z objektově orientovaného programování).

```
TRIDA *prom = new TRIDA; // vytvoření jedné proměnné,
s inicializací (libovolným) konstruktorem
delete prom; // „zrušení“ jedné proměnné, s „úklidem“
destruktorem
TRIDA *prom = new TRIDA[kolik]; // vytvoření pole proměnných,
s inicializací (implicitními) konstruktory
delete[] prom; // „zrušení“ pole proměnných, s „úklidem“ všech
proměnných destruktory
```

„kdo šetří … (ten plýtvá) “

– pokud mám jednoduché funkce, je možné je použít tak, aby se nevolaly jako funkce ale provedly se přímo na daném místě. Ušetříme tedy funkční volání (práci se zásobníkem), ale prodloužíme kód -> hodí se pro jednoduché a krátké funkce (bez cyklů, bez volání funkcí …). Napsaná funkce je prakticky předpis (jako makro v C), podle kterého se v daném místě vytvoří kód -> lze ji uvádět v hlavičkovém souboru.

```
inline int secti(a,b) {return a+b;} // jednoduchá funkce,
která bude pouze rozvinuta a nebude se volat
c = secti(d,e);  // zde se prakticky provede c = d + e; podle
předpisu pro definici funkce, funkce se nezavolá
```

„má to logiku“

– je zaveden nový typ pro logické proměnné. Jmenuje se `bool` a proměnné nabývají hodnot `true` a `false`. (Dříve to byly hodnoty 0 a 1, ještě dříve 0 a nenula (tj. cokoli jiného) – tyto staré lze stále použít (zpětná kompatibilita).

```
bool hodnota= a > b;  // uloží se true nebo false podle
hodnoty
return true; // použití konstanty typu bool
```

„operátorujte s vlastními typy“

– je možné napsat operátor pro vlastní typ. Operátor je vlastně speciálně pojmenovaná funkce, pro kterou existuje zkrácené volání. Překladač volá operátory podle kontextu.

```
bool operator>(KOMPLEX &a, double b) {return
((a.real*a.real+a.imag*a.imag) >b*b);}
// srovnání komplexních čísel s doublem podle velikosti
KOMPLEX c;
double d;
if (c>d) … // můžu použít operátor i pro nově vytvořený typ
(strukturu) – zkrácená verze
if (operator>(c,d)) .. // plná verze stejného volání
```

„tam a zpět“

– C++ zavádí nový typ vstupu a výstupu. Díky přetěžování operátorů << a >> je možné realizovat vstup a výstup pro dané typy.

```
#include <fstream.h> // příslušná knihovna
char a; double b;
cout << a << “ teplota “ << b << “ tlak “; // výstup na
konzolu proměnných různých typů prokládaných texty
// na příkladu je vidět i „zřetězení“ příkazů, které je
// možné díky tomu, že operátor vrátí referenci (tedy cout),
// který je následně použit pro následující operátor
cin >> a >> b; // vstup proměnných různých typů z konzoly
```

“každý na svém písečku”

– umožňuje “zabalit” jednotlivé celky dat a funkcí (např. podle autorů). Balíčky se nazývají prostory jmen (namespace). Pokud chceme použít data z balíčku, musíme ke jménu proměnných či funkcí přidat i příjmení – název balíčku. Za takový balíček je možné považovat i třídu či strukturu.

```
namespace Matice {
int iii;    // tyto funkce a data patří do prostoru matice
```

```
int fff(int ab) {iii=4;} // proměnná je ve stejném prostoru
}
Matice::iii=5; // proměnná je v jiném prostoru
using Matice::iii; // zveřejnění proměnné z jiného prostoru
iii=5; // používání zveřejněné proměnné
```

„všichni kdo patří k sobě mají stejné příjmení“

– příjmení (jméno prostoru) připojujeme před jméno dat pomocí oddělovače „::“, kterému říkáme operátor příslušnosti.

`Matice::Nastav(a,b);` funkce Nastav z prostoru Matice
`Vektor::Nastav(a,b);` funkce Nastav z prostoru Vektor – nebudou spolu kolidovat. Mají stejné jméno ale různé příjmení

„všechno na jednom místě“

– spojení dat a funkcí, které s nimi pracují do jednoho celku. Možnost vytvořit nový datový typ, který obsahuje data i metody (funkce).

```
class TTT{
float a, b,c ; // data
int Nastav(float cc,float dd) {… } //metoda
};
```

„data jsou moc důležitá na to aby k nim byl přístup“

– zavádí se přístupová práva, která říkají, kdo má právo přistupovat k datům a metodám. Data je lepší mít pod kontrolou, proto k nim přistupujeme pomocí metod, která je mohou před zápisem a čtením kontrolovat nebo modifikovat.

„Každý má přístup pouze přes jednu mez“

– okolí může k veřejným metodám, veřejná metoda může k privátním datům třídy

private	public	globální prostor
	přístup k private položkám stejné třídy, (může i k public jiných tříd)	přístup k public položkám

---

„nejlépe když se věci nemění (na pohled zvenčí) ale přitom jsou stále lepší (vnitřně)“

- Interface zůstává zachován, (vnitřní) implementace se mění. Změna dat na jiný typ (jinou reprezentaci), není zvenčí vidět, protože přístupové funkce data modifikují tak aby to nebylo poznat. Vnitřně tedy dojde ke zlepšení ale způsob použití se nemění.

varianta	privátní data	veřejná data	okolí
1	Reálná a imaginární složka	GetReal() {return Reálnou složku}	zavoláme GetReal
2	Amplituda a úhel	GetReal() {return Amplitud*cos(uhel)}	zavoláme GetReal

---

„přátelé si mohou dovolit více“

- pokud třída označí někoho (třídu, funkci) za přítele – friend, potom tento má právo přistupovat k privátním datům dané třídy

```
class TTT {
int i;
public:
friend funkce(TTT &pom);  // zde se označí funkce za přítele
};

funkce (TTT &pom)
{
 pom.i = 4; // a zde využívá svého privilegia (výjimky)
k přístupu k privátním datům třídy
}
```

„s daty je nutné cílevědomě pracovat“

– společně s daty jsou ve třídě také metody – funkce, které pracují s daty třídy. Metoda se od funkce liší tím, že je v ní implicitně přítomen prvek (proměnná), která si ji zavolala.

```
class TTT {
public:
int Nastav(float cc,float dd) {… } //metoda
};
TTT aa;
aa.Nastav(3,4); // proměnná aa třídy TTT si zavolá metodu
Nastav  této třídy
```

„uvnitř se všichni jmenují stejně“

– proměnná, která metodu vyvolala je uvnitř metody přístupná přes ukazatel this. Tato proměnná je přítomna implicitně (zařídí překladač). Předávané proměnné fungují jako u normálních funkcí.

```
class TTT {
int x,y;
public:
```

```
int Nastav(float cc,float dd) {this->x = cc; this->y = dd;} //
this je ukazatel na proměnnou, která
// metodu vyvolala. tj. nejprve aa a potom bb
};
TTT aa,bb;
aa.Nastav(3,4); // vstupu do Nastav je proměnná aa, která
metodu vyvolala přístupná přes this
bb.Nastav(7,8); // po vstupu do Nastav je proměnná bb, která
metodu vyvolala přístupná přes this
```

„zrození a smrt (vznik a zánik) – nejdůležitější okamžiky v životě“

– speciální metody ovlivňují vznik a zánik objektu. Tyto metody je nutné napsat, ale volá je překladač. Konstruktory zajišťují inicializaci – vznik objektu. Destruktor zajišťuje zánik objektu. Je to vlastně speciální metoda.

```
class TTT {
int x,y;
public:
TTT(void) {this->x=0;this->y=0;} // konstruktor má stejný
název jako jméno třídy, volá se pro TTT a;
TTT(float aa,float bb) {x=aa; y = bb;} // konstruktor se dvěma
parametry, volá se pro TTT b(3,4);
// this není nutné uvádět – vyplývá z kontextu
TTT(TTT const &bb) {x=bb.x; y = bb.y;} // copy konstruktor –
vytvoří kopii proměnné stejného typu
TTT(double ee) {  } // konverzní konstruktor (má jeden
parametr) – „změní“ proměnnou double na TTT
~TTT(void) {} // destruktor – zde se vyřeší zánik objektu
};
TTT aa,bb=aa;
// volá se konstruktor bez parametrů a kopykonstruktor
// překladač vlastně doplní na aa.Komplex(), bb.Komplex(aa)
// (!! programátor to takhle volat nemůže!!)
TTT dd (6.7), ff(4,8); // volá se konverzní konstruktor a
konstruktor se dvěma parametry
```

"musí to mít hlav(ičk)u"

v hlavičce je definice třídy. vše co je zde uvedeno je jen předpis pro tvorbu objektu a jeho vlastností. metody zde uvedené jsou realizovány jako inline. metody, které zde mají pouze hlavičku nejsou inline. tělo mají definováno ve zdrojové části a jsou volány funkčním voláním.

```
====== trida.h ======

class trida {
int x; // v paměti se vyhradí místo až s definicí proměnné
static int pocet; // deklarace - nevyhrazuje místo v paměti
```

```
public:
int metoda1(void) {return 1;} // má tělíčko v definici třídy -
// inline metoda
inline int metoda2 (void) ; // nemá tělíčko, ale je označena
// jako inline -  tento zápis vede ke zpřehlednění hlavičky –
// nepletou se tělíčka do prototypů funkcí
int metoda3(void); // není označeno inline ani nemá tělíčko –
// funkční volání
}

int trida::metoda2(void){return  2;} // je to inline, definuje
// se v hlavičce  je mimo třídu, proto zde musí být plné jméno
// metody i s trida::

===== trida.h  konec ===

===== trida.cpp ======
#include "trida.h"
int trida::pocet=0; // definice statické proměnné s
// inicializací zde je vyhrazana paměť pro uložení proměnné

int trida::metoda3(void) {return 3;} // metoda volaná funkčním
// voláním zde se tvoří kod metody v paměti. Je nutné uvést
// plné jméno

===== trida.cpp konec ====

===== pouziti.cpp =====
int main()
{
trida::pocet ++; // statická prměnná "žije" i bez objeků
třídy. je v "globální" paměti
trida aaa;
// zápis volání je stejný pro všechny případy
aaa.metoda1();  // zde je vložen kod na základě předpisu
aaa.metoda2();  // zde je vložen kod na základě předpisu
aaa.metoda3();// zde je provedeno funkční volání

return 0;
}
===== pouziti.cpp konec ====
```

„konstanta musí být konstantní“

- konstanta nesmí být změněna. Nelze jí tedy přímo měnit. Nelze ji měnit ani pomocí funkcí / metod. Má-li být metoda použita na konstantní objekt, musí to být metoda označena jako konstantní – metoda, která objekt nezmění. Jinak překladač volání nepovolí.

```
T {
int i;
public:
int Metoda1(void) const {return 1;} // nemění objekt
int Metoda2(void) {i=0;return 2;} // mění objekt
}
const T oo;
oo.Metoda1() ; // lze použít na konstantní prvek – nemění ho
oo.Metoda2(); // nelze použít na konstantní prvek – mění ho
```

„vyrobte si operátory“

- pro tvorbu operátorů se používají speciální metody. Jejich název je dán klíčovým slovem operator a znakem operátoru. To je plné volání. I když toto plné volání lze použit i k volání, většinou se používá zkrácená verze. Operátor má vlastnosti standardního operátor (priorita, počet parametrů …) a měl by se činností blížit jeho významu (operátor + sčítá, spojuje …) tak aby na první pohled bylo jasné co se děje.

```
class T {
int a;
public:
T& operator=(T const &bb)
     {this->a = bb.a;return *this;} // definice operátoru =
 // až na speciální jméno s operátor, je to stejné jako
// jiné metody.
// Má to návratovou hodnotu, název a seznam parametrů
// následovaný tělíčkem
}

T c,d,e;

c.operator=(d.operator=(e)); // plné volání – stejné jako
// ostatní (normální) metody
// díky návratové hodnotě – výsledný prvek „se předá ven“ – je
// možné zřetězení operátorů =, kdy výsledek jednoho přiřazení
// je parametrem dalšího
c = d = e;  // stejné volání ve zkráceném zápisu
```

„směna je základem pokroku“

- ke změnám typu se používají (konverzní) konstruktory (to jsou konstruktory s jedním parametrem) a přetypování (konverzní operátor). Tyto předpisy se používají ze strany operátora – explicitní konverze – i překladače – implicitní konverze.

```
T{
int a;
public:
T(int i) {a=i;} // konverze z int na T
```

```
T(float f){a=f;} // konverze z float na T

operator double() {return (double)a;} // konverzní operátor
}

T a=4, b=3.34; // inicializace proměnné třídy T intem a
// floatem – je to vlastně změna typu

double bb = double(b); // použití konverzního operátoru –
// přetypování
```

„bez práce koláče – získáme děděním“

dědění je mechanizmus, kdy se vlastnosti jedné třídy použijí jako základ pro jinou třídu.
První řádek určuje definici třídy a způsob dědění.
Ve třídě A jsou prvku s různými přístupovými právy. Ve zděděných třídách je naznačeno jaká práva mají po dědění. Private není „vidět“ nikdy – nutno přistupovat přes metody Get, Set. Protected je možné přistupovat přímo – je to pohodlné ale nebezpečné – porušení ochrany dat.

class A	**class B:private A**	**class C:protected A**	**class D:public A**
public a private b protected c	private a - private c	protected a - protected c	public a - protected c

Při dědění se nic neztrácí. Lze překrýt. Vyhodnocení je od potomka k bázi.
Základní způsoby při dědění

class Base {
public:
Metoda1();
Metoda2();
Metoda3();
}
	class Dedeni:public Base {
	public:
	
	Metoda2();
	Metoda3() {Base::Metoda3();}
	Metoda4();
	}
		
		// necháme původní metodu z báze
		//vytvoříme novou metodu,původní je skrytá
		// doplníme původní metodu
		// vytvoříme novou metodu

Objekt třídy Dedení má tedy Metodu1 1x, Metoda2 2x, Metoda3 2x a Metoda4 1x.

postup volání konstruktorů - konstruktor bázové třídy, konstruktory lokálních proměnných (třídy) v pořadí jak jsou uvedeny v hlavičce, konstruktor (tělo) dané třídy
destruktory se volají v opačném pořadí než konstruktory

Jelikož objekt se vytváří postupně, je na jeho začátku uložena bázová třída. Všechny třídy, které jsou odvozeny od stejné třídy, mají tedy stejný začátek. Na prvek zděděné třídy se můžeme odkazovat i pomocí (ukazatelů) na jeho předky

---

„míchat jablka s hruškama“

virtuální metody – umožňují pracovat společným způsobem s různými třídami, pokud vycházejí ze společného základu.
Při „klasickém“ překladu se přímo doplní adresa volané funkce či metody – záleží na tom, jak je proměnná definována.
U virtuálních metod dochází k „postupnému“ volání, kdy se až za chodu programu zjišťuje adresa metody. Ta se zjistí tak, že se u prvku program za chodu podívá do tabulky virtuálních metod a podle toho odskočí. Tabulka virtuálních metod se „přiděluje“ prvku při vzniku – záleží tedy na tom jak objekt vznikl, a ne na tom čemu je aktuálně přiřazen.
Pokud různé prvky zděděné z jedné třídy dáme do pole ukazatelů této základní třídy, pak při volání virtuálních metod se volají tyto podle toho jak objekt vznikl (tedy se volají metody příslušné typu)

```
class A {
virtual Metoda (){cout << „a“;}
}
class B:A{
virtual Metoda() {cout << „b“;}
}
class C:A{
virtual Metoda() {cout << „c“;}
}

fce () {
A* pole[2];
B b;
C c;
pole [0] = &b; pole [1] = &c;

pole[0].Metoda(); // tiskne b – podle vzniku ne podle toho čemu je přiřazeno
pole[1].Metoda(); // tiskne c
}
```

„interface“

Metody (virtuální) bázové třídy jsou společným prvkem, všech tříd zděděných a přitom jsou v nich specializované. Protože jsou společné pro všechny třídy, hovoříme o interface pro danou skupinu tříd – jsou to metody, které najdeme u všech těchto tříd a můžeme s nimi zacházet společně = stejně.

„nepoužitelný interface“

pokud virtuální metody v bázové třídě nemají „tělíčka“, hovoříme o abstraktní bázové třídě. Nejde vytvořit její objekt. Slouží pouze jako definice interface.

„kolikrát zdědíš, tolikrát …“

Je možné dědit i z více tříd najednou. Pokud má více předků stejně pojmenovaná data, je nutné pomocí operátoru příslušnosti určit, která máme na mysli např. BA::a, BB::a – proměnná a, která patří k bázové třídě BA, nebo BB. Nelze dědit na jedné úrovni dvakrát z jedné třídy. Je možné ale zdědit jednu třídu ve více předcích.

„pište to přes kopírák“

Užitečným nástrojem je mechanizmus šablon. Šablona je předpis=návod (a proto je v hlavičkovém souboru) na základě kterého si překladač „napíše“ metody nebo třídy podle potřeby. Programátor napíše funkci nebo třídu s obecným typem a v případě nutnosti si překladač (nebo programátor) určí konkrétní typ – na jeho základě podle předpisu (šablony) si překladač napíše vše sám. Napíše se to tedy pro jeden obecný typ a překladač si potřebné pro (libovolné) konkrétní typy napíše sám.

```
template < typename T >
double max ( T h1, T h2 ) {return h1>h2?h1:h2;}
```

T značí onen obecný typ. Při volání (se dvěma parametry shodného typu) si překladač „vytáhne“ tento typ, dosadí ho za T a „napíše“ si příslušnou verzi funkce. Toto dokáže pro všechny typy. Problémem je, že to nedokáže pro parametry různých typů.

```
template < class T, class S >
double max ( T h1, S h2 ) {return h1>h2?h1:h2;}
```

Starší zápis s označením obecných typů pomocí class a ne typename. Nyní si překladač vytvoří funkci i pro různé parametry.

Třída pomocí šablony (opět pouze předpis s obecným typem)

```
template < typename T >
class A {
T a , b ; // za T se dosadí až při konkrétním použití
T fce ( double, T, int )
}

T A<class T>:: fce (double a, T b,int c) {}
```

potom

A <double> c, d; // označení konkrétního typu musí být při definici
bude c,d třídy A, kde a,b jsou double
A <int> g, h;
bude g,h třídy A, kde a,b jsou int,
Tučně jsou vyznačeny názvy typu – jsou to tedy dva různé typy A<double> a A<int>

„chyby řešíme výjimečně“

nový mechanizmus ošetření chyb se nazývá výjimka. Tento mechanizmus spočívá v tom, že chyby se většinou neřeší v místě kde vznikly (zanořený výpočet v knihovně), ale někde v uživatelské části programu. Proto se chyba „neřeší“ ale „vypustí“ se objekt = výjimka. Následně se ukončují metody a funkce (včetně destruktorů objektů) až do doby, než výjimku „chytíme“. „chytat“ můžeme pouze v označeném bloku.

```
try
  {
…
volání funkce, která hodí výjimku pomocí trow
…
} catch(X:V) {zde je řešení pro výjimku X:V}
catch (X:V2) {zde je řešení pro výjimku X:V2}
```
