Jazyk C++

Význam názvu

- *C* … založen na syntaxi jazyka C
- ++ … je jeho nadstavbou (výrazným rozšířením)

Historie

- Jazyk v první polovině 80. let navrhl Bjarne Stroustrup a publikoval ho v roce 1985 v knize "The C++ Programming Language". Tento popis se stal neoficiálním standardem jazyka C++ a byl používán pro implementaci překladačů.
- Od roku 1990 se jeho dalším vývojem a standardizací zabývá odborná skupina ISO–WG21.
- V roce 1998 byl zveřejněn standard jazyka označovaný C++98 .
- V roce 2003 byl zveřejněn standard označovaný C++03 .
- V roce 2011 byl zveřejněn standard označovaný C++11 .
- Na rok 2014 je naplánován další standard dnes označovaný C++14 .

Jazyka C++ je určen pro profesionální programy – rozsáhlé úlohy, složité algoritmy, výkonově náročné programy, časově kritické programy, systémové programy.

Programy napsané v jazyce C++:

- Windows XP, Vista, Windows 7
- Microsoft Office (Word, Excel, Access, PowerPoint, Outlook)
- Visual Studio (některé knihovny jsou napsané v C#)
- Microsoft SQL Server
- MySQL
- Apple MAC OS
- Acrobat,
- Google – vyhledávací systém, prohlížeč Chromium
- Mozilla Firefox
- Winamp
- VLC media player, KMPlayer
- YouTube, Facebook, Paypal
- akční hry, simulátory
- a další

Standardy jazyka C: C89, C99, C11

Program = deklarace + příkazy

Deklarace – 2 druhy:

- Jsou zároveň i definice. Tento druh deklarací zavádí (definuje) nové entity (proměnné, funkce, datové typy). Ve stejném rozsahu platnosti nelze takovou deklaraci uvést vícekrát.
- Jsou jen deklarace, nejsou definice. Popisují něco, co už je definováno nebo bude definováno, či by mělo být někde definováno. Takovou deklaraci lze uvést vícekrát i ve stejném rozsahu platnosti.

```
int a;              // je definice

extern int a;       // není definice

struct Bod { float x,y; };   // je definice – nový datový typ

float sqrt0(float x)
{ return x>=0 ? sqrt(x) : 0; }   // je definice

float sqrt0(float);              // není definice

typedef unsigned char byte;      // není definice
```

Nové základní datové typy

Logický datový typ: `bool`

Hodnoty: `false`, `true`

```
bool jeCeleCislo(float a) { return a == floor(a); }
```

Široký znak: `wchar_t`

Kódování Unicode, implementace UTF-16 nebo UTF-32. Konstanty mají na začátku označení `L`.

```
const wchar_t srdce = L'♥';

const wchar_t karty[] = L"♠♣♥♦";
```

Celočíselné datové typy

`long long int`, `long long unsigned` – VS - implementovány 64 bity.

(`int` – VS - implementován 32 bity, `long int` – VS - implementován 32 bity).

Konstanty mají přípony `ll`, `ull`.

Označení datového typu `int` nelze v deklaracích vynechat.

```
const int SEDM=7;   // C lze: const SEDM=7;

int foo();          // C lze: foo();
```

Definice strukturovaných datových **struct**, **union** definují kompletní nové datové typy. Při použití takto definovaného strukturovaného datového typu stačí uvést jméno datového typu (není zapotřebí uvádět klíčové slovo **struct** nebo **union**).

```
struct Bod { float x,y; };

Bod b;

float foo(Bod *b);
```

Deklarace s klíčovým slovem **auto**

```
auto jméno = výraz;
```

Datový typ deklarované proměnné je odvozen z inicializačního výrazu.

```
auto i = 0;     // int i = 0;

auto k = i+1u; // unsigned k = i+1u;
```

Lokální deklarace lze uvést v libovolném místě bloku.

```
void Quicksort(int a[], int k, int l)
{ int x = a[(k+l)/2];
  int i=k,j=l;
  do { while (a[i]<x) ++i;
       while (x<a[j]) --j;
       if (i>j) break;
       int w = a[i];
           a[i] = a[j];
           a[j] = w;
    ++i; --j;
  } while (i<=j);
  if (k<j) Quicksort(a, k, j);
  if (i<l) Quicksort(a, i, l);
}
```

V příkazu cyklu **for** lze uvést deklaraci proměnné (nebo proměnných).

```
int s=0;
for (int i=1; i<=10; ++i) s += i;
```

Platnost proměnné je omezena na tělo cyklu.

Deklarace reference (na výraz typu *lvalue*).

```
datový_typ & jméno = výraz_lvalue ;
```

Reference reprezentuje odkaz na výraz typu *lvalue*, který je uveden v deklaraci reference. Je vedle ukazatele dalším typem odkazu v jazyce C++.

Rozdíly mezi referencí a ukazatelem:

- V deklaraci reference musí být výraz typu *lvalue*, na který je reference odkazem. (V deklaraci ukazatele nemusí být žádný výraz, který mu přiřazuje hodnotu).
- Odkaz daný v deklaraci reference nelze změnit. (Hodnotu ukazatele lze změnit, ukazatel může postupně ukazovat na různá místa v paměti).
- Pro přístup k hodnotě, která je uložena v paměti v místě, na které je reference, se u reference nepoužívá žádný operátor. (U ukazatele používáme operátor dereference *, jestliže pracujeme s hodnotou, která je uložena v paměti v místě, na které ukazatel ukazuje.)

```
int i;

int &ri=i;

ri = 1;      // i = 1;

ri = 1-ri;   // i = 1-i;

unsigned p[10];

for (int i=0;i<10;++i) p[i] = 2*p[i] + p[i]&1;

for (int i=0;i<10;++i) { unsigned &r=p[i]; r = 2*r + r&1; }
```

Hlavní použití referencí je v deklaracích parametrů funkcí a deklaracích typů funkčních hodnot.

Volání parametru	*Deklarace parametru*	*Druh parametru*
hodnotou	Proměnná	vstupní
odkazem	Ukazatel	vstupní i výstupní
	Reference	vstupní i výstupní

```
struct Obdelnik { float a,b; };

float obsah(const Obdelnik &o) { return o.a * o.b; }

float obsah(const Obdelnik *o) { return o->a * o->b; }
```

Nulový ukazatel `nullptr`

Používá se místo NULL ve výrazech s ukazateli.

```
int *p=nullptr;        // C: int *p=NULL;
```

```
if (p==nullptr) { }   // C: if (p==NULL) { }
```

Na rozdíl od konstanty NULL, která je jen symbolické označení nulové hodnoty a lze ji použít i v aritmetických a logických výrazech, nulový ukazatel **nullptr** je výlučně pro ukazatele.

```
int i=NULL;      // int i=0;

int i=nullptr;   // chyba !!

if (i==NULL) { }      // if (i==0) { }

if (i==nullptr) { }   // chyba !!
```

Alokace paměti – **new**, **delete**

```
  ukazatel = new datový_typ;

  delete ukazatel;
```

```
struct Bod { float x,y; };

Bod *u = new Bod;  // Bod *u = (Bod *)malloc(sizeof(Bod));

delete u;          // free(u);
```

Alokace pole

```
  ukazatel = new datový_typ [počet];

  delete [] ukazatel;
```

```
Bod *pole;

pole = new Bod[10];  // pole = (Bod *)calloc(10,sizeof(Bod));

delete [] pole;      // free(pole);

float (*matice)[4] = new float [5][4];

auto matice = new float [5][4];   // stejný význam jako předchozí deklarace

delete [] matice;
```

Operátor **typeid**

Umožňuje zjistit nebo vypsat datový typ výrazu.

```
typeid(matice).name()    // float (*)[4]

int i;

typeid(i+1u).name()      // unsigned int
```

```
auto &ri=i;
typeid(ri).name()          // int
auto *pi=&i;
typeid(pi).name()          // int *
typeid(i)==typeid('a')     // false
typeid(i).name()           // int
typeid('a').name()         // char
```

V prototypu funkce je nutno uvést deklarace jejích parametrů (v C je lze vynechat).

```
float foo(float);  // C lze: float foo();
int foo();         // C: int foo(void);
```

Parametry funkcí mohou být *implicitní*, kdy v jejich deklaraci uvedeme jejich implicitní hodnoty (zadané výrazem). Implicitní parametry mohou začínat od libovolného parametru funkce, přičemž všechny parametry následující za implicitním parametrem musí být již implicitní. Při volání funkce lze vynechat parametry od libovolného implicitního parametru. Za neuvedené implicitní parametry se při volání dosadí jejich implicitní hodnoty (výrazy, kterými jsou implicitní hodnoty zadány, se vypočítají v okamžiku volání funkce).

```
int i;
void foo(float a, char c='x', int d=2*i, const char *s="C++")
{ … }
foo(1.2)                   // foo(1.2, 'x', 2*i, "C++")
foo(1.2, 'a')              // foo(1.2, 'a', 2*i, "C++")
foo(1.2, 'a', 5)           // foo(1.2, 'a', 5, "C++")
foo(1.2, 'a', 5, "\n")     // foo(1.2, 'a', 5, "\n")
```

Implicitní hodnoty parametrů se uvádí jen jednou. Jsou buďto (v prvním) prototypu funkce, je–li před definicí funkce uveden její prototyp, nebo jen v definici funkce, není-li před funkcí její prototyp.

```
void foo(float, char ='x', int =2*i, const char * ="C++");
void foo(float a, char c, int d, const char *s)
{ … }
```

Polymorfismus funkcí

Význam pojmu polymorfismus funkce: funkce má různé chování pro různé typy parametrů.

Můžeme mít více funkcí se stejným jménem, které se vzájemně liší datovým typy nebo počty parametrů. Na typech funkčních hodnot jednotlivých funkcí nezáleží, Mohou být stejné nebo mohou být různé.

```
char c(int i) { if (i<CHAR_MIN) return CHAR_MIN;     // 1. funkce
                if (i>CHAR_MAX) return CHAR_MAX;
                return i; }

char c(unsigned u) { return u<=CHAR_MAX ? u : CHAR_MAX; }
                                                      // 2. funkce
```

Při volání polymorfní funkce musí překladač být schopen ze skutečných parametrů uvedených v zápisu volání funkce rozpoznat, která z funkcí je volána.

```
c(1)    // volání 1. funkce – datový typ konstanty 1 je int

unsigned u;

c(2*u+1)      // volání 2. funkce – datový typ výrazu 2*u+1 je unsigned

c('a')     // datový typ konstanty 'a' je char
           // možné konverze: char → int
           //                 char → unsigned
           // konverze char → int je preferována ⇒ volání 1. funkce

float v;

c(v)       // chyba !! – datový typ proměnné je float
           // možné konverze: float → int
           //                 float → unsigned
           // žádná z nich není preferována

c((int)v)  // volání 1. funkce – explicitně stanovena konverze na typ int
```

---

inline funkce

Při překladu programu je funkce přeložena a následně uložena na určitém místě v přeloženém programu.

Volání funkce má fáze:

- Uložení skutečných operandů na stanovené místo (do registrů, na zásobník).
- Uložení návratové adresy (na zásobník).
- Skok na místo v paměti, kde je funkce.
- Po ukončení funkce návrat z funkce na místo, odkud byla volána.

inline funkce není samostatně umístněna v přeloženém programu, ale při překladu její tělo je vloženo do místa, kde je funkce volána. Její použití je vhodné v případech, kdy:

- funkce je velmi krátká

- funkce není příliš rozsáhlá a přitom volání funkce v průběhu vykonávání programu proběhne mnohokrát

```
inline float sqrt0(float x) { return x>=0 ? sqrt(x) : 0; }

float a,b=-2.1;

a=sqrt0(b);     // a= b>=0 ? sqrt(b) : 0;
```