JAZYK C++ – ÚVOD

Historie

Jazyk C++ je objektově orientovaným rozšířením jazyka C.

Autor C++ – Bjarne Stroustrup z firmy AT&T Bell Laboratories.

Rok 1982 – Stroustrup rozšířil jazyk C o objekty pomocí preprocesoru – nazval jej „C s třídami“ (C with classes).

Rok 1985 – první verze C++.

Rok 1998 – norma ISO/IEC 14882:1998 pro jazyk C++.

Rok 1999 – norma ISO/IEC 9899:1999 – nová norma pro jazyk C.

Rok 2001 – norma ISO/IEC 9899:1999/Cor. 1:2001 (E) – opravy normy pro jazyk C.

Syntaktické definice v přednáškách

Při popisu programovacího jazyka se rozlišují *terminální* a *neterminální symboly*.

Terminální symboly se mohou přímo objevit v programu, např. klíčové slovo `int` nebo operátor `++`. V syntaktických definicích budou terminální symboly formátovány **tučně**.

Neterminální symboly se musí definovat. Neterminálním symbolem je např. *identifikátor*. V syntaktických definicích budou terminální symboly formátovány *kurzívou*.

V záhlaví syntaktické definice bude vždy jméno neterminálního symbolu, který definice popisuje a za ním dvojtečka. Potom budou v jednotlivých řádcích vypsány různé možné varianty významu definovaného symbolu. Bude-li definice obsahovat velké množství položek, budou tyto možnosti uvedeny na jednom řádku za sebou a na začátku řádku bude uveden text *jeden z*. Nepovinné části budou označeny dolním indexem $_{nep}$.

Např. definice množiny znaků je následující:

znak:

písmeno
číslice
speciální_znak
bílý_znak

číslice: jedna z

0 1 2 3 4 5 6 7 8 9

Deklarace a definice

Deklarace – přesněji *informativní deklarace* – informuje překladač o existenci nějakého objektu.

Definice – přesněji *definiční deklarace* – přikazuje překladači daný objekt vytvořit.

NEOBJEKTOVÉ ROZDÍLY MEZI C A C++

Identifikátory

Délka identifikátoru v C++ není dle normy omezena a všechny znaky jsou v něm *signifikantní* (významné). V jazyce C je signifikantních pouze 31 znaků. Překladače však mají omezení délky identifikátoru v C++. V prostředí Visual C++ 2003 je signifikantních implicitně 247 znaků.

Komentáře

Kromě komentáře

```
/* komentář
na více řádcích */
```

existuje v C++ komentář

```
// komentář
```

Veškerý text počínaje `//` do konce řádku se považuje za komentář.

Komentář začínající dvěma lomítky lze vnořit do komentáře začínajícího znaky `/*`.

Komentáře začínající znaky `/*` nelze dle normy C++ vnořovat. V některých překladačích lze vnořování komentářů povolit, ne však v překladači Visual C++ 2003.

Podle nové normy pro jazyk C lze psát i komentáře začínající znaky `//`.

Příkazy

Příkazem je též deklarace (informativní i definiční), tj. deklarace se může vyskytnout všude tam, kde syntaxe jazyka povoluje příkaz.

Blok (*složený příkaz*) v C++ je definován následovně:

blok:

{ *posloupnost_příkazů*$_{nep}$ **}**

posloupnost_příkazů:

příkaz posloupnost_příkazů$_{nep}$

Blok v jazyce C je definován takto:

blok_v_C:

{ *posloupnost_deklarací*$_{nep}$ *posloupnost_příkazů*$_{nep}$ **}**

Např.

```
int a[10];
a[5] = 10;
int j = a[5]*2; // v C++ je povoleno
```

Deklarace proměnné je dále povolena ve výrazu *podmínka* těchto příkazů:

- **if (***podmínka***)** *příkaz_1*
- **if (***podmínka***)** *příkaz_1* **else** *příkaz_2*
- **switch (***podmínka***)** *příkaz*

- **while (***podmínka***)** *příkaz*
- **for (***inicializační_příkaz***;** *podmínka***;** *výraz***)** *příkaz*

V cyklu `for` může být proměnná deklarována též v části *inicializační_příkaz*.

Ve všech těchto příkazech je deklarovaná proměnná lokální proměnou v rámci tohoto příkazu (bloku).

V prostředí Visual C++ 2003 deklarovaná proměnná v těchto příkazech implicitně není lokální proměnou v rámci tohoto příkazu. Toto nesprávné chování lze však změnit pomocí menu **Project | Properties**, část **C/C++, Language**, položka **Force Conformance In For Loop Scope**.

Proměnnou nelze deklarovat v příkazu cyklu

do *příkaz* **while (***výraz***)**;

ve výrazu *výraz*.

Např.:

```
if (int i = f()) Zpracuj(i); // OK
b = i; // CHYBA – i už neexistuje
for (int i = 0; i < 10; ++i) a[i] = i; // OK
b = i; // CHYBA – i už neexistuje
```

Podle nové normy pro jazyk C je možné i v C deklarovat proměnnou ve výše uvedených případech (včetně deklarace v bloku za jiným příkazem).

Skoky

Nelze přeskočit definici s inicializací, pokud se nepřeskočí celý blok, ve kterém se takový příkaz nachází. Toto omezení se týká příkazů `goto` a `switch`.

Následující zápis je nesprávný:

```
if (x) goto Pokracovani;
int y = 10;
//...
Pokracovani:
// ...
```

Následující zápisy jsou správné:

```
if (x) goto Pokracovani;
int y;
y = 10;
//...
Pokracovani:
// ...
```

```
if (x) goto Pokracovani;
{
   int y = 10;
   //...
}
Pokracovani:
// ...
```

Nesprávná konstrukce příkazu `switch`:

```
switch (c) {
   case 0:
      int x = 10;
      // ...
      break;
   case 1:
      // ...
}
```

Správné konstrukce příkazu `switch`:

```
switch (c) {
   case 0:
      int x;
      x = 10;
      // ...
      break;
   case 1:
      // ...
}
```

```
switch (c) {
   case 0: {
      int x = 10;
      // ...
      break;
   }
   case 1:
      // ...
}
```

Překladač Visual C++ 2003 povoluje přeskočit definici s inicializací u příkazu `goto`, u příkazu `switch` však nikoliv.

Datové typy

Celočíselné typy

Logický typ

V C++ existuje typ `bool` pro logické hodnoty. Má dvě možné hodnoty: `false` a `true`. V paměti zabírá 1 byte. Platí `false < true`.

Použije-li se číslo (reálné, celé, znak) na místě, kde překladač očekává logickou hodnotu, automaticky je konvertuje, a to:

- libovolné nenulové číslo na hodnotu `true`,
- nulu na `false`.

Podobně se převede hodnota ukazatele.

Použije-li se místo celého čísla logická hodnota, automaticky se převede na celé číslo, a to:

- `false` na nulu,
- `true` na 1.

Překladač Visual C++ 2003 při převodu čísla na logickou hodnotu zobrazí implicitně varování.

Znakové typy

Kromě jednobytových znakových typů `char`, `unsigned char` a `signed char` existuje dvoubajtový typ `wchar_t`. Ten se používá pro práci s asijskými znaky nebo s kódem UNICODE.

Znakové konstanty, např. `'a'`, jsou typu `char` (v jazyce C jsou typu `int`).

Znakové konstanty typu `wchar_t` se zapisují s prefixem L, např. `L'a'`.

Řídicí posloupnosti pro znak v kódu UNICODE:

`\Uxxxxxxxx`

`\uxxxx` – zkrácený zápis pro `\U0000xxxx`

kde `x` představuje číslici šestnáctkové soustavy.

Znaková konstanta v kódu UNICODE je potom `'\Uxxxxxxxx'` nebo `'\uxxxx'`.

Typ `wchar_t` je v C++ základní typ, v nové normě C je implementován pomocí deklarace `typedef`.

Funkce pro práci s dvoubajtovými znaky (tzv. širokými znaky – wide characters) mají ve svém názvu `w`, např. `wprintf`.

Celá čísla

Nová norma C zavádí celočíselné typy `long long int` a `unsigned long long int` (`int` lze vynechat), jejichž rozsah nesmí být menší než rozsah `long int` a `unsigned long int`. V prostředí Visual C++ 2003 mají velikost 8 bytů, tj.:

`long long int` $-2^{64}/2$ až $2^{64}/2 - 1$

`unsigned long long int` 0 až $2^{64} - 1$

Literály typu `long long` mají příponu `LL` nebo `ll`, modifikátor `unsigned` má příponu standardní, tj. `U` resp. `u`.

Nová norma jazyka C dále zavádí následující celočíselné typy:

- typy s přesně určenou šířkou:

  `int8_t`, `int16_t`, `int32_t`, `int64_t` – typy se znaménkem

  `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` – typy bez znaménka
- typy s určitou minimální šířkou – typy se šířkou alespoň *N* bitů:

  `int_least8_t`, `int_least16_t`, `int_least32_t`, `int_least64_t`

  `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, `uint_least64_t`
- typy s určitou minimální šířkou, pro něž jsou v dané architektuře výpočty nejrychlejší:

  `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `int_fast64_t`

  `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, `uint_fast64_t`
- typy, které mohou obsahovat ukazatele na objekty, neboli do nichž lze uložit hodnotu typu `void*` a pak ji přenést zpět, aniž se změní (nejsou povinné):

  `intptr_t`

  `uintptr_t`
- typy s maximální šířkou (nejsou povinné):

  `intmax_t`

  `uintmax_t`

Zpravidla se jedná jen o jiná jména pro základní celočíselné typy, zavedená pomocí deklarace `typedef` v hlavičkovém souboru `<stdint.h>`.

V hlavičkovém souboru `<stdint.h>` jsou mj. definována následující makra, poskytující minimální a maximální hodnotu daného datového typu:

- `INTN_MIN`, `INTN_MAX`, `UINTN_MAX` – např. `INT16_MAX`
- `INT_LEASTN_MIN`, `INT_LEASTN_MAX`, `UINT_LEASTN_MAX` – např. `INT_LEAST16_MIN`
- `INT_FASTN_MIN`, `INT_FASTN_MAX`, `UINT_FASTN_MAX` – např. `INT_FAST16_MIN`
- `INTPTR_MIN`, `INTPTR_MAX`, `UINTPTR_MAX`
- `INTMAX_MIN`, `INTMAX_MAX`, `UINTMAX_MAX`.

V prostředí Visual C++ 2003 hlavičkový soubor `<stdint.h>` neexistuje a tudíž neexistují ani typy v něm deklarované.

V prostředí Visual C++ 2003 existují dále celočíselné znaménkové typy s přesně určenou šířkou:

`__int8`, `__int16`, `__int32`, `__int64`

Mohou se používat i ve spojení s modifikátorem `unsigned`. Jedná se o klíčová slova.

Celočíselný konstantní výraz

Celočíselný konstantní výraz je výraz, který dokáže překladač vyhodnotit již v době překladu. Ve výrazu se může jako operand použít:

- literál,
- manifestační konstanta (makro),
- konstanta deklarovaná pomocí modifikátoru `const` inicializovaná konstantním výrazem,
- výčtová konstanta,
- operátor `sizeof`.

Pole

Meze polí lze v deklaraci zadat libovolným celočíselným konstantním výrazem.

Např.

```
const int n = 10;
enum { m = 20 };
#define p 30
int a[n], b[m], c[m+n], d[p+m]; // v jazyce C nelze
double e[p]; // lze v C i C++
```

V jazyce C lze použít pouze manifestační konstantu (makro) nebo literál.

Ukazatele

Ukazatel bez doménového typu, tj. `void*` nelze v C++ přiřadit ukazateli s doménovým typem, např.:

```
void* v;
int a, *b;
v = &a; // OK v C i C++
b = v;  // v C lze, ale v C++ nikoli
```

V C++ se musí přetypovat:

```
b = (int*)v; // OK i v C++
```

Ukazatel nikam

Pro ukazatel, který neukazuje na žádnou platnou adresu, se v C++ používá `0` (nula). Nulu lze přiřadit libovolnému ukazateli a libovolný ukazatel lze s nulou porovnávat.

V jazyce C se pro „ukazatel nikam“ používá manifestační konstanta (makro) `NULL`, která obvykle stejně znamená `0` nebo `((void *)0)`. V C++ ji lze většinou použít také, ale existují situace, kdy může `NULL` vzhledem k přísnější typové kontrole v C++ způsobit problémy, a proto se doporučuje používat raději `0`. V prostředí Visual C++ 2003 je konstanta NULL definována např. v hlavičkovém souboru `<stddef.h>` takto:

```
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#endif
```

Z uvedené definice vyplývá, že ve zdrojových souborech jazyka C++ je konstanta NULL synonymem pro `0`.

Ukazatele lze použít všude tam, kde se očekává logická hodnota. Hodnota `0` se automaticky konvertuje na `false`, jiná hodnota na `true`.

Standardní vstupy a výstupy

V C++ lze pro výstup do standardního výstupního proudu `stdout` (implicitně obrazovka) použít konstrukci

```
cout << výraz << výraz ... ;
```

např.:

```
cout << "Obsah obdélníku = " << a*b << '\n';
```

Pro čtení dat ze standardního vstupního proudu `stdin` (implicitně klávesnice) lze použít konstrukci:

```
cin >> proměnná >> proměnná ...;
```

např.

```
int a, b;
cin >> a >> b;
```

Před použitím uvedených konstrukcí se musí do zdrojového souboru zahrnout hlavičkový soubor `<iostream>` a uvést příkaz

```
using namespace std;
```

Podrobné informace o objektových datových proudech – viz přednášky Jazyk C++ II.

Reference

Reference je proměnná odkazující se na jinou proměnnou. V jazyce C neexistují.

Reference se musí v deklaraci inicializovat. Inicializátorem musí být l-hodnota stejného typu, jako je deklarovaná reference. Např.:

```
int i;
int& ri = i;
```

Cokoli se od této chvíle provede s proměnnou `ri`, bude mít stejný výsledek, jako kdyby se provedlo totéž s proměnnou `i`, to znamená, že

```
ri = 12;
```

je naprosto totéž jako

```
i = 12;
```

Proměnná `i` se nazývá v tomto případě *odkazovaná proměnná.*

Je-li `ui` ukazatel na `int`, jsou tyto příkazy ekvivalentní:

```
ui = &ri;
ui = &i;
```

Nelze deklarovat ukazatele na referenci, referenci na referenci a pole referencí. Nelze také deklarovat proměnnou typu `void&`.

Lze ale deklarovat referenci na ukazatel, referenci na pole nebo referenci na funkci.

Např.

```
int* ui = &i;
int*& rui = ui; // reference na ukazatel – OK
```

```
int&* uri; // ukazatel na referenci - chyba
*ui = 10;
*rui = 10; // dtto: *ui = 10
```

Reference na funkci:

```
void f();
void (&rf)() = f; // reference na funkci void f()
rf(); // volání funkce f()
```

Reference se nejčastěji používají pro předávání parametrů funkce odkazem – viz kapitola o funkcích.

Funkce může referenci vracet – viz kapitola o funkcích.

Konstantní reference

Použije-li se v deklaraci reference modifikátor `const`, jako např. v zápisu:

```
int j;
const int& rj = j;
```

vznikne *konstantní reference* (reference na konstantu).

Konstantní referenci nelze po inicializaci přidělit jinou hodnotu:

```
rj = i; // Chyba
ri = j; // OK
```

Jako inicializátor konstantní reference lze použít libovolný výraz, který lze přiřadit deklarovanému typu reference. Nejedná-li se však o l-hodnotu deklarovaného typu reference, vytvoří překladač pomocnou proměnnou, do které uloží hodnotu inicializátoru a reference bude odkazovat na tuto pomocnou proměnnou:

```
int a = 10;
const int& b = 2*a; // b odkazuje na pomocnou proměnnou s hodnotou 20
```

Implicitní int

V jazyce C lze vynechat v deklaraci proměnných a funkcí specifikaci typu. Překladač si v tom případě doplní `int`. Např. lze zapsat:

```
const x = 10;
extern y;
f(const x, register y)
{
   int vysledek;
   // ...
   return vysledek;
}
```

Pravidlo implicitního typu `int` norma C++ a nová norma jazyka C nezná. Avšak překladač Visual C++ 2003 toto pravidlo povoluje.

Funkce

Funkci, která nemá parametry, lze zapsat v C++ dvěma způsoby, např.:

```
int f(void);
int f();
```

V jazyce C je přípustná pouze první varianta. Druhá varianta znamená funkci, u níž není známý počet a typ parametrů.

Informativní deklarace funkce = *prototyp* funkce.

V C++ musí být před voláním funkce deklarován alespoň její prototyp.

V jazyce C může volání funkce předcházet její deklaraci.

Parametry funkce

Parametry funkce mohou být předávány:

- hodnotou – lze použít v C i C++, např.:

```
int f(double a);
void swap(int* a, int* b);
```

- odkazem – existuje pouze v C++.

Parametry volané odkazem mají formální parametr typu reference.

Je-li formální parametr typu `T&`, musí být skutečným parametrem l-hodnota typu `T`, ne však `const T`.

Je-li formální parametr typu `const T&` (konstantní reference), může být skutečným parametrem libovolný výraz, který lze použít k inicializaci proměnné typu `T`, tj. nemusí to být l-hodnota. Nejedná-li se však o l-hodnotu typu `T`, vytvoří překladač pomocnou proměnnou typu `T`, do které uloží hodnotu skutečného parametru. Formální parametr potom bude odkazovat na tuto pomocnou proměnnou.

Příklad

Definiční deklarace funkce `Vymena`:

```
void Vymena(int& a, int& b)
{
   int p = a; a = b; b = p;
}
```

Funkce `Vymena` má dva parametry volané odkazem a provede výměnu hodnot těchto dvou parametrů. Volání funkce může být následující:

```
int x = 10, y = 20;
Vymena(x, y);
```

Implicitní hodnoty parametrů

V C++ se mohou definovat implicitní hodnoty parametrů funkce. Implicitní hodnoty se předepisují v první deklaraci funkce v daném oboru viditelnosti (prototyp nebo definiční deklarace, zpravidla jsou předepsány v prototypech funkcí v hlavičkových souborech).

Deklarace implicitní hodnoty se podobá deklaraci s inicializací, např.:

```
void f(int a, int b = 0, int c = getch());
```

Místo konstanty lze použít i výraz.

Předepíše-li se implicitní hodnota pro některý z parametrů, musí se předepsat implicitní hodnoty i pro všechny parametry, které za ním následují.

Příklady použití:

```
x = f(3, 5, 7); // implcitní hodnoty se nepoužijí
x = f(3, 5); // pro c se použije implicitní hodnota
x = f(3); // pro b a c se použije implicitní hodnota
```

Pokud se některý skutečný parametr vynechá, musí se vynechat i všechny následující.

Referenční funkce

Funkce, která vrací referenci, se nazývá *referenční funkce*.

Výraz v příkazu `return` musí představovat l-hodnotu vráceného typu, která bude existovat i po návratu z funkce. Může to být např. globální proměnná, lokální statická proměnná nebo formální parametr předaný odkazem.

Zápis referenční funkce může stát i na levé straně přiřazovacího příkazu, tj. vytváří l-hodnotu.

Příklad

```
const int N = 10;
int Pole[N] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
int& PrvekPole(int index)
{
   if (index < 0 || index >= N) chyba();
   return Pole[index];
}
PrvekPole(4) = 20;
PrvekPole(5) = PrvekPole(4)*10;
```

NEOBJEKTOVÉ ROZDÍLY MEZI C A C++ – POKRAČOVÁNÍ

Funkce – pokračování

Vložené funkce (inline functions)

Je-li tělo funkce malé, např.:

```
int DruhaMocnina(int x) { return x*x; }
```

volání takovéto funkce není příliš efektivní, neboť počítač spotřebuje více času na předávání parametrů, skok do funkce a návrat z ní, než na vykonání příkazů těla funkce.

V C++ lze použít vloženou funkci, která se deklaruje pomocí specifikátoru `inline`:

```
inline int DruhaMocnina(int x) { return x*x; }
```

Překladač vloží namísto např. výrazu

```
DruhaMocnina(a + 1)
```

vlastní tělo funkce

```
(a + 1)*(a + 1)
```

Specifikátor `inline` (podobně jako `register`) není pro překladač závazný. Překladač může vloženou funkci přeložit i jako „obyčejnou“. Jedná se např. o tyto případy:

- ve funkci jsou použity některé příkazy, které by způsobovaly problémy,
- v programu je použit ukazatel na tuto funkci,
- funkce je rekurzivní.

Vypnutí/zapnutí překladu vložených funkcí jako vložených v prostředí Visual C++ 2003: menu **Project | Properties**, část **C/C++, Optimization**, položka **Inline Function Expansion**. Položka může obsahovat volby:

- **Default** (implicitní stav) – způsob překladu vložených funkcí závisí na tom, zda program se překládá s ladícími informacemi nebo bez nich. Pokud se program překládá s ladícími informacemi, je použití vložených funkcí vypnuto, aby bylo možné krokovat do těla vložených funkcí. Pokud se program překládá bez ladících informací, je použití vložených funkcí zapnuto.
- **Any Suitable –** použití vložených funkcí je zapnuto.
- **Only __inline** – použijí se pouze vložené funkce mající specifikátor `__inline`.

Přetěžování funkcí (overloading functions)

V C++ lze v jednom programu definovat několik funkcí se stejným identifikátorem, pokud se liší počtem nebo typem parametrů. Lze tedy vedle sebe deklarovat např. funkce

```
void f();                  // #1
int f(int);                // #2
double f(double);          // #3
void f(int, double);       // #4
```

Při volání vybere překladač funkci, jejíž formální parametry nejlépe odpovídají skutečným parametrům v zápisu funkce. Např.:

```
f(); // volá se funkce #1
f(10); // volá se funkce #2
f(3, 3.14); // volá se funkce #4
```

Pokud typy skutečných parametrů neodpovídají přesně typům formálních parametrů žádné z funkcí, vybere překladač tu, pro kterou je konverze nejsnazší, výsledek rozhodování však musí být jednoznačný, jinak překladač oznámí chybu.

Např.:

```
f('a');  // volá se funkce #2
f(1, 3); // volá se funkce #4
```

Následující příkaz je chybný

```
f(0L);
```

protože převod z `long` na `double` nebo `int` je stejně obtížný.

K rozlišení přetížených funkcí nestačí rozdíl:

- v typu vrácené hodnoty,
- mezi parametrem předávaným hodnotou a parametrem předávaným odkazem (referencí),
- mezi typy `T` a `const T`.

Stačí ale rozdíl mezi:

- typy `T*` a `const T*`
- typy `T&` a `const T&`.

Výčtové typy

Syntaktický popis:

deklarace_výčtového_typu:

enum *jmenovka*$_{nep}$ **{** *seznam_enumerátorů* **}** *seznam_deklarátorů*$_{nep}$**;**

seznam_enumerátorů:

enumerátor

*enumerátor***,** *seznam_enumerátorů*

enumerátor:

identifikátor

identifikátor **=** *konstantní_výraz*

Konstantní_výraz je celočíselný konstantní výraz (viz 1. přednáška). Ve výrazu se může použít i výčtová konstanta stejného nebo jiného výčtového typu.

Např.

```
enum barvy { cerna = 22, bila = -33, modra = 2*cerna, zelena };
```

V jazyce C se na výčtový typ musí odvolávat „úplným zápisem“, tj. konstrukcí:

```
enum barvy b = cerna;
```

V C++ stačí jmenovka, tj.

```
barvy b = cerna;
```

Aby se v jazyce C mohlo klíčové slovo `enum` vynechat, musí se deklarovat výčtový typ pomocí klíčového slova `typedef`:

```
typedef enum barvy { cerna = 22, bila = -33, modra = 2*cerna, zelena
} tbarvy;
```

Na tento typ se lze v C odvolávat jak konstrukcí `enum barvy`, tak i konstrukcí `tbarvy`:

```
tbarvy b = cerna;
```

V C++ lze také použít „úplný zápis“ nebo deklaraci `typedef enum`, ale je to zbytečné.

V jazyce C lze do proměnné výčtového typu přiřadit libovolnou celočíselnou hodnotu. V jazyce C++ se musí celočíselná hodnota přetypovat na příslušný výčtový typ:

```
b = 22; // OK v jazyce C, chyba v jazyce C++
b = (barvy)22; // OK v jazyce C i C++
```

Rozsah výčtového typu

Proměnná výčtového typu může obsahovat nejen hodnoty všech jeho výčtových konstant, ale všechny hodnoty nejmenšího bitového pole, do kterého lze uložit hodnoty všech výčtových konstant. Rozsah je tedy následující:

- jsou-li všechny výčtové konstanty nezáporné:

  0 až $b_{max} - 1$

  kde b_{max} je nejmenší číslo tvaru 2^N větší než největší z výčtových konstant.
- je-li některá výčtová konstanta záporná:

  $-b_{max}$ až $b_{max} - 1$

  kde b_{max} je nejmenší číslo tvaru 2^N větší než maximum z absolutních hodnot výčtových konstant.

V dříve uvedeném příkladu je největší konstanta `zelena = 45` a nejmenší `bila = -33`. Rozsah je tedy –64 (2^6) až 63.

Podle uvedeného rozsahu se zároveň určuje velikost výčtového typu – jedná se o velikost nejmenšího celočíselného typu, do kterého se vejde rozsah výčtového typu. Velikost tohoto celočíselného typu vrací operátor `sizeof`.

Výčtový typ `barvy` je uložen do celočíselného typu `signed char` (rozsah –128 až 127). Výraz `sizeof(barvy)` má tedy hodnotu 1.

V jednotlivých vývojových prostředích je rozsah a velikost výčtového typu implementována různým způsobem.

V prostředí Visual C++ 2003 je velikost výčtového typu vždy rovna velikosti výčtového typu `int`.

Struktury a unie

Struktury a unie patří v C++ mezi objektové typy, v této kapitole jsou zatím popsány jako neobjektové typy.

Struktury

Syntaxe:

deklarace_struktury:

struct *jmenovka*$_{nep}$ **{** *deklarace_složek* **}** *seznam_deklarátorů*$_{nep}$**;**

Např.:

```
struct Osoba {
   char Jmeno[30];
   int Cislo;
};
```

V jazyce C se na strukturu musí odvolávat „úplným zápisem“, tj. konstrukcí

```
struct Osoba osoba;
```

V C++ stačí jmenovka, tj.

```
Osoba osoba;
```

Aby se v jazyce C mohlo klíčové slovo `struct` vynechat, musí se deklarovat nový typ pomocí klíčového slova `typedef`:

```
typedef struct Osoba TOsoba;
```

Na tento typ se lze v C odvolávat jak konstrukcí `struct Osoba`, tak i konstrukcí `TOsoba`:

```
TOsoba osoba;
```

V C++ lze také použít „úplný zápis“ nebo deklaraci `typedef struct`, ale je to zbytečné.

Unie

V C++ lze deklarovat anonymní unie, v jazyku C nejsou k dispozici. Anonymní unie nemá ani jmenovku ani deklaraci proměnných. Ke složce anonymní unie se přistupuje pomocí samotného identifikátoru složky.

Globální anonymní unie musí být statické.

Příklad

```
void main()
{
   union {
      int i;
      unsigned char c[4];
   };
   i = 0x11223344;
   cout << i << '\n';
   for (int j = 0; j < 4; ++j)
      cout << (usnigned)c[j] << ' ';
   getch();
}
```

Pole `c` bude obsahovat tyto hodnoty:

```
c[0] = 0x44
c[1] = 0x33
c[2] = 0x22
c[3] = 0x11
```

Zarovnání dat v paměti

Složky struktur a tříd (třídy – viz některá z dalších přednášek) a celá struktura nebo třída (dále jen struktura) jsou v paměti zarovnány dle nastavení překladače.

Složka struktury začíná ve struktuře na x-tém bytu, přičemž x je definováno vztahem

$$x = y + z$$

kde:

y ..... byte, na kterém končí předchozí složka struktury,

z ..... nejmenší číslo takové, aby platilo:

$$x \bmod s = 0$$

$$s = \min\{st, da\}$$

kde:

mod ..... zbytek po celočíselném dělení;

st ..... velikost datového typu složky; je-li složkou jiná struktura, tak velikost největšího datového typu složky této vnořené struktury;

da ..... zarovnání dat dle nastavení překladače.

Vzorec pro výpočet *x* lze také zapsat následovně:

$$x = \text{int}(\frac{y}{s}) \cdot s + w$$
$$w = 0 \quad (y \bmod s = 0)$$
$$= s \quad (y \bmod s > 0)$$

kde:

int(*x*)..... celá část z čísla *x*.

Celá struktura se zarovná na *a* bytů, přičemž *a* je definováno vztahem

$$a = b + c$$

kde:

b..... byte, na kterém končí poslední složka struktury,

c..... nejmenší číslo takové, aby platilo:

$$a \bmod \min\{\max\{st_1, st_2, \ldots st_n\}, da\} = 0$$

kde:

$st_1, st_2, \ldots st_n$ .....velikost datových typů jednotlivých složek struktury resp. složek vnořených struktur.

Zarovnání dat v prostředí Visual C++ 2003: menu **Project | Properties**, část **C/C++, Code Generation**, položka **Struct Member Alignment:**

- **Default** (implicitní stav) – zarovnání dat je na 8 bytů,
- **1**, **2**, **4**, **8** nebo **16** bytů – zarovnání na zvolený počet bytů.

Příklad

Je deklarována struktura:

```
struct Struktura {
   char     c;
   bool     b1;
   __int64  i1;
   bool     b2;
   __int16  i2;
   bool     b3;
};
```

V následující tabulce jsou uvedeny pozice, na kterých začínají složky struktury a celková velikost struktury pro jednotlivé typy zarovnání dat v paměti.

Složka	Byte, na kterém začíná složka pro *da*			
	1	2	4	8
b1	1	1	1	1
i1	2	2	4	8
b2	10	10	12	16
i2	11	12	14	18
b3	13	14	16	20
Velikost struktury	14	16	20	24

Bitová pole

Složky bitového pole mohou být v jazyce C pouze typu `int` nebo `unsigned`. V jazyce C++ mohou být jakéhokoliv celočíselného typu (včetně `bool`, `char`).

Paměťová reprezentace bitových polí může být (a také je) v různých implementacích jazyka C++ odlišná.

V prostředí C++ Builder je následující.

Složky bitového pole jsou rozděleny do skupin. Skupinu tvoří po sobě jdoucí složky stejného typu bez ohledu na znaménko. Každá skupina je zarovnána podle typu dané skupiny a aktuálního nastavení zarovnání dat v paměti. Uvnitř každé skupiny překladač vytvoří podskupiny složek, které se vejdou do velikosti typu dané skupiny. Celková velikost bitového pole je nakonec zarovnána podle aktuálního nastavení zarovnání dat v paměti.

Např.:

```
struct BitovePole {
   int           s1 : 8;
   unsigned      s2 : 16;
   unsigned long s3 : 8;
   long          s4 : 16;
   long          s5 : 16;
   char          s6 : 4;
};
```

První skupinu tvoří složky `s1` a `s2`. Zabírají celkem 24 bitů. Je-li zarovnání dat na 1 byte, za tuto skupinu se nevloží žádný prázdný bit, je-li zarovnání dat na 4 byty, za tuto skupinu se vloží 8 bitů.

Druhou skupinu tvoří složky `s3`, `s4` a `s5`, které zabírají celkem 40 bitů. Typ `long` ale zabírá 32 bitů, to znamená, že se musí skupina rozdělit na 2 podskupiny – 1. podskupina `s3` a `s4` (24 bitů), 2. podskupina `s5` (16 bitů). Je-li zarovnání dat na 1 byte, před druhou podskupinu se vloží 0 bitů, je-li zarovnání dat na 4 byty, před druhou podskupinu se vloží 8 bitů.

Třetí skupinu tvoří složka s6 typu `char`, který se vždy zarovnává na byte, tudíž se před tuto složku nevloží žádný prázdný bit. Za složku `s6` se vloží 4 prázdné bity, je-li zarovnání dat na 1 byte nebo 12 bitů, je-li zarovnání dat na 4 byty.

Celková velikost bitového pole bude 9 bytů při zarovnání dat na 1 byte a 12 bytů při zarovnání na 4 byty.

V prostředí Visual C++ 2003 není zarovnání složek bitového pole popsáno.

V obou prostředích se doporučuje, aby všechny složky byly stejného datového typu, jehož velikost je větší nebo rovna součtu velikostí jednotlivých složek.

Ukazatel na funkci

Ukazateli na funkci lze přiřadit hodnotu příkazem ve tvaru:

ukazatel = **&**$_{nep}$ *identifikátor_funkce*

Příklad

Je-li deklarován následující ukazatel na funkci:

```
double (*f) (double);
```

lze do něj přiřadit funkci `sin` dvěma způsoby:

```
f = sin;
f = &sin;
```

Rozlišovací operátor (angl. scope resolution operator)

Může být unární nebo binární. Syntaxe:

:: *identifikátor*

*obor***::***identifikátor*

Binární rozlišovací operátor vyznačuje, že *identifikátor* náleží do daného *oboru*. *Obor* je jméno objektového typu nebo jméno prostoru jmen. Bližší informace – viz některá z dalších přednášek.

Unární rozlišovací operátor umožňuje přístup k zastíněným globálním identifikátorům.

Příklad

```
int i = 11;

void main()
{
   int i = 0;
   cout << "lokalni i: " << i << '\n';
   cout << "globalni i: " << ::i << '\n';
}
```

Operátory přetypování

Kromě standardního operátoru *(typ)* existují čtyři nové operátory: `const_cast`, `dynamic_cast`, `static_cast` a `reinterpret_cast`.

Operátor (typ)

Syntaxe:

(*označení_typu***)** *cast_výraz*

jméno_typu **(***seznam_výrazů*$_{nep}$**)**

V jazyce C lze použít pouze první variantu. *Označení_typu* je identifikátor typu deklarovaný pomocí `typedef`, kombinace klíčových slov označujících základní typy (např. `unsigned short`) nebo složitější konstrukce (např. `unsgined short*`).

Jméno_typu ve druhé variantě je jednoslovné jméno základního typu, výčtového typu, struktury, unie, třídy nebo typu deklarovaného pomocí `typedef`. *Seznam_výrazů* může představovat jeden nebo více výrazů. Je-li uvedeno více výrazů, jedná se o volání konstruktoru objektového typu se jménem *jméno_typu* a s parametry danými *seznamem_výrazů* – viz některá z dalších přednášek.

Příklad

```
double d = 1.5;
int i = (int)d*10;  // do i se ulozi 10
int j = int(d*10);  // do j se ulozi 15
void* v = &i;       // OK
int* ui = (int*)v;  // OK
int* ui2 = int*(v); // Chyba: int* neni jmeno typu
```

Operátory const_cast, static_cast, dynamic_cast a reinterpret_cast

Tyto operátory si rozdělují úlohy standardního operátoru *(typ)* a nabízí další možnosti. V této kapitole je uveden jejich stručný popis, podrobný popis – viz některá z dalších přednášek.

Všechny operátory mají stejnou syntaxi, např. pro operátor `const_cast`:

`const_cast`**<***označení_typu***>(***výraz***)**

Označení_typu je cílový typ a *výraz* je výraz, jehož výsledný typ se má změnit.

Operátor `const_cast` slouží k přetypování *výrazu* přidáním nebo odebráním modifikátoru `const` nebo `volatile`..

Např.:

```
const int i = 10;
int* ui;
ui = &i;                      // Chyba
ui = const_cast<int*>(&i); // OK
```

Operátor `static_cast` slouží k běžným přetypováním, např. mezi celými a reálnými čísly, mezi různými typy ukazatelů (např. z `void*` na `int*`). Nekontroluje se, zda má požadovaná operace smysl.

Operátor `dynamic_cast` slouží k přetypování referencí a ukazatelů na objektové typy v rámci dědické hierarchie. Kontroluje se, zda má požadované přetypování smysl.

Operátor `reinterpret_cast` slouží k různým „nečistým“ přetypováním, např. přetypování ukazatele na celé číslo a naopak.

Operátor sizeof

Syntaxe:

sizeof *unární_výraz*

sizeof (*označení_typu***)**

V jazyce C lze použít pouze druhou variantu.

Unární_výraz má složitou definici – může se jednat např. o identifikátor proměnné, výraz s unárním operátorem, volání funkce apod. Libovolný výraz lze převést na unární výraz uzavřením výrazu do kulatých závorek.

Výsledkem operátoru je konstanta typu `size_t`. Typ `size_t` je deklarovaný pomocí `typedef` v hlavičkovém souboru `stddef.h` jako `unsigned int` (může být deklarován i jako typ `unsigned long`). Použije-li se operátor `sizeof` na výraz, tento výraz se nevyhodnotí, pouze se určí jeho typ a z něho se určí velikost. Použije-li se operátor `sizeof` na referenci, výsledkem je velikost odkazovaného objektu.

Příklady

```
__int16   i = 10;
__int16&  ri = i;
size_t    n;
n = sizeof i;         // n = 2
n = sizeof ri;        // n = 2
n = sizeof &i;        // n = 4
n = sizeof ++i;       // n = 2, hodnota i se nezmění
n = sizeof sin(0.5); // n = 8, tj. sizeof(double)
                      // funkce sin se nevolá
n = sizeof(i*10);     // n = 4, tj. sizeof(int)
           // výraz i*10 není unární – musí být v závorkách
n = sizeof(bool);     // n = 1
n = sizeof bool;      // Chyba
```

Dynamické proměnné

V jazyce C se používají pro alokaci a uvolnění paměti funkce `malloc`, `calloc`, `realloc` a `free`. Ty lze použít i v C++, ale zpravidla se místo nich používají operátory `new` a `delete`.

Operátor new

Slouží pro alokaci proměnných.

Zjednodušená syntaxe:

new *žádná_výjimka*$_{nep}$ *označení_typu new_rozsah*$_{nep}$ *inicializátor*$_{nep}$

new *umístění*$_{nep}$ *označení_typu new_rozsah*$_{nep}$ *inicializátor*$_{nep}$

žádná_výjimka:

(nothrow)

new_rozsah:

[*výraz* **]**

new_rozsah **[** *konstantní_výraz* **]**

kde:

umístění ..... adresa, od které se má alokace provést v závorkách,

označení_typu ..... označení typu proměnné, která se má alokovat (v případě nejednoznačnosti se musí uvést v kulatých závorkách),

new_rozsah ..... počet prvků alokovaného pole – viz dále,

inicializátor ..... počáteční hodnota proměnné v kulatých závorkách. Není-li uvedeno, proměnná má nedefinovanou hodnotu.

Výsledkem operátoru `new` je ukazatel na *označení_typu*.

Alokace jednoduché proměnné

Bez inicializace:

```
int* a = new int;
```

Výraz `*a` má nedefinovanou hodnotu.

S inicializací:

```
int* b = new int(10);
```

Výraz `*b` má hodnotu 10.

Alokace jednorozměrných polí

Příkaz

```
int* c = new int[10];
```

alokuje pole 10 prvků typu `int`, vrací ukazatel na první prvek pole.

Inicializaci pole nelze provést pomocí operátoru `new`. Musí se provést po alokaci, např.:

```
for (int i = 0; i < 10; ++i) c[i] = 0;
```

Mez jednorozměrného pole v operátoru `new` může být nekonstantní celočíselný výraz.

Alokace vícerozměrných polí

Příklad:

```
typedef int tradek[10]; // deklarace typu řádku matice
tradek* d = new int[5][10]; // alokace matice
for (int i = 0; i < 5; ++i) for (int j = 0; j < 10; ++j)
   d[i][j] = 0;
```

V uvedeném příkladu se nejprve deklaruje typ `tradek`, potom se alokuje matice o 5 řádcích a 10 sloupcích a nakonec se inicializují prvky na nulu.

Jiný způsob zápisu alokace v uvedeném příkladu:

```
tradek* d = new tradek[5];
```

Bez použití deklarace typu `tradek` by alokace matice vypadala takto:

```
int (*d)[10] = new int[5][10];
```

První mez vícerozměrného pole může být nekonstantní celočíselný výraz, ostatní meze musí být konstantní celočíselné výrazy.

Pokud se alokace nepodaří

Operátor `new` v případě neúspěchu:

- ve starších překladačích vracel hodnotu 0,
- v novějších překladačích včetně Visual C++ 2003 a C++ Builder 6 vyvolá výjimku typu `bad_alloc` (výjimky – viz některá z dalších přednášek) – toto chování předepisuje norma C++.

Uvedené chování lze změnit pomocí funkce `set_new_handler`, deklarované v hlavičkovém souboru <`new`>:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler my_handler);
```

Parametr `my_handler` je ukazatel na funkci, která provede určité činnosti, pokud se alokace pomocí operátoru `new` nepodaří. Tato funkce (handler) by měla ukončit program, vyvolat výjimku nebo se pokusit uvolit potřebnou paměť. Pokud tato funkce neukončí program nebo nevyvolá výjimku, pokusí se operátor `new` po návratu z ní alokovat paměť znovu. Funkce `set_new_handler` vrací předchozí handler. Pro zpřístupnění identifikátorů deklarovaných v hlavičkovém souboru <`new`> se musí uvést např. příkaz

```
using namespace std;
```

Příklad:

```
#include <iostream>
#include <conio.h>
#include <new> // nemusí být uveden, pokud je zahrnut soubor <iostream>
using namespace std;

void Chyba()
{
   cout << "Nedostatek pameti. Program bude ukoncen.";
   getch();
   exit(1); // program se okamžitě ukončí
}

int main()
{
   set_new_handler(Chyba);
   double* e = new double[0xFFFFFFF];
   // ...
   return 0;
}
```

Je-li parametr `my_handler` roven nule, pro operátor `new` se použije implicitní chování (v novějších překladačích tedy vyvolá výjimku).

Pokud má operátor `new` v případě neúspěchu vracet nulu, lze použít parametr `(nothrow)`:

```
int* f = new (nothrow) int[0xFFFFFFF];
if (!f) Chyba();
```

Pokud je však nastaven nenulový handler pomocí funkce `set_new_handler`, operátor `new` s parametrem `(nothrow)` v případě neúspěchu zavolá nastavený handler a nulu nevrací.

Parametr umístění

Uvedením adresy v parametru umístění se předepisuje, aby se alokace provedla od této adresy, např.:

```
char* adresa;
// ...
int* g = new (adresa) int[10];
```

Operátor delete

Slouží pro uvolnění paměti alokovaných proměnných.

Zjednodušená syntaxe:

delete *ukazatel*

delete [] *ukazatel*

Ukazatel je výraz, jehož hodnotou je adresa vrácená operátorem `new`.

Operátor `delete` uvolní paměť, na kterou ukazuje *ukazatel*, ale hodnotu ukazatele nezmění. Pokud *ukazatel* obsahuje hodnotu `0`, nejedná se o chybu. Operátor `delete` v tom případě neudělá nic.

První možnost slouží pro uvolnění jednoduché proměnné, např. pro dříve alokovanou proměnnou `a`:

```
delete a;
```

Druhá možnost se používá pro uvolňování polí, např. pro dříve alokovaná pole `c` a `d`:

```
delete [] c;
delete [] d;
```

Pro proměnnou, která byla alokována pomocí operátoru `new` s parametrem umístění, se nevolá operátor `delete`.

Přetěžování operátorů new a delete

Viz některá z dalších přednášek o přetěžování operátorů.

OBJEKTOVÉ TYPY

Třída

Jedním ze základních pojmů objektově orientovaného programování (OOP) je *třída* (angl. *class*). Jako *třídy* ve smyslu OOP lze v C++ použít struktury, unie a třídy, tj. objektové datové typy deklarované pomocí klíčových slov `struct`, `union` a `class`.

Pojem *třída* ve smyslu OOP bude dále uveden pod pojmem *třída* bez dalšího upřesnění nebo místo něho bude použit pojem *objektový typ*. Pojem *třída* ve smyslu typu deklarovaného pomocí klíčového slova `class` bude v následujícím textu uveden s upřesněním, např. *třída deklarovaná pomocí class*.

Struktury (deklarované pomocí `struct`) a třídy (deklarované pomocí `class`) jsou v podstatě ekvivalentní, rozdíly mezi nimi jsou v implicitních přístupových právech ke složkám a při dědění. Unie se liší více – odlišnosti jsou popsány zvlášť. Syntaxi deklarace všech těchto typů lze zapsat společně:

Deklarace_objektového_typu:

*klíč jmenovka***;**

klíč jmenovka$_{nep}$ **{** *tělo_třídy* **}** *seznam_deklarátorů*$_{nep}$**;**

*klíč jmenovka***:** *seznam_předků* **{** *tělo_třídy* **}** *seznam_deklarátorů*$_{nep}$**;**

klíč:

class

struct

union

tělo_třídy:

specifikace_přístupu$_{nep}$ *deklarace_atributu*$_{nep}$**;** *tělo_třídy*$_{nep}$

specifikace_přístupu$_{nep}$ *deklarace_metody*$_{nep}$**;** *tělo_třídy*$_{nep}$

specifikace_přístupu$_{nep}$ *deklarace_typu*$_{nep}$**;** *tělo_třídy*$_{nep}$

specifikace_přístupu$_{nep}$ *deklarace_šablony*$_{nep}$**;** *tělo_třídy*$_{nep}$

specifikace_přístupu$_{nep}$ *deklarace_přátel*$_{nep}$**;** *tělo_třídy*$_{nep}$

specifikace_přístupu:

public:

protected:

private:

Jmenovka je identifikátor nově deklarovaného typu. *Seznam_deklarátorů* jsou deklarátory proměnných tohoto typu, polí, ukazatelů na tento typ apod. *Seznam_předků* určuje třídy, od kterých je právě deklarovaná třída odvozena. Dědění viz některá z dalších přednášek.

Proměnné, konstanty a parametry objektových typů (ve funkcích) se nazývají *instance*. Pojem *objekt* zpravidla označuje manipulovatelnou oblast paměti, tedy proměnnou jakéhokoliv typu (základního typu, objektového typu, pole hodnot, ukazatele na funkci apod.).

Tělo_třídy obsahuje seznam deklarací *složek* (*členů*) třídy (angl. *class members*), kterými mohou být:

- atributy = datové složky (angl. *data members*)
- metody = funkční složky = členské funkce (angl. *member functions*)

- typy – tzv. vnořené typy (angl. *nested types*)
- šablony – viz některá z dalších přednášek.

První způsob deklarace třídy představuje informativní deklaraci. Ta se může vyskytnout i vícekrát. Další dva způsoby deklarace představují definiční deklaraci.

Z uvedeného plyne, že definiční deklarace objektového typu bez předků začíná klíčovým slovem `class`, `struct` nebo `union`, za kterým následuje jméno typu (jmenovka). Potom ve složených závorkách je uvedena posloupnost deklarací členů, které mohou být deklarovány v libovolném pořadí. Před každou deklarací může předcházet specifikace přístupových práv.

Deklarace objektového typu může být:

- globální,
- lokální v deklaraci jiné třídy – tzv. *vnořená třída* (angl. *nested class*),
- lokální ve funkci – tzv. *lokální třída* (angl. *local class*).

Pokud se v deklaraci vynechá *jmenovka*, jedná se o *nepojmenovanou třídu* (angl. *unnamed class*). V deklaraci nepojmenované třídy se nesmí vynechat *seznam_deklarátorů*, výjimkou je anonymní unie nebo deklarace vnořené třídy.

Pro přístup ke složkám třídy neuvedených v metodách této třídy se používá operátor:

- kvalifikace (tečka) pro přístup pomocí instance třídy nebo reference na třídu,
- nepřímé kvalifikace (`->`) pro přístup pomocí ukazatele na třídu.

Tento způsob přístupu je stejný jako přístup ke složkám struktury v jazyku C.

Atributy

Deklarace atributů je analogická deklaraci obyčejných proměnných.

V atributu lze specifikovat paměťovou třídu `static` (tzv. statické atributy) nebo `mutable` (tzv. měnitelné atributy) – podrobnosti viz dále. Nelze specifikovat paměťovou třídu `auto`, `register` nebo `extern`.

Nestatickou datovou složkou třídy `TA` nemůže být:

a) instance nebo pole instancí třídy `TA`,

b) instance jiné třídy, která obsahuje jako atribut instanci nebo pole instancí třídy `TA`,

c) instance jiné třídy, která je potomkem třídy `TA`.

Nestatickou datovou složkou ale může být ukazatel nebo reference na třídu `TA`.

Příklad

```
struct TA {
   TA   A;         // Chyba
   TA   &RA, *UA; // OK
   TB   B;         // Chyba, pokud TB:
                   //  - je potomkem TA
                   //  - má jako atribut instanci TA
   TB   &RB, *UB; // OK
};
```

Metody

Deklarace metody je buď prototyp anebo definiční deklarace. Pokud se uvede pouze prototyp, musí se metoda definovat později za deklarací třídy. Definuje-li se metoda mimo tělo třídy, musí se její jméno kvalifikovat jmenovkou třídy s použitím rozlišovacího operátoru, tj. např.

```
void TA::Vypocet() { ... }
```

kde `TA` je jméno třídy.

Metodou je též konstruktor, destruktor nebo přetížený operátor.

V deklaraci metody lze specifikovat paměťovou třídu `static` (tzv. statické metody) a uvést specifikátory `inline`, `explicit` nebo `virtual`. Specifikátor `explicit` lze uvést jen u konstruktoru.

Pokud má být metoda deklarována jako vložená, musí se provést jedna z těchto možností:

- v těle třídy se musí zapsat celá definice metody,
- u prototypu nebo definice metody se musí použít specifikátor `inline`.

Příklad

```
struct TA {
   int        x, y;

   int        DejSoucin { return x*y; }
   inline int DejSoucet();
   void       Vypis();
};

inline int TA::DejSoucet()
{
   return x+y;
}
void TA::Vypis()
{
   cout << x << ' ' << y;
}
```

Metody `DejSoucin` a `DejSoucet` jsou metody vložené, metoda `Vypis` vložená není.
V metodách třídy není nutné kvalifikovat datové a funkční složky této třídy ani jména vnořených typů a výčtových konstant této třídy.

Přístupová práva

K omezení přístupu různých částí programu ke složkám třídy slouží specifikátory `private`, `protected a public` a deklarace přátel.

Specifikace přístupu určuje přístupová práva pro všechny deklarace, které za ní následují, až do další specifikace přístupu.

Význam specifikátorů:

- `public` – *veřejně přístupné* (*veřejné*) *složky* – jsou přístupné pro kteroukoli část programu bez omezení.
- `private` – *soukromé složky* – jsou přístupné pouze pro metody této třídy a její přátele.
- `protected` – *chráněné složky* – jsou přístupné pouze pro metody této třídy a jejích potomků a pro její přátele.

Složky, jejichž deklaraci nepředchází žádná specifikace přístupu, jsou:

- soukromé ve třídách deklarovaných pomocí `class`,
- veřejné ve strukturách.

Příklad

```
class TB {
   int a;
   int b;
protected:
   int c;
   int d;
public:
   int Soucet() { return a + b; }
protected:
   void Init();
};
```

V třídě `TB` jsou atributy `a`, `b` soukromé, atributy `c`, `d` chráněné, metoda `Soucet` je veřejná a metoda `Init` chráněná.

Ukazatel this

Klíčové slovo `this` představuje konstantní ukazatel na danou třídu. V těle metody třídy `TA` je to tedy ukazatel typu `TA* const`.

Lze ho použít v nestatických metodách. Nestatické metody mají tento ukazatel jako skrytý parametr. Překladač v nestatické metodě zachází ve skutečnosti s atributy třídy pomocí ukazatele `this`.

Např. definuje-li se instance `A` typu `TA`

```
TA A;
```

při volání metody

```
A.Vypis();
```

bude v těle metody `Vypis` ukazatel `this` představovat ukazatel na instanci `A` a tělo této metody lze zapsat též takto:

```
void TA::Vypis()
{
   cout << this->x << ' ' << this->y;
}
```

Lokální proměnné a atributy

Identifikátor lokální proměnné definované v metodě včetně parametru této metody může být shodný s identifikátorem atributu třídy. V takovémto případě lokální proměnná metody zastiňuje atribut třídy. Pokud je potřebné v takovéto metodě přistoupit k atributu třídy, musí se atribut třídy kvalifikovat pomocí jmenovky třídy a rozlišovacího operátoru nebo pomocí ukazatele `this`.

Např. struktura `TA` obsahuje metodu `SetX`:

```
void SetX(int x) { TA::x = x; }
void SetX(int x) { this->x = x; }
```

V obou uvedených případech se do atributu `x` uloží hodnota parametru `x`.

Statické atributy

Statické atributy mají ve své deklaraci specifikovánu paměťovou třídu `static`. Statické atributy třídy `TA` jsou atributy společné pro všechny instance třídy `TA`. Statická datová složka třídy `TA` v programu existuje i v případě, že neexistuje žádná instance třídy `TA`.

Na statický atribut se lze dívat jako na globální proměnnou ukrytou v dané třídě. Na statický atribut se vztahuje specifikace přístupu (`private`, `protected`, `public`).

V metodách dané třídy se statické atributy nemusí kvalifikovat. Mimo metody dané třídy se musí kvalifikovat, a to jednou z těchto možností:

- jmenovkou třídy a rozlišovacím operátorem,
- identifikátorem instance třídy a operátorem kvalifikace (tečka) resp. operátorem nepřímé kvalifikace (->).

Deklarace statického atributu v rámci definice třídy ještě nevyhrazuje paměť pro tento atribut, má pouze informativní význam. Definiční deklarace statické datové složky třídy musí následovat někde za definicí třídy. V definiční deklaraci se musí identifikátor statického atributu kvalifikovat jmenovkou třídy a rozlišovacím operátorem bez specifikace `static` a může se v ní i inicializovat.

Statickým atributem nemůže být bitové pole. Statickým atributem třídy `TA` narozdíl od nestatických atributů může být:

a) instance nebo pole instancí třídy `TA`,

b) instance jiné třídy, která obsahuje jako atribut instanci nebo pole instancí třídy `TA`,

c) instance jiné třídy, která je potomkem třídy `TA`.

Statický atribut nemůže být současně měnitelným atributem (s paměťovou třídou `mutable`).

Lokální třídy nemohou mít statické atributy.

Je-li statický atribut konstantního celočíselného nebo výčtového typu (včetně typu `bool` a `char`, ne však typu `float` nebo `double`), může se přímo v těle třídy inicializovat konstantním celočíselným výrazem a v tom případě se tato deklarace stává definiční deklarací statického atributu a mimo tělo třídy se již atribut nedefinuje.

Příklad

Je definována třída `TIntArr` a instance `A` této třídy, určená pro práci s dynamickým polem celých čísel. Třída obsahuje statický atribut `PocAlokPrvku`, který udává počet alokovaných prvků všech instancí typu `TIntArr` a konstantní statický atribut `MaxPocPrvku`, který představuje maximální počet prvků pole, které lze alokovat.

```
class TIntArr {
   int        *a, n;
public:
   static int PocAlokPrvku;
   static const int MaxPocPrvku = 1000;

   void Alokuj(int _n);
   void Dealokuj() { PocAlokPrvku -= n; /* ... */ }
};

void TIntArr::Alokuj(int _n)
{
   if (_n <= MaxPocPrvku) {
      PocAlokPrvku += n;
      // ...
   }
}

int TIntArr::PocAlokPrvku = 0; // static se před int neuvádí
TIntArr A;
```

Počet alokovaných prvků lze vypsat jedním z těchto způsobů:

```
cout << A.PocAlokPrvku;
cout << TIntArr::PocAlokPrvku;
```

Inicializátor v definici statického atributu třídy `TA` patří do oblasti platnosti třídy `TA`. Pokud by v rámci dříve uvedené třídy `TIntArr` byl deklarován ještě statický atribut `PocAlokBytu`:

```
class TIntArr {
   ...
public:
   static int PocAlokBytu;
};
```

lze jej inicializovat mj. výrazem, obsahujícím již dříve inicializovaný atribut `PocAlokPrvku` bez kvalifikace:

```
int TIntArr::PocAlokPrvku = 0;
int TIntArr::PocAlokBytu = PocAlokPrvku * sizeof(int);
```

Statické metody

Statické metody mají ve své deklaraci v těle třídy specifikovánu paměťovou třídu `static`. V definici statické metody mimo deklaraci třídy se paměťová třída `static` neuvádí. Statické metody třídy `TA` lze volat, i když neexistuje žádná instance třídy `TA`. Statické metody třídy `TA` se nevztahují k instancím třídy `TA` a tudíž mohou přímo používat jen statické atributy a statické metody třídy `TA`. Není v nich k dispozici ukazatel `this`.

V metodách dané třídy se statické metody nemusí kvalifikovat. Mimo metody dané třídy se musí kvalifikovat, a to jednou z těchto možností:

- jmenovkou třídy a rozlišovacím operátorem,
- identifikátorem instance třídy a operátorem kvalifikace (tečka) resp. operátorem nepřímé kvalifikace (->).

Konstruktory a destruktory nemohou být statické. Statické metody nemohou být virtuální (viz některá z dalších přednášek).

Příklad

```
class TIntArr {
   int        *a, n;
   static int PocAlokPrvku;
pulic:
   void Alokuj(int _n) { PocPrvku += _n; /* ... */ }
   void Dealokuj() { PocPrvku -= n; /* ... */ }
   static int DejPocAlokPrvku() { return PocAlokPrvku; }
   static int DejPocAlokBytu();
};

int TIntArr::PocAlokPrvku = 0;

int TIntArr::DejPocAlokBytu() // static se před int neuvádí
{
   return PocAlokBytu * sizeof(int);
}

TIntArr A;
```

Počet alokovaných prvků lze vypsat jedním z těchto způsobů:

```
cout << A.DejPocAlokPrvku();
cout << TIntArr::DejPocAlokPrvku();
```

Metody konstantních a nestálých instancí

Instance tříd mohou být deklarovány s cv-modifikátory, tj. `const` (*konstantní instance*), `volatile` (*nestálé instance*) nebo `const volatile` (*konstantní nestálé instance*), např.:

```
const TA A;
```

je deklarace konstantní instance `A` třídy `TA`.

Konstantní instance dané třídy:

- nemohou měnit nestatické atributy této třídy kromě měnitelných atributů (mají paměťovou třídu `mutable`), tj. mohou měnit jen nekonstantní statické atributy a měnitelné atributy,
- mohou volat jen konstantní a statické metody této třídy.

Konstantní metody lze volat pro konstantní i nekonstantní instance.

Nestálé instance dané třídy mohou volat jen nestálé a statické metody této třídy. Nestálé metody lze volat pro instance deklarované s nebo bez modifikátoru `volatile`.

Pro konstantní nestálé instance platí kombinace uvedených omezení.

Konstantní metoda (angl. *const member function*) třídy se deklaruje s modifikátorem `const` za závorkou uzavírající seznam formálních parametrů, např.:

```
struct TA {
   // ...
   void Vypis() const;
};
void TA::Vypis() const
{ /* ... */ }
```

Obdobně se deklaruje nestálá nebo konstantní nestálá metoda, tj. místo modifikátoru `const` se uvede modifikátor `volatile` nebo `const volatile`.

Cv-modifikátor je součástí typu metody a musí se tedy uvádět v prototypu i v definiční deklaraci metody.

V těle konstantní metody je ukazatel `this` konstantní ukazatel na konstantu. V těle konstantní metody třídy `TA` je to tedy ukazatel typu `const TA* const`. Obdobně v těle nestálé (resp. konstantní nestálé) metody třídy `TA` je to ukazatel typu `volatile TA* const` (resp. `const volatile TA* const`).

V jedné třídě lze deklarovat dvě metody se stejným identifikátorem a stejnými parametry, které se liší jen tím, že jedna z nich je konstantní a druhá nekonstantní. Pro konstantní instanci se bude volat konstantní verze této metody a pro nekonstantní instanci se bude volat nekonstantní verze této metody.

Konstruktory, destruktory a statické metody nelze deklarovat jako konstantní nebo nestálé, ale lze je volat i pro konstantní nebo nestálé instance.

Příklad

Je deklarována struktura `TA` a instance `A` a `B` této struktury:

```
struct TA {
   int            a;
   mutable int    b;
   static int     x;

   static void    MocninaX() { x *= x; }
   int            NasobekA(int n) { return a *= n; }
   int            NasobekA(int n) const { return a*n; }
   void           Set(int _a, int _b) { a = _a, b = _b; }
   void           SoucetA(int n) const { a += n; } // Chyba
   void           SoucetB(int n) const { b += n; } // OK
};

const TA A = { 10, 20 }; // konstatní instance, a = 10, b = 20
TA       B;              // nekonstatní instance
```

Příklady použití instancí:

```
A.a = 100;      // Chyba
A.b = 30;       // OK
A.x = 40;       // OK - kontantní instance může měnit statické atributy
B.a = 1;        // OK
A.MocninaX();   // OK - konstantní instance může volat statické metody
A.Set(10, 20);  // Chyba
B.Set(1, 2);    // OK
A.NasobekA(10); // OK - volá se konstatní metoda
B.NasobekA(10); // OK - volá se nekonstatní metoda

void f(TA& c, const TA& d, const TA* e, int a)
{
   c.Set(a, a);  // OK
   d.Set(a, a);  // Chyba
   e->Set(a, a); // Chyba
}
```

Přátelé

Přítelem dané třídy může být funkce nebo třída, která není členem dané třídy, které je povoleno přistupovat ke všem složkám dané třídy. Tyto funkce a třídy se nazývají *spřátelené funkce* a *spřátelené třídy*.

Syntaxe:

Deklarace_přátel:

friend *prototyp_funkce*;

friend *definiční_deklarace_funkce*;

friend *klíč identifikátor_objektového_typu*;

V dané třídě se může deklarovat prototyp spřátelené funkce nebo celá její definice. Uvede-li se definice, bude s ní překladač zacházet jako s vloženou funkcí bez ohledu na to, zda se deklaruje s nebo bez specifikátoru `inline`. Pokud definice spřátelené funkce je uvedena mimo tělo třídy, spřátelená funkce je vložená tehdy, pokud je specifikátor `inline` uveden buď v jejím prototypu (za klíčovým slovem `friend` nebo mimo tělo třídy) nebo v její definici nebo na obou místech.

Spřátelenou funkcí dané třídy může být nejen „obyčejná“ funkce, ale i metoda jiné třídy. V tom případě se musí její identifikátor kvalifikovat pomocí jmenovky třídy a rozlišovacího operátoru.

Spřátelená třída je třída, jejíž všechny metody jsou spřátelené.

Před deklarací přátel může být uvedena specifikace přístupu, ta se však na deklaraci přátel nevztahuje.

Přátelství není tranzitivní. To znamená, že pokud třída `B` je přítelem třídy `A` a třída `C` je přítelem třídy `B`, nevyplývá z toho, že třída `C` je také přítelem třídy `A`. Přátelství není ani dědičné. To znamená, že pokud je třída `B` přítelem třídy `A` a třída `D` je potomkem třídy `B`, nevyplývá z toho, že třída `D` je také přítelem třídy `A`. Přátelství se automaticky nevztahuje ani na vnořené třídy. Např. je-li třída `B` přítelem třídy `A` a třída `E` je vnořenou třídou třídy `B`, nevyplývá z toho, že třída `E` je přítelem třídy `A`.

Příklad

```
class TKomplexCislo {
   double r, i;
public:
   void Set(double _r, double _i) { r = _r; i = _i; }

   friend double abs(const TKomplexCislo& t) // vložená funkce
    { return sqrt(t.r*t.r + t.i*t.i); }
   friend void FriendSet(TKomplexCislo& t, double r, double i);
   friend class TKomplexCisloArr;
    // v metodách třídy TKomplexCisloArr lze přistupovat
    // ke všem složkám třídy TKomplexCislo
   friend void TKomplexCisloList::SetAll(double r, double i);
    // metoda SetAll třídy TKomplexCisloList může přistupovat
    // ke všem složkám třídy TKomplexCislo
};

void FriendSet(TKomplexCislo& t, double r, double i)
{ t.r = r; t.i = i; }

TKomplexCislo kc;
kc.Set(10, 20);
double a = abs(kc);
FriendSet(kc, 1, 2);
double b = abs(kc);
```

OBJEKTOVÉ TYPY – POKRAČOVÁNÍ

Třída – pokračování

Vnořené typy

V těle třídy lze deklarovat vnořený typ, a to:

- pomocí `typedef`,
- výčtový typ,
- objektový typ.

Na vnořené typy se vztahují přístupová práva. Při použití vnořeného typu mimo obklopující třídu se musí jeho jméno kvalifikovat:

- jmenovkou obklopující třídy a rozlišovacím operátorem,
- případně identifikátorem instance obklopující třídy a operátorem kvalifikace nebo nepřímé kvalifikace.

Stejným způsobem se musí kvalifikovat i identifikátory výčtových konstant vnořených výčtových typů.

Příklad

Je deklarována třída `TOsoba`:

```
class TOsoba {
public:
   enum { MaxJmeno = 31 };
private:
   char Jmeno[MaxJmeno]; // MaxJmeno se nemusí kvalifikovat
...
};
```

Při použití výčtové konstanty `MaxJmeno` mimo třídu `TOsoba` je nutné `MaxJmeno` kvalifikovat:

```
TOsoba::MaxJmeno
```

Metody vnořené třídy nemají při přístupu ke složkám obklopující třídy žádné výjimky z pravidel o přístupových právech (tj. jako kdyby vnořená třída byla deklarována mimo obklopující třídu). Podobně ani obklopující třída nemá při přístupu ke složkám vnořené třídy žádné výjimky z pravidel o přístupových právech. Pokud by obklopující třída měla mít právo přístupu k chráněným nebo soukromým složkám vnořené třídy, musela by být deklarována jako přítel vnořené třídy.

Příklad

```
class TDList {
public:
   class TUzel { // deklarace vnořeného typu
      TData Data;
      TUzel *Dalsi, *Predch;
   public:
      void SetData(TData& _Data);
      // ...
   };
private:
   TUzel *Prvni, *Posledni;
```

```
public:
   // ...
};

void TDList::TUzel::SetData(TData& _Data)
{ Data = _Data; }
```

Vnořená třída se může v obklopující třídě deklarovat pouze klíčem (`class`, `struct`, `union`) a jmenovkou (informativní deklarace) a její definiční deklarace může být uvedena mimo obklopující třídu. V tom případě se musí jmenovka vnořené třídy v definiční deklaraci kvalifikovat *jmenovkou* obklopující třídy.

Příklad

```
class TDList {
public:
   class TUzel; // informativní deklarace vnořené třídy
private:
   TUzel *Prvni, *Posledni;
public:
   // ...
};

class TDList::TUzel { // definiční deklarace vnořené třídy
   TData Data;
   TUzel *Dalsi, *Predch;
public:
   void SetData(TData& _Data);
   // ...
};

void TDList::TUzel::SetData(TData& _Data)
{ Data = _Data; }
```

Inicializace bez konstruktoru

Instance objektového typu lze inicializovat stejným způsobem jako složky struktury nebo unie v jazyku C, tj. pomocí složených závorek, pokud tento objektový typ:

- nemá konstruktor,
- všechny jeho atributy jsou veřejné a nekonstantní,
- ani jeden z jeho nestatických atributů není referencí,
- nemá předka,
- nemá virtuální metodu.

Atributy objektových typů a pole hodnot deklarované v rámci třídy se inicializují pomocí vnitřních složených závorek.

Příklad

```
class TA {
public:
   int x[2];
   int y;
   struct TB {
      int i, j;
   } b;
} A = { { 1, 2 }, 3, { 4, 5 } };
// A.x[0] = 1, A.x[1] = 2, A.y = 3, A.b.i = 4, A.b.j = 5
```

Pokud by se deklarovala vnořená struktura `TB` bez atributu `b`, neinicializovaly by se atributy struktury `TB` při inicializaci instance `A`, protože by se nejednalo o atribut třídy `TA`.

Může být inicializováno i méně nestatických atributů třídy. V tom případě se zbývající nestatické atributy inicializují nulovou hodnotou, např.:

```
TA A = { { 1, 2 }, 3 };
// A.x[0] = 1, A.x[1] = 2, A.y = 3, A.b.i = 0, A.b.j = 0
```

Uvedou-li se při inicializaci jen prázdné složené závorky, všechny nestatické atributy se inicializují nulovou hodnotou:

```
TA A = { };
// A.x[0] = 0, A.x[1] = 0, A.y = 0, A.b.i = 0, A.b.j = 0
```

Statické atributy nejsou inicializovány při inicializaci ostatních nestatických atributů:

```
class TC {
   int x;
   static int s;
   int y;
};
TC C = { 1, 2 }; // A.x = 1, A.y = 2
```

Konstruktory

Konstruktory se volají při vytváření instance, a to v rámci její definiční deklarace nebo alokace pomocí operátoru `new`, při konverzích, při předávání parametrů objektových typů hodnotou apod.

Konstruktory jsou metody třídy s těmito specifiky:

- Jméno konstruktoru je tvořeno identifikátorem (jmenovkou) třídy.
- Deklarace konstruktoru nesmí obsahovat specifikaci typu vrácené hodnoty, a to ani `void`.
- Nedědí se. Konstruktor odvozené třídy používá ke své konstrukci konstruktory předků.
- Nesmí být virtuální, statické, konstantní nebo nestálé.
- Nelze získat jejich adresu.
- Mohou být volány pro instance deklarované s cv-modifikátory (`const`, `volatile`).

Instanci např. `A` třídy `TA` lze definovat ve funkci `f()`, pokud je splněna alespoň jedna z následujících podmínek:

- konstruktor třídy `TA`, volaný definicí instance `A`, je veřejný,
- funkce `f()` je ve třídě `TA` deklarována jako přítel.

Podobná omezení platí i pro ostatní akce, které vedou k volání konstruktoru.

Deklarace konstruktoru

Syntaxe:

deklarace_konstruktoru:

modifikátory$_{nep}$ *jméno_třídy*$_{nep}$**::**$_{nep}$*jmenovka* **(***spec_form_par*$_{nep}$**)**;

modifikátory$_{nep}$ *jméno_třídy*$_{nep}$**::**$_{nep}$*jmenovka* **(***spec_form_par*$_{nep}$**)** *inicializační_část*$_{nep}$ *tělo*

modifikátory:

inline

explicit

inline explicit

explicit inline

První možnost představuje prototyp konstruktoru, druhá pak definiční deklaraci.

Jméno_třídy je jmenovka třídy, tj. její identifikátor. Pro vnořenou třídu se jedná o jmenovku vnořené třídy kvalifikovanou jménem obklopující třídy a rozlišovacím operátorem. *Jméno_třídy* lze spolu s rozlišovacím operátorem **::** vynechat, jedná-li se o deklaraci v těle třídy. *Jmenovka* je identifikátor třídy, jejíž součástí je deklarovaný konstruktor.

Spec_form_par představuje specifikaci formálních parametrů konstruktoru. Pokud se vynechá, jedná se o konstruktor bez parametrů. Parametry mohou být libovolného typu kromě typu deklarované třídy. Parametrem ale může být reference nebo ukazatel na deklarovanou třídu.

Např.:

```
class TA {
public:
   TA(); // prototyp konstruktoru
   // ...
};
TA::TA() // definice konstruktoru
{
   //...
}

class TB {
public:
   TB(int x) { /* ... / } // vložený konsturktor
   // ...
};
```

Inicializační část konstruktoru

Slouží pro inicializaci nestatických atributů instancí a k předání parametrů konstruktorům předků. Zapisuje se mezi hlavičku a tělo konstruktoru. Statické atributy se v inicializační části nesmí uvádět.

Syntaxe:

inicializační_část:

: *seznam_inicializací*

seznam_inicializací:

identifikátor **(***seznam_výrazů*$_{nep}$**)**

identifikátor **(***seznam_výrazů*$_{nep}$**),** *seznam_inicializací*

Identifikátor je identifikátor atributu nebo předka. *Seznam_výrazů* je:

- pro atribut neobjektového typu – jeden výraz, jehož hodnota se přiřadí jako počáteční hodnota tomuto atributu,
- pro atribut objektového typu – seznam parametrů konstruktoru tohoto objektového typu,
- pro předka – seznam parametrů konstruktoru tohoto předka.

Atributy neobjektového typu, které nejsou v inicializační části uvedeny:

- u globálních instancí jsou inicializovány nulovou hodnotou,
- u lokálních a dynamických instancí mají nedefinovanou hodnotu.

Atributy objektového typu, které nejsou v inicializační části uvedeny, jsou inicializovány svým konstruktorem bez parametrů.

U globálních instancí jsou vlastně všechny atributy nejprve inicializovány nulovou hodnotou a potom případně hodnotou uvedenou v inicializační části konstruktoru.

Inicializační část konstruktoru proběhne před vstupem do těla konstruktoru.

Příklad

```
class TA {
   int x, y;
public:
   TA(int _x, int _y) : x(_x), y(_y) {}
};

class TB {
   int z;
   TA A;
public:
   TB(int i, int j, int k);
};
TB::TB(int i, int j, int k) : A(i, j) { z = k; }
```

Atribut `A` objektového typu `TA` je v konstruktoru třídy `TB` inicializován voláním konstruktoru třídy `TA` s parametry `i` a `j`. Atribut `z` není inicializován v inicializační části, ale až v těle konstruktoru.

Atributy deklarované třídy se inicializují v pořadí, v jakém jsou deklarovány, nikoli v pořadí v jakém jsou inicializovány. V inicializační části se zpravidla uvádějí atributy v takovém pořadí, v jakém jsou deklarovány, aby bylo zřejmé pořadí jejich inicializace.

Konstantní a referenční atributy musí být v inicializační části uvedeny (inicializovány). Hodnoty konstantních atributů nelze již jinde měnit.

V inicializační části nelze inicializovat pole.

Příklad

```
class TC {
   int x;
   const int y;
   int& ri;
   static int z;
public:
   TC(int& i) : ri(i), x(0), y(10) {}
};
```

Atributy `ri` a `y` musí být v inicializační části inicializovány, naopak atribut `z` nesmí být v inicializační části uveden. Atributy jsou inicializovány v pořadí `x`, `y` a `ri`.

Definiční deklarace instance

Definice instance třídy znamená vždy volání konstruktoru. Parametry konstruktoru se zapisují za identifikátor instance do kulatých závorek podobně jako skutečné parametry při volání funkce. Pokud se má volat konstruktor bez parametrů, definuje se instance bez prázdných kulatých závorek.

Příklad

```
class TA {
   int x, y;
public:
   TA() : x(0), y(0) {} // konstruktor K1
   TA(int _x, int _y = 10) : x(_x), y(_y) {} // konstruktor K2
};

TA A1(10, 20); // volá se K2, x = 10, y = 20
TA A2(30);     // volá se K2, x = 30, y = 10
TA A3;         // volá se K1, x = 0, y = 0
TA A4();       // prototyp funkce bez parametrů vracející TA
               // nejedná se o definici instance A4
```

Instanci lze definovat také následujícím zápisem:

```
TA A5 = TA(10, 20); // dtto TA A5(10, 20);
TA A6 = TA(); // dtto TA A6;
```

Implicitní konstruktor

Implicitní konstruktor (angl. *default constructor*) je konstruktor, který může být volán bez parametrů. Může mít parametry, ale ty musí mít předepsány implicitní hodnoty.

Např.

```
class TA {
   int x, y;
   TA(int _x = 0, int _y = 0) : x(_x), y(_y){} // implicitní konstruktor
};
```

Implicitně deklarovaný implicitní konstruktor (angl. *implicitly-declared default constructor*) je konstruktor, který vytvoří automaticky překladač pro třídu, která nemá žádný uživatelem deklarovaný neboli explicitně deklarovaný konstruktor. Implicitně deklarovaný implicitní konstruktor je vložený (`inline`) a veřejně přístupný.

Implicitně deklarovaný implicitní konstruktor dané třídy je *implicitně definovaný implicitní konstruktor* (angl. *implicitly-defined default constructor*), když je použit k vytvoření nějaké instance dané třídy. Jedná se o konstruktor, který neobsahuje inicializační část a má prázdné tělo. Uživatelem zapsaná obdoba takovéhoto konstruktoru by vypadala následovně:

```
class TA {
   // ...
public:
   TA() {}
};
```

Jestliže třída obsahuje alespoň jeden uživatelem deklarovaný konstruktor, nebude překladač vytvářet implicitní konstruktor a třída potom nemusí mít implicitní konstruktor (pokud ho uživatel nedeklaroval).

Např.:

```
class TA {
   int x, y;
public:
   TA(int _x, int _y = 10) : x(_x), y(_y) {}
};
```

```
class TB {
   TA A1, A2;
public:
   TB(int i, int j) : A1(i, j) {} // Chyba
};
```

Třída `TA` nemá implicitní konstruktor, a proto se musí atribut `A2` v inicializační části konstruktoru třídy `TB` také inicializovat – překladač by v tomto případě oznámil chybu.

Definiční deklarace pole instancí

Při definici pole instancí třídy zavolá překladač konstruktory jednotlivých prvků v pořadí, v jakém jsou tyto prvky uloženy v paměti.

Např.:

```
class TA {
   int x, y;
public:
   TA() : x(0), y(0) {} // konstruktor K1
   TA(int _x, int _y = 10) : x(_x), y(_y) {} // konstruktor K2
};
TA A[10];
```

Při definici pole `A` se zavolají implicitní konstruktory `K1` pro prvky `A[0]`, `A[1]` až `A[9]`.

Pokud se mají pro jednotlivé prvky volat specifické konstruktory, předepíše se jejich volání v inicializátorech jednotlivých prvků, např.:

```
TA A[10] = { TA(10, 20), TA() };
```

Prvek `A[0]` bude inicializován konstruktorem `TA(10, 20)` a ostatní prvky konstruktorem `TA()`.

Dynamické instance

Dynamické instance jsou alokovány pomocí operátoru `new`. Při alokaci instance se volá příslušný konstruktor. Syntaxe použití operátoru `new` v tomto případě je následující:

::$_{nep}$ **new** *umístění*$_{nep}$ *jméno_třídy inicializátor*$_{nep}$

inicializátor:

(*seznam_parametrů*$_{nep}$**)**

Seznam_parametrů udává skutečné parametry konstruktoru. Pokud se má volat implicitní konstruktor, lze ponechat prázdné kulaté závorky nebo uvést *jméno_třídy* bez závorek. Parametr *umístění* – viz dřívější přednáška. Rozlišovací operátor před slovem `new` se používá u přetížených verzí operátoru `new` – viz některá z dalších přednášek.

Např. (třída `TA` je deklarována v kapitole „Definiční deklarace pole instancí“):

```
TA* A1 = new TA(10, 20); // volá se K2
TA* A2 = new TA();       // volá se K1
TA* A3 = new TA;         // volá se K1
```

Lze alokovat i pole instancí třídy. Pro inicializaci prvků pole se ale vždy použijí implicitní konstruktory.

Např.:

```
TA* A = new TA[10]; // alokuje se 10 prvků typu TA,
                    // pro každý prvek se volá K1
```

Kopírovací konstruktor

Kopírovací konstruktor je konstruktor, který lze volat s jedním parametrem typu reference na danou třídu. Kopírovací konstruktor třídy `TA` může mít tedy prototyp např.:

```
TA::TA(TA& t);
TA::TA(TA& t, int i = 0);
```

Reference může být deklarována s cv-modifikátory (`const`, `volatile`). Všechny formy kopírovacího konstruktoru z hlediska typu reference mohou být v třídě deklarovány současně.

Překladač volá kopírovací konstruktor vždy, když se k inicializaci instance použije jiná instance téže třídy. To znamená, že se použije v těchto případech:

a) v definici instance ve tvaru:
```
TA A1 = A2; // A2 je instance třídy TA
```
b) při předávání parametru objektového typu hodnotou, např.:
```
void f(TA A);
TA A1;
f(A1);
```
c) při vracení instance objektového typu funkcí, např.:
```
TA f();
```

Pokud v dané třídě není uživatelem deklarovaný kopírovací konstruktor, překladač vytvoří *implicitně deklarovaný kopírovací konstruktor* (angl. *implicitly-declared copy constructor*), který je vložený a veřejně přístupný. Pro třídu `TA` by jeho prototyp byl následující:

```
TA::TA(const TA&);
```

nebo

```
TA::TA(TA&);
```

podle okolností.

Implicitně deklarovaný kopírovací konstruktor dané třídy je *implicitně definovaný kopírovací konstruktor* (angl. *implicitly-defined copy constructor*), jestliže je použit k inicializaci instance dané třídy.

Implicitně definovaný kopírovací konstruktor zkopíruje každý nestatický atribut třídy. Je-li atribut objektového typu, použije se k jeho zkopírování jeho kopírovací konstruktor. V mnoha případech to postačí. Uživatel si musí vytvořit kopírovací konstruktor např. tehdy, je-li součástí třídy dynamicky alokovaný atribut.

Příklad

```
class TIntArr {
   int *a, n;
public:
   TIntArr(int _n) : n(_n) { a = new int[n]; }
};
TIntArr A(10);
TIntArr B = A;
```

Třída `TIntArr` neobsahuje explicitně definovaný kopírovací konstruktor, a proto se při deklaraci instance `B` použije implicitně definovaný kopírovací konstruktor, který pouze zkopíruje hodnotu atributů `a` a `n`. Atribut `B.a` bude tudíž ukazovat na stejné pole jako atribut `A.a`, což není zpravidla žádané, a proto se musí explicitně definovat kopírovací konstruktor:

```
class TIntArr {
   int *a, n;
public:
   TIntArr(int _n) : n(_n) { a = new int[n]; }
   TIntArr(const TIntArr& t);
};
TIntArr::TIntArr(const TIntArr& t)
{
   n = t.n;
   a = new int[n];
   for (int i = 0; i < n; ++i) a[i] = t.a[i];
}
```

Uvedený kopírovací konstruktor alokuje nové pole a překopíruje do něj hodnoty z pole `t.a`.

Konverzní konstruktory

Konstruktor třídy `TA` bez modifikátoru `explicit`, který lze volat s jedním parametrem, může překladač použít k implicitním nebo explicitním konverzím typu prvního parametru na typ třídy `TA`. Takovýto konstruktor se nazývá *konverzní konstruktor* (angl. *converting constructor*). Kopírovací konstruktor je konverzním konstruktorem.

Příklad

```
class TA {
   int x, y;
public:
   TA(int _x, int _y = 0) : x(_x), y(_y) {}
   // ...
};
void f(TA A);

TA A = 10; // A.x = 10, A.y = 0
TA A2 = static_cast<int>(10.5); // A2.x = 10, A2.y = 0
TA A3 = 10.5; // #1
TA A4 = (TA)10; // explicitní konverze
TA A5 = static_cast<TA>(10); // explicitní konverze
TA A6 = TA(10); // přímá definice instance bez konverze
f(10); #2
```

Instance `A`, `A2` a `A3` jsou definovány prostřednictvím implicitní konverze z typu `int` resp. `double` na typ `TA`. Instance `A4` a `A5` jsou vytvořeny pomocí explicitní konverze z celočíselného typu na typ `TA`. Instance `A6` je inicializována přímo, bez použití konverze.

Příkaz `#1` je správný, ale některé překladače mohou zobrazit varování týkající se konverze z typu `double` na typ `int`. Překladač Visual C++ 2003 zobrazí varování.

Při volání funkce `f()` příkazem `#2` se zavolá pouze konverzní konstruktor. Kopírovací konstruktor se nevolá.

Explicitní konstruktory

Explicitní konstruktor (angl. *explicit constructor*) je deklarován s modifikátorem `explicit`. Explicitní konstruktor narozdíl od konverzního konstruktoru nelze použít k implicitním konverzím, ale pouze k explicitním. Modifikátor `explicit` má význam specifikovat jen u konstruktorů, které lze volat s jedním parametrem. Modifikátor `explicit` může být uveden pouze v deklaraci konstruktoru v těle třídy, nikoliv v jeho definici mimo tělo třídy.

K zamezení implicitní konverze se zabraňuje nechtěným chybám. Např. v návaznosti na předchozí příklad:

```
int a;
f(a);
```

Volání funkce `f(a)` je pro překladač v pořádku, ale co když se uživatel spletl a napsal omylem malé `a` místo velkého `A`. Aby překladač v případě volání `f(a)` oznámil chybu, musí se konstruktor třídy `TA` deklarovat s modifikátorem `explicit`:

```
class TA {
   // ...
   explicit TA(int _x, int _y = 0) : x(_x), y(_y) {}
   // ...
};
```

Pro volání funkce `f(a)` se musí parametr `a` přetypovat na `TA` jedním z následujících způsobů:

```
f((TA)a);
f(static_cast<TA>(a));
f(TA(a));
```

Ve všech uvedených případech se volá pouze konverzní konstruktor – nevolá se po něm ještě kopírovací konstruktor.

Definice instancí s implicitní konverzí jsou při použití explicitního konstruktoru nesprávné, např.:

```
TA A = 10; // Chyba – implicitní konverze
TA A4 = (TA)10; // OK - explicitní konverze
TA A5 = static_cast<TA>(10); // OK – explicitní konverze
TA A6 = TA(10); // OK – přímá definice instance bez konverze
```

Příklad

Následující příklad ilustruje další případ, kdy nepoužití explicitního konstruktoru v třídě `TA` způsobí neočekávanou chybu.

```
class TA {
public:
   TA(bool Inicializace = false);
   // ...
};
void f(TA& A);

class TB {
public:
   TA& GetTA();
};

TB* B;
```

Programátor chtěl volat funkci f příkazem

```
f(B->GetTA());
```

ale omylem napsal příkaz

```
f(B);
```

Výraz `f(B)` se provede, protože lze implicitně přetypovat hodnotu typu `TB*` na hodnotu typu `bool`. Takže do funkce `f()` se předá dočasná instance třídy `TA`, která byla vytvořena konstruktorem `TA(bool Inicializace = false)`.

Nepojmenovaná instance

Zápis konstruktoru znamená příkaz k vytvoření nepojmenované instance. Lze ho použít v přiřazovacích příkazech, v příkazu `return`, v seznamech skutečných parametrů funkcí apod. Při použití nepojmenované instance v místě, kde se volá pro pojmenovanou instanci kopírovací konstruktor, se pro nepojmenovanou instanci kopírovací konstruktor nevolá.

Příklad

```
class TA {
   int x, y;
public:
   TA() {}
   TA(int _x, int _y) : x(_x), y(_y) {}
   friend TA Pricti(TA A, int n);
   // ...
};
TA Pricti(TA t, int n)
{
   return TA(t.x+n, t.y+n); // #1
}
TA A;
A = Pricti(TA(10,20), 5); // #2
```

Nepojmenovaná instance je vytvořena v příkazu `#1` a pro první skutečný parametr funkce `Pricti` v příkazu `#2`. V obou případech se nevolá kopírovací konstruktor. Pokud by však byla vytvořena instance `B`, která by byla prvním parametrem funkce `Pricti`

```
TA B(10, 20);
A = Pricti(B, 5);
```

volal by se pro parametr `t` funkce `Pricti` kopírovací konstruktor.

OBJEKTOVÉ TYPY – POKRAČOVÁNÍ

Destruktory

Destruktory jsou metody, které se volají automaticky (implicitně) při zániku:

- lokální nestatické instance – ve chvíli, kdy program opustí oblast platnosti definice této instance,
- globální instance a lokální statické instance – při ukončení programu (po ukončení funkce `main`),
- dynamické instance – při volání operátoru `delete` pro tuto instanci.

Destruktory jsou metody třídy s těmito specifiky:

- Jméno destruktoru je tvořeno identifikátorem (jmenovkou) třídy, před kterým je znak ~ (tilda).
- Nemá žádný parametr.
- Deklarace destruktoru nesmí obsahovat specifikaci typu vrácené hodnoty, a to ani `void`.
- Nedědí se. Odvozená třída ovšem použije ke své destrukci destruktory svých předků.
- Nesmí být statické, konstantní nebo nestálé.
- Nelze získat jejich adresu.
- Mohou být volány pro instance deklarované s cv-modifikátory (`const`, `volatile`).

Ve funkci např. `f()` se může volat destruktor třídy `TA`, pokud je splněna jedna z těchto podmínek:

- destruktor třídy `TA` je veřejný,
- funkce `f()` je ve třídě `TA` deklarována jako přítel.

Syntaxe:

deklarace_destruktoru:

modifikátory$_{nep}$ *jméno_třídy*$_{nep}$**::**$_{nep}$*~jmenovka* **( )**;

modifikátory$_{nep}$ *jméno_třídy*$_{nep}$**::**$_{nep}$*~jmenovka* **( )** *tělo*

modifikátory:

virtual

inline

virtual inline

inline virtual

První možnost představuje prototyp destruktoru, druhá pak definiční deklaraci. Význam jednotlivých symbolů – viz 4. přednáška, kapitola „Deklarace konstruktoru“.

Např.:

```
class TA {
   // ...
public:
   ~TA(); // prototyp destruktoru
};

TA::~TA() // definice destruktoru
{
   // ...
}
```

Jestliže daná třída nemá uživatelem deklarovaný destruktor, překladač vytvoří *implicitně deklarovaný destruktor* (angl. *implicitly-declared destructor*), který je vložený (`inline`) a veřejně přístupný.

Implicitně deklarovaný destruktor je implicitně definovaný (angl. *implicitly defined destructor*), je-li použit ke zrušení nějaké instance. Implicitně definovaný destruktor má prázdné tělo.

Destruktor lze volat stejně jako jiné metody, ale prakticky se tato možnost nevyužívá. Např. pro instanci `A` třídy `TA` by explicitní volání destruktoru bylo následující:

```
A.~TA();
```

Po provedení těla destruktoru dané třídy se automaticky volají destruktory atributů objektových typů dané třídy a destruktory předků.

Skončí-li program voláním funkce `exit()`, nezavolají se destruktory lokálních nestatických instancí. Globální a lokální statické instance budou zrušeny obvyklým způsobem.

Skončí-li program voláním funkce `abort()`, nezavolají se žádné destruktory.

V případě dynamických instancí, zánik ukazatele nezpůsobí volání destruktoru instance, na který ukazoval. Musí se použít operátor `delete` resp. `delete[]`.

Destruktory pro prvky pole jsou volány v opačném pořadí jejich konstrukce.

Příklad

```
class TA {
   char* s;
   int n;
public:
   TA(int _n) : n(_n) { s = new char[n]; }
   ~TA() { delete[] s; }
   // ...
};

void main()
{
   TA* A1 = new TA(5);
   TA A2(5);
  // ...
   delete A1; // volá se destruktor třídy TA pro instanci A1
} // po ukončení main se volá destuktor pro instanci A2
```

ODVOZENÉ TŘÍDY

Jazyk C++ podporuje vícenásobnou dědičnost (angl. *multiple inheritance*), takže jedna třída může mít několik *předků* (*bázových tříd*) (angl. *base classes*).

Deklarace_objektového_typu_s_předky:

klíč jmenovka **:** *seznam_předků* **{** *tělo_třídy* **}** *seznam_deklarátorů*$_{nep}$**;**

klíč:

class

struct

seznam_předků:

specifikátory_předka$_{nep}$ *jmenovka_předka*

specifikátory_předka$_{nep}$ *jmenovka_předka***,** *seznam_předků*

specifikátory_předka:

specifikátor_přístupu$_{nep}$ **virtual**$_{nep}$

virtual$_{nep}$ *specifikátor_přístupu*$_{nep}$

specifikátor_přístupu:

public

protected

private

Má-li daná třída předky, v deklaraci odvozené třídy (angl. *derived class*) se za její jmenovkou uvede dvojtečka a seznam jmenovek předků oddělených čárkou. Před každou jmenovkou předka může být uveden jeden ze specifikátorů přístupu `public`, `protected` nebo `private` a případně klíčové slovo `virtual`. Uvede-li se obojí, nezáleží na jejich pořadí.

V seznamu předků nesmí být uvedena jmenovka právě deklarované třídy nebo vícekrát stejná jmenovka předka.

Pokud daná třída v deklaraci svých předků nemá u žádného z předků uveden specifikátor `virtual`, říká se, že daná třída nemá žádné virtuální předky resp. má jen nevirtuální předky. V takovém případě odvozená třída zdědí všechny složky svých předků (atributy, metody, typy) kromě konstruktorů, destruktorů a přetížených kopírovacích operátorů přiřazení =.

Příklad

```
class TA {
   int x, y;
public:
   void f();
   // ...
};
class TB {
   double a, b;
public:
   void g();
   // ...
};
```

```
class TC : public TA, public TB {
   int z;
public:
   void h();
   // ...
};
```

Třída `TC` má dva nevirtuální předky: třídy `TA` a `TB` a obsahuje atributy `x`, `y`, `a`, `b`, `z` a metody `f()`, `g()`, `h()`.

Přístupová práva ke složkám předků

Specifikátor přístupu uvedený před jmenovkou předka určuje nejširší hodnotu přístupových práv, které mohou mít složky zděděné třídy. Konkrétně, bude-li před jmenovkou předka specifikátor:

- `public` – zděděné složky budou mít stejná přístupová práva, jako mají v předkovi.
- `protected` – veřejně přístupné a chráněné složky budou chráněné, soukromé složky zůstanou soukromé.
- `private` – všechny zděděné složky budou v odvozené třídě soukromé.

Pokud se před jmenovkou předka neuvede specifikace přístupu, je to totéž, jako kdyby se před jmenovkou předka uvedl specifikátor:

- `private` – v deklaraci odvozené třídy deklarované s klíčem `class`,
- `public` – v deklaraci odvozené struktury (klíč `struct`).

Přístupová práva pro zděděnou složku, ke které má potomek přístup (složky, které nejsou v předkovi soukromé), lze v potomkovi změnit opětovnou deklarací zděděné složky s novou specifikací přístupu. Existují dva způsoby deklarace.

1. způsob

Zděděná složka se deklaruje uvedením jmenovky předka, rozlišovacím operátorem a identifikátorem zděděné složky. U atributu se tedy neuvádí jeho typ, u metody se neuvádí ani návratový typ ani kulaté závorky se seznamem formálních parametrů.

Tímto způsobem lze pouze obnovit přístupová práva pro zděděnou složku, jaké měla ve své třídě.

Např.:

```
class TA {
protected:
   int x, y;
public:
   int GetX() const { return x; }
   enum TBarva { modra, cervena };
   // ...
};
class TB : private TA {
protected:
   TA::x;
public:
   TA::GetX;
   TA::cervena;
   // ...
};
```

Složky třídy `TA` jsou ve třídě `TB` soukromé, kromě atributu `x`, metody `GetX()` a výčtové konstanty `cerverna`. Výčtové konstanty nemá ale smysl tímto způsobem zpřístupňovat, protože lze k nim přistupovat pomocí jmenovky jejich třídy a rozlišovacího operátoru, např. `TA::cervena`.

2. způsob

Deklarace zděděné složky je stejná, jako u 1. způsobu s tím rozdílem, že začíná klíčovým slovem `using`. Tímto způsobem lze změnit přístup bez omezení.

Např.:

```
class TB : public TA {
private:
   using TA::x;
   // ...
};
```

Ve třídě `TB` je atribut `x` soukromý, `y` chráněný a metoda `GetX()` veřejně přístupná.

Nevirtuální a virtuální dědění

Jedna třída nemůže být vícekrát přímým předkem jiné třídy. Může se ale stát, že od jedné třídy, např. `TA` budou odvozeny dvě třídy `TB` a `TC` a ty budou přímými předky třídy `TD`. Podobjekt třídy `TA` bude obsažen ve třídě `TD` jednou nebo dvakrát v závislosti na typu dědění.

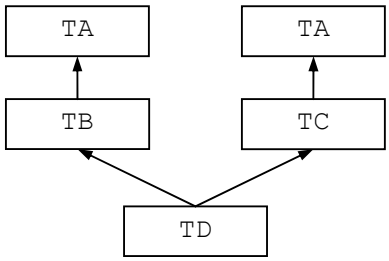
Statické složky třídy jsou v paměti uloženy pouze jednou bez ohledu na typ dědění, počtu instancí dané třídy a jejich potomků. Obdobně identifikátory vnořených typů a výčtových konstant dané třídy existují v jejích potomcích pouze jednou bez ohledu na typ dědění.

Příklad – nevirtuální dědění

Jsou deklarovány třídy:

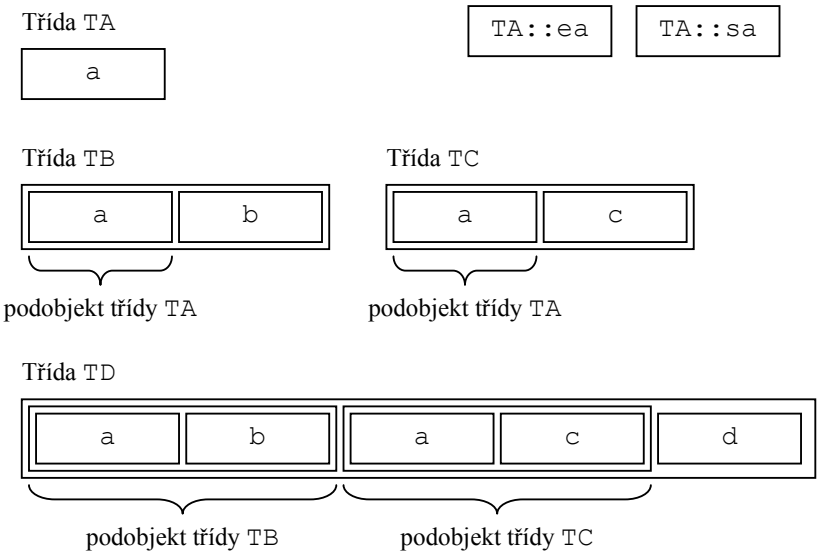
```
struct TA { int a; static int sa; enum { ea = 10 }; };
struct TB : TA { int b; };
struct TC : TA { int c; };
struct TD : TB, TC { int d; };
```

Dědická hierarchie tříd se znázorňuje orientovaným acyklickým grafem – viz obr. 1.



Obr. 1 Graf hierarchie tříd s nevirtuálním předkem TA

Norma jazyka C++ nepředepisuje přesné uspořádání instance v paměti. Jediný požadavek normy je, že složky deklarované bezprostředně za sebou, které nejsou odděleny specifikátorem přístupových práv, musí být v paměti uloženy bezprostředně za sebou. Struktura uspořádaní složek instancí uvedených tříd v paměti v prostředí Visual C++ 2003 je znázorněna na obr. 2.



Obr. 2 Paměťová reprezentace instancí tříd – nevirtuální dědění

K atributu `a` v instanci třídy `TD` se v takovémto případě musí přistupovat pomocí jmenovky třídy `TB` nebo `TC` a rozlišovacího operátoru, např.:

```
TD D;
D.TB::a = 0;
D.TC::a = 1;
D.TA::a = 0; // #1 Chyba
D.a = 0; // Chyba
```

K statickému atributu `sa` lze přistupovat těmito způsoby:

```
D.sa = 1;
TA::sa = 2;
TB::sa = 3;
TC::sa = 4;
TD::sa = 5;
```

Stejnými způsoby lze přistupovat i k výčtové konstantě `ea`.

Může být sice deklarována i takováto třída:

```
struct TE : TA, TB { int e; };
```

ale potom lze přistoupit pouze ke složkám třídy `TA`, která je podobjektem třídy `TB`:

```
TE E:
E.TB::a = 0; // OK
E.TA::a = 0; // #2 Chyba
E.a = 0; // Chyba
```

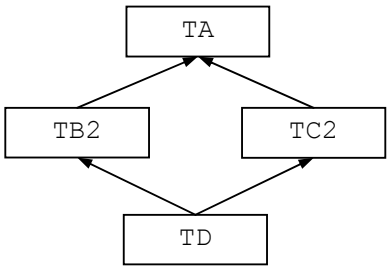
Příkazy #1 a #2 jsou chybné. Překladač Visual C++ 2003 je přesto označí za správné a hodnotu uloží do prvního nalezeného atributu `a` příslušné instance.

Příklad – virtuální dědění

Pokud mají být složky třídy `TA` ve třídě `TD` uvedeny jen jednou, musí se třída `TA` specifikovat jako virtuální předek tříd `TB` a `TC`:

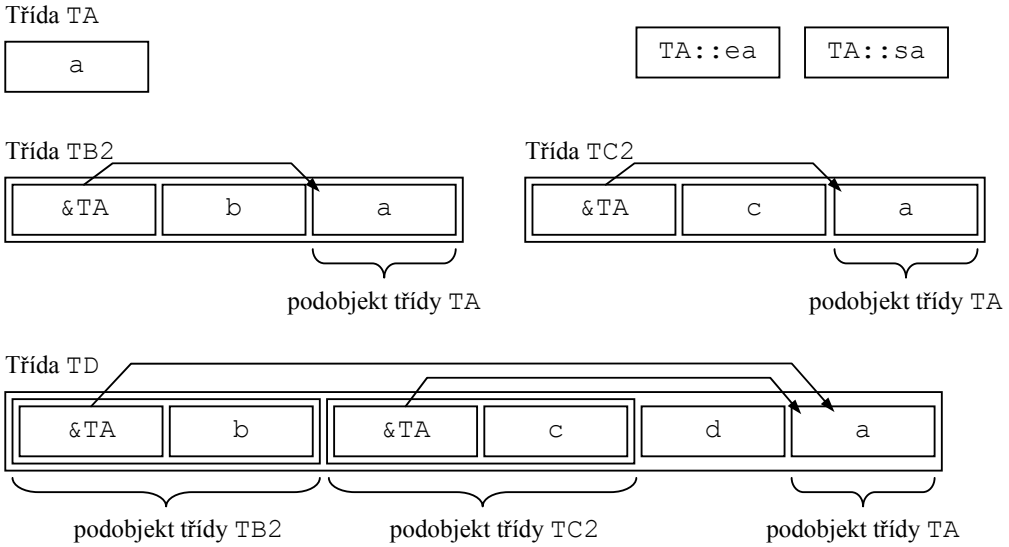
```
struct TA { int a; static int sa; enum { ea = 10 }; };
struct TB2 : virtual TA { int b; };
struct TC2 : virtual TA { int c; };
struct TD : TB2, TC2 { int d; };
```

Graf dědické hierarchie těchto struktur je znázorněn na obr. 3.



Obr. 3 Hierarchie tříd s virtuálním předkem TA

Paměťová reprezentace těchto struktur v prostředí Visual C++ 2003 je uvedena na obr. 4.



Obr. 4 Paměťová reprezentace instancí tříd – virtuální dědění

Na začátku podobjektu třídy `TB2` a `TC2` je skrytý atribut, obsahující adresu virtuálního podobjektu třídy `TA`. Tato adresa je v obou podobjektech stejná. Operátor `sizeof` vrací velikost instance resp. třídy včetně skrytých atributů, obsahujících adresy virtuálních podobjektů. Výsledkem výrazů `sizeof D` a `sizeof(TD)` je tedy hodnota 24.

K atributu `a` v instanci třídy `TD` lze přistupovat bez kvalifikace pomocí jmenovky předka a rozlišovacího operátoru, např.:

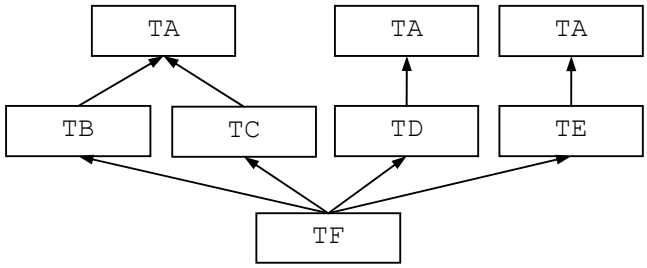
```
TD D;
D.a = 0; // OK
```

K statickému atributu `sa` a výčtové konstantě `ea` lze přistupovat stejnými způsoby jako při nevirtuálním dědění.

Daná třída může mít virtuální i nevirtuální předky stejného typu. V tom případě obsahuje jednoho virtuálního předka a všechny nevirtuální předky stejného typu. Např.:

```
struct TA { int a; };
struct TB : virtual TA { int b; };
struct TC : virtual TA { int c; };
struct TD : TA { int d; };
struct TE : TA { int e; };
struct TF : TB, TC, TD, TE { int f; };
```

Třída `TA` je ve třídě `TF` obsažena celkem třikrát. Jednou jako virtuální předek tříd `TB` a `TC` a dvakrát jako nevirtuální předek tříd `TD` a `TE`. Graf dědické hierarchie těchto struktur je znázorněn na obr. 5.



Obr. 5 Hierarchie tříd s virtuálním i nevirtuálním předkem TA

K jednotlivým atributům `a` v instanci třídy `TF` se v takovémto případě musí přistupovat pomocí jmenovky příslušného přímého předka třídy `TF` a rozlišovacího operátoru, např.:

```
TF F;
F.TB::a = 0; // dtto F.TC::a = 0;
F.TD::a = 0;
F.TE::a = 0;
```

Konflikt jmen

Třída představuje oblast platnosti a viditelnosti jejích složek. Odvozená třída se chová jako oblast vnořená do rodičovské třídy. To znamená, že identifikátory složek odvozené třídy zastiňují stejně pojmenované identifikátory složek jejích předků. To se týká i stejně pojmenovaných metod, které se mohou (ale nemusí) lišit typem nebo počtem parametrů. K takovéto složce předka lze přistupovat pomocí jmenovky předka a rozlišovacího operátoru `::`.

Příklad

```
class TA {
public:
   int x, x2;
   int g() { return x + x2; }
};
```

```
class TB {
public:
   double x, y;
   double f() { return x*y; }
};
class TC : public TA, public TB {
public:
   char x, y;
   double f(int Trida);
   int g() { return x + y; }
};
double TC::f(int Trida)
{
   switch (Trida) {
      case 0:  return TA::x * x2;
      case 1:  return TB::f();
      default: return x * y;
   }
}
int main()
{
   TC C;
   C.x = 1;
   C.y = 2;
   C.TA::x = 3;
   C.x2 = 4;
   C.TB::x = 5.5;
   C.TB::y = 6.5;
   cout << C.f(0) << ' ' << C.TB::f() << ' ' << C.g() << ' ' <<
           C.TA::g() << '\n';
   cout << C.f() << '\n'; // chyba
}
```

Výstup programu bude následující:

```
12 35.75 3 7
```

Identifikátory složek přímého předka dané třídy zastiňují stejně pojmenované identifikátory složek předka přímého předka dané třídy. Např.:

```
class TA {
public:
   int x;
   void f();
};
class TB : public TA {
public:
   int x;
   void f();
};
class TC : public TB { /*...*/ };

TC C;
C.x = 1; // TB::x
C.TA::x = 2;
C.f(); // TB::f()
C.TA::f();
```

Podobně identifikátory složek třídy zastiňují stejně pojmenované globální identifikátory, ke kterým se v dané třídě přistupuje pomocí unárního rozlišovacího operátoru:

```
void f();
struct TA {
   void f(bool);
   void g() { ::f() + f(true); }
}
```

Pokud by se při volání funkce `f()` neuvedl unární rozlišovací operátor, překladač by zápis `f()` ve třídě `TA` chápal jako volání metody f třídy `TA` a oznámil by chybu, že volaná metoda `f()` nemá parametr typu `bool`.

Potomek může zastoupit předka

Potomek může vždy zastoupit předka.

Překladač může automaticky konvertovat instanci potomka na předka, jestliže:

- toto přetypování je jednoznačné,
- předek je v místě operace veřejně přístupný.

Výsledkem je instance tvořená zděděným podobjektem.

Příklad

```
struct TA { int a; };
struct TB : TA { int b; };
struct TC : TA { int c; };
struct TD : TB, protected TC { int d; };
TD D;
TC C;
TB B;
TA A;
C = D; // Chyba – TC není veřejný předek TD
B = D; // OK
A = D; // Chyba – podobjekt TA je obsažen v TD dvakrát
A = static_cast<TB>(D); // OK
A = static_cast<TC>(D); // Chyba - TC není veřejný předek TD #1
```

Příkaz #1 není správný, přesto jej překladač Visual C++ 2003 označí za správný.

Pokud by třída `TA` byla virtuálním předkem tříd `TB` a `TC`, je možné následující přiřazení:

```
A = D;
```

Podobně překladač provede automatickou konverzi ukazatele na potomka na ukazatele na předka při splnění stejných pravidel jako pro instance. Stejně funguje i přetypování referencí:

```
TB* UB = &D; // OK
TB& RB = D; // OK
```

Výsledkem konverze ukazatele na potomka na ukazatel na předka je ukazatel, který ukazuje na podobjekt předka. tj. může obsahovat jinou adresu než ukazatel na potomka.

Toto automatické přetypování lze využít při předávání skutečných parametrů funkcí – formálním parametrem funkce může být reference (ukazatel) na předka, skutečným parametrem může být l-hodnota (ukazatel) na potomka.

Opačné přetypování z předka na potomka překladač automaticky neprovede. Musí se provést pomocí operátorů přetypování `static_cast` a `dynamic_cast` – viz některá z dalších přednášek.

Konstruktory, destruktory a dědičnost

Pořadí konstrukce a destrukce

Při vytváření instance odvozené třídy se nejprve zkonstruují zděděné podobjekty. Přitom se nejdříve volají konstruktory virtuálních předků a potom nevirtuálních. V obou těchto skupinách se postupuje podle pořadí, ve kterém jsou předkové zapsáni v deklaraci odvozené třídy. Jsou-li předkové také odvozené třídy, budou nejprve volány konstruktory jejich předků atd.

Destruktory jsou volány v opačném pořadí než konstruktory.

Příklad – nevirtuální dědění

```
class TA {
public:
   TA() { cout << "TA"; }
   ~TA() { cout << "TA"; }
};
class TB : public TA {
public:
   TB() { cout << "TB"; }
   ~TB() { cout << "TB"; }
};
class TC : public TA {
public:
   TC() { cout << "TC"; }
   ~TC() { cout << "TC"; }
};
class TD : public TB, public TC {
public:
   TD() { cout << "TD"; }
   ~TD() { cout << "TD"; }
};
void f()
{
  TD D;
  cout << "\n";
}
```

Po provedení funkce `f()` program vypíše následující text:

```
TATBTATCTD
TDTCTATBTA
```

Příklad – virtuální dědění

```
class TA { ... };
class TB { ... };
class TC : public TB, public virtual TA { ... };
class TD : public TB, public virtual TA { ... };
class TE : public TC, public virtual TD { ... };
TE E;
```

Při vytváření instance `E` se nejprve se zkonstruuje virtuální předek `TD` a potom nevirtuální předek `TC`. Při konstrukci `TD` se nejprve vytvoří virtuální předek `TA`, potom nevirtuální předek `TB` a nakonec třída `TD`. Při konstrukci `TC` se vytvoří pouze nevirtuální předek `TB`, protože předek `TA` již existuje a potom se zkonstruuje `TC`. Nakonec se vytvoří třída `TE`. Konstruktory tedy budou volány v následujícím pořadí: `TA()`, `TB()`, `TD()`, `TB()`, `TC()`, `TE()`.

Inicializační část konstruktorů

V inicializační části konstruktoru odvozené třídy se uvádějí konstruktory přímých předků se seznamem parametrů. Pokud se v inicializační části odvozené třídy neuvede konstruktor přímého předka, použije překladač k inicializaci předka implicitní konstruktor. Pokud není v daném předkovi deklarovaný nebo je v potomkovi nepřístupný (je v předkovi soukromý), ohlásí překladač chybu.

Příklad

```
class TA {
   int a1, a2;
public:
   TA(int _a1 = 0, int _a2 = 0) : a1(_a1), a2(_a2) {}
};
class TB {
   int b;
public:
   TB(int _b) : b(_b) {}
};
class TC : public TA, public TB {
   int c;
public:
   TC(int _a1, int _a2, int _b, int _c) : TA(_a1, _a2), TB(_b), c(_c) {}
   TC() : TB(0), c(0) {} // konstruktor TB se musí uvést
};
```

POLYMORFISMUS

S instancemi se pracuje často pomocí ukazatelů. Přitom zpravidla vznikají situace, kdy není známý přesný typ instance, na kterou daný ukazatel ukazuje a je potřebné vyvolat metodu skutečné instance.

Časná a pozdní vazba

Příklad

```
class TObjektSite {
public:
   enum TTyp { osBod, osUsek };
protected:
   TTyp Typ;
public:
   TObjektSite(TTyp _Typ) : Typ(_Typ) {}
   void Vypis() {}
};
class TBod : public TObjektSite {
   int Cislo;
public:
   TBod(int _Cislo) : TObjektSite(osBod), Cislo(_Cislo) {}
   void Vypis() { cout << "Bod: " << Cislo << '\n'; }
};
class TUsek : public TObjektSite {
   int BodCislo[2];
public:
   TUsek(int Cislo1, int Cislo2);
   void Vypis()
      { cout << "Usek: " << BodCislo[0] << ' ' << BodCislo[1] << '\n'; }
};
TUsek::TUsek(int Cislo1, int Cislo2) : TObjektSite(osUsek)
{
   BodCislo[0] = Cislo1; BodCislo[1] = Cislo2;
}
int i;
enum { PocOS = 3 };
TObjektSite* OS[PocOS];
OS[0] = new TBod(100);
OS[1] = new TBod(200);
OS[2] = new TUsek(100, 200);
for (i = 0; i < PocOS; i++) OS[i]->Vypis();
//...
for (i = 0; i < PocOS; i++) delete OS[i];
```

Program pracuje s polem ukazatelů na objekty sítě, kterými mohou být body (třída `TBod`) a úseky (třída `TUsek`). Bod a úsek má společného předka – třídu `TObjektSite`, na který ukazují prvky pole `OS`. Prvky pole jsou sice typu `TObjektSite*`, obsahují však ukazatele na objekty různých potomků třídy `TObjektSite`. Prvky pole mají tzv. *statický typ* `TObjektSite*`, a *dynamický typ* podle okolností `TBod*` nebo `TUsek*`.

Příkaz `OS[i]->Vypis()` volá pro všechny prvky pole metodu `TObjektSite::Vypis()`, která nic nevypíše.

Překladač použil v uvedeném příkladě tzv. *časnou vazbu*. To znamená, že vyšel ze statického typu ukazatele `TObjektSite*` a podle toho zavolal metodu `TObjektSite::Vypis()`.

V tomto případě je však potřebná tzv. *pozdní vazba*, při níž se použije dynamický typ, tj. je potřebné volat metodu `Vypis` skutečného typu instance, na který ukazatel ukazuje. Pozdní vazba pro nějakou metodu se použije, pokud se tato metoda označí jako *virtuální*. Virtuální metoda je deklarována se specifikátorem `virtual`. Definice virtuální metody mimo tělo třídy nesmí obsahovat specifikátor `virtual`.

Třídy, obsahující virtuální metody, se nazývají *polymorfní* (angl. *polymorphic*).

Jestliže je v určité třídě `TA` deklarována metoda jako virtuální, bude virtuální i ve všech potomcích třídy `TA`. Při nové definici metody v potomkovi se metoda nemusí (ale může) deklarovat se specifikátorem `virtual`. Za účelem přehlednosti se však doporučuje slovo `virtual` uvádět i v deklaracích virtuální metody v potomcích.

Virtuální metoda, která je v odvozené třídě nově definována, se nazývá *předefinovaná virtuální metoda* (angl. *overridden virtual function*).

Aby uvedený program fungoval správně, musí být třída `TObjektSite` deklarována s virtuální metodou `Vypis`:

```
class TObjektSite {
public:
   enum TTyp { osBod, osUsek };
protected:
   TTyp Typ;
public:
   TObjektSite(TTyp _Typ) : Typ(_Typ) {}
   virtual void Vypis() {}
};
```

Metoda `Vypis` je virtuální i ve třídě `TBod` a `TUsek`, i když není slovo `virtual` u metody `Vypis` v těchto třídách uvedeno.

Virtuální metodou mohou být i destruktory, i když nemají v předkovi a v potomkovi stejné jméno. Virtuální nemohou být konstruktory, statické metody ani spřátelené funkce.

Pokud se volá destruktor pro ukazatel na předka (explicitně nebo pomocí operátoru `delete`), který ve skutečnosti ukazuje na instanci potomka, měl by být destruktor předka virtuální, aby se volal destruktor potomka a z něho potom destruktor předka.

Pokud by v uvedeném příkladě třída `TBod` obsahovala dynamický řetězec znaků reprezentující jméno bodu, musel by být definován destruktor, který by řetězec dealokoval:

```
class TBod : public TObjektSite {
   int Cislo;
   char *Jmeno;
public:
   TBod(int _Cislo, const char* _Jmeno);
   ~TBod() { delete[] Jmeno; };
   // ...
};
```

Aby se destruktor třídy `TBod` volal při dealokaci ukazatele na `TObjektSite` pomocí příkazu

```
delete OS[i];
```

musí se deklarovat virtuální destruktor třídy `TObjektSite`:

```
class TObjektSite {
   // ...
public:
   virtual ~TObjektSite() {}
};
```

I když destruktor není v potomkovi uživatelem definovaný, měl by se v předkovi deklarovat destruktor jako virtuální, aby se zavolal implicitně definovaný destruktor potomka, který mj. zavolá destruktory atributů objektových typů, které jsou složkami potomka.

Příklad

```
class TA {
   int x;
public:
   TA(int _x) : x(_x) {}
   virtual ~TA() { cout << "destruktor TA\n"; }
};
class TB {
   int y;
public:
   TB(int _y) : y(_y) {}
   ~TB() { cout << "destruktor TB\n"; }
};
class TC : public TA {
   TB b;
public:
   TC(int _x, int _y) : TA(_x), b(_y) {}
};
void f()
{
   TC* C = new TC(10, 20);
   TA* A = C;
   delete A;
}
```

Provedením příkazu `delete A;` ve funkci `f()` se nejprve zavolá implicitně definovaný destruktor třídy `TC`, který zavolá destruktor pro atribut `b` a potom destruktor předka, třídy `TA`. Na obrazovku se vypíše text:

```
destruktor TB
desturktor TA
```

Pokud by ale třída `TA` měla nevirtuální destruktor, nezavolal by se destruktor atributu `b` při provedení příkazu `delete A;` a na obrazovku by se vypsal text:

```
destruktor TA
```

V konstruktoru dané třídy lze sice volat virtuální metodu, ale ta se chová jako by byla nevirtuální, tj. volá se metoda dané třídy, nikoliv metoda potomka. Je to proto, že v okamžiku, kdy se volá konstruktor předka ještě není zkonstruován potomek.

V destruktoru dané třídy lze také volat virtuální metodu, ale ta volá vždy metodu dané třídy. Je to proto, že nejprve dojde k destrukci potomka a potom předka.

Virtuální metody musí být v předkovi i v potomkovi deklarovány se stejným jménem, počtem a typem parametrů. Typ vrácené hodnoty virtuální metody `f` by měl být identický v předkovi `TA` i v potomkovi `TB` nebo musí splňovat všechna následující omezení:

a) v obou třídách `TA` i `TB` se jedná o ukazatel nebo referenci na třídu;

b) třída uvedená v návratovém typu metody `TA::f` musí být přímým nebo nepřímým předkem třídy uvedené v návratovém typu metody `TB::f` a tento předek je ve třídě `TB` veřejně přístupný;

c) v obou třídách mají ukazatele nebe reference v návratovém typu stejné cv-modifikátory nebo v metodě `TA::f` je uveden cv-modifikátor a v metodě `TB::f` uveden není.

Pozdní vazba se uplatňuje při volání metod prostřednictvím ukazatelů a referencí. To znamená, že se uplatňuje také:

a) pro parametry funkcí předávané odkazem,

b) pro metody volané v těle jiných metod – prostřednictvím ukazatele `this`.

Virtuální metoda může být předefinována, i když není v potomkovi viditelná, např.:

```
struct TA { virtual void f(); };
struct TB : TA { void f(int); };
struct TC : TB { void f(); };
```

Metoda `f(int)` třídy `TB` zastiňuje virtuální metodu `f()` třídy `TA`. Metoda `f(int)` není virtuální. Metoda `f()` třídy `TC` je shodná s metodou `f()` třídy `TA` a tudíž je virtuální a předefinovává metodu `TA::f()`, i když metoda `TA::f()` není ve třídě `TC` viditelná.

Je-li definována virtuální metoda v nevirtuálním předkovi, který je vícekrát obsažen v potomkovi, existuje v potomkovi i více definic této virtuální metody, např.:

```
struct TA { virtual void f(); };
struct TB : TA { virtual void f(); };
struct TC : TA { virtual void f(); };
struct TD : TB, TC { };

TD D;
TA* A = static_cast<TB*>(&D);
A->f(); // volá se metoda TB::f()
A = &D; // Chyba – TA je v TD dvakrát
D.f(); // Chyba – metoda f() je v TD dvakrát
```

U uvedeném příkladě existují ve třídě `TD` dvě virtuální metody `f()` zděděné z předků `TB` a `TC`. Pokud by se ve třídě `TD` předefinovala virtuální metoda `f()`, existovala by v této třídě jen jedna verze této metody a výrazy `A->f()` a `D.f()` by zavolaly metodu `TD::f()`.

Pokud je virtuální metoda definována ve virtuálním předkovi, mohou nastat případy uvedené v následujícím příkladě:

```
struct TA { virtual void f(); };
struct TB : virtual TA { virtual void f(); };
struct TC : virtual TA { virtual void f(); };
struct TD : TB, TC { }; // Chyba
struct TE : TB, TC { virtual void f(); }; // OK
```

Definice třídy `TD` není správná, protože oba její předci `TB` a `TC` mají předefinovanou virtuální metodu `f()`, která není ve třídě `TD` předefinována a překladač by nevěděl, zda má pro ukazatel nebo referenci na `TA` volat metodu `f()` třídy `TB` nebo `TC`.

Následující příklad ukazuje možnost volání virtuální metody jiného předka daného potomka prostřednictvím virtuálního předka:

```
struct TA { virtual void f(); };
struct TB : virtual TA { virtual void f(); };
struct TC : virtual TA { };
struct TE : TB, TC { };

TE E;
TC* C = &E;
C->f(); // volá se TB::f()
```

Pokud se volání virtuální metody kvalifikuje jmenovkou třídy a rozlišovacím operátorem, potlačuje se tím pozdní vazba, např.:

```
class TA { public: virtual void f(); };
class TB : public TA { public: void f(); };
void TB::f()
{
   // ...
   TA::f(); // volá TA::f() a ne TB::f()
}
```

Abstraktní třídy

Abstraktní třída (angl. *abstract class*) je třída, která může být použita jen jako předek jiné třídy. Nelze deklarovat instanci abstraktní třídy, ale lze deklarovat ukazatele nebo referenci na abstraktní třídu. Třída je abstraktní, jestliže má alespoň jednu *čistou virtuální metodu* (angl. *pure virtual function*). Čistá virtuální metoda nemá definici a její prototyp má tvar:

virtual *prototyp* = 0;

kde *prototyp* je vlastní prototyp nevirtuální metody.

Třída `TObjektSite` z dříve uvedeného příkladu obsahuje metodu `Vypis()`, která nic nedělá, a proto nemá smysl vytvářet instanci této třídy. Třída tudíž může být abstraktní a metoda `Vypis()` čistá virtuální. Deklarace třídy `TObjektSite` by byla následující:

```
class TObjektSite {
public:
   enum TTyp { osBod, osUsek };
protected:
   TTyp Typ;
public:
   TObjektSite(TTyp _Typ) : Typ(_Typ) {}
   virtual ~TObjektSite() {}
   virtual void Vypis() = 0; // čistá virtuální metoda
};
```

Abstraktní třídou je i potomek jiné abstraktní třídy, pokud neobsahuje definici zděděné čisté virtuální metody, např.:

```
class TObjektSite2 : public TObjektSite {
   // ...
   // neobsahuje definici metody Vypis
}
TObjektSite2 os2; // Chyba - TObjektSite2 je abstraktní
```

Abstraktní třídu lze použít jako typ formálního parametru funkce předávaného referencí nebo ukazatelem, nikoli hodnotou. Lze ji použít i jako typ výsledku funkce vráceného referencí nebo ukazatelem, nikoli hodnotou.

Abstraktní třída může být potomkem neabstraktní třídy a její čistá virtuální metoda může předefinovat virtuální metodu předka.

V konstruktoru nebo destruktoru abstraktní třídy nelze přímo či nepřímo (prostřednictvím jiných funkcí) volat čisté virtuální metody.

Destruktor může být čistou virtuální metodou, ale takový destruktor není prakticky použitelný. Pokud by destruktor třídy `TObjektSite` byl čistou virtuální metodou, dealokaci ukazatele na `TObjektSite` by překladač označil za chybný, i kdyby ukazatel ve skutečnosti obsahoval adresu potomka třídy `TObjektSite`, např.:

```
TObjektSite* OS = new TBod(100, "aaa");
delete OS[0]; // Chyba – ~TObjektSite() je čistá virtuální metoda
```

Paměťová reprezentace polymorfismu

Pro každou třídu, která obsahuje alespoň jednu virtuální metodu, vytvoří překladač jednu tabulku virtuálních metod (VMT – *virtual method table*), která obsahuje adresy všech virtuálních metod dané třídy. Kromě toho překladač vloží do každé instance dané třídy skrytý atribut, který bude obsahovat adresu VMT. Tento atribut je umístěn na začátku každé instance a jeho inicializace se provede automaticky při konstrukci instance (při volání konstruktoru).

Při volání metody, např. `Vypis` z dříve uvedeného příkladu

```
OS[i]->Vypis();
```

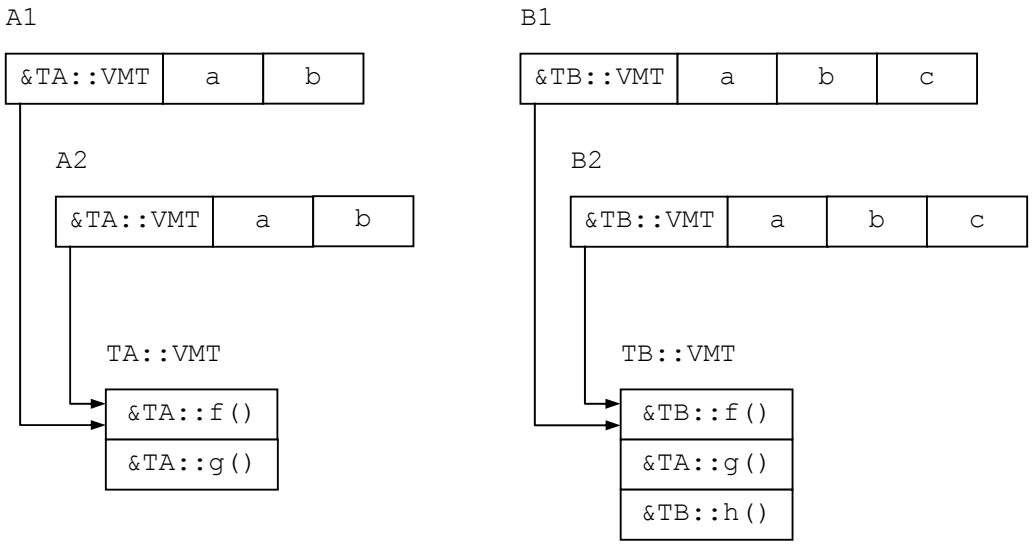
se program nejprve přemístí do instance, na kterou ukazuje `OS[i]` a v ní získá adresu VMT. Na základě adresy VMT se přemístí do VMT a v ní vyhledá adresu metody `Vypis()`, kterou zavolá.

Při volání virtuální metody z konstruktoru nebo z destruktoru se pozdní vazba neuplatňuje. Nejprve se zkonstruují předkové a potom odvozená třída. Nejprve konstruktor předka vloží adresu své VMT (předka) a teprve potom konstruktor potomka tento odkaz přepíše adresou své VMT (potomka). Opačný postup se provede při destrukci.

Na obr. 1 je znázorněna paměťová reprezentace následujících polymorfních tříd, jejich instancí a VMT:

```
class TA {
   int a, b;
public:
   virtual void f();
   virtual void g();
};
TA A1, A2;
```

```
class TB : public TA {
   int c;
public:
   virtual void f();
   virtual int h();
};
TB B1, B2;
```



Obr. 1 Paměťová reprezentace polymorfních tříd

Velikost instancí polymorfních tříd je o 4 byty větší než nepolymorfních. Výsledkem výrazu `sizeof A1` resp. `sizeof(TA)` je tedy hodnota 12.

TŘÍDNÍ UKAZATELE

S atributy tříd lze pracovat pomocí obyčejných ukazatelů, např.:

```
class TA {
public:
   int x;
   int f();
   // ...
};
TA A;
int *ux;
ux = &A.x; // OK
```

S nestatickými metodami však takto pracovat nelze:

```
int (*uf)();
uf = &A.f; // Chyba
uf = A.f; // Chyba
```

V C++ však existují i tzv. *třídní ukazatele* neboli *ukazatele do třídy* (angl. *pointers to members*). Ty umožňují pracovat s adresami atributů i metod. Syntaxe deklarace třídního ukazatele je následující:

deklarace_třídního_ukazatele:

*deklarační_specifikátory jméno_třídy***::***deklarátor***;**

Třídnímu ukazateli lze přiřadit výsledek operátoru získání adresy `&`, jehož operandem je identifikátor atributu nebo metody, kvalifikovaný jménem třídy. Tím se získá ukazatel na složku libovolné instance dané třídy.

Třídní ukazatel obsahuje relativní adresu určité složky třídy vzhledem k začátku instance. Vnitřní struktura třídních ukazatelů je však zpravidla složitější.

Příklad

Je dána třída `TA` a její instance:

```
class TA {
public:
   int x, y;
   int f();
   // ...
};
TA A1, A2;
TA *UA2 = &A2;
```

Deklarace třídního ukazatele `ui` na atribut typu `int` třídy `TA` a třídního ukazatele `uf` na metodu bez parametrů vracející typ `int` třídy `TA` je následující:

```
int TA::*ui;
int (TA::*uf)();
```

Deklaraci třídního ukazatele lze spojit s inicializací, např. `ui2` bude ukazovat na atribut `x` třídy `TA`:

```
int TA::*ui2 = &TA::x;
```

Přiřazení hodnoty do ukazatele `ui` a `uf` mimo deklaraci se provede následovně:

```
ui = &TA::y; // operátor & se musí uvést
uf = TA::f; // dtto uf = &TA::f;
```

Dereference třídního ukazatele se musí vztáhnout ke konkrétní instanci. K tomu slouží operátory dereferencování třídních ukazatelů s následující syntaxí:

*instance***.***třídní_ukazatel*

*ukazatel_na_instanci***->***třídní_ukazatel*

Operátor `.*` se použije pro instanci, operátor `->*` pro ukazatel na instanci.

Např. lze uložit hodnotu do atributu `y` instance `A1` a `A2` prostřednictvím ukazatele `ui`:

```
A1.*ui = 10; // dtto A1.y = 10
A2.*ui = 15; // dtto A2.y = 15
UA2->*ui = 20; // dtto UA2->y = 20 resp. A2.y = 20
```

Obdobně lze zavolat metodu `f` libovolné instance třídy `TA` pomocí ukazatele `uf`:

```
int i1 = (A1.*uf)();
int i2 = (UA2->*uf)();
```

Pomocí třídních ukazatelů nelze pracovat se statickými složkami třídy.

UNIE

Unie není plnohodnotný objektový typ. Omezení jsou následující:

- Unie nemůže být předkem ani potomkem jiného objektového typu.
- Složky unie, pro než nejsou specifikována přístupová práva, jsou veřejně přístupné, podobně jako složky struktury.
- Metody unie nemohou být virtuální.
- Atributem unie nesmí být objektový typ s konstruktorem, destruktorem nebo přetíženým operátorem =, ale unie sama může mít konstruktor, destruktor nebo přetížený operátor =.
- Anonymní unie nesmí mít metody a statické složky.

PŘETĚŽOVÁNÍ OPERÁTORŮ

Základní pravidla

Jazyk C++ umožňuje rozšířit definici většiny operátorů na objektové a výčtové typy. Nelze tedy zavést nové operátory a nelze také změnit význam existujících operátorů pro vestavěné typy (s výjimkou operátorů `new` a `delete`). Nelze také změnit prioritu a asociativitu operátorů. Při přetěžování nelze ani změnit počet operandů (s výjimkou operátorů `new` a `delete`).

Operátory se z hlediska přetěžování rozdělují do 4 skupin:

1. Operátor podmíněného výrazu `?:`, rozlišovací operátor `::`, operátor přímé kvalifikace `.` (tečka), dereferencování třídních ukazatelů `.*`, `sizeof`, `typeid`, `dynamic_cast`, `static_cast`, `reinterpret_cast` a `const_cast` nelze přetěžovat.
2. Operátor indexování `[]`, volání funkce `()`, přiřazení `=`, nepřímé kvalifikace `->` a přetypování `(`*typ*`)` lze přetěžovat pouze jako nestatické metody objektových typů.
3. Operátory `new` a `delete` lze přetěžovat jako obyčejné funkce nebo jako statické metody objektových typů.
4. Všechny ostatní operátory lze přetěžovat jako nestatické metody objektových typů nebo jako obyčejné funkce, které mají alespoň jeden parametr objektového nebo výčtového typu.

Přetížený operátor může změnit svůj význam, který má pro vestavěné typy, např. operátor `+` lze definovat jako násobení apod., ale zpravidla je snahou neměnit smysl přetíženého operátoru, aby se dal odhadnout jeho výsledek.

Přetížený operátor se deklaruje jako tzv. *operátorová funkce.* Oproti jiným funkcím se liší svým jménem, které má následující syntaxi:

jméno_operátorové_funkce:

operator *symbol_operátoru*

symbol_operátoru: jeden z

`new`	`-`	`|`	`-=`	`<<`	`>=`	`->`
`delete`	`*`	`~`	`*=`	`>>`	`&&`	`()`
`new []`	`/`	`!`	`/=`	`<<=`	`||`	`[]`
`delete []`	`%`	`=`	`%=`	`>>=`	`++`	
`+`	`^`	`<`	`^=`	`==`	`--`	
	`&`	`>`	`&=`	`!=`	`,`	
		`+=`	`|=`	`<=`	`->*`	

Pro parametry operátorové funkce nelze předepisovat implicitní hodnoty.

Unární operátor se definuje jako obyčejná funkce s jedním parametrem nebo jako metoda bez parametrů. Obdobně binární operátor se definuje jako obyčejná funkce se dvěma parametry nebo jako metoda s jedním parametrem.

Přetížený operátor lze použít stejným způsobem jako operátor původní nebo jej lze volat zápisem operátorové funkce: jménem operátorové funkce, za kterým následují parametry v kulatých závorkách:

```
TKomplexCislo z = a + b;
TKomplexCislo z = a.operator +(b);
```

Pokud existuje více verzí téhož operátoru, použijí se k jejich rozlišení stejná pravidla jako v případě přetížených funkcí, přičemž se při jejich rozlišení uvažují jak operátory definované jako obyčejné funkce nebo metody, tak i operátory pro vestavěné typy.

Skupina 4 – ostatní operátory

Unární operátory

Unární operátory 4. skupiny se přetěžují definováním:

- obyčejné funkce s jedním parametrem objektového nebo výčtového typu, nebo
- nestatické metody dané třídy bez parametrů – operandem je instance dané třídy.

Je-li `@` nějaký unární operátor a `A` instance třídy, mohou zápisy `@A` resp. `A@` být interpretovány jako:

```
A.operator @()
```

nebo

```
operator @(A)
```

podle toho, která z variant byla deklarována. Najde-li překladač obě varianty, rozliší je podle typu operandu.

Příklad

Je definována třída `TBitKalendar` představující bitovou mapu nějakého kalendáře. Prvek `Kal[i]` představuje dny *i*-tého týdne. Dny jsou v prvku pole bitově zakódované – 1. bit představuje pondělí, 2. bit úterý atd. Pro tuto třídu je definován unární operátor ~, který vrací bitový doplněk kalendáře, tj. dny uvedené v původním kalendáři nebudou obsaženy ve výsledném kalendáři a naopak.

```
class TBitKalendar {
   enum { PocTydnu = 52 };
   uint8_t Kal[PocTydnu];
public:
   TBitKalendar() { memset(Kal, 0, sizeof Kal); }
   TBitKalendar operator ~() const; // doplnek
};

TBitKalendar TBitKalendar::operator ~() const
{
   TBitKalendar t;
   for (int i = 0; i < PocTydnu; i++) t.Kal[i] = ~Kal[i];
   return t;
}
TBitKalendar A, B;
```

Bitový doplněk kalendáře `A` lze uložit do kalendáře `B` jedním z těchto způsobů:

```
B = ~A;
B = A.operator ~();
```

Operátor ~ nemění instanci, na kterou je použit, a proto může být deklarován jako konstantní metoda. Tak funguje tento operátor i pro vestavěné typy. Nicméně lze definovat navíc i operátor ~, který změní instanci, která je jeho operandem.

```
class TBitKalendar {
   // ...
   TBitKalendar& operator ~(); // nestandardní verze
};
TBitKalendar& TBitKalendar::operator ~()
{
   for (int i = 0; i < PocTydnu; i++) Kal[i] = ~Kal[i];
   return *this;
}
```

Potom lze změnit kalendář `A` na doplněk zápisem:

```
~A;
```

Operátory inkrementace ++ a dekrementace –-

Pro tyto operátory platí odlišná pravidla než pro jiné unární operátory této skupiny. Lze totiž přetížit jejich prefixovou i postfixovou verzi.

Prefixový operátor `++` nebo `--` se deklaruje jako obyčejná funkce s jedním parametrem nebo jako metoda bez parametrů.

Postfixový operátor `++` nebo `--` se deklaruje jako obyčejná funkce se dvěma parametry, z nichž druhý je typu `int`, nebo jako metoda s jedním parametrem typu `int`. Parametr typu `int` slouží pouze k rozlišení prefixové a postfixové verze a nelze jej v operátorové funkci použít.

Přetížený i standardní prefixový operátor `++` nebo `--` se zapisuje před operand, zatímco postfixový za něj. Chování prefixové a postfixové verze přetíženého operátoru však může být odlišné od chování standardních operátorů inkrementace a dekrementace, ale není to vhodné.

Příklad

Je definován výčtový typ `TDen` a pro něj prefixový a postfixový operátor inkrementace, který změní příslušný den na následující den.

```
enum TDen { pondeli, utery, streda, ctvrtek, patek, sobota, nedele };

TDen operator ++ (TDen& Den) // prefixový operátor
{
   int d = Den + 1;
   return Den = (d == nedele+1) ? pondeli : static_cast<TDen>(d);
}

TDen operator ++ (TDen& Den, int) // postfixový operátor
{
   TDen DenPuv = Den;
   int d = Den + 1;
   Den = (d == nedele+1) ? pondeli : static_cast<TDen>(d);
   return DenPuv;
}
```

Po provedení následujících příkazů bude v proměnné `d` i `d2` hodnota `ctvrtek`:

```
TDen d = streda;
TDen d2 = ++d; // prefixový operátor
```

Po provedení následujících příkazů bude v proměnné `d` hodnota `nedele` a v `d2` hodnota `pondeli`:

```
d2 = nedele;
d = d2++; // postfixový operátor
```

Volání operátorové funkce pro postfixovou verzi operátoru inkrementace by vypadalo takto:

```
d = operator ++(d2, 0); // dtto d = d2++
```

Druhý parametr může být libovolné celé číslo, které se vejde do typu `int`.

Binární operátory

Binární operátory 4. skupiny se přetěžují definováním:

- obyčejné funkce se dvěma parametry, z nichž alespoň jeden je objektového nebo výčtového typu, nebo
- nestatické metody objektového typu s jedním parametrem.

Obyčejné funkci bude předán levý operand jako první parametr a pravý operand jako druhý parametr. U metody je levým operandem instance, pro níž se tato metoda zavolá, a pravým operandem je parametr této metody.

To znamená, že je-li `@` nějaký binární operátor, může překladač interpretovat zápis `A @ B` jako:

```
A.operator @(B)
```

nebo

```
operator @(A, B)
```

Pokud jsou deklarovány obě varianty, překladač je rozliší podle typu parametrů.

Přetížení binárních operátorů `+`, `-` a dalších neznamená automatické přetížení složených přiřazovacích operátorů `+=`, `-=` atd. Tyto operátory se musí přetížit samostatně a mohou mít jiný význam než by plynulo z významů jejich dílčích operátorů. Je ovšem rozumné definovat složené přiřazovací operátory tak, aby jejich chování odpovídalo významu jejich dílčích operátorů.

Složené přiřazovací operátory lze přetěžovat jako metody i jako obyčejné funkce, ale prostý operátor přirazení `=` lze přetížit jen jako metodu.

Příklad

Je definována třída `TMatice` s těmito atributy: `m` – počet řádků matice, `n` – počet sloupců, `a` – dynamické pole ukazatelů na dynamická pole. Kromě konstruktorů a destruktoru, jejichž definice není zde uvedena, obsahuje tato třída operátorovou funkci pro násobení matice reálným číslem. Dále je definována instance `A` třídy `TMatice`, jejíž prvky jsou inicializovány hodnotou `3`.

```
class TMatice {
   int    m, n;
   double **a;
public:
   TMatice(int _n = 0, int _m = 0, double c = 0);
   TMatice(const TMatice& t);
   ~TMatice();
   TMatice operator* (const double c) const; // násobení matice číslem
}
TMatice TMatice::operator* (const double c) const
{
   int     i, j;
   TMatice t(*this);
   for (i = 0; i < m; i++) for (j = 0; j < n; j++) {
      t.a[i][j] *= c;
   }
   return t;
}
TMatice A(5, 2, 3);
```

Operátor násobení nemění své operandy, a proto je definován jako konstantní metoda. V těle tohoto operátoru je nejprve definována pomocná instance matice `t`, která obsahuje kopii levého operandu (instance `*this`) a potom jsou všechny její prvky vynásobeny pravým operandem (formální parametr `c`).

Použití operátoru násobení může být následující:

```
TMatice B = A*2;
```

nebo

```
TMatice B = A.operator * (2);
```

V obou případech se při vracení výsledku operátoru volá jedenkrát kopírovací konstruktor třídy `TMatice`. Prvky matice `B` budou mít hodnotu `6`.

Operace násobení matice číslem je komutativní, tj. levým operandem může být číslo a pravým operandem matice nebo naopak. Následující zápis však překladač vyhodnotí jako chybu:

```
TMatice B = 2*A;
```

Důvodem je skutečnost, že není definován operátor, který by měl levý operand číslo a pravý operand matici. Takovýto operátor ale nemůže být metoda (levý operand není objektový typ), musí to být obyčejná funkce, např. vložená funkce, která volá metodu operátoru násobení:

```
inline TMatice operator* (const double c, const TMatice& t)
{
   return t*c;
}
```

Skupina 2 – operátory přetěžované jen jako metody

Každý z operátorů této skupiny má svá specifika, proto jsou popsány samostatně.

Operátor přiřazení =

Operátor přiřazení se nedědí.

Přetížený (uživatelem deklarovaný) operátor přiřazení je nestatická metoda mající tvar:

```
operator = (parametr)
```

kde *parametr* je jeden parametr libovolného typu. Typ vrácené hodnoty může být libovolného typu.

Uživatelem deklarovaný kopírovací operátor přiřazení (angl. *user-declared copy assignment operator)* třídy `TA` je přiřazovací operátor, který má parametr typu `TA`, `TA&`, `const TA&`, `volatile TA&` nebo `const volatile TA&`.

Pokud v dané třídě není uživatelem deklarovaný kopírovací operátor přiřazení, překladač zpravidla vytvoří *implicitně deklarovaný kopírovací operátor přiřazení* (angl. *implicitly-declared copy assignment operator*), který je vložený a veřejně přístupný. Pro třídu `TA` by jeho prototyp byl následující:

```
TA& TA::operator = (const TA&);
```

nebo

```
TA& TA::operator = (TA&);
```

podle okolností.

Implicitně deklarovaný kopírovací operátor přiřazení vrací referenci na jeho levý operand, tj. na instanci, do níž se přiřazuje instance jeho parametru.

Implicitně deklarovaný kopírovací operátor přiřazení dané třídy je *implicitně definovaný* (angl. *implicitly-defined*), jestliže je použit k přiřazení do jedné instance dané třídy z instance stejné nebo odvozené třídy.

Implicitně definovaný kopírovací operátor přiřazení třídy `TA` volá kopírovací operátory přiřazení (implicitně nebo uživatelem deklarované) nejprve pro přímé předky třídy `TA` v pořadí jejich uvedení v definici třídy `TA` a potom pro nestatické atributy třídy `TA`.

Překladač nemůže vytvořit implicitně definovaný kopírovací operátor přiřazení pro třídu, která má:

- nestatický konstantní atribut,
- nestatický referenční atribut,
- nestatický atribut objektového typu, který nemá veřejně přístupný kopírovací operátor přiřazení, nebo
- předka, jenž nemá veřejně přístupný kopírovací operátor přiřazení.

Aby bylo možné přetížený operátor přiřazení zřetězit (tj. aby vytvářel l-hodnotu), musí vracet referenci na jeho levý operand podobně jako implicitně deklarovaný operátor přiřazení.

V dané třídě může být deklarováno více přetížených operátorů přiřazení, lišících se typem formálního parametru. Implicitně deklarovaný kopírovací operátor přiřazení překladač vytvoří i v případě, že v třídě `TA` jsou deklarovány nějaké operátory přiřazení, ale ani jeden z nich není kopírovacím operátorem.

Přetížený operátor přiřazení nemění význam operátorů `+=`, `*=` atd. Tyto složené operátory se musí přetížit zvlášť.

Kopírovací operátor přiřazení má smysl definovat, pokud se jedná o třídu, pro níž nemohl překladač vytvořit implicitní kopírovací operátor přiřazení nebo pokud obsahuje dynamicky alokovaný atribut.

Příklad

Třída `TMatice`, definovaná dříve, má atribut `a`, který obsahuje adresu dynamicky alokované matice. Pokud by se kopírovací operátor přiřazení nepřetížil, příkaz

```
B = A;
```

by pro instance `A` a `B` třídy `TMatice` mj. provedl zkopírování atributu `a` příkazem:

```
B.a = A.a;
```

v důsledku čehož by atribut `B.a` ukazoval na stejnou matici jako atribut `A.a` a původní hodnota atributu `B.a` by se ztratila, tj. nedošlo by k dealokaci paměti, na kterou atribut `B.a` případně ukazoval.

Toto chování není zpravidla žádoucí, a proto se musí kopírovací operátor přiřazení definovat.

```
class TMatice {
   // ...
   TMatice& operator = (const TMatice& t);
};
TMatice& TMatice::operator = (const TMatice& t)
{
   if (this != &t) {
      this->~TMatice();
      m = t.m;
      n = t.n;
      a = new double*[m];
      for (int i = 0; i < m; i++) {
         a[i] = new double[n];
         memcpy(a[i], t.a[i], n*sizeof(double));
      }
   }
   return *this;
}
```

Pokud levý i pravý operand představuje tutéž instanci, přiřazení se v uvedené metodě neprovede. V opačném případě se nejprve dealokuje matice instance, na níž ukazuje `this` explicitním voláním jejího destruktoru a potom se provede vlastní zkopírování, které je shodné s příkazy uživatelem

definovaného kopírovacího konstruktoru. V takovémto případě by bylo vhodné definovat metodu pro kopírování, která by se volala jak z kopírovacího konstruktoru, tak i z operátoru přiřazení.

Takovýto operátor přiřazení by bylo možné zřetězit, např. přiřazení instance `A` třídy `TMatice` do instance `B` a `C` třídy `TMatice` by se mohlo provést příkazem

```
C = B = A;
```

Nejprve se provede přiřazení `B = A` a potom `C = B`.

Operátor indexování `[]`

Operátor indexování je binární – levým operandem je instance objektového typu, pravým operandem je index zapsaný mezi závorky `[]`. Přetížený operátor indexování je nestatická metoda mající tvar:

```
operator [] (parametr)
```

kde *parametr* je jeden parametr libovolného typu.

Je-li `A` instance třídy, znamená zápis

```
A[i]
```

totéž co

```
A.operator[](i)
```

Příklad 1

Je definována třída `TIntArr` obsahující dynamické pole celých čísel. Její součástí je operátor indexování, sloužící k zpřístupnění prvku pole zadaného indexu.

```
class TIntArr {
   int *a, n;
public:
   TIntArr(int _n) : n(_n) { a = new int[n]; }
   ~TIntArr() { delete[] a; }
   int& operator [] (int Index);
};
int& TIntArr::operator [] (int Index)
{
   if (Index < 0 || Index >= n) Chyba();
   return a[Index];
}
```

Aby mohl být operátor indexování použit i na levé straně přiřazení (jako l-hodnota), musí vracet referenci na typ prvku. Tím se mohou měnit data instance této třídy, a proto nemůže být operátor deklarován jako konstantní metoda.

Použití operátoru pro instanci `A` může být např. následující:

```
TIntArr A(10);
A[0] = 1;
```

Vedle uvedeného operátoru indexování lze deklarovat ještě jeden operátor indexování, který se použije pro konstantní instance:

```
class TIntArr {
   // ...
   int operator [] (int Index) const;
};
int TIntArr::operator [] (int Index) const
{
   if (Index < 0 || Index >= n) Chyba();
   return a[Index];
}
```

Tento operátor vrací výsledek hodnotou (r-hodnota), a proto nemůže způsobit změnu dat instance této třídy. Pro konstantní instanci lze operátor indexování použít jen na pravé straně přiřazení:

```
const TIntArr B(5);
A[1] = B[0]; // OK
B[0] = 2;    // Chyba
```

Pro instanci `A` se zavolá nekonstantní metoda, pro instanci `B` konstantní metoda operátoru indexování.

Překladač neoznámí chybu, pokud konstantní metoda operátoru indexování bude také vracet nekonstantní referenci na typ `int`, ale potom by tato metoda ztrácela svůj význam.

Konstantní metoda operátoru indexování by mohla vracet konstantní referenci na typ prvku pole, zejména v případě, kdy prvky pole jsou objektového typu.

Příklad 2

Pro třídu `TMatice`, definovanou dříve, lze také definovat operátor indexování. Ten zpřístupní pole prvků na zadaném řádku:

```
class TMatice {
   // ...
   double* operator[] (int Index) { return a[Index]; }
};
TMatice A(5, 2, 3);
```

Operátor by mohl být použit např. takto:

```
A[i][j] = 11;
```

Pro instanci `A` se nejprve zavolá přetížený operátor indexování, který vrátí ukazatel na pole *i*-tého řádku. To je hodnota typu `double*`, pro níž se použije standardní operátor indexování, který zpřístupní *j*-tý prvek v řádku. V tomto operátoru indexování nelze kontrolovat překročení hranice pro index sloupce.

Konstantní metoda operátoru indexování pro matici by mohla být definována následovně:

```
class TMatice {
   // ...
   const double* operator[] (int Index) const { return a[Index]; }
};
```

Operátor volání funkce `()`

Přetížený operátor volání funkce je nestatická metoda mající tvar

`operator ()` (*seznam_parametrů*$_{nep}$)

kde *seznam_parametrů* je libovolný počet parametrů libovolného typu.

Je-li `A` instance třídy, znamená zápis

```
A(x, y)
```

totéž co

```
A.operator()(x, y)
```

Příklad

Ve třídě `TMatice`, definované dříve, lze použít operátor volání funkce pro zpřístupnění prvku matice:

```
class TMatice {
   // ...
   double& operator() (int i, int j);
};
TMatice A(5, 2, 3);

double& TMatice::operator() (int i, int j)
{
   if (i < 0 || i >= m || j < 0 || j >= n) Chyba();
   return a[i][j];
}
```

Operátor volání funkce vrací referenci, aby mohl být použit na levé straně přiřazovacího příkazu.

Přiřazení hodnoty `10` do prvku matice `A` na *i*-tém řádku v *j*-tém sloupci se provede zápisem:

```
A(i, j) = 10;
```

Pro tento operátor by se mohla definovat i konstantní metoda, vracející hodnotu typu `double`.

PŘETĚŽOVÁNÍ OPERÁTORŮ

Skupina 2 – operátory přetěžované jen jako metody – pokračování

Operátor nepřímé kvalifikace ->

Tento operátor se pro účely přetěžování považuje za unární, to znamená, že operátorová funkce je nestatická metoda bez parametrů, mající tvar:

```
operator -> ()
```

Je-li `A` instance třídy, bude překladač interpretovat zápis `A->m` jako

```
(A.operator -> ()) -> m
```

To znamená, že operátor `->` musí vracet buď ukazatel na nějakou třídu nebo instanci jiné třídy, v níž je tento operátor také přetížen.

Příklad

Třída `TAutoPtrMat` zapouzdřuje ukazatel na dynamicky alokovanou matici třídy `TMatice` a zajišťuje její automatickou dealokaci definováním destruktoru. Pro přístup k ukazateli slouží operátor nepřímé kvalifikace.

```
class TAutoPtrMat {
   TMatice* Matice;
public:
   TAutoPtrMat(TMatice* t) : Matice(t) {}
   ~TAutoPtrMat() { delete Matice; }
   TMatice* operator ->() { return Matice; }
   // ...
};
void f() {
   TAutoPtrMat A(new TMatice(5, 3, 0));
   A->Vypis();
}
```

Ve funkci `f()` se nejprve definuje instance `A` obsahující ukazatel na dynamicky alokovanou matici. Potom se prostřednictvím přetíženého operátoru nepřímé kvalifikace volá metoda `Vypis()` pro dynamickou instanci `A.Matice`. Matice se ukončením funkce `f()` automaticky dealokuje.

Operátor přetypování

Tato operátorová funkce definuje konverzi objektového typu, v němž je deklarována, na jakýkoli jiný typ. Metoda je bez parametrů, bez návratového typu a její jméno se skládá z klíčového slova `operator` a z označení cílového typu. Cílový typ se může skládat z několika slov a může obsahovat i znaky `*` nebo `&`, např.:

```
operator char* ();
operator long double();
```

Deklaruje-li se ve třídě `TA` operátor přetypování na typ `TB`, překladač použije tento operátor všude tam, kde očekává hodnotu typu `TB` a kde najde hodnotu typu `TA`. Použije ji také při explicitním přetypování, např. `(TB)A` nebo `static_cast<TB>(A)`, kde `A` je instance typu `TA`.

Příklad

```
class TB {
   long i;
public:
   TB(long _i) : i(_i) {}
   // ...
};
class TA {
   int i;
public:
   TA(int _i) : i(_i) {}
   operator TB() { return TB(i); }
   operator int() { return i; }
};
void f(TB B);
```

Je-li `A` instance třídy `TA`, funkci `f` lze volat kterýmkoli z následujících zápisů:

```
f(static_cast<TB>(A));
f((TB)A);
f(TB(A));
f(A);
```

Ve všech případech se nejprve zavolá metoda `A.operator TB()`, jejíž výsledek se předá do funkce `f`.

Podobně lze napsat příkaz:

```
cout << A;
```

který vypíše hodnotu atributu `A.i`, protože se zavolá metoda `A.operator int()` a její výsledek se použije pro operátor << určený pro výstup hodnot typu `int`.

Skupina 3 – operátory new a delete

Operátory pro práci s pamětí `new`, `delete`, `new[]` a `delete[]` jsou jediné, které lze nejen přetěžovat, ale i předefinovat, tj. lze nahradit jejich standardní verzi svojí vlastní verzí. Tyto operátory lze definovat jako:

a) obyčejné funkce,
b) statické metody objektového typu.

Obyčejné operátorové funkce

Použije-li se v programu operátor `new` (pro alokaci jednoduché proměnné) bez parametru `(nothrow)`, překladač zavolá standardní funkci

```
void* operator new(size_t size) throw(bad_alloc);
```

Použije-li se v programu operátor `new` s parametrem `(nothrow)`, překladač zavolá standardní funkci

```
void* operator new(size_t size, const nothrow_t&) throw();
```

Specifikace `throw(bad_alloc)` a `throw()` představuje specifikaci výjimek, které se mohou z dané funkce rozšířit. Bližší informace viz přednášky v dalším semestru.

Použije-li se v programu operátor `new[]` (pro alokaci pole), překladač zavolá jednu z těchto standardních funkcí, podle toho, zda se použije parametr `(nothrow)`:

```
void* operator new[](size_t size) throw(bad_alloc);
void* operator new[](size_t size, const nothrow_t&) throw();
```

Parametr `size` ve všech uvedených funkcích obsahuje velikost požadované paměti v bajtech. O jeho předání se postará překladač.

Použije-li se v programu operátor `new` nebo `new[]` s parametrem umístění (udává adresu, od které se má alokovat), překladač zavolá jednu z těchto standardních funkci:

```
void* operator new(size_t size, void* ptr) throw();
void* operator new[](size_t size, void* ptr) throw();
```

Tyto dvě funkce nelze předefinovat.

Použije-li se v programu operátor `delete` (pro dealokaci jednoduché proměnné), překladač zavolá standardní funkci

```
void operator delete(void* ptr) throw();
```

Použije-li se v programu operátor `delete[]` (pro dealokaci pole), překladač zavolá standardní funkci

```
void operator delete[](void* ptr) throw();
```

Pokud je v programu definována vlastní funkce, mající některý z uvedených prototypů, použije ji překladač místo odpovídající standardní funkce, a to již od začátku běhu programu. Specifikace `throw(bad_alloc)` a `throw()` může být v předefinované funkci vynechána.

Operátory `new`, `new[]` lze přetížit, tj. deklarovat v programu jejich další verze, lišící se počtem a typem parametrů, přičemž první parametr musí být typu `size_t` a návratový typ musí být `void*`. Dodatečné parametry přetíženého operátoru se při jeho volání zapisují do kulatých závorek za klíčové slovo `new`, podobně jako při použití parametru s umístěním.

Příklad

V uvedeném příkladu jsou předefinovány operátory `new[]` a `delete[]`. Předefinovaný operátor `new[]` narozdíl od standardní verze inicializuje přidělenou paměť hodnotou 0 a v případě neúspěchu vrací hodnotu 0. Pokud se předefinuje operátor `new[]`, měl by se předefinovat i operátor `delete[]`.

```
#include <stdlib.h>
#include <memory.h>

void* operator new[](size_t size)
{
   void *ptr = malloc(size);
   if (ptr) memset(ptr, 0, size);
   return ptr;
}

void operator delete[](void* ptr)
{
   free(ptr);
}
```

```
void f()
{
   int *a = new int[10]; // volá předefinovanou verzi
   // ...
   delete [] a; // volá předefinovanou verzi
}
```

Chování uvedeného operátoru `new[]` se liší od chování předepsaného pro standardní verzi tohoto operátoru. Standardní operátor v případě:

a) nulové hodnoty parametru `size` vrací adresu, která bude různá od adres ostatních objektů;

b) v případě neúspěšné alokace zavolá funkci, jejíž adresu obsahuje ukazatel typu `new_handler` (ukazatel se nastavuje pomocí funkce `set_new_handler`). Je-li tento ukazatel nulový, vyvolá výjimku `bad_alloc`. Ukazatel na funkci typu `new_handler` je globální proměnná, jejíž jméno může být v jednotlivých překladačích odlišné.

Pro splnění chování ad a) se může pro nulovou hodnotu parametru `size` alokovat 1 bajt.

Pro splnění chování ad b) se musí kromě úpravy operátorové funkce `new[]` definovat globální proměnná typu `new_handler`, která se bude nastavovat v předefinované funkci `set_new_handler`.

Příklad

```
#include <stdlib.h>
#include <memory.h>
#include <conio.h>
#include <new>
#include <iostream>
using namespace std;

static new_handler new_hand;
// proměnná new_hand je přístupná pouze v tomto souboru

new_handler set_new_handler(new_handler my_handler)
{
   new_handler predchozi = new_hand;
   new_hand = my_handler;
   return predchozi;
}

void* operator new[](size_t size)
{
   void *ptr;
   if (!size) size = 1;
   while ((ptr = malloc(size)) == 0 && new_hand) new_hand();
   if (ptr) memset(ptr, 0, size);
   return ptr;
}

void operator delete[](void* ptr)
{
   free(ptr);
}
```

```
void my_handler()
{
   cout << "Nedostatek pameti";
   getch();
   exit(1);
}

int f()
{
   int *a = new int[1000000000]; // vrací 0
   delete[] a;
   ::set_new_handler(my_handler);
     // pomocí operátoru :: se volá předefinovaná verze
   int *b = new int[1000000000];  // volá my_handler
   delete[] b;
}
```

Pokud by měl operátor `new[]` inicializovat alokovanou paměť zadanou hodnotou, musí se přetížit, např. takto:

```
void* operator new[](size_t size, unsigned char c)
{
   void *ptr;
   if (!size) size = 1;
   while ((ptr = malloc(size)) == 0 && new_hand) new_hand();
   if (ptr) memset(ptr, c, size);
   return ptr;
}
```

Předefinovaný operátor `new[]` by potom mohl volat přetíženou verzi. Hodnota 0 se musí přetypovat na typ parametru `c`, jinak překladač nedokáže rozlišit, kterou verzi operátorové funkce má zavolat, zda s parametrem typu `void*` (parametr umístění), nebo s parametrem typu `unsigned char`.

```
void* operator new[](size_t size)
{
   return operator new[](size, static_cast<unsigned char>(0)));
}
```

Alokace pole s inicializací paměti hodnotou 255 lze provést např. zápisem

```
int *a = new (255) int[10];
```

Operátory new a delete jako metody

Operátory `new`, `new[]`, `delete` a `delete[]` lze také definovat jako metody objektových typů. Jedná se o statické metody, i když není v jejich deklaraci uvedeno klíčové slovo `static`.

Pro deklaraci operátorů `new` a `new[]` platí následující požadavky:

- musí vracet hodnotu typu `void*`
- jejich první parametr musí být typu `size_t`
- případné další parametry mohou být libovolného typu.

Deklarace operátorů `delete` a `delete[]` musí splňovat následující požadavky:

- musí vracet hodnotu typu `void`
- jejich první parametr musí být typu `void*`

- mohou mít ještě jeden parametr typu `size_t`.

Tyto operátory slouží pro alokaci a uvolňování instancí nebo polí objektového typu, v němž jsou deklarovány.

Příklad

V programu je potřeba velmi mnoho instancí třídy `TA`, ale každou z nich jen krátkou dobu. Proto se alokují dynamicky za sebou do předem připraveného pole `bazen`. Neprovádí se jejich dealokace, nýbrž ve chvíli, kdy se dojde s alokací na konec bazénu, začne se znovu od jeho počátku. Nové instance tím budou přepisovat staré. Předpokládá se, že tou dobou již přepisované instance nebudou v programu potřebné.

```
class TA {
   int x; // nějaká data
   enum { n = 1000 }; // velikost bazénu
public:
   void* operator new(size_t size);
   void operator delete(void* ptr) {}
};

void* TA::operator new(size_t size)
{
   static char bazen[n]; // oblast pro alokaci
   static int  pos = 0;  // ukazatel na aktuální pozici
   int i = (pos+size > n) ? 0 : pos;
   pos = i+size;
   return &bazen[i];
}
```

Proměnná `bazen` je definována jako lokální statická, čímž se zabraňuje jejímu zneužití k jiným účelům. Lokální statická proměnná `pos` obsahuje ukazatel na aktuální pozici v bazénu, od které by se měla alokovat příští instance.

Příkaz

```
TA* A = new TA;
```

způsobí volání metody `TA::operator new`.

Příkazem

```
TA* A = ::new TA;
```

se však zavolá obyčejná operátorová funkce `new`, ať již standardní nebo předefinovaná.

Pokud se provede alokace pole instancí třídy `TA`

```
TA* A = new TA[3];
```

překladač použije obyčejnou operátorovou funkci `new[]`. Aby se v tomto případě volala metoda třídy `TA`, musí se ve třídě `TA` definovat ještě operátor `new[]` a operátor `delete[]` např. takto:

```
class TA {
   // ...
   void* operator new[](size_t size) { return operator new(size); }
   void operator delete[](void* ptr) {}
};
```

Operátory `delete` a `delete[]` třídy `TA` nic nedělají. Jsou definovány proto, aby se při případné omylem provedené dealokaci instance třídy `TA` nevolala obyčejná operátorová funkce `delete` resp. `delete[]`.

Operátory `delete` a `delete[]` mohou být v dané třídě deklarovány se dvěma parametry:

```
void operator delete(void*, size_t)
void operator delete[](void*, size_t)
```

Překladač je použije pro dealokaci instance, která byla alokována pomocí operátoru `new` resp. `new[]` s dodatečnými parametry nebo vždy, pokud v dané třídě není definován operátor `delete` resp. `delete[]` s jedním parametrem.

V prostředí Visual C++ 2003 sice mohou být v dané třídě deklarovány dvě verze operátoru `delete` resp. `delete[]`, jedna s parametrem `size_t` a druhá bez tohoto parametru, ale verze s parametrem `size_t` není nikdy volána.

Volání operátorové funkce

Klasický zápis operátoru `new`, např.

```
TB* B = new TB;
```

provede alokaci a volání implicitního konstruktoru třídy `TB`. Výsledkem operátoru `new` je v tomto případě typ `TB*`.

Explicitní volání operátorové funkce `new`, např.

```
TB* B = (TB*)operator new(sizeof(TB));
```

provede alokaci instance třídy `TB` bez volání jejího konstruktoru. Výsledkem je hodnota typu `void*`, která se musí přetypovat.

Podobně zápis

```
delete B;
```

nejen uvolní paměť, ale zavolá i destruktor.

Zápis

```
operator delete (B);
```

dealokuje paměť, ale destruktor se nezavolá.

ŠABLONY

Šablony (angl. *templates*) jsou základem generického programování. *Generické programování* je jeden z programovacích stylů, který je založen na vytváření abstraktních vzorů funkcí a tříd pomocí generických konstrukcí (šablon). Dalšími programovacími styly jsou *strukturované programování* (programování pomocí funkcí, které operují nad daty – neobjektový přístup) a *objektově orientované programování*.

Šablony umožňují popsat najednou celou množinu funkcí, které se liší jen např. typem jejich parametru, nebo množinu objektových typů, které se liší jen např. typem jejich atributu. Mají podobný význam jako makra, ale poskytují více možností. Na rozdíl od maker jsou šablony zpracovávány překladačem.

Šablona představuje vzor, podle kterého překladač vytvoří funkci nebo objektový typ. Takto vytvořená funkce nebo objektový typ se nazývá *instance šablony*.

Deklarace šablony

Deklarace šablony se v programu může objevit na úrovni souboru, uvnitř objektového typu nebo uvnitř šablony objektového typu. Šablona se nemůže deklarovat jako lokální v bloku.

Syntaxe deklarace šablony:

deklarace_šablony:

template<*seznam_parametrů*> *deklarace*

Seznam_parametrů představuje jednotlivé formální parametry šablony oddělené čárkami, podobně jako v deklaraci funkce. Formálními parametry šablony mohou být hodnotové typy (podobně jako v deklaraci funkce), formální typy (typové parametry) nebo šablony objektových typů.

Deklarace je:

- deklarace nebo definice obyčejné funkce,
- deklarace nebo definice třídy,
- definice metody třídy,
- definice vnořené třídy,
- definice statického atributu šablony třídy nebo definice statického atributu třídy vnořené uvnitř šablony třídy,
- definice vnořené šablony.

V deklaraci mohou být některá jména typů a některé konstanty nahrazeny formálními parametry šablony. Deklarace musí končit středníkem, nejde-li o definiční deklaraci funkce nebo metody.

Parametry šablony

Syntaxe formálních parametrů šablony:

parametr_šablony:

hodnotový_parametr

class *identifikátor*

class *identifikátor* **=** *označení_typu*

typename *identifikátor*

typename *identifikátor* **=** *označení_typu*

template<*seznam_parametrů***> class** *identifikátor*

template<*seznam_parametrů***> class** *identifikátor* **=** *jméno_šablony*

Pro všechny druhy parametrů šablony lze předepsat implicitní hodnoty, pokud se nejedná o šablonu obyčejné funkce. Přitom platí, že pokud pro nějaký parametr šablony je předepsána implicitní hodnota, musí být předepsána i pro všechny parametry, které za ním následují.

Implicitní hodnota může být předepsána jen v jedné z deklarací šablony stejného typu:

```
template<class T = int> class TA;
template<class T> class TA { ... }; // OK

template<class T> class TB;
template<class T = int> class TB { ... }; // OK

template<class T = int> class TC;
template<class T = int> class TC { ... }; // Chyba
```

Parametr šablony lze použít v deklaraci následujících parametrů této šablony, např.:

```
template<class T, T hodnota, class U = T> class TA { ... };
```

Typové parametry

Typové parametry se předepisují pomocí klíčového slova `class` nebo `typename`. Obě možnosti znamenají totéž. Např. zápisem

```
template<class T, class V = int> class TA;
```

se deklaruje šablona třídy `TA` se dvěma typovými parametry `T` a `V`. Pro parametr `V` je předepsána implicitní hodnota typu `int`. I když je u typového parametru uvedeno klíčové slovo `class`, může být skutečným parametrem šablony jakýkoli typ, nejen objektový. Tutéž šablonu lze deklarovat i zápisem:

```
template<typename T, typename V = int> class TA;
```

Skutečným parametrem pro typový parametr je označení typu. Nesmí se jednat o označení lokální třídy (deklarované ve funkci).

Hodnotové parametry

Hodnotové parametry šablon se deklarují podobně jako formální parametry funkcí, musí však být jednoho z následujících typů:

- Celočíselný typ. Skutečným parametrem musí být celočíselný konstantní výraz, např.:
  ```
  template<class T, int i> class TA { ... };
  TA<double, 10*5> A;
  ```

- Výčtový typ. Skutečným parametrem musí být celočíselný konstantní výraz, jehož výsledkem je daný výčtový typ, např.:
```
enum TB { b1, b2, b3 };
template<class T, TB b> class TA { ... };
TA<int, b2> A;
```
- Ukazatel na objekt. Skutečným parametrem musí být konstantní výraz představující adresu pojmenovaného objektu s externím linkováním (s paměťovou třídou `extern`). Nesmí to být adresa prvku pole.
- Reference na objekt. Skutečným parametrem musí být l-hodnota představující pojmenovaný objekt s externím linkováním.
- Ukazatel na funkci. Skutečným parametrem musí být výraz představující funkci s externím linkováním, např.:
```
template <class T, T (*f)(T) > class TA { ... };
int f(int x) { ... }
TA<int, f> A;
```
- Třídní ukazatel. Skutečným parametrem musí být konstantní adresový výraz představující pojmenovanou složku třídy, např.:
```
struct TB { int x, y; }
template <class T, class U, T U::*u> class TA { ... };
TA<int, TB, &TB::x> A;
```

Hodnotové parametry se v šabloně chovají jako konstanty, tj. lze je používat v konstantních výrazech (např. ve specifikaci mezí polí).

Hodnotovým parametrem nemůže být typ reálných čísel (`double`, `float`), třída nebo `void`.

V prostředí Visual C++ 2003 nelze použít hodnotový parametr typu třídní ukazatel.

Šablonové parametry

Parametrem šablony může být jiná šablona třídy. Skutečným parametrem v tomto případě musí být jméno existující šablony třídy.

Příklad

```
template <class T, template<class U> class W> class TA {
   W<T> a, b;
public:
   TA(W<T> _a, W<T> _b) : a(_a), b(_b) {}
};

template <class T> struct TB { T x, y; };

TB<int> b1 = { 10, 20 }, b2 = { 30, 40 };
TA<int, TB> A(b1, b2);
```

Třídy

Šablona objektového typu (třídy, struktury nebo unie) (angl. *class template*) může obsahovat stejné druhy složek jako objektový typ, tj. i statické atributy, vložené, statické a virtuální metody, vnořené typy apod.

Příklad

Je deklarována šablona dynamického pole `TVektor`. Prvky pole jsou typu `T`, což může být jakýkoliv typ.

```
template <class T> class TVektor {
   T  *a;
   int n;
   static int PocInstanci;
public:
   TVektor(int _n = 0) : n(_n) { a = new T[n]; PocInstanci++; }
   ~TVektor() { delete[] a; PocInstanci--; }
   T&         operator[](int index) { return a[index]; }
   bool       operator == (const TVektor& t);
   static int GetPocInstanci() { return PocInstanci; }
};
```

Metody

Metody, které nejsou definovány přímo v těle šablony, se musí definovat jako šablony. Šablona metody musí obsahovat stejné formální parametry jako šablona její třídy a v témže pořadí. Jména formálních parametrů ale mohou být odlišná. Jméno metody se musí kvalifikovat jmenovkou třídy, za níž následují v lomených závorkách jména formálních parametrů v témže pořadí.

Příklad

```
template<class T1, class T2, int z> struct TA {
   void f1();
   void f2();
   void f3();
   void f4();
   // ...
};
template<class T1, class T2, int z>
void TA<T1, T2, z>::f1() { ... } // OK

template<class T2, class T1, int z>
void TA<T2, T1, z>::f2() { ... } // OK

template<class T1, class T2, int z>
void TA<T2, T1, z>::f3() { ... } // Chyba

template<class T1, int z, class T2>
void TA<T1, z, T2>::f4() { ... } // Chyba
```

Příklad

Operátorová funkce == šablony `TVektor` bude mít následující definici:

```
template<class T>
bool TVektor<T>::operator == (const TVektor& t)
{
   if (n != t.n) return false;
   for (int i = 0; i < n; i++) {
      if (a[i] != t.a[i]) return false;
   }
   return true;
}
```

Statické atributy

Statické atributy se musí definovat jako šablony. Pro šablonu statického atributu platí stejná pravidla jako pro šablonu metody.

Statický atribut `PocInstanci` šablony `TVektor` se může definovat např. takto:

```
template<class T> int TVektor<T>::PocInstanci = 0;
```

Vnořené třídy

Vnořené třídy definované mimo obklopující šablonu třídy se musí definovat jako šablony, pro které platí stejná pravidla jako pro šablonu metody, např.:

```
template<class T> struct TA {
   struct TB;
   // ...
};
template<class T> struct TA<T>::TB {
   T z;
   T& Pricti(T _z);
};
template<class T>
T& TA<T>::TB::Pricti(T _z)
{
   z += _z;
   return z;
}

TA<int>::TB B; // instance vnořené třídy
B.z = 10;
```

Instance

Instancí šablony třídy je třída, vytvořená podle této šablony, dosazením skutečných parametrů namísto formálních. Jméno typu instance se skládá ze jména šablony a ze skutečných parametrů v lomených závorkách. Přitom parametry, pro které jsou definovány implicitní hodnoty, lze vynechat.

Příklad

```
template <class T = int, int n = 10> class TA { ... };
TA<char, 12> B; // OK
TA<char> A;     // OK - TA<char, 10>
TA<> C;         // OK - TA<int, 10>
TA D;           // Chyba
```

Příklad

Příklady deklarace instance šablony `TVektor`:

```
TVektor<int> A(10), B(20); // pole 10 a 20 celých čísel
TVektor<double> C(10); // pole 10 čísel typu double
TVektor< TVektor<int> > D(10);
// pole 10 prvků typu pole celých čísel
```

Jméno typu instance `A` a `B` je `TVektor<int>`, instance `C` je `TVektor<double>`. Výpis počtu vytvořených instancí pro tyto typy by mohl být následující:

```
cout << TVektor<int>::GetPocInstanci() << '\n';
cout << TVektor<double>::GetPocInstanci() << '\n';
```

Vytvoření instance šablony třídy ještě neznamená vytvoření instancí všech jejích metod a statických atributů. Překladač vytvoří instance těch metod a statických atributů, které jsou pro daný typ v programu použity. Vždy se ale generují virtuální metody instance šablony třídy.

To znamená, že v případě deklarace

```
TVektor<int> A(10), B(20);
```

vytvoří překladač pouze instanci typu `TVektor<int>`, statického atributu `TVektor<int>::PocInstanci`, konstruktoru s jedním parametrem typu `int` a destruktoru.

Dvě instance šablony třídy jsou stejného typu, pokud všechny jejich skutečné typové parametry jsou stejného typu, jejich hodnotové parametry mají stejnou hodnotu a jejich šablonové typy se odkazují na stejnou šablonu.

Příklad

Instance `A`, `B` a `C` v následujícím příkladu jsou stejného typu.

```
template<class T, int n = 7> struct TA { ... };

typedef unsigned int uint;
TA<unsigned, 5+2> A;
TA<uint, 10-3> B;
TA<uint> C;
TA<uint, 10> D;
```

Pomocí `typedef` se nedeklaruje nový typ, nýbrž se deklaruje pouze synonymum pro existující typ.

Instanci šablony třídy lze použít jako předka jiné třídy nebo šablony třídy. Lze též deklarovat šablonu třídy jako potomka jiné třídy.

Vnořené šablony

Vnořená šablona (*členská šablona*) (angl. *member template*) je šablona, která je složkou třídy nebo složkou šablony třídy, přičemž složkou může být v tomto případě metoda, tzv. *vnořená šablona metody* (angl. *member function template*) nebo vnořená třída, tzv. *vnořená šablona třídy* (angl. *member class template*).

Šablonou metody může být i konstruktor. Šablonou metody nemůže být destruktor nebo virtuální metoda. Překladač nepoužije šablonu konstruktoru s jedním parametrem k vytvoření kopírovacího konstruktoru. Šablona metody muže mít stejný název s obyčejnou metodou. Šablona metody se použije v případě, pokud typy nebo počet skutečných parametrů neodpovídají typům nebo počtu formálních parametrů obyčejné metody.

Lokální třídy (deklarované v těle funkce) nemohou mít vnořené šablony.

V obklopující šabloně třídy nelze použít formální parametr vnořené šablony, naopak to však lze.

Příklad

V následujícím příkladu je deklarována šablona třídy `TA`, v níž je deklarována vnořená šablona třídy `TB` se dvěma atributy a metodou `Pricti`. Šablona `TA` má kromě jiného atribut typu vnořené šablony.

```
template <class T, class U> struct TA {
   template <class V> struct TB {
      V z;
      U z2;
      V Pricti(V x);
   };
   T  x, y;
   TB<U> b;
};
```

```
template <class T, class U> template <class V>
V TA<T, U>::TB<V>::Pricti(V x)
{
   z += x;
   return z;
}

TA<int, double> A;
A.x = 10; // x je typu int
A.b.z = 5.4; // z je typu double
```

Následující zápis představuje deklaraci instance vnořené šablony `TB`, jejíž atribut `z` je typu `char` a `z2` typu `double`. Typ `int` v `TA<int, double>` nemá na deklaraci šablony `TB` žádný vliv – může být libovolného typu:

```
TA<int, double>::TB<char> B;
B.z = 'c';
B.z2 = 5.4;
```

Příklad

Součástí deklarované šablony třídy `TVektor` může být např. šablona metody `assign`, která přiřadí do pole třídy `TVektor` hodnoty prvků jiné datové struktury, reprezentované iterátory `first` a `last`, které jsou typu `InputIterator`. Pro typ `InputIterator` musí být definovány operátory `++` (prefixová verze – posun na následující prvek datové struktury), `*` (dereference ukazatele – l-hodnota prvku datové struktury) a `!=` (vrací `true`, pokud dva iterátory neukazují na stejný prvek datové struktury).

```
template <class T> class TVektor {
   //...
   template <class InputIterator>
   void assign(InputIterator first, InputIterator last);
};
template <class T> template <class InputIterator>
void TVektor<T>::assign(InputIterator first, InputIterator last)
{
   InputIterator iter;
   T*            iter2;
   delete[] a;
   n = 0;
   for (iter = first; iter != last; ++iter, n++);
   a = new T[n];
   for (iter = first, iter2 = a; iter != last; ++iter, ++iter2)
      *iter2 = *iter;
}
```

Metoda `assign` by mohla být použita např. k přiřazení prvků obyčejného pole:

```
TVektor<int> A;
int b[] = { 10, 20, 30, 40 };
A.assign(b, b+3);
```

Vektor `A` bude obsahovat první 3 prvky pole `b`.

Obdobně by mohla být definována šablona konstruktoru:

```
template <class T> class TVektor {
   ...
   template <class InputIterator>
   TVektor(InputIterator first, InputIterator last) : a(0)
      { assign(first, last); PocInstanci++; }
};
TVektor<int> C(b+1, b+4);
```

Vektor `C` bude obsahovat hodnoty prvků `b[1]`, `b[2]` a `b[3]`.

ŠABLONY – POKRAČOVÁNÍ

Třídy – pokračování

Vnořený typ typového parametru šablony

Typovým parametrem šablony může být třída, která může obsahovat deklaraci vnořeného typu. Vnořený typ lze použít v šabloně třídy, např. pro deklaraci jejího atributu:

```
template<class T> class TA {
   T::TC b; // ?
   // ...
};
```

V uvedeném příkladě se předpokládá, že parametrem `T` šablony `TA` je třída, v níž je deklarován vnořený typ `TC`. Pokud by překladač kontroloval správnost deklarace až v okamžiku použití, bylo by vše v pořádku. Současná norma jazyka C++ ale předpokládá, že překladač bude kontrolovat vše, co lze, již v okamžiku, kdy narazí na deklaraci šablony. Proto se musí překladači sdělit, že `T::TC` je typ. K tomu slouží klíčové slovo `typename`, které se uvede před jménem neznámého typu:

```
template<class T> class TA {
   typename T::TC b; // OK
   // ...
};
```

Bylo by možné deklarovat synonymum pro vnořený typ pomocí `typedef` a ten potom použít:

```
template<class T> class TA {
   typedef typename T::TC TTC;
   TTC b;
   // ...
};
```

Funkce

Šablona funkce (angl. *function template*) umožňuje popsat najednou celou množinu funkcí, které se liší jen např. typem jejich parametru.

Šablona funkce `Max`, která vrací větší ze dvou parametrů, by se mohla definovat následovně:

```
template<class T>
inline T Max(T a, T b) { return a > b ? a : b; }
```

Instance

Jestliže se zavolá funkce, kterou překladač nezná, ale kterou umí vytvořit podle šablony, vytvoří si překladač potřebnou instanci této šablony. Přitom si ale musí dokázat odvodit typy parametrů šablony. Např.:

```
cout << Max(1, 2);      // vytvoří se Max<int>
cout << Max(1.4, 1.5); // vytvoří se Max<double>
cout << Max(1, 'b');    // Chyba
cout << Max(2.2, 1);    // Chyba
```

Poslední dvě použití funkce `Max` nejsou správná, protože překladač nedokáže rozhodnout, kterou instanci použít resp. vytvořit. Lze mu ale napomoci tím, že jméno šablony funkce se při volání kvalifikuje skutečnými parametry v lomených závorkách připojenými za jméno funkce:

```
cout << Max<int>(1, 'b'); // zavolá se Max<int>
cout << Max<double>(2.2, 1); // zavolá se Max<double>
```

Překladač neumí odvodit typ skutečného parametru šablony, pokud je použit jen jako návratový typ, např.:

```
template <class T> T f(int i) { return T(i*2); }

double b = f(10); // Chyba
double b = f<double>(10); // OK
```

Vytvoření instance šablony funkce není rovnocenné prototypu funkce. Jestliže si překladač vytvoří instanci `int Max(int, int)` a poté narazí na volání `Max(1.4, 1.5)`, nepokusí se o konverzi parametrů, ale vytvoří novou instanci pro typ `double`.

Pokud se šablona funkce při jejím volání kvalifikuje skutečnými parametry, lze vynechat koncové parametry, které si překladač dokáže odvodit. Např.: šablonu funkce

```
template <class T, class U>
T Konverze(U u) { return static_cast<T>(u); }
```

lze zavolat zápisem:

```
Konverze<char>(a);
```

neboť formální typ `U` si dokáže překladač odvodit z typu skutečného parametru `a`.

Přetěžování

V jednom oboru viditelnosti lze deklarovat několik šablon funkcí se stejným jménem. Lze také deklarovat šablonu funkce a obyčejnou funkci se stejným jménem. Pravidla pro rozlišování mezi šablonami jsou podobná jako pravidla pro rozlišování přetížených funkcí. Má-li překladač na vybranou mezi obyčejnou funkcí a instancí šablony, dá přednost obyčejné funkci, pokud typy skutečných parametrů přesně odpovídají typům formálních parametrů obyčejné funkce. Pokud typy skutečných parametrů přesně neodpovídají typům formálních parametrů obyčejné funkce ani formálním parametrům šablon, překladač zavolá obyčejnou funkci, pokud lze typy skutečných parametrů automaticky zkonvertovat na typy formálních parametrů.

Např. šablona funkce `Max` nebude fungovat pro řetězce znaků, a proto se může definovat přetížená funkce:

```
inline char* Max(char* a, char* b)
{
   return strcmp(a, b) > 0 ? a : b;
}
```

Zápisem

```
cout << Max("aaa", "abb");
```

se zavolá obyčejná funkce `Max(char*, char*)` a ne instance šablony pro typ `char*`.

Pokud by existovala deklarace obyčejné funkce `Max` s parametry typu `int`:

```
int Max(int a, int b);
```

příkaz

```
cout << Max(1, 'b');
```

by zavolal funkci `Max(int, int)`.

Vazba jmen

V těle šablony lze použít také globální jména, viditelná v místě deklarace šablony. Může jít o jména typů, proměnných apod. Táž jména (s jiným významem) se mohou vyskytnout i v místě použití šablony. Při vytváření instance šablony se však použijí jména viditelná v místě deklarace šablony.

Příklad

```
int i = 10;
template<class T> inline T GetI() { return i; }

int main()
{
   int i = 50;
   cout << GetI<int>();
   return 0;
}
```

Uvedený program vypíše na obrazovku hodnotu globální proměnné `i`, tj. `10`.

Explicitní specializace

Explicitní specializace šablony umožňuje deklarovat specifickou verzi šablony pro její určité skutečné parametry. Explicitní specializace může být deklarována pro:

- šablonu obyčejné funkce,
- šablonu třídy,
- metodu šablony třídy,
- statický atribut šablony třídy,
- třídu vnořenou do šablony třídy,
- šablonu třídy vnořenou do jiné šablony třídy,
- šablonu metody vnořenou do šablony třídy.

Syntaxe:

explicitní_specializace:

template < > *deklarace*

Deklarace je definiční nebo informativní deklarace funkce, metody nebo třídy. V této deklaraci se za jménem funkce (metody) nebo jmenovkou třídy uvedou skutečné parametry šablony v lomených závorkách. V případě šablony funkce (metody) se mohou vynechat koncové parametry, které si překladač dokáže odvodit. Explicitní specializace dané šablony třídy nemusí obsahovat stejné složky (atributy, metody apod.) jako primární šablona třídy.

Explicitní specializace šablony obyčejné funkce

Deklarace explicitní specializace šablony obyčejné funkce musí začínat specifikací `template < >`. Pokud by se specifikace `template < >` neuvedla, jednalo by se o deklaraci obyčejné funkce.

Příklad

```
template<class T> void Vypis(T t1, T t2)
{ cout << t1 << ", " << t2 << '\n'; }
```

```
// explicitní specializace šablony funkce
template<> void Vypis(double t1, double t2)
{ printf("%.1lf, %.1lf\n", t1, t2); }

Vypis(10, 12);     // OK – volá se primární šablona
Vypis(10.0, 12.5); // OK - volá se explicitní specializace
Vypis(10, 12.5);   // Chyba
```

Poslední volání funkce `Vypis(10, 12.5)` oznámí překladač jako chybu, protože nenalezl funkci `Vypis(int, double)`. Pokud by však funkce `Vypis(double, double)` byla deklarována bez specifikace `template < >`, jednalo by se o deklaraci obyčejné funkce a volání `Vypis(10, 12.5)` by bylo správné, protože pro obyčejnou funkci by se skutečný parametr 10 automaticky zkonvertoval na typ `double`.

Explicitní specializace metody šablony třídy

Příklad

Součástí v předchozí přednášce deklarované šablony třídy `TVektor` by mohla být metoda `Vypis`, která vypíše na obrazovku hodnoty jednotlivých prvků pole:

```
template <class T> class TVektor {
   // ...
   void Vypis();
};

template <class T>
void TVektor<T>::Vypis()
{
   for (int i = 0; i < n; i++) cout << a[i] << ' ';
   cout << '\n';
}
```

Pro prvky typu `double` by mohla být deklarována explicitní specializace metody `Vypis`, která vypíše hodnoty prvků pole zaokrouhlené na 1 desetinné místo:

```
template < > void TVektor<double>::Vypis()
{
   for (int i = 0; i < n; i++) printf("%.1lf ", a[i]);
   cout << '\n';
};
```

Specifikace `template < >` může být v definici metody `Vypis()` vynechána. Pro instance typu `TVektor<double>` se bude volat explicitně specializovaná šablona metody `Vypis()`, pro ostatní instance typu `TVektor<T>` se bude volat primární šablona metody `Vypis()`.

Explicitní specializace statického atributu šablony třídy

Příklad

Statický atribut `PocInstanci` šablony třídy `TVektor` definované v předchozí přednášce by mohl mít pro typ `TVektor<double>` jinou počáteční hodnotu než pro ostatní typy:

```
template<class T> int TVektor<T>::PocInstanci; // = 0
template<> int TVektor<double>::PocInstanci = 5;
```

Specifikace `template < >` může být v definici statického atributu vynechána.

Po vytvoření instancí

```
TVektor<int> A;
TVektor<double> C;
```

následující příkazy vypíší na obrazovku hodnoty `1` a `6`:

```
cout << TVektor<int>::GetPocInstanci() << '\n';
cout << TVektor<double>::GetPocInstanci() << '\n';
```

Explicitní specializace šablony třídy

Příklad

Pro typ `char` může být deklarována explicitní specializace šablony třídy `TVektor`, která obsahuje nulou zakončený řetězec znaků a kromě implicitního konstruktoru obsahuje ještě konstruktor s parametrem typu `const char*`:

```
template<> class TVektor<char> {
   char* a;
public:
   TVektor() { a = 0; }
   TVektor(const char* s);
  ~TVektor() { delete[] a; }
   // ...
};

TVektor<char>::TVektor(const char* s)
{
   a = new char[strlen(s)+1];
   strcpy(a, s);
}
```

Specifikace `template < >` nesmí být uvedena v definici konstruktoru `TVektor(const char* s)` a v případných dalších metodách definovaných mimo tělo šablony třídy, protože tyto metody nejsou explicitně specializovány.

Příkaz

```
TVektor<char> C("test");
```

vytvoří instanci typu `TVektor<char>` pomocí konstruktoru `TVektor(const char* s)`.

Explicitní specializace vnořené šablony

Explicitně specializovaná definice šablony třídy nebo šablony metody, vnořené do obklopující šablony třídy, musí být uvedena mimo tělo obklopující šablony třídy. Definice obklopující šablony třídy musí být v tomto případě také explicitně specializovaná. Ale pokud je pro obklopující šablonu třídy definována explicitní specializace, nemusí být definována explicitní specializace pro vnořenou šablonu.

Příklad

```
template <class T> struct TA {
   template <class U> struct TB { U z; };
   T x;
};
```

```
template <> template<> struct TA<int>::TB<int> {
   int z1, z2; // OK
};

TA<int>::TB<double> B;
B.z = 2.3;
TA<int>::TB<int> C; // instance explicitní specializace TB
C.z1 = 3;
C.z2 = 4;
```

Následující deklarace však není přípustná, protože obklopující třída není explicitně specializovaná:

```
template <class T> template<> struct TA<T>::TB<char> {
   char z3;
};
```

Příklad

```
template <class T> struct TA {
   T x;
   template <class U> void f(U u) { x = T(u); }
};

template <> template <>
void TA<char>::f(const char* c) { x = c[0]; }

TA<int> A;
A.f(10.5); // volá se primární vnořená šablona
TA<char> B, C;
B.f('A'); // volá se primární vnořená šablona
C.f("text"); // volá se explicitní specializace vnořené šablony
             // C.x = 't'
```

ŠABLONY – POKRAČOVÁNÍ

Třídy – pokračování

Vnořený typ typového parametru šablony

Typovým parametrem šablony může být třída, která může obsahovat deklaraci vnořeného typu. Vnořený typ lze použít v šabloně třídy, např. pro deklaraci jejího atributu:

```
template<class T> class TA {
   T::TC b; // ?
   // ...
};
```

V uvedeném příkladě se předpokládá, že parametrem `T` šablony `TA` je třída, v níž je deklarován vnořený typ `TC`. Pokud by překladač kontroloval správnost deklarace až v okamžiku použití, bylo by vše v pořádku. Současná norma jazyka C++ ale předpokládá, že překladač bude kontrolovat vše, co lze, již v okamžiku, kdy narazí na deklaraci šablony. Proto se musí překladači sdělit, že `T::TC` je typ. K tomu slouží klíčové slovo `typename`, které se uvede před jménem neznámého typu:

```
template<class T> class TA {
   typename T::TC b; // OK
   // ...
};
```

Bylo možné deklarovat synonymum pro vnořený typ pomocí `typedef` a ten potom použít:

```
template<class T> class TA {
   typedef typename T::TC TTC;
   TTC b;
   // ...
};
```

Překladač C++ Builder 6 umožňuje použít vnořený typ typového parametru šablony bez nebo s klíčovým slovem `typename`. Překladač Visual C++ 2003 správně vyžaduje použití klíčového slova `typename`.

Funkce

Šablona funkce (angl. *function template*) umožňuje popsat najednou celou množinu funkcí, které se liší jen např. typem jejich parametru.

Šablona funkce `Max`, která vrací větší ze dvou parametrů, by se mohla definovat následovně:

```
template<class T>
inline T Max(T a, T b) { return a > b ? a : b; }
```

Instance

Jestliže se zavolá funkce, kterou překladač nezná, ale kterou umí vytvořit podle šablony, vytvoří si překladač potřebnou instanci této šablony. Přitom si ale musí dokázat odvodit typy parametrů šablony. Např.:

```
cout << Max(1, 2);      // vytvoří se Max<int>
cout << Max(1.4, 1.5); // vytvoří se Max<double>
cout << Max(1, 'b');    // Chyba
cout << Max(2.2, 1);    // Chyba
```

Poslední dvě použití funkce `Max` nejsou správná, protože překladač nedokáže rozhodnout, kterou instanci použít resp. vytvořit. Lze mu ale napomoci tím, že jméno šablony funkce se při volání kvalifikuje skutečnými parametry v lomených závorkách připojenými za jméno funkce:

```
cout << Max<int>(1, 'b'); // zavolá se Max<int>
cout << Max<double>(2.2, 1); // zavolá se Max<double>
```

Překladač neumí odvodit typ skutečného parametru šablony, pokud je použit jen jako návratový typ, např.:

```
template <class T> T f(int i) { return T(i*2); }

double b = f(10); // Chyba
double b = f<double>(10); // OK
```

Vytvoření instance funkce podle šablony není rovnocenné prototypu funkce. Jestliže si překladač vytvoří instanci `int Max(int, int)` a poté narazí na volání `Max(1.4, 1.5)`, nepokusí se o konverzi parametrů, ale vytvoří novou instanci pro typ `double`.

Pokud se šablona funkce při jejím volání kvalifikuje skutečnými parametry, lze vynechat koncové parametry, které si překladač dokáže odvodit. Např.: šablonu funkce

```
template <class T, class U>
T Konverze(U u) { return static_cast<T>(u); }
```

lze zavolat zápisem:

```
Konverze<char>(a);
```

neboť formální typ `U` si dokáže překladač odvodit z typu skutečného parametru `a`.

Přetěžování

V jednom oboru viditelnosti lze deklarovat několik šablon funkcí se stejným jménem. Lze také deklarovat šablonu funkce a obyčejnou funkci se stejným jménem. Pravidla pro rozlišování mezi šablonami jsou podobná jako pravidla pro rozlišování přetížených funkcí. Má-li překladač na vybranou mezi obyčejnou funkcí a instancí šablony, dá přednost obyčejné funkci, pokud typy skutečných parametrů přesně odpovídají typům formálních parametrů obyčejné funkce. Pokud typy skutečných parametrů přesně neodpovídají typům formálních parametrů obyčejné funkce ani formálním parametrům šablon, překladač zavolá obyčejnou funkci, pokud lze typy skutečných parametrů automaticky zkonvertovat na typy formálních parametrů.

Např. šablona funkce `Max` nebude fungovat pro řetězce znaků, a proto se může definovat přetížená funkce:

```
inline char* Max(char* a, char* b)
{
   return strcmp(a, b) > 0 ? a : b;
}
```

Zápisem

```
cout << Max("aaa", "abb");
```

se zavolá obyčejná funkce `Max(char*, char*)` a ne instance šablony pro typ `char*`.

Pokud by existovala deklarace obyčejné funkce `Max` s parametry typu `int`:

```
int Max(int a, int b);
```

příkaz

```
cout << Max(1, 'b');
```

by zavolal funkci `Max(int, int)`.

Vazba jmen

V těle šablony lze použít také globální jména, viditelná v místě deklarace šablony. Může jít o jména typů, proměnných apod. Táž jména (s jiným významem) se mohou vyskytnout i v místě použití šablony. Při vytváření instance šablony se však použijí jména viditelná v místě deklarace šablony.

Příklad

```
int i = 10;
template<class T> inline T GetI() { return i; }

int main()
{
   int i = 50;
   cout << GetI<int>();
   return 0;
}
```

Uvedený program vypíše na obrazovku hodnotu globální proměnné `i`, tj. `10`.

Explicitní specializace

Explicitní specializace šablony umožňuje deklarovat specifickou verzi šablony pro její určité skutečné parametry. Explicitní specializace může být deklarována pro:

- šablonu obyčejné funkce,
- šablonu třídy,
- metodu šablony třídy,
- statický atribut šablony třídy,
- třídu vnořenou do šablony třídy,
- šablonu třídy vnořenou do jiné šablony třídy,
- šablonu metody vnořenou do šablony třídy.

Syntaxe:

explicitní_specializace:

template < > *deklarace*

Deklarace je definiční nebo informativní deklarace funkce, metody nebo třídy. V této deklaraci se za jménem funkce (metody) nebo jmenovkou třídy uvedou skutečné parametry šablony v lomených závorkách. V případě šablony funkce (metody) se mohou vynechat koncové parametry, které si překladač dokáže odvodit. Explicitní specializace dané šablony třídy nemusí obsahovat stejné složky (atributy, metody apod.) jako primární šablona třídy.

Explicitní specializace šablony obyčejné funkce

Deklarace explicitní specializace šablony obyčejné funkce musí začínat specifikací `template < >`. Pokud by se specifikace `template < >` neuvedla, jednalo by se o deklaraci obyčejné funkce.

Příklad

```
template<class T> void Vypis(T t1, T t2)
{ cout << t1 << ", " << t2 << '\n'; }

// explicitní specializace šablony funkce
template<> void Vypis(double t1, double t2)
{ printf("%.1lf, %.1lf\n", t1, t2); }

Vypis(10, 12);     // OK – volá se primární šablona
Vypis(10.0, 12.5); // OK - volá se explicitní specializace
Vypis(10, 12.5);   // Chyba
```

Poslední volání funkce `Vypis(10, 12.5)` oznámí překladač jako chybu, protože nenalezl funkci `Vypis(int, double)`. Pokud by však funkce `Vypis(double, double)` byla deklarována bez specifikace `template < >`, jednalo by se o deklaraci obyčejné funkce a volání `Vypis(10, 12.5)` by bylo správné, protože pro obyčejnou funkci by se skutečný parametr 10 automaticky zkonvertoval na typ `double`.

Explicitní specializace metody šablony třídy

Příklad

Součástí v předchozí přednášce deklarované šablony třídy `TVektor` by mohla být metoda `Vypis`, která vypíše na obrazovku hodnoty jednotlivých prvků pole:

```
template <class T> class TVektor {
   // ...
   void Vypis();
};

template <class T>
void TVektor<T>::Vypis()
{
   for (int i = 0; i < n; i++) cout << a[i] << ' ';
   cout << '\n';
}
```

Pro prvky typu `double` by mohla být deklarována explicitní specializace metody `Vypis`, která vypíše hodnoty prvků pole zaokrouhlené na 1 desetinné místo:

```
template < > void TVektor<double>::Vypis()
{
   for (int i = 0; i < n; i++) printf("%.1lf ", a[i]);
   cout << '\n';
};
```

Specifikace `template < >` může být v definici metody `Vypis()` vynechána. Pro instance typu `TVektor<double>` se bude volat explicitně specializovaná šablona metody `Vypis()`, pro ostatní instance typu `TVektor<T>` se bude volat primární šablona metody `Vypis()`.

Explicitní specializace statického atributu šablony třídy

Příklad

Statický atribut `PocInstanci` šablony třídy `TVektor` definované v předchozí přednášce by mohl mít pro typ `TVektor<double>` jinou počáteční hodnotu než pro ostatní typy:

```
template<class T> int TVektor<T>::PocInstanci; // = 0
template<> int TVektor<double>::PocInstanci = 5;
```

Specifikace `template < >` může být v definici statického atributu vynechána.

Po vytvoření instancí

```
TVektor<int> A;
TVektor<double> C;
```

následující příkazy vypíší na obrazovku hodnoty `1` a `6`:

```
cout << TVektor<int>::GetPocInstanci() << '\n';
cout << TVektor<double>::GetPocInstanci() << '\n';
```

Explicitní specializace šablony třídy

Příklad

Pro typ `char` může být deklarována explicitní specializace šablony třídy `TVektor`, která obsahuje nulou zakončený řetězec znaků a kromě implicitního konstruktoru obsahuje ještě konstruktor s parametrem typu `const char*`:

```
template<> class TVektor<char> {
   char* a;
public:
   TVektor() { a = 0; }
   TVektor(const char* s);
  ~TVektor() { delete[] a; }
   // ...
};

TVektor<char>::TVektor(const char* s)
{
   a = new char[strlen(s)+1];
   strcpy(a, s);
}
```

Specifikace `template < >` nesmí být uvedena v definici konstruktoru `TVektor(const char* s)` a v případných dalších metodách definovaných mimo tělo šablony třídy, protože tyto metody nejsou explicitně specializovány.

Příkaz

```
TVektor<char> C("test");
```

vytvoří instanci typu `TVektor<char>` pomocí konstruktoru `TVektor(const char* s)`.

Explicitní specializace vnořené šablony

Explicitně specializovaná definice šablony třídy nebo šablony metody, vnořené do obklopující šablony třídy, musí být uvedena mimo tělo obklopující šablony třídy. Definice obklopující šablony třídy musí být v tomto případě také explicitně specializovaná. Ale pokud je pro obklopující šablonu třídy definována explicitní specializace, nemusí být definována explicitní specializace pro vnořenou šablonu.

Příklad

```
template <class T> struct TA {
   template <class U> struct TB { U z; };
   T x;
};

template <> template<> struct TA<int>::TB<int> {
   int z1, z2; // OK
};

TA<int>::TB<double> B;
B.z = 2.3;
TA<int>::TB<int> C; // instance explicitní specializace TB
C.z1 = 3;
C.z2 = 4;
```

Následující deklarace však není přípustná, protože obklopující třída není explicitně specializovaná:

```
template <class T> template<> struct TA<T>::TB<char> {
   char z3;
};
```

Příklad

```
template <class T> struct TA {
   T x;
   template <class U> void f(U u) { x = T(u); }
};

template <> template <>
void TA<char>::f(const char* c) { x = c[0]; }

TA<int> A;
A.f(10.5); // volá se primární vnořená šablona
TA<char> B, C;
B.f('A'); // volá se primární vnořená šablona
C.f("text"); // volá se explicitní specializace vnořené šablony
             // C.x = 't'
```

ŠABLONY – POKRAČOVÁNÍ

Parciální specializace šablon tříd

Parciální specializace šablony třídy (angl. *class template partial specialization*) poskytuje alternativní definici k primární definici šablony pro určité druhy parametrů. Primární šablona musí mít alespoň informativní deklaraci před deklaracemi parciálních specializací této šablony.

Každá parciální specializace šablony představuje odlišnou šablonu, a proto musí být kompletně definována. Může obsahovat jiné složky než primární šablona.

Parciálně specializované deklarace se od primární deklarace liší tím, že za jménem šablony následují v lomených závorkách formální parametry, které určují způsob specializace. Tyto parametry mohou mít odlišné názvy než formální parametry primární šablony, např.:

```
template<class T, class U, int n> class TA             { }; // #1
template<class T, int n>          class TA<T, T*, n>   { }; // #2
template<class T, class U, int n> class TA<T*, U, n>   { }; // #3
template<class V>                 class TA<int, V*, 5> { }; // #4
template<class T, class U, int n> class TA<T, U*, n>   { }; // #5
```

První definice definuje primární (nespecializovanou) šablonu třídy. Další definice definují parciální specializace primární šablony.

Překladač při generování instancí použije tu verzi šablony, která odpovídá skutečným parametrům a je nejvíce specializovaná, např.:

```
TA<int, int, 1>   a1; // použije se #1
TA<int, int*, 1>  a2; // použije se #2
TA<int, char*, 5> a3; // použije se #4
TA<int, char*, 1> a4; // použije se #5
TA<int*, int*, 2> a5; // chyba – lze použít #3 i #5
```

Složky (metody, statické atributy, vnořené třídy, šablony) parciálně specializované šablony třídy definované mimo tělo šablony, musí obsahovat stejný seznam formálních parametrů (uváděných před jmenovkou třídy) a stejný seznam specializovaných parametrů (uváděných za jmenovkou třídy) jako parciálně specializovaná šablona. Explicitní specializace složky parciálně specializované šablony třídy se deklaruje stejným způsobem jako explicitní specializace složky primární šablony třídy. Např.:

```
// primární šablona
template <class T, int I> struct TA {
   void f();
};

// definice metody primární šablony
template <class T, int I> void TA<T, I>::f(){ ... } // #1

// parciální specializace šablony
template <class T> struct TA<T, 2> {
   void f();
};

// definice metody parciálně specializované šablony
template <class T> void TA<T, 2>::f() { ... } // #2

// explicitní specializace metody parciálně specializované šablony
template <> void TA<char, 2>::f() { ... } // #3
```

```
void g()
{
   TA<char, 1> a1;
   TA<int, 2>  a2;
   TA<char, 2> a3;
   a1.f(); // použije se #1
   a2.f(); // použije se #2
   a3.f(); // použije se #3
}
```

Příklad

Je deklarována primární šablona třídy `TKonstVektor`, obsahující pole `n` prvků typu `T`:

```
template <class T, int n> class TKonstVektor { // #1
   T a[n];
public:
   TKonstVektor(const T& t = T())
      { for (int i = 0; i < n; i++) a[i] = t; }
   T& operator[](int i) { return a[i]; }
};
```

Pokud `T` bude typu ukazatel, šablona bude obsahovat pole dynamicky alokovaných prvků. Deklaruje se tedy tato parciální specializace:

```
template <class T, int n> class TKonstVektor<T*, n> { // #2
   T* a[n];
public:
   TKonstVektor(const T& t = T())
      { for (int i = 0; i < n; i++) a[i] = new T(t); }
   T& operator[](int i) { return *a[i]; }
};
```

Bude-li navíc `n` rovno hodnotně `2`, bude šablona deklarována ještě jinak:

```
template <class T> class TKonstVektor<T*, 2> { // #3
   T* a[2];
public:
   TKonstVektor(const T& t = T()) { a[0] = new T(t), a[1] = new T(t); }
   T& operator[](int i) { return *a[i]; }
};
```

Příkaz

```
TKonstVektor<int, 10> A(0); // použije se #1
```

způsobí použití primární šablony, zatímco příkazy

```
TKonstVektor<long*, 10> B(1); // použije se #2
TKonstVektor<double*, 2> C;  // použije se #3
```

způsobí použití specializovaných deklarací této šablony.

Explicitní vytvoření instance

Překladači lze přikázat, aby vytvořil instanci šablony, aniž by se hned použila. To lze provést pro šablony funkcí, tříd, metod a statických atributů. Pro explicitně vytvořenou instanci šablony třídy vytvoří překladač všechny její metody (včetně statických) a statické atributy, i když nejsou

v programu použity – budou součástí souboru .obj a .exe. Obdobně se postupuje u dalších typů šablon.

Syntaxe je následující:

Explicitní_vytvoření_instance:

template *deklarace***;**

Deklarace představuje prototyp funkce nebo metody, informativní deklaraci třídy bez jejího těla nebo deklaraci statického atributu. V této deklaraci se za jmenovkou třídy nebo jménem funkce uvedou skutečné parametry šablony v lomených závorkách. V případě šablony funkce se mohou vynechat koncové parametry, které si překladač dokáže odvodit.

Příklad

Příkaz

```
template class TVektor<int>;
```

explicitně vytvoří instanci dříve definované šablony třídy `TVektor` pro typ `int`. Tím se vytvoří instance všech metod a statických atributů pro typ `int`. Nevytvoří se však instance vnořených šablon, a to jak metod, tak i tříd. To znamená, že se nevytvoří instance vnořených šablon metod `assign` a konstruktoru `TVektor(InputIterator, InputIterator)`. Explicitní vytvoření jejich instancí se musí provést zvlášť, protože se musí specifikovat typ pro parametr `InputIterator`. Např. pro typ `int*` by explicitní vytvoření těchto instancí bylo následující:

```
template void TVektor<int>::assign<int*>(int*, int*);
template TVektor<int>::TVektor(int*, int*);
```

Specifikaci `<int*>` za jménem metody `assign` lze vynechat.

Lze explicitně vytvořit i instance jednotlivých metod a statických atributů šablony třídy bez explicitního vytvoření instance celé třídy. Např. pro šablonu `TVektor`:

```
template bool TVektor<double>::operator == (const TVektor& t) const;
template int TVektor<double>::PocInstanci;
```

Příkaz

```
template int TVektor<int>::PocInstanci;
```

se nemusí uvádět, pokud byl předtím uveden příkaz `template class TVektor<int>;`

Překladač C++ Builder 6 oznámí varování, že již existuje instance tohoto atributu pro typ `TVektor<int>`. Překladač Visual C++ 2003 varování neoznámí.

Explicitní vytvoření instancí dříve uvedených obyčejných funkcí `Max` pro typ `int` a `Konverze` pro typ `char` a `double` je následující:

```
template int Max(int, int);
template char Konverze<char>(double);
```

Přátelé

Přítelem třídy nebo šablony třídy může být:

- šablona obyčejné funkce,
- šablona jiné třídy,

- specializace šablony obyčejné funkce,
- specializace šablony jiné třídy,
- obyčejná funkce,
- jiná třída.

Pokud má být přítelem šablony třídy `TA` obyčejná funkce, zpravidla jeden z jejích parametrů je typu reference nebo ukazatel na šablonu třídy `TA`. V takovém případě se ale jedná o šablonu obyčejné funkce, která má jako formální šablonové parametry formální parametry šablony třídy `TA`.

Příklad

```
template <class T> class TA;
template <class T> void f2(TA<T>&);
template <class T> class TB;

template <class T> class TA {
   friend void f1(TA<int>&);
   friend void f2<T>(TA<T>&);
   template <class T, class U> friend void f3(TA<T>&, U);
   friend class TB<long>;
   template <class U> friend class TC;
   T x;
public:
   TA(T _x) : x(_x) {}
};
void f1(TA<int>& A)
{ A.x += 2; }

template<class T> void f2(TA<T>& A)
{ A.x += A.x; }

template <class T, class U> void f3(TA<T>& A, U u)
{ A.x += static_cast<T>(u); }

TA<int> A(2);
TA<float> A2(10.5);
```

Obyčejná funkce `f1` je přítelem pouze instance šablony třídy `TA<int>`. Funkci `f1` lze použít např. pro instanci `A`, nelze ji však volat pro instanci `A2`:

```
f1(A); // OK
f1(A2); // Chyba
```

Přítelem šablony třídy `TA` je dále šablona obyčejné funkce `f2` včetně jejích explicitních specializací. Takto uvedená deklarace spřátelené šablony funkce vyžaduje deklaraci prototypu šablony funkce `f2` před definicí šablony třídy `TA`. Protože v tomto prototypu je uvedena šablona `TA<T>`, musí být před ním uvedena informativní deklarace šablony třídy `TA`. Šablona funkce `f2` má jeden formální šablonový parametr `T`, jenž je identický s formálním parametrem `T` šablony třídy `TA`. To znamená, že např. přítelem instance typu `TA<int>` je instance šablony funkce `f2`, mající tvar

```
void f2<int>(TA<int>&);
```

Použití funkce `f2` může být např. následující:

```
f2(A); // A.x = 4
f2(A2); // A2.x = 7
```

Šablonu funkce `f2` jako přítele šablony třídy `TA` lze deklarovat také následujícím způsobem:

```
template <class T> class TA {
   template <class T> friend void f2(TA<T>&);
   // ...
};
```

V tomto případě není nutná informativní deklarace šablony funkce `f2` před definicí šablony třídy `TA`.

Dalším přítelem šablony třídy `TA` je šablona funkce `f3`, mající dva typové parametry `T` a `U`. Parametr `T` je identický s formálním parametrem `T` šablony třídy `TA` a mohl by mít i jiné označení. Např. přítelem instance `TA<int>` je šablona funkce `f3`, která má parametr `T` rovný `int` a parametr `U` je libovolného typu.

Přítelem šablony třídy `TA` je dále specializace šablony třídy `TB` pro typ `long`. Šablona třídy `TB` je definována takto:

```
template <class T> class TB {
   T y;
public:
   TB(T _y) : y(_y) {}
   template <class U> void g(TA<U>& A) { y += A.x; }
};
```

Metodu `g` lze volat jen pro instanci typu `TB<long>`:

```
TB<long>  B1(3);
TB<float> B2(3.5);
B1.g(A); // OK
B2.g(A); // Chyba
```

Přítelem šablony třídy `TA` je také libovolná instance šablony třídy `TC`, která je definována takto:

```
template <class T> class TC {
   T z;
public:
   TC(T _z) : z(_z) {}
   template <class U> void h(TA<U>& A) { z += A.x; }
};
```

Metodu `h` lze volat pro libovolnou instanci šablony `TC`:

```
TC<double> C(3);
C.h(A); // OK
```

Příklad

```
class TD;
template <class T> void fd2(TD&, T);

class TD {
   int x;
   template <class T> friend void fd1(TD&, T);
   friend void fd2<int>(TD&, int);
   friend void fd2<char>(TD&, char);
public:
   TD(int _x) : x(_x) {}
};
```

```
template <class T> void fd1(TD& D, T t)
{ D.x += static_cast<int>(t); }

template <class T> void fd2(TD& D, T t)
{ D.x -= static_cast<int>(t); }
```

Přítelem třídy `TD` je šablona obyčejné funkce `fd1`, kterou lze volat pro libovolnou instanci třídy `TD`:

```
TD D(2);
fd1(D, 4.5); // OK
```

Přítelem třídy `TD` jsou dále dvě specializace šablony funkce `fd2` pro typy `int` a `char`. Funkci `fd2` lze volat jen, pokud jejím druhým skutečným parametrem je typ `int` nebo `char`:

```
fd2(D, 4);   // OK – druhý parametr je typu int
fd2(D, 'a'); // OK – druhý parametr je typu char
fd2(D, 4.5); // Chyba – druhý parametr je typu double
```

Organizace programu

Používání šablon lze realizovat dvěma způsoby:

1. Deklarace i definice šablony se umístí do hlavičkového souboru, který se vloží pomocí direktivy `#include` do každého modulu, kde se šablona používá.
2. Deklarace šablony (u šablon tříd i definice šablony třídy) se umístí do hlavičkového souboru a její definice do samostatného modulu. Do jiných modulů, kde se šablona používá, se vloží hlavičkový soubor, obsahující deklaraci šablony, pomocí direktivy `#include`.

1. způsob

Hlavičkový soubor `Vektor.h` obsahuje deklaraci i definici šablony třídy `TVektor`.

```
// soubor Vektor.h
#ifndef VektorH
#define VektorH

template <class T, int n> class TVektor {
   T   a[n];
public:
   TVektor(const T& t = T()) { for (int i = 0; i < n; i++) a[i] = t; }
   T& operator[](int i) { return a[i]; }
   void Vypis() const;
};

template <class T, int n>
void TVektor<T, n>::Vypis() const
{
   for (int i = 0; i < n; i++) cout << a[i] << ' ';
   cout << '\n';
}

#endif
```

V každém souboru .cpp (modulu), v němž bude uveden řádek

```
#include "Vektor.h"
```

bude mít překladač k dispozici definici šablony `TVektor` a proto si vytvoří hned na místě potřebné instance. Může se ale stát, že ve dvou různých modulech téhož programu vznikne stejná instance,

např. `TVektor<int>`. Spojovací program (linker) pak při spojování modulů do výsledného programu sloučí identické instance z různých modulů.

Pokud by byla definována explicitní specializace metody `Vypis`, její informativní deklarace se uvede v hlavičkovém souboru `Vektor.h` a definice v modulu `Vektor.cpp`:

```
// soubor Vektor.h
template <>
void TVektor<double, 2>::Vypis() const;

// soubor Vektor.cpp
#include "Vektor.h"
template <>
void TVektor<double, 2>::Vypis() const
{
   printf("%.2f, %.2f\n", a[0], a[1]);
}
```

Pokud by byla definována explicitní specializace šablony třídy `TVektor`, její definice se uvede v hlavičkovém souboru `Vektor.h`. Definice jejích metod se uvede v modulu `Vektor.cpp`.

Parciální specializace šablony třídy `TVektor` se uvedou v hlavičkovém souboru `Vektor.h`.

2. způsob – varianta A

Hlavičkový soubor `Vektor.h` obsahuje pouze informativní deklaraci šablony třídy `TVektor`:

```
// soubor Vektor.h
#ifndef VektorH
#define VektorH

template <class T, int n> class TVektor {
   T   a[n];
public:
   TVektor(const T& t = T()) { for (int i = 0; i < n; i++) a[i] = t; }
   T& operator[](int i) { return a[i]; }
   void Vypis() const;
};

#endif
```

Modul `Vektor.cpp` obsahuje definice složek šablony třídy `TVektor`. Každá tato definice začíná klíčovým slovem `export`:

```
// soubor Vektor.cpp
#include "Vektor.h"

export template <class T, int n>
void TVektor<T, n>::Vypis() const
{
   for (int i = 0; i < n; i++) cout << a[i] << ' ';
   cout << '\n';
}
```

V každém modulu, v němž se tato šablona používá, vytvoří překladač odkaz na příslušnou instanci jako na externí objekt, který existuje někde jinde.

V prostředí C ++ Builder 6 sice existuje klíčové slovo `export`, ale je rezervováno pro pozdější použití. V prostředí Visual C++ 2003 klíčové slovo export ani neexistuje. To znamená, že nelze tento 2. způsob v uvedených prostředích používat.

2. způsob – varianta B

Na rozdíl od varianty A definice složek šablony třídy `TVektor` v modulu `Vektor.cpp` neobsahují klíčové slovo `export`. Všechny instance šablony `TVektor` použité v programu musí být v modulu `Vektor.cpp` explicitně vytvořeny, např. pro typ `int`:

```
template class TVektor<int>;
```