

C++0x

1234 Co je C++0x



začátečník

Standard jazyka C++, jak byl přijat v roce 1998 a revidován v roce 2003 [ISO Cpp], je neúplný. Přijetí nové verze standardu, která by měla rozšířit jeho standardní knihovnu a upravit některé syntaktické konstrukce, se očekávalo do konce prvního desetiletí nového století, a proto se pro něj vžilo označení C++0x, kde „x“ mělo být nahrazeno rokem. V době závěrečných korektur této knihy (květen 2010) stále ještě nebyl publikován, je k dispozici pouze návrh [CPP0x], současné překladače – např. Visual C++ 2010 nebo g++ – však už některé z novinek implementují.

1235 Podpora C++0x v překladači g++



pokročilý

Překladače g++ verzí 4.3, 4.4, 4.5 a 4.6 podporují některé z novinek připravovaného standardu, nikoli ovšem automaticky. Chceme-li tuto podporu zapnout, použijeme v příkazové řádce přepínač `-std=c++0x`.

Chceme-li zároveň povolit i rozšíření GNU, použijeme přepínač `-std=gnu++0x`.

1236 Nové znakové typy a řetězcové literály



začátečník

Vzhledem ke stále širší podpoře kódování Unicode je pochopitelné, že C++0x podporuje i nové typy řetězcových literálů, a to ve formátu UTF-8, UTF-16 a UTF-32. Za tím účelem jsou k dispozici znakové typy `char16_t` a `char32_t`.

Odpovídající znakové literály vytvoříme pomocí předpon `u8`, `u` a `U` takto:

```
u8"Toto je řetězec v UTF-8."
```

```
u"Toto je řetězec v UTF-16."
```

```
U"Toto je řetězec v UTF-32."
```

První z těchto řetězců představuje vlastně obvyklé pole typu `const char[]`. Druhý je pole typu `const char16_t[]` a třetí je pole typu `const char32_t[]`.

1237 Ukazatel nikam



začátečník

V jazycích C a C++ se pro „ukazatel nikam“ tradičně používá hodnota 0 nebo makro `NULL` definované v hlavičkovém souboru `<stdlib.h>`, jehož typ ale není standardem přesně stanoven. Nový standard by pro měl pro ukazatel nikam zavést klíčové slovo `nullptr`. Představuje hodnotu typu `std::nullptr_t` a znamená samozřejmě hodnotu.

1238 Nová podoba příkazu for (foreach)



V mnoha programovacích jazycích najdeme příkazy, které umožňují projít celý obsah kontejneru. Nyní bude takovýto příkaz – obvykle označovaný jako příkaz *foreach* – i v jazyce C++. To znamená, že budeme moci napsat

```
int pole[3] = {1, 2, 3};
for(int n : pole)
{
    cout << n << endl;
}
```

Za deklarací iterační proměnné (parametru cyklu) následuje dvojtečka a pak kontejner, který chceme projít. Takto by mělo být možno projít jak klasická pole z jazyka C, tak i všechny kontejnery ze STL (přesněji všechny kontejnery, které implementují metody `begin()` a `end()`, jež vracejí iterátory ukazující na první a za poslední prvek).



Poznámka: Chceme-li v tomto cyklu měnit hodnoty uložené v procházeném kontejneru, deklarujeme parametr cyklu jako referenci.

1239 Zapomeňte na klíčové slovo auto pro paměťovou třídu



Jednou z novinek C++0x je zrušení klíčového slova `auto` jako specifikátoru paměťové třídy automatických proměnných. Pro tyto proměnné se nyní nepoužívá žádné klíčové slovo. (Upřímně řečeno, neumím si představit, že by to někomu chybělo.)

V nové verzi C++ má klíčové slovo `auto` jiný význam – viz následující tip 1240.

1240 Jsme líní specifikovat typ proměnné



Jestliže proměnnou v deklaraci inicializujeme, nemusíme v C++0x psát typ proměnné; místo toho použijeme klíčové slovo `auto`. Překladač si typ proměnné odvodí podle typu inicializátoru. To znamená, že můžeme napsat

```
auto x = 67;
```

a překladač pochopí, že `x` je proměnná typu `int` s počáteční hodnotou 67.

Tuto konstrukci oceníme například při použití iterátorů jako parametrů cyklu: Je-li seznam proměnná typu `list<string>`, je zápis

```
for(auto i = seznam.begin(); i != seznam.end(), i++)
{ // zpracování...
```

jistě přehlednější než

```
for(list<string>::iterator i = seznam.begin(); i != seznam.end(), i++)
{ // zpracování...
```

V některých situacích – např. při práci s lambda-výrazy – se bez tohoto klíčového slova neobejdeme.



Poznámka: Použití klíčového slova `auto` neznamena, že deklarovaná proměnná nemá typ. C++ je a zůstává typovým jazykem, beztypová proměnná v něm není možná. Znamená pouze, že typ nechceme psát, že si ho má překladač odvodit sám.

1241 Reference na r-hodnotu



pokročilý

Nová verze C++ by měla umožňovat deklaraci referencí na r-hodnotu; k její specifikaci slouží symbol `&&`. Deklarujeme-li např. funkci

```
void vypis(int&& x)
{
    x++;
    cout << x << endl;
}
```

a zavoláme-li ji příkazem

```
vypiš(x+1);
```

vypíše hodnotu $x + 2$.

V podstatě jde o referenci inicializovanou pomocnou proměnnou, do které program uloží hodnotu použitou při inicializaci (tedy hodnotu skutečného parametru).

1242 Aserce v době překladu



začátečník

Chceme-li zajistit, že v době překladu bude splněna jistá podmínka, můžeme místo direktiv pro podmíněný překlad použít klíčové slovo `static_assert`, a to takto:

```
static_assert(konstantní-výraz, řetězcový-literál);
```

Konstantní výraz musí umět vyhodnotit překladač. Je-li nenulový, nezpůsobí tento příkaz nic, jinak se vypíše zadaný *řetězcový literál* a překlad skončí.

Jde vlastně o analogii makra `assert()`, ovšem vyhodnocovanou v době překladu.

1243 Inicializace kontejnerů



začátečník

V C++0x bychom měli smět použít v deklaraci instance kontejneru inicializaci pomocí složených závorek, např.

```
vector<int> vek = {3, 4, 6, 7, 99};
```

1244 Volání jiného konstruktoru téže třídy



Současné C++ neumožňuje zavolat z jednoho konstruktoru jiný konstruktore téže třídy. C++0x by to však již mělo dovolovat, a to tak, že v inicializační části konstruktoru uvedeme volání konstruktoru téže třídy s potřebnými parametry. Například takto:

```
class numero {
    int num;
public:
    numero(int n): num(n){}
    numero(): numero(0){} // Využije služeb jiného konstruktoru
    // a další metody
};
```

1245 Inicializace složek v deklaraci



V C++0x by také mělo být možné zapisovat inicializaci datových složek přímo v deklaraci, například

```
class počet {
    int n = 5;
    // ...
};
```

1246 Když nevíme přesně typ



Představte si, že máme proměnné typů `typ1` a `typ1`, které jsou někde zavedeny pomocí deklarace `typedef`,

```
typ1 x = 86;
typ2 y = 75;
```

a chceme deklarovat funkci, která vrátí výsledek stejného typu jako je součet `x + y`. Pak nám pomůže nové klíčové slovo `decltype`:

```
decltype(x+y) f(typ1 a, typ2 b)
{
    return a+b;
}
```

Toto klíčové slovo představuje specifikátor typu a označuje typ výrazu, který za ním uvedeme v závorkách.

1247 Alternativní zápis deklarace funkce



Někdy se nám bude hodit zapsat v deklaraci funkce typ výsledku až za seznam parametrů; to nyní jde, musí mu ale předcházet symbol `->`; před identifikátor funkce napíšeme

klíčové slovo `auto`. Například následující deklarace zavádí funkci typu `int`, jež počítá druhou mocninu předané hodnoty:

```
auto sqr(int x) -> int { return x*x; };
```

Tyto možnosti oceníme zejména v souvislosti se šablonami.

1248 Funkce, jejíž typ výsledku závisí na několika parametrech šablony



Chceme napsat šablonu funkce, jež vrátí součet dvou čísel různých typů. Výsledek musí být většího z typů, tedy stejného typu jako součet. To jazyk C++ dosud neumožňoval. Řešení založené na nové syntaxi deklarace funkce a operátoru `decltype` vypadá takto:

```
template<typename T, typename U>
auto fun(T x, U y) -> decltype (x+y)
{
    return x+y;
}
```

Argumentem operátoru `decltype` musí být výraz, a tak je třeba typ uvést až v místě, kde máme k dispozici formální parametry.



Poznámka: V literatuře se doporučuje použít zde pomocnou funkci `forward<>()`, jež přetypuje dané parametry na odkaz na r-hodnotu zadaného typu:

```
template<typename T, typename U>
auto fun(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
{
    return forward<T>(t) + forward<U>(u);
};
```

Volání `forward<A>(b)` vrátí `static_cast<A&&>(b)`.



Poznámka: Tato funkce by měla být deklarována v hlavičkovém souboru `<algorithm>`.

1249 Co je to lambda-výraz



Lambda-výraz, nebo také λ -výraz, je konstrukce, která se hojně používá především ve funkcionálních programovacích jazycích; v poslední době se ovšem lambda-výrazy objevují i v jazycích středního proudu, a jazyk C++ samozřejmě nemůže zůstat stranou.

Jde v podstatě o alternativní zápis definice funkce, který ovšem umožňuje vytvořit i nepojmenovanou funkci (typicky „pro jedno použití“, tedy např. jako parametr předaný některému z algoritmů ze standardní knihovny).

1250 Transformace vektoru (nepojmenovaná funkce v C++0x)



Máme vektor celých čísel,

```
vector<int> celá
```

a chceme všechna čísla v něm nahradit jejich druhými mocninami. K tomu můžeme v C++0x použít příkaz

```
transform(celá.begin(), celá.end(), celá.begin(),
          [=](int x)->int{return x*x;});
```

Zápis `[=](int x)->int{return x*x;}` je jednoduchým příkladem lambda-výrazu, tedy objektu představujícího nepojmenovanou funkci, kterou předáváme algoritmu `transform<>()` jako parametr. Zde tato nepojmenovaná funkce počítá druhou mocninu.



Poznámka: Funkce `transform<>()` je deklarována v hlavičkovém souboru `<algorithm>`.

1251 Lambda-výraz



Podívejme se na pravidla pro vytvoření lambda-výrazu.

1. Lambda-výraz začíná hranatými závorkami `[]`, v nichž může být specifikace *záchytu* (capture). Zpravidla tam je znak `=` nebo `&`.
2. Za ním následuje seznam parametrů podobně jako v běžné deklaraci funkce.
3. Pak může, ale nemusí následovat klíčové slovo `mutable`, které říká, že lambda-výraz smí měnit hodnoty „zachycených“ lokálních proměnných viditelných v místě jeho deklarace.
4. Dále může, ale nemusí následovat specifikace výjimek (podobná jako v běžné deklaraci funkce).
5. Za touto hlavičkou následuje `->` a typ výsledku.
6. Nakonec uvedeme tělo funkce.

Schématicky to můžeme vyjádřit takto:

```
[=] (int r) mutable throw() -> int { return 11; }
(1)  (2)      (3)      (4)  (5)          (6)
```

Čísla v závorkách odkazují na body předchozího výčtu.



Poznámka: Lambda-výraz v tomto schématu slouží jako příklad. Neznamená to, že by lambda výraz mohl mít jen jeden parametr nebo že specifikace výjimek musí být prázdná. Také specifikace záchytu může být složitější.

1252 Okolní proměnné



Lambda-výraz může pracovat i s proměnnými, které jsou viditelné v místě jeho deklarace. Tomu se říká *záchyt* (capture). Uvedeme-li jako první ve specifikaci záchytu znak `=`, znamená to, že se budou lambda-výrazu předávat hodnotou (bude tedy pracovat s jejich kopiemi).

Uvedeme-li zde `&`, znamená to, že se budou zachycené proměnné předávat odkazem.

Podívejme se na příklad. Platí-li např. ve funkci `main()` deklarace

```
int x = 67; // Lokální proměnná
auto f = [&]() -> void {x++;}; // Lambda-výraz, který pracuje s x
```

a napíšeme-li dále

```
f(); // Volání lambda-výrazu
```

bude `x` obsahovat hodnotu 68, neboť tato proměnná je zachycena odkazem, takže se změní.

Upravíme-li deklaraci lambda-výrazu do tvaru

```
auto f = [=]() -> void {x++;};
```

program se nepřeloží, neboť lambda-výraz – není-li v jeho deklaraci uvedeno klíčové slovo `mutable` – nesmí měnit lokální automatickou proměnnou zachycenou hodnotou.

Napišeme-li

```
auto f = [=]() mutable -> void {x++;};
```

program se přeloží, avšak lambda-výraz změní pouze lokální kopii proměnné `x`, nikoli samotnou proměnnou `x`.

1253 Podrobnější specifikace záchytu



Ve specifikaci záchytu můžeme uvést způsob zacházení s jednotlivými proměnnými. Úvodní `=`, resp. `&` ve specifikaci záchytu určuje způsob zacházení s proměnnými, pro které nebude explicitně specifikován. Vedle toho ale můžeme přesně určit zacházení s jednotlivými proměnnými. Napišeme-li např.

```
auto f = [=,&x]() -> void {x++;};
```

říkáme tím, že se všechny okolní lokální automatické proměnné, které jsou v místě deklarace vidět, chceme předávat hodnotou, pouze proměnnou `x` chceme předávat odkazem. Napišeme-li

```
auto f = [y,&x]() -> ...
```

říkáme tím, že `y` chceme předávat hodnotou a `x` odkazem.



Poznámka: Je třeba zdůraznit, že specifikace záchytu se týká lokálních automatických proměnných, nikoli globálních proměnných.

1254 Jakého typu je lambda-výraz



znalec

Lambda-výraz představuje hodnotu jistého volatelného typu, a to objektového typu s přetíženým operátorem volání funkce. Přesný typ není specifikován a není programátorovi ani dostupný. Proto jsme v předchozích tipech použili klíčové slovo `auto`.

1255 Deklarace šablony s proměnným počtem parametrů



znalec

C++0x umožňuje deklarovat šablony s proměnným počtem parametrů. Například takto:

```
template<typename... HODNOTY> class entice;
```

Specifikace proměnného počtu parametrů se skládá z klíčového slova `template`, za nímž následuje symbol výpustky – tři tečky bezprostředně za sebou – a jméno „balíku parametrů“ (parameter pack).

Poznamenejme, že pro šablony s proměnným počtem parametrů se používá označení „variadické šablony“ (variadic templates). Může se jednat jak o šablony funkcí, tak o šablony objektových typů.

1256 Definice šablony s proměnným počtem parametrů



znalec

V definici šablony s proměnným počtem parametrů využíváme mechanismu částečné a úplné specializace. Ukážeme si to na jednoduchém příkladu. Předpokládejme, že platí deklarace

```
template<typename... PARAM> struct počet;
```

`PARAM` je jméno „balíku parametrů“. Nejprve definujeme specializaci bez parametrů:

```
template<>
struct počet<> {
    static const int hodnota = 0;
};
```

Pak definujeme rekurzivně ostatní případy:

```
template<typename T, typename... Args>
struct count<T, PARAM...> {
    static const int hodnota = 1 + počet<PARAM...>::hodnota;
};
```



Poznámka: Uvedený příklad je v podstatě převzat z [variad].

1257 Speciální funkce ve standardní knihovně



Jazyk C++ pomalu nahrazuje Fortran při vědeckotechnických výpočtech, a tak se nelze divit, že nezi požadavky na rozšíření byly i implementace tzv. speciálních funkcí. Pod tímto označením se skrývá řada matematických funkcí, které se v matematické fyzice objevují jako řešení jistých diferenciálních rovnic a často se používají v technických výpočtech.

Jejich přehled ukazuje následující tabulka:

Funkce	Popis
<code>laguerre(unsigned n, double x)</code>	Laguerreův polynom $L_n(x)$
<code>assoc_laguerre(unsigned n, unsigned m, double x)</code>	přidružený Laguerreův polynom $L_n^m(x)$
<code>legendre(unsigned r, double x)</code>	Legendreův polynom $P_r(x)$
<code>assoc_legendre(unsigned r, unsigned m, double x)</code>	přidružená Legendreova funkce $P_r^m(x)$
<code>sph_legendre(unsigned r, unsigned m, double theta)</code>	přidružená sférická Legendreova funkce $Y_r^m(\theta, 0)$
<code>tgamma(double x)</code>	funkce $\Gamma(x)$
<code>lgamma(double x)</code>	logaritmus absolutní hodnoty $\Gamma(x)$
<code>beta(double x, double y)</code>	funkce $B(x, y)$
<code>erf(double x)</code>	chybová funkce $\Phi(x) = \text{erf}(x)$
<code>erfc(double x)</code>	doplňková chybová funkce $\text{erfc}(x)$
<code>ellint_1(double k, double phi)</code>	neúplný eliptický integrál prvního druhu $F(k, x)$
<code>ellint_2(double k, double phi)</code>	neúplný eliptický integrál druhého druhu $E(k, \varphi)$
<code>ellint_3(double k, double nu, double phi)</code>	neúplný eliptický integrál třetího druhu $\Pi(u, k, \varphi)$
<code>comp_ellint_1(double phi)</code>	úplný eliptický integrál prvního druhu $K(k) = F(k, \pi/2)$
<code>comp_ellint_2(double k)</code>	úplný eliptický integrál druhého druhu $E(k, \pi/2)$
<code>comp_ellint_3(double k, double nu)</code>	úplný eliptický integrál třetího druhu $\Pi(u, k, \pi/2)$
<code>hyperg(double a, double b, double c, double x)</code>	hypergeometrická funkce $F(a, b; c; x)$
<code>conf_hyperg(double a, double c, double x)</code>	konfluentní hypergeometrická funkce $F(a, c; x)$
<code>cyl_bessel_i(double nu, double x)</code>	regulární modifikovaná cylindrická Besselova funkce $I_\nu(x)$
<code>cyl_bessel_j(double nu, double x)</code>	cylindrická Besselova funkce prvního druhu $J_\nu(x)$

Funkce	Popis
<code>cyl_bessel_k(double nu, double x)</code>	iregulární modifikovaná cylindrická Besselova funkce $K_\nu(x)$
<code>sph_bessel(double nu, double x)</code>	sférická Besselova funkce prvního druhu $j_\nu(x)$
<code>cyl_neumann(double nu, double x)</code>	cylindrická Neumannova funkce $N_\nu(x)$ (cylindrická Besselova funkce druhého druhu)
<code>sph_neumann(double n, double x)</code>	sférická Neumannova funkce $n_\nu(x)$ (sférická Besselova funkce druhého druhu)
<code>expint(double x)</code>	exponenciální integrál $Ei(x)$
<code>hermite(unsigned n, double x)</code>	Hermiteův polynom $H_n(x)$
<code>reimann_zeta(double x)</code>	Reimannova funkce $\zeta(x)$

Všechny tyto funkce jsou definovány pro typy `float`, `double` a `long double` a vracejí hodnoty stejného typu. Návrh obsahuje jak přetížené verze, které se v souladu s pravidly jazyka C++ rozlišují podle typu parametru, tak i verze odpovídající zvyklostem jazyka C, tedy rozlišené v identifikátoru koncovým `f`, resp. `l`.

Tyto funkce mohou vedle proměnné x záviset na dalších parametrech. Například Besselovy funkce jsou vlastně funkcemi dvou proměnných – vedle proměnné x závisejí ještě na „indexu“ ν . Implementace může stanovit rozsah hodnot proměnné x a indexu, pro které má výpočet smysl. Na podobná omezení můžeme narazit i u dalších speciálních funkcí.