[Základy OOP v C++: Od C k C++](http://www.builder.cz/rubriky/c/c--/zaklady-oop-v-c-od-c-k-c--155657cz)

<http://www.builder.cz/rubriky/c/c--/zaklady-oop-v-c-od-c-k-c--155657cz>

* [Základy OOP v C++: Od C k C++](http://www.builder.cz/rubriky/c/c--/zaklady-oop-v-c-od-c-k-c--155657cz)
* [Základní pojmy objektově orientovaného programování](http://www.builder.cz/rubriky/c/c--/zakladni-pojmy-objektove-orientovaneho-programovani-155665cz)
* [Vytváření tříd, instance třídy, zasílání zpráv v C++](http://www.builder.cz/rubriky/c/c--/vytvareni-trid-instance-tridy-zasilani-zprav-v-c--155667cz)
* [Vytváření instancí - konstruktory, destruktory](http://www.builder.cz/rubriky/c/c--/vytvareni-instanci-konstruktory-destruktory-155678cz)
* [Kopírovací konstruktor v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-konstruktor-v-c--155685cz)
* [Jednoduchá dědičnost v C++](http://www.builder.cz/rubriky/c/c--/jednoducha-dedicnost-v-c--155692cz)
* [Časná versus pozdní vazba - úvod do polymorfismu v C++](http://www.builder.cz/rubriky/c/c--/casna-versus-pozdni-vazba-uvod-do-polymorfismu-v-c--155697cz)
* [Polymorfismus - dokončení](http://www.builder.cz/rubriky/c/c--/polymorfismus-dokonceni-155701cz)
* [Vícenásobná dědičnost v C++](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c--155707cz)
* [Vícenásobná dědičnost v C++ - opakovaná dědičnost](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-opakovana-dedicnost-155717cz)
* [Vícenásobná dědičnost v C++ - volání konstruktorů a destruktorů](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-volani-konstruktoru-a-destruktoru-155723cz)
* [Přetěžování operátorů v C++ 1.díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-1-dil-155728cz)
* [Přetěžování operátorů v C++ 2. díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-2-dil-155729cz)
* [Vstupní a výstupní operace pomocí datových proudů v C++](http://www.builder.cz/rubriky/c/c--/vstupni-a-vystupni-operace-pomoci-datovych-proudu-v-c--155739cz)
* [Přetěžování operátorů << a >> pro datové proudy v C++](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-lt-lt-a-gt-gt-pro-datove-proudy-v-c--155741cz)
* [Neformátovaný vstup a výstup v C++](http://www.builder.cz/rubriky/c/c--/neformatovany-vstup-a-vystup-v-c--155754cz)
* [Paměťové proudy v C++](http://www.builder.cz/rubriky/c/c--/pametove-proudy-v-c--155755cz)
* [Prostory jmen v C++](http://www.builder.cz/rubriky/c/c--/prostory-jmen-v-c--155766cz)
* [Řetězce v C++](http://www.builder.cz/rubriky/c/c--/retezce-v-c--155767cz)
* [Výjimky v C++](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c--155798cz)
* [Výjimky v C++ - výjimky tvoří dědičnou hierarchii](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-vyjimky-tvori-dedicnou-hierarchii-155809cz)
* [Výjimky v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-dokonceni-155810cz)
* [Dynamická identifikace typů v C++](http://www.builder.cz/rubriky/c/c--/dynamicka-identifikace-typu-v-c--155828cz)
* [Přetypování v C++](http://www.builder.cz/rubriky/c/c--/pretypovani-v-c--155842cz)
* [Problémy s typy při vícenásobné dědičnosti](http://www.builder.cz/rubriky/c/c--/problemy-s-typy-pri-vicenasobne-dedicnosti-155865cz)
* [Šablony funkcí v C++](http://www.builder.cz/rubriky/c/c--/sablony-funkci-v-c--155874cz)
* [Šablony datových typů v C++](http://www.builder.cz/rubriky/c/c--/sablony-datovych-typu-v-c--155883cz)
* [Vnitřní typy u parametrů šablon, vnořené šablony v C++](http://www.builder.cz/rubriky/c/c--/vnitrni-typy-u-parametru-sablon-vnorene-sablony-v-c--155889cz)
* [Pole s libovolným intervalem indexování v C++](http://www.builder.cz/rubriky/c/c--/pole-s-libovolnym-intervalem-indexovani-v-c--155896cz)
* [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/datove-kontejnery-v-c-uvod-do-stl-155907cz)
* [Vector - datový kontejner v C++](http://www.builder.cz/rubriky/c/c--/vector-datovy-kontejner-v-c--155916cz)
* [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/iteratory-v-c--155928cz)
* [Šablona vector v C++ a iterátory](http://www.builder.cz/rubriky/c/c--/sablona-vector-v-c-a-iteratory-155932cz)
* [Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/asociativni-pole-v-c--155938cz)
* [Množina v C++](http://www.builder.cz/rubriky/c/c--/mnozina-v-c--155952cz)
* [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/funkcni-objekty-v-c--155955cz)
* [Standardní funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/standardni-funkcni-objekty-v-c--155969cz)
* [Úvod do standardních algoritmů v C++](http://www.builder.cz/rubriky/c/c--/uvod-do-standardnich-algoritmu-v-c--155983cz)
* [Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-a-presouvaci-algoritmy-v-c--156005cz)
* [Vyhledávací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/vyhledavaci-algoritmy-v-c--156011cz)
* [Skenovací (prohlížecí) algoritmy v C++](http://www.builder.cz/rubriky/c/c--/skenovaci-prohlizeci-algoritmy-v-c--156016cz)
* [Transformační algoritmy v C++](http://www.builder.cz/rubriky/c/c--/transformacni-algoritmy-v-c--156022cz)
* [Řadící algoritmy v C++](http://www.builder.cz/rubriky/c/c--/radici-algoritmy-v-c--156023cz)
* [Halda v C++](http://www.builder.cz/rubriky/c/c--/halda-v-c--156036cz)
* [Standardní algoritmy v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/standardni-algoritmy-v-c-dokonceni-156045cz)
* [Automatické ukazatele v C++](http://www.builder.cz/rubriky/c/c--/automaticke-ukazatele-v-c--156054cz)
* [Inteligentní ukazatel - čítač referencí v C++](http://www.builder.cz/rubriky/c/c--/inteligentni-ukazatel-citac-referenci-v-c--156071cz)
* [Použití čítače referencí v C++](http://www.builder.cz/rubriky/c/c--/pouziti-citace-referenci-v-c--156074cz)
* [Kopírování velkých objektů v C++](http://www.builder.cz/rubriky/c/c--/kopirovani-velkych-objektu-v-c--156115cz)
* [Řízené kopírování prvků v poli v C++](http://www.builder.cz/rubriky/c/c--/rizene-kopirovani-prvku-v-poli-v-c--156130cz)
* [**Dokončení seriálu objektově orientované programování v C++**](http://www.builder.cz/rubriky/c/c--/dokonceni-serialu-objektove-orientovane-programovani-v-c--156137cz)

**6. prosince 2000, 00.00 | první díl seriálu o objektově orientovaném programování v C++. Podobně jako v ostatních článcích se nejprve podíváme jaký je rozdíl mezi klasickým Céčkem a C++.**

**Úvod**

   Úvodem musím říci, že můj seriál článků "Základy objektově orientovaného programováni v C++" je určen lidem, kteří znají jazyk C, a chtějí se naučit objektově programovat v jazyce C++. Pro ty, kteří jazyk C neznají, bych nejprve doporučil seriál článků mého kolegy pana Růžičky, který také vychází na serveru Builder.cz. Dále bych chtěl také upozornit, že veškerý můj výklad se bude týkat ANSI normy jazyka C++, a budu-li vycházet z jazyka C, budu vycházet jen z ANSI normy jazyka C. Je tak zaručen bezproblémový přenos zdrojových textů mezi překladači různých firem, a dokonce i mezi různými operačními systémy. Bohužel ANSI norma C++ narozdíl od ANSI C, je známa teprve 2 roky, a mnoho starších překladačů ji nedodržuje, i když to o sobě prohlašují. Já všechny příklady zdrojových textů, které zde budu prezentovat, budu zkoušet na překladačích Borland C++ Builder BETA 0.0.4.212, a GNU C/C++ Compiler. Druhý zmiňovaný je standardní překladač pod OS Linux, který by měl ANSI C++ dodržovat. První zmiňovaný je testovací verze Builderu (ani ne verze 1) fy Borland, která jej uvolnila pro otestování. Můžete jej získat volně na některém z CD časopisu Chip, které vyšlo v roce 1996, nebo 1997. Tento překladač je již poměrně starý, ale dosud jsem nenarazil na nic z ANSI normy co by nepodporoval. Narazím-li na něco v průběhu psaní tohoto seriálu článku, upozorním vás na to. Máte-li však novější verzi Borland C++ Builderu, je to dobře pro vás. Od fy Borland by měl prý vyhovovat i překladač Borland C/C++ 5. Starší už ale asi ne. O překladačích jiných firem bohužel nevím nic, proto si budete muset zjistit sami nakolik podporují ANSI normu.

   Protože snad 99.9% všech knih, či článků začíná na úvod malým programem, který pozdraví svět, začnu takto i já. Protože čtenář, který teprve začíná s C++ asi nebude vědět co se děje, doporučuji tento program jen opsat bez přemýšlení. Vstupně výstupním operacím se chci věnovat až někdy kolem 10. článku. A nyní slibovaný program pozdrav.cpp:  
  
*#include <iostream.h>  
  
void main(void)  
{  
   cout << "Ahoj svete" << endl;  
}*

**Je C podmnožinou C++?**

   Tuto otázku si klade asi každý, kdo s C++ začíná. Odpověď není vůbec tak jednoduchá jak se zdá. Já bych odpověděl: "Až na několik málo vyjímek si ano". Obecně platí, že C++ je rozšíření jazyka C. Jazyk C je čistě strukturovaný programovací jazyk. V jazyce C++ lze programovat objektově, nebo také strukturovaně. Bohužel jazyk C++ umožňuje i jakýsi mix strukturovaného, a objektově orientovaného programování. Tento mix vám rozhodně nedoporučuji. Zbytek tohoto článku budu věnovat strukturovanému programování v C++, a počínaje příštím článkem se budu věnovat jen objektově orientovanému programování.

**Problémy při přechodu z C do C++**

   Zde popíšu v čem není C s C++ kompatibilní. Je to těch několik málo vyjímek, o kterých jsem se zmínil v minulém odstavci. Není totiž každý program napsaný pro C kompilovatelný jako program napsaný v C++.

   **Znakové typy** - C++ zná 3 typy reprezentující znaky: *char, signed char, unsigned char*. Zatímco *char* napsaný v C je totéž jako *signed char*. Například předáte-li funkci, která jako parametr očekává *signed char*, parametr *char*, pravděpodobně vás upozorní překladač na nekompatibilitu typů. *Char* bude nakonec implementován nejspíš jako *signed char*, nebo jako *unsigned char*, ale překladač k nim přistupuje jako k různým typům. Platí to však jen pro typ znak. U všech ostatních typů, jako třeba *int*, zůstává vše stejné jako u jazyka C. Tedy např. *int* je stejný typ jako *signed int*.

   **Implicitní návratová hodnota** - Další nekompatibilitou je implicitní návratová hodnota funkce. Nenapíšete-li návratovou hodnotu funkce v jazyce C, překladač C podle ANSI normy předpokládá, že návratová hodnota je *int*. Překladač C++ však v takovém případě předpokládá jako návratový typ *void* - tedy "nic". Takže deklarace funkce:  
  
*MojeFunkce(int cislo);*  
  
v jazyce C je funkce, která vrací *int*, kdežto v C++ se jedná o deklaraci funkce vracející void.

Možná, že těchto drobných problémů bude více. Já jsem však přišel jen na tyto. Jinak lze program napsaný v jazyce C považovat za program v jazyce C++. Pro strukturované programování však C++ nabízí řadu příjemných rozšíření.

**Rozšíření jazyka C++ pro strukturované programování**

   Protože se v tomto seriálu článků chci zabývat objektově orientovaným programováním, popíšu jen letmo ty rozšíření, které se mi zdají opravdu nejvýznamnější.

   **Jednořádkový komentář** - Jednořádkové komentáře začínají dvěma znaky *//* a končí koncem řádku. Je asi dost překvapující, že tento komentář jazyk C podle ANSI normy nepodporuje, protože jej stejně snad všechny překladače jazyka C dovolují. Přesto v popisu ANSI normy C není, je až součástí jazyka C++. Abych uvedl nějaký příklad:  
  
*int  PocetCtvercu; // Udává počet čtverců*

   **Deklarace proměnných** - V C++ oproti C se mohou proměnné deklarovat kdekoliv v textu, nejenom na začátku aktuálního bloku.

   **Datový typ bool** - Dalším příjemným rozšířením jazyka C je datový typ *bool*. Typ *bool* může nabývat dvou hodnot *true* (logická 1), *false* (logická 0). Nad datovým typem *bool* jsou definovány logické operace *&&* (logický and), *||* (logický or), *!* (negace), *^* (logický xor). Tedy všechny logické operace, které lze použít v jazyce C v podmínkách. Příklad:   
  
*bool a;  
bool b = true;  
int c = 20;  
bool d = ((c == 20) || ( c<=5));  
a = !b;  
  
if (a)   
{  
. . .   
}*

   **Návratový typ funkce main** - Návratový typ funkce main už v C++ nemusí být int, ale mže být také void. Viz můj první program pozdrav.cpp.

   **Implicitní hodnoty parametrů funkcí** - Implicitní hodnoty parametrů v C++ jsou již asi pro programátora v jazyce C novinkou. Jestliže předpokládáte, že budete funkci často volat s nějakou hodnotou parametru, můžete tuto hodnotu uvést jako implicitní, a při volání ji v seznamu parametrů neuvádět. Nejlépe asi demonstruji na příkladu:  
  
*int soucet(int a, int b, int c = 0)     /\* 3. parametr je implicitně 0. \*/  
{  
  return (a+b+c);     /\* vracím součet parametrů\*/  
}  
  
void main(void)  
{  
int cislo;  
. . . // Nějaký program   
cislo = soucet(1,2,3);     /\* Zavolá se funkce soucet s parametry a = 1 , b = 2, c = 3 - To asi nikoho nepřekvapí.\*/  
. . . // Nějaký program   
cislo = soucet(2,3);     /\* Zavolá se funkce soucet s parametry a = 2 , b = 3, c = 0. Tedy je to stejný výsledek, jako bych napsal cislo = soucet(2,3,0). \*/  
}*

Implicitní hodnoty parametrů mohu zadávat jen v seznamu parametrů zprava. Pokud by tedy v naší funkci součet nemel parametr c implicitní hodnotu, parametr b by ji také nemohl mít. Parametr c (1. zprava) implicitní hodnotu má, proto lze definovat i implicitní hodnotu parametru b (2. zprava).

 **Přetěžování funkcí** - Přetížení funkcí je možnost deklarovat, i definovat více funkcí stejného jména s různým počtem, nebo s různými typy parametrů. Tuto vlastnost jazyk C neměl. Příklad: Mohu definovat dvě funkce se stejným jménem *max*.  
  
*int max(int a, int b)  
{  
  return (a>b)? a:b;  
}  
  
float max(float a, float b)  
{  
  return (a>b)? a:b;  
}*

Zavolám-li funkci max s parametry typu *float* vrátí mi větší z nich jako typ *float*. Zavolám-li funkci max s parametry typu *int* vrátí mi větší z nich jako typ *int*. Název funkce *max* jsem přetížil. Při přetěžování funkcí musíte dávat pozor na případné nejednoznačnosti. Kdyby jste v našem příkladě zavolali funkci *max(1.0,2.0)*, překladač by vás upozornil na chybu. Neví totiž, zda volat max s parametry float, nebo parametry implicitně přetypovat na int a volat max s parametry int. Tedy volání funkce je nejednoznačné, překladač neví, kterou funkci vybrat. Tento problém vyřešíte, jestliže budete volat *max((float)2.0,(float)3.5)*. Nejlepší způsob jak předcházet případným nejednoznačnostem je nepřetěžovat funkce tak, aby mezi parametry šlo provést implicitní přetypování.

   **Reference** - V C++ existuje vedle proměnné nějakého typu a ukazatele na proměnnou nějakého typu také reference na proměnnou nějakého typu. Mám-li to říci hodně neformálně, tak reference je ukazatel, se kterým se pracuje stejně jako se statickou proměnnou. S pojmem ukazatel se jistě každý programátor v jazyce C již musel setkat. Referenci demonstruji na příkladu:  
  
*int a;    // Proměnná typu int  
int &b = a;    /\* Reference na proměnnou a, b je nyní jiný název pro číslo, které reprezentuje proměnná a, b je přezdívka a (alias). \*/  
a++;     /\* Nyní jsem změnil hodnotu a, i hodnotu b. Změnil jsem vlastně jen jednu hodnotu, proměnná a i b reprezentují tutéž proměnnou. \*/*Jedná se vlastně o jakousi obdobu klasického ukazatele (pointeru) v jazyce C. Ale POZOR reference NENÍ ukazatel. Narozdíl od ukazatele nemůže ukazovat "nikam", tedy mít hodnotu NULL. Právě tento fakt je důvod, proč já osobně reference nepoužívám, ale používám "staré a dobré" pointery - ostatně jak uvidíte v následujících článcích. I když jsem někde četl, že právě tato vlastnost je výhodou referencí od ukazatelů, mně se naopak zdá, že je nevýhodou. U reference totiž musím již v době deklarace znát proměnnou, kterou bude zastupovat, což u ukazatele nemusím. Chci-li někde nějakým způsobem použít sdílení proměnných, použiji raději ukazatele. Reference má výhodu oproti ukazatelům snad jen při předávání parametrů funkcí odkazem. Například funkci swap, která vymění hodnotu dvou čísel, je zbytečné psát pomocí ukazatelů, když v C++ lze zapsat následovně:  
  
*swap(int &a, int &b)   
{  
  int c = a;  
  a = b;  
  b = c;  
}  
  
// A někde v programu ji volat takto:   
int a = 2, b = 9;  
swap(a,b);*Tento zápis je přece jenom čitelnější a "lehčí" než implementace stejné funkce v jazyce C.

**Závěr**

   Závěrem bych chtěl jen dodat, že rozšíření jazyka C++ oproti C pro strukturované programování je jistě mnohem více. Tento článek byl jen orientační, a hlavně odbočení od tématu seriálu o objektově orientovaném programování v C++. Zdálo se mi jen vhodné se o možnosti strukturovaného programování v jazyce C++ zmínit. Příště se chci podívat na základní pojmy objektově orientovaného programování. Vysvětlím v něm pojmy OOP jako jsou objekt, třída objektů, zpráva atd... A poté v dalších článcích chci ukázat, jak objektově programovat v jazyce C++.

[Základní pojmy objektově orientovaného programování](http://www.builder.cz/rubriky/c/c--/zakladni-pojmy-objektove-orientovaneho-programovani-155665cz)

**18. prosince 2000, 00.00 | V tomto článku bych se chtěl zmínit o základních pojmech objektově orientovaného programování.**

Základní pojmy objektově orientovaného programování

V tomto článku bych se chtěl zmínit o základních pojmech objektově orientovaného programování (Dále jenom OOP). Možná se bude tento článek někomu zdát příliš teoretický, protože zde nebude ani řádka zdrojového textu. Mě se ale zdá vhodné zmínit se o významu jednotlivých pojmů jako je třída, instance třídy a podobně, než začnu popisovat jejich implementace v jazyce C++. V dalších kapitolách svého seriálu článků se budu odkazovat na zde definované pojmy. U jednotlivých pojmů úmyslně pro vaší lepší představu vybírám příklady nejen z oblasti programování, ale i z reálného světa.

Objekt

Jak již asi každého napadne, základem OOP je objekt. Co je to objekt? - Objekt je entita, která má svou identitu (Každý objekt lze jednoznačně odlišit od objektu jiného.) a své vlastnosti (Každý objekt má nějaký vnitřní stav a nějak se chová vůči svému okolí). Každý asi i bez této definice podvědomě tuší, co to objekt je. Objekty jsou například: strom, auto, člověk, ale také datový soubor, okno v grafickém operačním systému, tlačítko v grafickém operačním systému, různé datové struktury jako je pole, fronta, zásobník atd... Je zřejmé, že všechny tyto objekty splňují podmínky uvedené v definici. Každý objekt, který má být užitečný, musí poskytovat svému okolí nějaké služby. Například auto se umí rozjet, zastavit atd. Okno v grafickém OS se umí minimalizovat, zavřít, aktivovat, atd. Datový soubor se umí otevřít, zavřít, lze do něj zapsat, lze z něj číst. Uživatel objektu k jednotlivým objektům přistupuje jako k tak zvaným "černým skříňkám". Nezajímá se, jakým způsobem objekty služby poskytují, ale zajímá se jen jaké služby poskytují. Například každý ví, co způsobí sešlápnutí brzdového pedálu v autě, ale málokdo by byl schopen přesně popsat, co se v motoru v tom okamžiku přesně děje. Pro zabrždění auta to prostě nemusí vědět. Stejně tak mnoho programátorů (já taky) ve svých programech běžně otevírá i zavírá soubor, ale příliš netuší, co všechno program a operační systém při otevírání souborů provádí. Právě tento přístup "černých skříněk" k objektům v OOP je dost podstatný. Programátor, který používá objekt, se nestará o to, jak byl naprogramován. Vytvořím-li objekt v OOP, vždy se jedná o jakýsi model objektu reálného světa.

Zpráva

Objekt nikdy neexistuje sám v nějakém vakuu. Objekt je vždy obklopen jinými objekty, kterým poskytuje svoje služby a po kterých služby požaduje. Požadavku na provedení služby se říká zpráva. "Zašlu-li" objektu zprávu, nastane jeden ze dvou případů:

 Objekt zprávě rozumí (Je schopen ji přijmout.) a zareaguje na ni nějakou svou metodou. Například sešlápnutím pedálu brzdy v autě vyvolám metodu motoru - brždění.

 Objekt zprávě nerozumí (Není schopen zprávu přijmout.) a nijak nezareaguje, případně jako svou reakci oznámí uživateli, že není schopen zprávu přijmout. Například u okna v grafickém OS nelze sešlápnout brzdový pedál.

Množina zpráv, které objekt může přijmout, se nazývá rozhraní (Anglicky interface). Popisu, jakým způsobem lze objektům zasílat zprávy (Pořadí a podobně), se nazývá komunikační protokol objektu. Například datový soubor lze nejprve otevřít pro zápis, až poté do něj zapisovat, poté jej zase zavřít. Pojmy rozhraní objektu a komunikační protokol objektu se v OOP rovněž velmi často používají. Jak jsem se zmiňoval v předchozím odstavci o "černých skříňkách", uživatel objektu zná jen rozhraní objektu, a komunikační protokol objektu. K datům (atributům) v objektu lze přistupovat jen pomocí rozhraní. Tedy vlastně vyvoláním metod. Tento jev se nazývá zapouzdření. Více o zapouzdření se budu věnovat v následujícím článku, kde zapouzdření ukážu přímo v jazyce C++.

Dalsím důležitým pojmem v OOP je polymorfismus (Mnohotvarost). Mohou existovat různé objekty, které na stejnou zprávu zareagují různě - polymorfismus. O polymorfismu se zmíním podrobněji v následujících kapitolách, kde ukážu, jak implementovat polymorfismus v C++.

Program napsaný v OOP je vlastně množina mezi sebou navzájem komunikujících objektů.

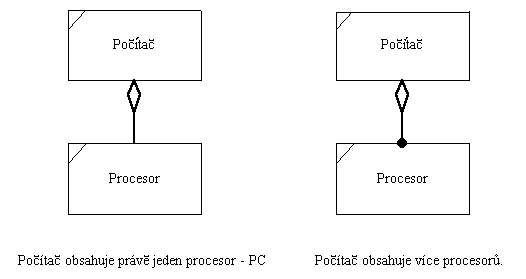
Třída objektů

Opět definice: "Třída objektů je abstrakce množiny podobných objektů". Je zřejmé, že některé objekty mají mnoho společných vlastností - jsou stejné třídy. Například objekt, na který se dívám při psaní tohoto článku, má mnoho společných vlastností s objektem, který sledujete vy při čtení tohoto článku - oba jsou třídy monitor. Nebo máte-li nějaký grafický OS, máte patrně na monitoru několik otevřených oken. Každé okno je objekt. Každý objekt vypadá jinak - každé okno zobrazuje něco jiného, přesto jsou si nějak podobné. Všechny objekty jsou stejné třídy. Každý objekt je nějaké třídy. Je-li objekt nějaké třídy, nazývá se instancí této třídy. Každá třída může mít libovolný počet instancí. Dále ve svých článcích budu místo pojmu objekt používat spíše pojem instance. Budu tím myslet objekt třídy, která je zřejmá, nebo kterou uvedu.

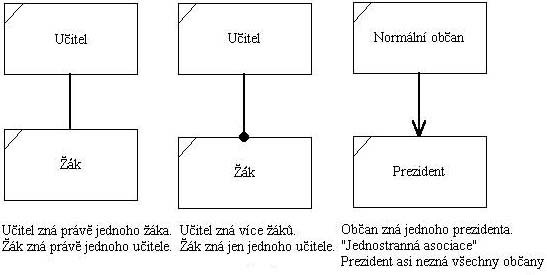
Vztahy mezi třídami

Pro popisy vztahů mezi třídami existuje mnoho druhů třídních diagramů. Mně se nejvíce zamlouvá syntaxe diagramů, které zavedl pan Rumbaugh, proto je zde budu používat. V těchto třídních diagramech jsou třídy znázorněny obdélníky, a vazby mezi třídami jsou nějak označené úsečky, které spojují obdélníky. Pro matematiky: Třídy jsou uzly grafu, vztahy jsou hranami grafu. Mezi třídami existují 3 druhy vztahů (vazeb):

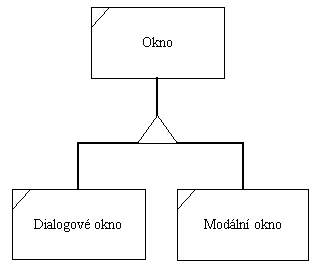
 Agregace - "obsažení" (Anglicky aggregation) Jestliže třída A agreguje třídu B, potom instance třídy A v sobě obsahují instance třídy B. Například v každé instanci třídy počítač existuje instance třídy procesor. Znázorněno v třídním diagramu:



 Asociace - "Link" (Anglicky association) Jestliže Třída A asociuje třídu B, potom instance třídy A nějakým způsobem musejí "vědět" o instancích třídy B. Zná-li objekt nějaký jiný objekt, znamená to, že jej může používat, zasílat mu zprávy a podobně. Například instance třídy učitel zná instance třídy žák. Není zde vhodná vazba agregace, protože učitel neobsahuje žáky uvnitř sebe. Asociace je obecnějsí vazba než agregace. Znázorněno v třídním diagramu:



 Dědičnost - "Specializace" (Anglicky inheritance) Dědičnost je doslovný překlad slova inheritance. Inheritance však někdy bývá přeložena jako specializace. Pojem specializace se mi líbí více, i když není přesným překladem. Zdá se mi, že lépe vystihuje daný vztah. Dále ve svých článcích budu používat zavedený pojem dědičnost. Třída A dědí ze třídy B, jestliže třída A je podmnožinou třídy B. Třída A se nazývá podtřídou třídy B, a třída B se nazývá nadtřídou třídy A. V tomto případě instance třídy A jsou zároveň instancemi třídy B, naopak to však nemusí platit. Například třída okna v grafickém OS je nadtřídou třídy dialogových oken, nadtřídou třídy modálních oken a podobně. Znázorněno v třídním diagramu:



Pro programátora je nutné, aby rozeznával mezi těmito vztahy. Rozdíl mezi asociaci a agregaci je zřejmý. Při agregaci objekty vždy obsahují jiné objekty v sobě. Při vztahu agregace (A agreguje B) by měly cizí objekty požadovat služby instancí agregované třídy (Třídy B) jen pomocí služeb instancí třídy A. Často se chybně zaměňuje mezi agregací, a dědičností. Je-li vztah mezi objekty "má", nebo "vlastní", potom se jedná agregaci. Například řetězec má (vlastní) znaky. Naopak je-li mezi objekty vztah "je", potom se jedná o dědičnost. Například dialogové okno je okno.

Upřímně děkuji všem, kteří tuto teorii dočetli až do konce. V příštích několika článcích naopak bude mnoho zdrojových textů, ve kterých budu používat jednotlivé pojmy, ale nebudu je již vysvětlovat. V příštím článku se dozvíte, jak vytvářet v C++ třídy a jejich instance.

[Vytváření tříd, instance třídy, zasílání zpráv v C++](http://www.builder.cz/rubriky/c/c--/vytvareni-trid-instance-tridy-zasilani-zprav-v-c--155667cz)

**28. prosince 2000, 00.00 | V minulém článku jsem vysvětlil základní pojmy z OOP. Nyní bych chtěl ukázat jak vše implementovat v C++.**

Vytváření tříd, instance třídy, zasílání zpráv

V minulém článku jsem vysvětlil základní pojmy z OOP. Nyní bych chtěl ukázat jak vše implementovat v C++. V tomto článku, i ve všech ostatních, již nebudu vysvětlovat význam jednotlivých pojmů, ale jen ukážu jak je implementovat v C++. Budete-li mít nějaké nejasnosti, doporučuji vám si přečíst můj předchozí článek.

Jak vytvořit (definovat) třídu?

K vytvoření třídy slouží klíčové slovo class. Definice vypadá následovně:

|  |  |
| --- | --- |
| |  | | --- | | class NázevTřídy [: předchůdci třídy]  { Položky třídy }; | |

Tato definice třídy není jediná. Mně se zdá nejlepší, proto ji budu dále používat. Třída je v C++ velmi podobná struktuře a také jsou k dispozici stejné možnosti pro vytváření třídy a struktury.

V tomto článku se dále nebudeme zabývat nepovinnou částí [: předchůdci třídy] . Týká se dědičnosti, které chci věnovat dva jiné články. Položky třídy jsou proměnné, nebo metody. Pojmu metoda se v C++ také někdy říká členská funkce (Member function). Metoda je vlastně funkce, která není globální, ale je volatelná jen v souvislosti s instancemi dané třídy. Položky mohou být soukromé ( private), chráněné (protected) a veřejné (public). Soukromé položky mohou být proměnné, ke kterým mají přístup pouze metody dané třídy, nebo metody, které mohou být volány jen jinými metodami dané třídy. Naopak veřejné položky může využívat kdokoliv. Veřejné položky tvoří rozhraní objektu dané třídy. Soukromé položky jsou zapouzdřeny, a přístup k nim se provádí jen přes veřejné rozhranní. Jako veřejné položky by neměli být proměnné, protože by docházelo k porušení zapouzdření. V praxi se ale na zapouzdření někdy moc nehledí. Chráněné položky jsou vlastně soukromé položky, které se někdy chovají jako veřejné. Jako veřejné se chovají k tak zvaným přátelským funkcím a vůči instancím tak zvaných přátelských tříd. Použití chráněných položek je vlastně také porušení zapouzdření. Budu se snažit chráněné položky nepoužívat, ale v praxi se v C++ někdy zapouzdření porušit musí.

Příklad třídy:

|  |  |
| --- | --- |
| |  | | --- | | class MojePrvniTrida {    private: /\* Následující položky jsou soukromé.\*/      int Cislo;      char Znak;    public: /\* Následující položky jsou veřejné.\*/      int VratMiTvojeCislo();      void NastavSiCislo(int noveCislo); }; | |

V této definici třídy jsem deklaroval hlavičky metod. Mohl jsem napsat těla metod hned v definici třídy. U větších tříd by takový zápis nebyl příliš přehledný. Také každá metoda, jejíž tělo je definováno v definici třídy je překládána, jako inline. Význam klíčového slova inline je stejný jako v C. Tedy funkce (nebo metoda) nebude volána, ale překladač vloží její tělo na místo jejího zavolání. Inline metoda je něco jako-by makro. Na rozdíl od makra, které je vkládáno na úrovni zdrojového textu, je inline funkce vkládána na úrovni přeloženého binárního kódu. Jen bych ještě měl upozornit, že slovo inline není pro překladač nijak závazné a pokud překladač "neuzná" za vhodné funkci překládat jako inline, tak ji přeloží normálně.

Nyní zbývá definovat těla metod:

|  |  |
| --- | --- |
| |  | | --- | | int MojePrvniTrida::VratMiTvojeCislo() /\* Operátor :: (čtyř-tečka) oznamuje překladači, že VratMiTvojeCislo() není globální funkce, ale členská metoda danné třídy.\*/ {     int a = 3; /\* Ukázka lokální proměnné\*/     return Cislo; } | |

V těle této metody je proměnná a lokální. Proměnná Cislo je proměnná, která je definována v třídě. Každá instance této třídy bude obsahovat svou proměnnou Cislo. V těle metod mohu také použít globální proměnné, kdyby nějaké byly.

|  |  |
| --- | --- |
| |  | | --- | | void MojePrvniTrida::NastavSiCislo(int noveCislo)  {     Cislo = noveCislo; } | |

V čistě objektově orientovaném programu neexistují globální funkce, ale jen metody. V C++ je trochu problematické napsat čistý objektově orientovaný program. Už jen proto, že funkce main je globální funkcí, není metodou žádné třídy.

Instance třídy

Procesu vytváření a rušení tříd věnuji celý příští článek. Zatím jen pro jednoduchou ilustraci naší jednoduché třídy vytvoříme instanci následujícím způsobem: MojePrvniTrida mojeprvniinstance;. Tedy stejně jako by jsme definovali "proměnnou typu MojePrvniTrida".

Objektu se zašle zpráva (Vyvolá se jeho metoda.) následujícím způsobem:

|  |  |
| --- | --- |
| |  | | --- | | Instance.NazevZpravy(argumenty - parametry); | |

Vyvolá se metoda, nebo se zašle zpráva? Jedná se jen o takové hraní si se slovíčky. V minulém článku jsem uvedl, že objekt na zprávu reaguje vyvoláním své metody. Podle teorie se objektům zasílají zprávy. Objekt na zprávu zareaguje voláním metody. Klidně tomu ale můžeme říkat zavolání metody. Nikomu snad nebude vadit, že přesně nedodržuji terminologii. Nyní příklad: (Napište před funkci main definici třídy a defici metod uvedené výše.)

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h>  void main(void) {    MojePrvniTrida a,b;    a.Cislo = 3; /\* CHYBA - proměnná Cislo je soukromá. Mají k ní přístup jen metody třídy MojePrvniTrida. Chcete-li program přeložit, odstraňte tento řádek. \*/     a.NastavSiCislo(5);    b.NastavSiCislo(4);    cout << a.VratMiTvojeCislo() << endl;     cout << b.VratMiTvojeCislo() << endl;  } | |

Ve svém prvním článku stejně jako zde používám zápis cout << něco, aniž bych vysvětlil o co jde. Vstupně výstupním operacím se chci věnovat v jednom svém článku podrobně. Zatím prosím berte jako fakt, že "to funguje" a že to něco vypíše. Nechcete-li psát něco, co neznáte, můžete zatím používat funkci printf z C knihovny stdio.h.

Implicitní parametr "this"

Každá metoda v C++ je volána s tak zvaným implicitním parametrem this. Tento parametr je ukazatelem na instanci třídy, jejíž položkou metoda je. Tento ukazatel ukazuje právě na instanci, pro kterou je volán. Tedy v některé metodě naší třídy je proměnná Cislo identická s proměnnou this->Cislo. Programátor může využít this při konfliktu jmen. Dopište prosím do definice třídy MojePrvniTrida mezi veřejné položky následující metodu: void pokus();. Dále definujte globální proměnnou int Cislo. Nyní ukážu jak přistupovat k jednotlivým proměnným:

|  |  |
| --- | --- |
| |  | | --- | | void MojePrvniTrida::pokus() { /\* Nyní je globální proměnná Cislo překrytá proměnnou Cislo deklarovanou v třídě. \*/    int Cislo;     /\* Nyní jsem lokální proměnnou překryl i proměnnou v třídě.\*/    Cislo++;   /\* Zvyšuji lokální proměnnou.\*/    this->Cislo++;   /\* Zvyšuji instanční proměnnou deklarovanou v třídě.\*/    ::Cislo++;   /\* Zvyšuji globální proměnnou. \*/  } | |

V příštím článku se podrobně podívám jak vytvářet a rušit instance. Povím něco o konstruktorech, destruktorech, operátorech new, delete, atd.

## [Vytváření instancí - konstruktory, destruktory](http://www.builder.cz/rubriky/c/c--/vytvareni-instanci-konstruktory-destruktory-155678cz)

**18. ledna 2001, 00.00 | V tomto článku si povíme něco o vytváření instancí, konstruktorech, destruktorech, operátorech new, delete a o věcech s těmito pojmy souvisejících.**

Konstruktor

Každá instance je vytvořena pomocí speciální metody - konstruktoru. Konstruktor se musí jmenovat stejně jako třída, jejíž instance bude vytvářet, nesmí mít návratovou hodnotu a měl by být veřejný. Každá třída může mít k dispozici více konstruktorů s různými parametry. Jedná se o přetěžování konstruktorů - o přetěžování jsem se zmiňoval v 1. článku "Od C k C++". Nemá-li třída žádný konstruktor (Například třída MojePrvniTrida v mém předchozím článku "Vytváření tříd, instance třídy, zasílání zpráv".) vytvoří překladač tak zvaný implicitní konstruktor bez parametrů. V konstruktoru by se měla provést inicializace dané instance. Nastavit všechny její atributy, vytvořit instance tříd, které obsahuje atd... Uvedu příklad jednoduché třídy Zlomek i s konstruktory.

|  |  |
| --- | --- |
| |  | | --- | | class Zlomek {     private:        int Jmenovatel, Citatel;     public:        Zlomek();  /\* Bezparametrický konstruktor \*/        Zlomek(int cislo);        Zlomek(int citatel, int jmenovatel);        /\* Ostatní metody třídy zlomek: \*/        int dejCitatel() const;        int dejJmenovatel() const;        void nastavCitatel(int citatel);        void nastavJmenovatel(int jmenovatel);        float hodnota() const; };   Zlomek::Zlomek() {     Jmenovatel = 1;     Citatel = 0; }  Zlomek::Zlomek(int cislo) {     Citatel = cislo;     Jmenovatel = 1; }  Zlomek::Zlomek(int citatel, int jmenovatel) {     Citatel = citatel;     if (jmenovatel != 0)        Jmenovatel = jmenovatel;     else        Jmenovatel = 1; }  int Zlomek::dejCitatel() const { return Citatel; }  int Zlomek::dejJmenovatel() const { return Jmenovatel; }  void Zlomek::nastavCitatel(int citatel)  { Citatel = citatel; }  void Zlomek::nastavJmenovatel(int jmenovatel)  {     if (jmenovatel != 0)        Jmenovatel = jmenovatel; }  float Zlomek::hodnota() const { return ((float)Citatel)/Jmenovatel; } | |

Nejprve vysvětlím některé možné nejasnosti ke zdrojovému textu, i když se přímo netýkají zaměření článku:

1. Klíčové slovo const za některými metodami. - Takto označená metoda nemění nijak vnitřní stav instance. V C++ existují "konstantní instance", stejně jako v C konstantní proměnné. V C++ může být konstantní jak proměnná "primitivního" datového typu, tak i instance. "Konstantní instanci" nelze nijak měnit vnitřní stav. Lze pro takovou instanci vyvolat pouze metody deklarované klíčovým slovem const .
2. Možná někoho napadne otázka, jestli není lepší napsat atributy jako veřejné a ušetřit psaní metod nastav... a dej... - Veřejné atributy porušují princip zapouzdření. Kdybych se nyní rozhodl nějak změnit reprezentaci čitatele a jmenovatele, musel bych změnit jen metody třídy Zlomek. Objekt se bere jako černá skříňka, která by se změnila jen uvnitř, vůči okolí by se jako celek nezměnila. S veřejnými atributy by to tak lehké nebylo. Musel by se změnit celý program všude tam, kde by byly používány nějaké instance třídy Zlomek. I když uznávám, že v takto jednoduché třídě jako je Zlomek bych velmi těžce odolával pokušení dát atributy jako soukromé. Podle principů OOP je ale určitě správné přistupovat k atributům přes metody. Jestliže máte obavu ze zpomalení programu voláním, můžete metody deklarovat jako inline.

Konstruktory v této třídě jsem napsal tak, aby bylo jasné, že slouží k inicializaci instance. Jsou napsány sice jasně a přehledně, ale ne nejlépe. Všechny konstruktory, které jsem zde napsal, lze napsat efektivněji. Je třeba vzít na vědomí, že před spuštěním samotného těla konstruktoru dojde k inicializaci (volání konstruktorů) všech členských dat třídy. Například v konstruktoru Zlomek(int citatel, int jmenovatel) dojde nejprve k inicializaci položek Citatel, Jmenovatel a poté k provedení těla konstruktoru, kde tyto dvě položky znovu změním. V tomto konkrétním případě to snad ani nevadí, protože int žádné konstruktory nemá. Kdyby se ale jednalo o instance nějaké třídy, zjistili by jste, že je pro položky nejprve zavolán konstruktor bez parametrů a poté přiřazena nějaká hodnota operátorem =. Obojí lze efektivněji provést v jednom kroku. Za název konstruktoru, před jeho tělo umístíme za dvojtečkou seznam položek s inicializačními hodnotami v závorce. Zmiňovaný konstruktor lze přepsat takto:

|  |  |
| --- | --- |
| |  | | --- | | Zlomek::Zlomek(int citatel, int jmenovatel) :Jmenovatel(jmenovatel),Citatel(citatel) {   /\* Zde provedu nějakou činnost, nebo nechám tělo prázdné. \*/ } | |

Při vytváření instance třídy již mohu použít její metody, i když vlastně instance je vytvořená až po dokončení konstruktoru.  
Někdy by se mohlo zdát, že je zbytečné vytvářet konstruktor bez parametrů. Bezparametrický konstruktor je ale nutný například při vytváření polí objektů a podobně. Vše ukážu dále.  
Ke konstruktorům se ještě vrátím.

Destruktor

Destruktor, jak již asi sám jeho název napovídá, slouží k likvidaci objektů. Destruktor (stejně jako konstruktor) je metoda, která je zavolána na instanci v momentě, kdy je instance likvidována. Destruktor se jmenuje stejně jako třída, jen před názvem třídy je znak ~ . Destruktor nesmí mít návratovou hodnotu a žádné parametry. V destruktoru by se měly uvolnit všechny zdroje, se kterými instance pracovala. Například uzavřít datové soubory, uvolnit případnou alokovanou paměť a podobně. Nemá-li třída destruktor, překladač vytvoří implicitní destruktor. Ve třídě Zlomek naprosto stačí implicitní destruktor. Já ale jako příklad uvedu. Připište prosím do třídy Zlomek mezi veřejné prvky deklaraci: ~Zlomek(); a dále připište metodu:

|  |  |
| --- | --- |
| |  | | --- | | Zlomek::~Zlomek() {     cout << "Loučí se s vámi instance třídy zlomek. " << endl << dejCitatel() << '/' << dejJmenovatel() << endl;     /\* Jinak tu není co dělat \*/ } | |

Protože požívám cout nezapomeňte na začátek zdrojového textu napsat: #include<iostream.h>

Jak vytvářet instance

Na instance lze v C++ někdy pohlížet jako na proměnné. Instance stejně jako proměnné mohou být "statické", nebo "dynamické". Statická instance má stejně jako proměnná platnost (viditelnost) v aktuálním bloku, kde byla vytvořena. V momentě, kdy její platnost končí, je automaticky zlikvidována zavoláním svého destruktoru. Příklad: (Připište jej do souboru, kde je definována třída Zlomek

|  |  |
| --- | --- |
| |  | | --- | | Zlomek Globalni(2,1);      /\* Globalni je vytvořeno konstruktorem Zlomek(int jmenovatel, int citatel); Tento konstruktor se provede dříve, než funkce main.\*/ void main(void) {      Zlomek a,b(1),c(1,2);     /\* a je vytvořeno bezparametrickým konstruktorem        b je vytvořeno konstruktorem Zlomek(int cislo);       b je vytvořeno konstruktorem Zlomek(int jmenovatel, int citatel);\*/      Zlomek pole[3]; /\* Pole objektů. Každý prvek v poli je vytvořen bezparametrickým konstruktorem. \*/      pole[2].nastavCitatel(10);      pole[2].nastavJmenovatel(2);      cout << a.hodnota() << " " << b.hodnota() << " " << c.hodnota() << " " << Globalni.hodnota() << endl;      for (int p = 0; p<3; p++)      {         cout << "Prvek " << p << " je " << pole[p].hodnota() << endl;      }    /\* Nyní končí viditelnost instancí a,b,c a všech prvků pole . Automaticky budou vyvolány jejich destruktory. \*/ }  /\* Nyní bude zavolán destruktor instance Globalni. \*/ | |

V programu jsou nejprve zavolány konstruktory globálních instancí, poté je až spuštěna funkce main , ve které jsou volány konstruktory a destruktory na lokální instance. Po skončení main jsou likvidovány globální instance.

Další možností je dynamické alokace paměti na haldě, a vytvoření instance. K takové instanci přistupujeme přes ukazatel, který na ní "ukazuje". Ukazatel deklarujeme jako v jazyce C. Pro alokaci paměti existuje nový operátor new . V takovém případě instance "přežije" konec bloku, a je potřeba ji zrušit operátorem delete . Stejně jako je tomu v C s dynamickými prom. Obdoba new a delete je v C malloc a free . Není správné ale používat funkce malloc a free , protože jen alokují paměť. narozdíl od toho new navíc zavolá konstruktor a delete zavolá destruktor. Jako příklad uvedu novou funkci main :

|  |  |
| --- | --- |
| |  | | --- | | void main(void) {      Zlomek \*a, \*b, \*c, \*pole;/\* Ukazatelé na instance třídy Zlomek.\*/      /\* Žádné konstruktory se nevolají.\*/      a = new Zlomek; /\* Instance je vytvořena bezparametrickým konstruktorem.\*/      b = new Zlomek(1,2); /\* Je zavolán konstruktor Zlomek(int jmenovatel, int citatel); \*/      c = b; /\* c ukazuje na stejnou instanci jako b! \*/      pole = new Zlomek[3]; /\* Takto se dynamicky vytváří pole instancí.\*/      cout << a->hodnota() << " " << b->hodnota() << c->hodnota() << endl;      for (int p = 0; p<3; p++)      {         cout << "Prvek " << p << " je " << pole[p].hodnota() << endl;      }     /\* Nyní instance likviduji. Neprovede se to automaticky.\*/     delete a;     delete b;     delete[] pole; /\* Takto se uvolňuje pole. \*/     /\* NENÍ správné napsat delete c; Tato instance již byla uvolněná příkazem delete b; \*/ } | |

V těchto příkladech jsem vždy přepokládal, že paměť pro instance je vždy k dispozici. To je samozřejmě velmi naivní. Operátor new v případě, že neuspěje s alokací paměti, vrací NULL , nebo vypustí tak zvanou vyjímku. Vyjímkám věnuji jeden z mých dalších článků.   
Článek je již dost dlouhý a já jsem se bohužel ještě nezmínil o jednom specielním a velmi důležitém konstruktoru, který se nazývá kopírovací konstruktor. Ve třídě Zlomek nebyl potřeba. Absence kopírovacího konstruktoru je velmi častou chybou a velmi častou příčinou nevysvětlitelných pádů programů. Někdy trvá dost dlouho, než si programátor uvědomí, že příčina nepochopitelných pádů programu je nepřítomnost kopírovacího konstruktoru. Proto vám doporučuji si přečíst můj další článek, který o kopírovacích konstruktorech bude.

## [Kopírovací konstruktor v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-konstruktor-v-c--155685cz)

**1. února 2001, 00.00 | Kopírovací konstruktor v C++ aneb Proč to pořád padá. V tomto článku popíši velmi důležitý konstruktor, o kterém jsem se v minulém díle nezmínil.**

V tomto článku popíšu velmi důležitý konstruktor, kopírovací konstruktor, o kterém jsem se v minulém článku nezmínil. Absence kopírovacího konstruktoru je častou chybou a je velmi častou příčinou "pádů" programů, nebo jejich nevysvětlitelného chování. Abych nejlépe ukázal, jak je kopírovací konstruktor důležitý, vymyslel jsem následující příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h>  class Vektor {   private:      int Rozmer;      int \*Slozky;   public:      Vektor();      Vektor(int rozmer);      ~Vektor();      void pricti(const Vektor druhyVektor);      int dejRozmer() const;      int dejPrvek(int index) const;      void nastavPrvek(int index, int hodnota); };  Vektor::Vektor():Rozmer(0),Slozky(NULL) {}  Vektor::Vektor(int rozmer):Rozmer(rozmer),Slozky(new int[rozmer]) {}  Vektor::~Vektor() {    delete[] Slozky; }  inline int Vektor::dejRozmer() const {    return Rozmer; }  inline int Vektor::dejPrvek(int index) const {    return ((Rozmer > index) && (index >=0))? Slozky[index] : 0; }  void Vektor::pricti(const Vektor druhyVektor) {    if (this->Rozmer != druhyVektor.dejRozmer())    {       return ;    }    for(register int p = 0; p<this->Rozmer; p++)    {       this->Slozky[p] += druhyVektor.dejPrvek(p);    } }  void Vektor::nastavPrvek(int index, int hodnota) {    if ((Rozmer > index) && (index >=0))    {       Slozky[index] = hodnota;    } }  int main(void) {    Vektor v1(4),v2(4);    int p;    for(p = 0; p<4; p++)    {       v1.nastavPrvek(p,p);       v2.nastavPrvek(p,2\*p+10);    }    v1.pricti(v2);    v2.nastavPrvek(2,0); /\* BUM!!!!!\*/    cout << "Vysledek: ";    for(p=0; p<4; p++)       cout << v1.dejPrvek(p) << '\t';    cout << endl;    return 0; } | |

O tom, jak je navržena třída vektor, by se asi dalo diskutovat. Jedná se jen o ilustrační příklad a k tomu je takto navržená třída dobrá. Na první pohled by se mohlo zdát, že je vše v pořádku. Přesto spustíte-li program, zjistíte, že program spadne vyvoláním metody instance v2 ( Tento řádek jsem okomentoval "BUM"). Abych byl přesný, program havaruje na 100% jen pod OS Linux, který má přísnou kontrolu a ochranu přístupu do paměti. Tento program pod Windows xx spadne jen někdy, což jsou velmi nepříjemné chyby, které se velmi obtížně ladí. Pod Windows by se dalo spíše očekávat, že se v programu nepochopitelně mění některé proměnné a podobně. V každém případě se ale jedná o chybu programátora, ne OS, nebo C++. V příkladu, který jsem uvedl, je chyba. V momentě, kdy je vykonáván řádek: v2.nastavPrvek(2,0); již není alokováno pole Slozky instance v2. Ukazatel složky v instanci v2 ukazuje na nealokovanou paměť ("nikam").  
Nyní nejprve vysvětlím, jak je to možné a poté, jak tomu zabránit.

Volání metody, nebo funkce

Problém mého příkladu není na onom kritickém řádku, ale při volání metody: v1.pricti(v2);. Dovolil bych si připomenout, jak probíhá volání metod, nebo funkcí. Program nejprve zkopíruje na svůj zásobník všechny parametry funkce. Jedná-li se o metodu, je navíc kopírován i implicitní parametr this . Také se na zásobník uloží návratová adresa, ale to nyní není podstatné. Po skončení funkce, nebo metody se uloží do nějakého registru (podle typu) návratová hodnota a předá se řízení na návratovou adresu volajícímu. Volající přečte návratovou hodnotu, "vyčistí" zásobník a pokračuje v činnosti.  
Nyní se dobře podívejme na metodu pricti a představme si, co se bude dít při jejím zavolání. Nejprve se na zásobník uloží (zkopíruje) instance třídy Vektor, která je při volání dána jako parametr. Instance je vlastně jen "obyčejný" kus paměti v našem případě délky dvou int, což je v 32-bitových OS 2\*32 bitů (8 bytů). Tím máme vytvořenu kopii parametru. Tato kopie parametru se v těle metody jmenuje druhyVektor. Ukazatel Slozky v instanci v2 je zkopírován na ukazatel složky v instanci druhyVektor. Není nijak zkopírován obsah paměti (V našem případě pole.), na kterou ukazatel ukazoval. Oba ukazatele (v1.Slozky i druhyVektor.Slozky) tedy ukazují na stejné pole. Po ukončení metody a návratu do volajícího (main) je zničena (odebraná ze zásobníku) instance druhyVektor. Likvidace instancí (viz můj minulý článek) provádí destruktor (v našem případě Vektor::~Vektor() ) a ten dealokuje pole, na které ukazoval ukazatel druhyVektor.Slozky. Na stejné pole ale ukazoval ukazatel v1.Slozky. Proto ukazatel v1.Slozky neukazuje na alokovanou paměť. Myslím si, že pochopit tuto "drobnost" je dost významné, takže není-li vám něco jasné, doporučuji si tento program podrobně krokovat v debuggeru a hlídat při tom hodnoty jednotlivých ukazatelů.

Kopírování instancí  
  
Druhy kopií

Existují dva druhy kopií. Tak zvaná "plytká" kopie a "hluboká" kopie. Jako kopii instance mám na mysli situaci, kdy vznikne nová instance podobná, nebo stejná originálu. Vznikne nový objekt s novou identitou. Tedy například:   
Vektor \*a,\*b;  
a = new Vektor(3);  
b = a; /\* NENÍ kopie !!! \*/   
Zde není vytvořená žádná kopie nějaké instance.  
"Plytká" kopie je druh kopie, kterou jsem popsal v odstavci "Volání metody, nebo funkce". Jedná se vlastně jen o takovou povrchní kopii, která nekopíruje "do hloubky". "Plytké" kopírování tedy zkopíruje hodnoty jednotlivých ukazatelů a nestará se o paměť, na kterou ukazovali. O proti tomu "hluboká" kopie zkopíruje vše, i blok paměti, na kterou ukazují jednotlivé ukazatele.

Kopírovací konstruktory v C++

Kopírovací konstruktor, jak asi nikoho nepřekvapí, se stará o vytváření kopií instancí. Není-li programátorem definován kopírovací konstruktor, je použit tak zvaný implicitní kopírovací konstruktor. Implicitní kopírovací konstruktor vytváří vždy plytkou kopii, což někdy je dobré, jindy není. Chceme-li, aby probíhalo kopírování více do hloubky, nebo aby se při kopírování stala nějaká další činnost, musíme kopírovací konstruktor napsat. Kopírovací konstruktor je konstruktor, který má jako svůj parametr konstantní referenci na instanci třídy, ze které má být vytvářena kopie (tedy své). Kopírovací konstruktor se mimo jiné použije při předávání parametrů hodnotou. Nyní dopíšu kopírovací konstruktor třídy Vektor. Mezi veřejné metody třídy dopište jeho deklaraci: Vektor(const Vektor& druhy); a dopište jeho tělo:

|  |  |
| --- | --- |
| |  | | --- | | Vektor::Vektor(const Vektor& druhy) :Rozmer(druhy.dejRozmer()),Slozky(new int[this->Rozmer]) {    for(register int p = 0; p<this->Rozmer; p++)    {       Slozky[p] = druhy.dejPrvek(p);    } } | |

Kopírovací konstruktor mohu samozřejmě i sám v programu volat. Například před zavoláním metody pricti mohu napsat Vektor kopie(v1); resp. Vektor \*ukazatel = new Vektor(v1); pro vytvoření kopie instance. Opět všem doporučuji si tento příklad dobře prohlédnout v debuggeru.   
Vždy si dobře rozmyslete, co se má a co se nemá kopírovat v kopírovacím konstruktoru. Je jasné, že jestliže instance pomocí ukazatelů sdílejí nějakou paměť, nemá smysl v kopírovacím konstruktoru dělat kopii této paměti. Naopak musíte dát pozor, aby nenastal případ jako v mém ukázkovém programu bez kopírovacího konstruktoru. Vše by se dalo shrnout asi do následující rady (spíše nepsaného pravidla): Používá-li instance nějakou "vnější" paměť, měla by VŽDY mít konstruktor, kde ji alokuje, destruktor, kde ji uvolní, a kopírovací konstruktor, kde vytvoří její kopii. Nepište kopírovací konstruktor v případě, že by prováděl stejnou činnost jako implicitní. Implicitní konstruktor vytvořený překladačem bude určitě rychlejší.  
Aby jsem nevytvořil mylný dojem, že je vše v pořádku, použijte třídu Vektor i s kopírovacím konstruktorem v nějakém programu, kde deklarujete: Vektor v1(3),v2;. Poté někde napište v2 = v1;. Zjistíte, že operátor přiřazení vytváří plytkou kopii. Toto lze potlačit přetížením tohoto operátoru. Proto by třída používající "vnější" paměť měla mít také přetížený operátor = . Přetěžování operátorů věnuji jeden celý článek, ale bohužel ne hned. Před tím se chci podívat na dědičnost a věci s ní spjaté, což se určitě nevejde do jednoho článku (nejspíš ani ne do dvou, nebo do tří).

Na závěr bych chtěl jen dodat trochu vtipnou radu. Až dostanete pocit, že váš program je správný, ale překladač je špatný (Co si budeme namlouvat, k takovým smělým hypotézám dojde každý programátor mnohokrát za život.), raději zkontrolujte, jestli každá třída má správně definované kopírovací konstruktory a přetížené přiřazovací operátory, než svůj pocit vyslovíte někomu nahlas. :-)

## [Jednoduchá dědičnost v C++](http://www.builder.cz/rubriky/c/c--/jednoducha-dedicnost-v-c--155692cz)

**15. února 2001, 00.00 | V tomto článku objasníme jednoduchou dědičnost v C++. Jak se chovaji konstruktory a destruktory a tak podobně.**

Jednoduchá dědičnost v C++

V tomto článku objasním jednoduchou dědičnost v C++. Obecně pojem dědičnost jsem vysvětlil ve svém článku "Základní pojmy OOP". Jedná se o jeden z možných vztahů mezi třídami. V tomto článku se budu věnovat pouze implementaci v C++. Pro ty z Vás, kterým dědičnost není zcela jasná doporučuji si nejprve přečíst článek "Základní pojmy OOP".

Dědičnost slouží ke rozšiřování vlastností tříd. V tomto článku se budu věnovat jen jednoduché dědičnosti, tedy dědičnosti, kdy nějaká třída dědí (Je speciálním případem.) jen jedné třídy. Jestliže třída B je speciálním případem třídy A, potom říkáme, že třída B dědí z třídy A. Třída B je vlastně třída A, ale má také nějaké vlastnosti navíc. Abychom metody třídy B, které jsou stejné jako metody třídy A, nemuseli psát podruhé, použijeme mechanismus dědění. Mějme například třídu vozidlo, která má soukromý atribut počet kol a veřejné metody které počet kol nastaví a vrátí. Z ní bude dědit třída nákladní vozidlo, která má navíc soukromý atribut nosnost a veřejné metody, které nosnost nastaví a vrátí. Třída nákladních vozidel má tedy také atribut počet kol a metody pro nastavení a vrácení počtu kol. Nemusí se psát znovu.

Možnosti jak dědit

Existují 3 způsoby jak v C++ dědit. Podle způsobu dědění se rozhoduje nakolik budou v potomkovi viditelné (přístupné) metody, nebo atributy předka. Způsoby jsou:

   public

   protected

   private

Jak vidíte, jsou použity stejná klíčová slova, jak pro způsob dědění, tak pro označení přístupových práv k atributům, nebo metodám. To asi proto, aby byl začínající programátor dokonale zmaten. Pro pochopení těchto způsobů uvedu následující tabulku. V horním řádku jsou přístupová práva položky třídy v předkovi. V levém sloupci jsou způsoby jakým byl potomek zděděn.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Položka v předkovi je private. | Položka v předkovi je protected. | Položka v předkovi je public. |
| Dědění je private. | Položka v potomkovi není přístupná. | Položka v potomkovi je private. | Položka v potomkovi je private. |
| Dědění je protected. | Položka v potomkovi není přístupná. | Položka v potomkovi je protected. | Položka v potomkovi je protected. |
| Dědění je public. | Položka v potomkovi není přístupná. | Položka v potomkovi je protected. | Položka v potomkovi je public. |

Položka v potomkovi nemusí být přístupná, ale vždy v potomkovi je. Abych přiznal pravdu, nevím v jakém případě by bylo vhodné použít jinou dědičnost, než veřejnou (public). Tím samozřejmě nechci říci, že dědění private či protected jsou k ničemu, nebo špatné. Já ale budu dále používat jen veřejnou dědičnost. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h>  class Vozidlo {  private:     int PocetKol;  public:     void nastavKola(int a){ PocetKol = a; }     int dejPocetKol(){ return PocetKol;} };  /\* Nákladní vozidlo dědí z vozidlo způsobem public. \*/  class NakladniVozidlo : public Vozidlo {  private:     int Nosnost;  public:     void nastavNosnost(int a){ Nosnost = a; }     int dejNosnost(){ return Nosnost;} }; | |

Instance třídy nákladní vozidlo je zároveň také instancí třídy vozidlo. Opačně to platit nemusí. Dostali jsme se k jedné ze základních vlastností dědičnosti: "na místě, kde je očekáván předek, může být dosazen potomek". Tedy ukazatel, nebo reference na instanci třídy předka může klidně ve skutečnosti ukazovat na instanci potomka. Pro příklad doplním předchozí zdrojový text o funkci main, kde ukážu mnoho možností:

|  |  |
| --- | --- |
| |  | | --- | | int main(void) {    Vozidlo \*v1 = new Vozidlo;    Vozidlo \*v2 = new NakladniVozidlo; /\* Naprosto korektní! \*/    NakladniVozidlo \*n1 = new NakladniVozidlo;    v1->nastavKola(4);    v2->nastavKola(6);    n1->nastavKola(8); /\* Instance n1 je vlastně i instancí třídy Vozidlo \*/    cout << v1->dejPocetKol() << '\t' << v2->dejPocetKol() << '\t' << n1->dejPocetKol() << endl;    n1->nastavNosnost(1000);    cout << n1->dejNosnost() << endl;    delete v1;    delete v2;    delete n1;    return 0; } | |

Můžete si lehce vyzkoušet, že v těle nějaké metody třídy nákladní vozidlo nelze přímo použít atribut int PocetKol, ale lze k němu přistupovat jen pomocí veřejných metod třídy vozidlo. To svědčí o tom, že třída nákladní vozidlo položku int PocetKol má, ale není k ní přístup. Položka v předkovi je private, dědění je public - viz tabulka. Bylo by chybou napsat v třídě nákladní vozidlo atribut int PocetKol, protože třída nákladní vozidlo by měla dvě proměnné int PocetKol. To by ani tak nevadilo jako fakt, že metody třídy vozidlo by pracovali s jiným atributem int PocetKol než metody třídy nákladní vozidlo. Tato chyba ale není syntaktickou chybou, takže na ni překladač v době kompilace neupozorní!

Samozřejmě, že nyní může dále ze třídy nákladní vozidlo dědit další třída, ze které může zase dědit jiná třída atd... Stejně tak může třída vozidlo mít více následovníků než jen nákladní vozidlo - třeba osobní vozidlo atd... Pořád se jedná o jednoduchou dědičnost.

Chování konstruktorů při dědění

Konstruktory (stejně jako destruktory) jsou metody, které se nedědí, ale lze implicitně vyvolat konstruktor předka v konstruktoru potomka. Implicitně při vytváření instance nějaké třídy se nejprve vyvolá bezparametrický konstruktor (Pokud není, tak implicitní konstruktor vytvořený překladačem.) nejvyšší nadtřídy, poté jejího potomka, atd, až se vyvolá požadovaný konstruktor pro instanci. Pro lepší pochopení uvedu příklad. Do výše uvedeného příkladu vepište mezi veřejné metody třídy vozidla konstruktor:

|  |  |
| --- | --- |
| |  | | --- | | Vozidlo():PocetKol(0) {    cout << "Tvořím vozidlo" << endl; } | |

A mezi veřejné metody třídy nákladní vozidlo vepište konstruktor:

|  |  |
| --- | --- |
| |  | | --- | | NakladniVozidlo():Nosnost(0) {     cout << "Tvořím nákladní vozidlo" << endl; } | |

Nyní po spuštění programu bude dobře vidět, jaké je pořadí volání jednotlivých konstruktorů. Mnohdy ale není vhodné volat bezparametrické konstruktory a poté nastavovat hodnoty. Implicitní volání konstruktorů předků lze potlačit a nahradit jej voláním konstruktorů s nějakými parametry. Provádí se to stejným způsobem jako se volají konstruktory atributů třídy (Viz můj článek "Vytváření instancí, konstruktory destruktory".). Pro příklad doplňte mezi veřejné metody třídy vozidlo konstruktor:

|  |  |
| --- | --- |
| |  | | --- | | Vozidlo(int kola):PocetKol(kola) {     cout << "Tvořím vozidlo s parametrem" << kola <<endl; } | |

Třída Vozidlo má nyní dva konstruktory. Konstruktor třídy nákladní vozidlo upravte následovně:

|  |  |
| --- | --- |
| |  | | --- | | NakladniVozidlo():Vozidlo(6),Nosnost(0) {     cout << "Tvořím nákladní vozidlo" << endl; } | |

Opět spusťte program a zjistíte, že při vytváření instance n1 je zavolán konstruktor Vozidlo(6) a potom konstruktor NakladniVozidlo().

Chování destruktorů při dědění

Destruktory se stejně jako konstruktory nedědí, ale stejně jako u konstruktorů jsou implicitně volány destruktory předků. Na rozdíl od konstruktorů se volají v opačném pořadí. V mém příkladě se při likvidaci instance n1 nejprve vyvolá destruktor ~NakladniVozidlo() a poté destruktor ~Vozidlo(). Můžete se o tom přesvědčit, jestliže definujete v mém příkladě destruktory třídám, které stejně jako konstruktory vypíšou na stdout informaci o tom, že proběhly. Bude možná jen překvapeni, že při likvidaci instance v2 nebude zavolán destruktor ~NakladniVozidlo(), ale jen destruktor ~Vozidlo(). Proč tomu tak je vysvětlím ve svém příštím článku věnovaném virtuálním metodám.

Není-li Vám zcela jasné v jakém pořadí se konstruktory a destruktory volají doporučuji Vám si opsat můj příklad i s doplňky a důkladně jej odkrokovat v debuggeru.

Dodatek

Musím ještě upozornit, že další metoda, která se nedědí, je přetížený operátor = . Přetíženým operátorům jednou věnuji celý článek, kde vše vysvětlím. Toto upozornění jsem napsal jen pro ty, kteří již přetěžování znají od jinud a zkoušeli by přetěžování i dědičnost ve svých programech.

V mém příštím článku se budu věnovat polymorfismu, tedy metodám volajících se tak zvanou pozdní vazbou (jsou deklarovány s klíčovým slovem virtual). Potom v dalším článku (spíše v dalších dvou článcích) se vrátím k dědičnosti, tentokráte vícenásobné.

## [Časná versus pozdní vazba - úvod do polymorfismu v C++](http://www.builder.cz/rubriky/c/c--/casna-versus-pozdni-vazba-uvod-do-polymorfismu-v-c--155697cz)

**19. února 2001, 00.00 | Jaký je rozdíl mezi časnou a pozdní vazbou? Dále si ukážeme jak a proč používat klíčové slovo virtual.**

Všechny metody, které jsem ve svých článcích doposud používal byly volány tak zvanou časnou vazbou. Tedy překladač v době překladu přesně věděl jaký podprogram (metoda) bude kdy vyvolán. Například v mém minulém článku "Jednoduchá dědičnost v C++" jsem v jednom ukázkovém programu uvedl řádek v1->nastavKola(4);. Pro překladač je jednoznačné, který podprogram se má vyvolat. Metoda Vozidlo::nastavKola(int) bude vyvolána i když v1 bude instance třídy Vozidlo, nebo instance třídy Nákladní vozidlo. V takovém případě říkáme, že metoda je volána časnou vazbou. Pojem "časná vazba" asi proto, že překladač zná adresu podprogramu na který má předat řízení včas (V době překladu.) Může ale nastat případ (nastává velmi často), kdy budeme potřebovat, aby metoda pro podtřídu vykonávala jinou činnost, než stejná metoda u nadtřídy. Tedy vlastně budeme potřebovat přepsat tělo metody v podtřídě. Jednoduše metodu přepsat sice jde, ale není to dobré. Uvedu příklad: (Metody void casna() jsou volány časnou vazbou.)

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h>  class Nadtrida {  public:    void casna() { cout << "Metoda tridy Nadtrida volana casnou vazbou" << endl; } };  class Podtrida : public Nadtrida {  public:    void casna() { cout << "Metoda tridy Podtrida volana casnou vazbou" << endl; } };  int main(void) {     Nadtrida \*n = new Nadtrida;     Podtrida \*p = new Podtrida;     n->casna();     p->casna(); /\* Zatím vypadá vše OK. Ale zkusme dál. \*/     Nadtrida \*problem = new Podtrida; /\* Na místo předka dám potomka - to je OK \*/     problem->casna(); /\* A tady je problém! \*/     return 0; } | |

Po spuštění zjistíte, že řádek problem->casna(); vyvolá metodu void Nadtrida::casna() což jsme nechtěli, protože ukazatel problém je sice ukazatel typu Nadtřída, ale ve skutečnosti ukazuje na instanci třídy Podtřída. Na místo, kde byl očekáván předek byl dosazen potomek - v OOP běžná a často používaná konstrukce.

Podívejme se, jak překladač postupoval při překladu řádku problem->casna(); . Nejprve zjistil typ ukazatele problém. Ukazatel je deklarován jako Nadtrida \*problem. Zjistil si, jestli třída Nadtřída (Případně některý její předek - v našem případě žádný není.) má definovanou metodu void casna();. Protože má, rozhodl již v době překladu o tom, že se bude volat void Nadtrida::casna();. Kdyby neměla, nahlásil by chybu. Fakt, že ukazatel problém může ukazovat na potomka nyní překladač nezajímalo. Proto abychom na řádku problem->casna();volali metodu void casna() podle toho, na jakou instanci ukazatel problém ukazuje, musíme použít pozdní vazbu.

Klíčové slovo virtual

Klíčové slovo virtual před deklarací metody překladači přikazuje použít tak zvanou pozdní vazbu při volání dané metody. Zkuste do mého příkladu mezi veřejné metody třídy Nadtřída vepsat řádek virtual void pozdni(); a mezi veřejné metody třídy Podtřída vepište řádek virtual void pozdni();. Potom do zdrojového textu vepište těla těchto metod:

|  |  |
| --- | --- |
| |  | | --- | | void Nadtrida::pozdni() /\* Tady se už virtual nepíše, jen v deklaraci. \*/ {    cout << "Metoda tridy Nadtrida volana pozdni vazbou" << endl; } void Podtrida::pozdni()  {    cout << "Metoda tridy Podtrida volana pozdni vazbou" << endl; } | |

A na konec funkce main (ale samozřejmě před řádek return 0;)dopište řádky:

|  |  |
| --- | --- |
| |  | | --- | | n->pozdni();    p->pozdni();    problem->pozdni(); /\* Je skutecne volana void Podtrida::pozdni() \*/ | |

O tom která metoda ( void Nadtrida::pozdni(); , nebo void Podtrida::pozdni(); ) ve skutečnosti bude volána se rozhoduje až při běhu programu podle toho, na jakou instanci je volána, ne v době kompilace.

Jak je možné, že to funguje?

Jak jsem ve svých předchozích článcích naznačil metody s časnou vazbou jsou překládány jako "obyčejné" céčkovské funkce, kde je přidán jako 1. parametr this - implicitní parametr, který ukazuje na instanci, pro kterou je metoda vyvolána. Rozdíl mezi "obyčejnou" funkcí a metodou volanou časnou vazbou je jen v tom, že překladač u metody kontroluje, zda je volána skutečně na správný objekt. Metody volané pozdní vazbou (virtuální metody) se překládají trochu jinak. Každá instance, která má alespoň jednu virtuální metodu má v sobě navíc ukazatel na tak zvanou tabulku virtuálních metod (TVM, někdy jsem také viděl zkratku VMT). Tento ukazatel je pro programátora nepřístupný (Alespoň ne korektní cestou.) a ukazuje na tabulku, ve které jsou uloženy adresy virtuálních metod. O tom, že v instanci je jeden ukazatel navíc se můžete lehce přesvědčit pomocí sizeof. Řádek problem->casna();bude přeložen ne jako jednoduché zavolání funkce, ale jako vyvolání 1. virtuální metody (metody, jejíž adresa je ve TVM na 1. místě) která je ve TVM pro instanci na kterou ukazatel problém ukazuje. O správnou inicializaci ukazatele na TVM se postará překladač, který při vytváření každé instance (před zavoláním samotného konstruktoru) přidá kód pro inicializaci ukazatele na TVM. Programátor, který chce používat virtuální metody nemusí vlastně vůbec pojmy jako TVM znát, prostě stačí, že to funguje jak má. Uvedl jsem je jen proto, aby byly jasné některé omezení:

 Metoda označená jako virtual nemůže být současně označená jako inline. Což je vlastně logické uvědomíte-li si význam slova inline a mechanismus pozdní vazby pomocí TVM.

 Konstruktor nemůže být virtuální. Volat konstruktor pozdní vazbou nemá žádný smysl a vlastně ani není jasné, jak by se měl lišit od konstruktoru volaného časnou vazbou. Vytvářím-li instanci, musí mě být jasné jaké třídy bude. Ukazatel na TVM je inicializován před vyvoláním konstruktoru, tedy v těle konstruktoru lze použít virtuální metody instance, kterou vytvářím.

 Instance musí být korektně inicializována. Jednou jsem napsal, že pro dynamickou inicializaci instancí máte používat operátor new a ne nějaké céčkovské funkce malloc a podobné. Dosud v případě že instance měla implicitní konstruktor to nevadilo. Nyní to již vadí i v tomto případě. Funkce mallocnevolá konstruktor a ani neinicializuje ukazatel na TVM, takže vyvolání jakékoliv virtuální metody se bude chovat hodně , ale opravdu hodně zvláštně. Ani Vám nedoporučuji to zkoušet, když tak jen na Vaší vlastní zodpovědnost. Funkce malloc a podobné patří do jazyka C, do C++ patří operátor new.

Konstruktor virtuální být nemůže, ale destruktor virtuální být může a dokonce by i měl být. Alespoň v případě, že třída má jiné virtuální metody. V mém předchozím článku jsem upozornil na jedno zvláštní vyvolání "nesprávného" destruktoru. Problém byl v tom, že destruktor byl překládán časnou vazbou.

Někteří uživatelé Borland C++ Builderu si mohou všimnout jedné zajímavosti, která jako-by vyvrací mé tvrzení. Například třída TForm z knihovny VCL má konstruktor deklarovaný s klíčovým slovem virtual. Nejedná se v žádném případě o konstruktor volaný pozdní vazbou. Konstruktor volaný pozdní vazbou není ani v ANSI C++, ani nemůže být žádným rozšířením jazyka C++. Klíčové slovo virtual u konstruktorů některých tříd z VCL má jiný význam než pozdní vazba, a nemá s ANSI C++ nic společného.

Tolik pro vysvětlení rozdílu mezi časnou a pozdní vazbou, příště dokončím téma polymorfismu. Poté v dalším článku se vrátím zpět k dědičnosti, tentokráte vícenásobné, což je téma na 2 - 3 články.

## [Polymorfismus - dokončení](http://www.builder.cz/rubriky/c/c--/polymorfismus-dokonceni-155701cz)

**26. února 2001, 00.00 | V tomto článku dokončíme téma polymorfismu, virtuálních metod, abstraktních tříd. Ukážeme si k čemu se používají abstraktní třídy a jak nejlépe využívat virtuální metody.**

Slovo polymorfismus znamená něco jako jako "vícetvarost", nebo "mnohotvarost". Polymorfismus je v programování velmi obecný pojem. V souvislosti s OOP se jedná o to, že instance různých tříd na stejný podnět (na vyvolání stejné metody) reagují různě. Instance více tříd poskytují svému okolí stejnou službu, ale každá instance na vyžádání této služby provede něco jiného. Troufám si říci, že právě pro spojení polymorfismu s dědičností se vlastně používá OOP. Pro třídy poskytující stejné služby se nabízí nadtřída, která má společné vlastnosti všech svých podtříd. Velká část návrhu programů v OOP je vlastně hledání těchto společných služeb (vlastností) pro různé třídy, a vytváření nadtříd podobných tříd. Mějme jako příklad třídu kruh, a třídu obdélník. U obou chceme mít nějakou metodu, vracející obvod tohoto geometrického útvaru. V OOP se podobný problém řeší většinou tak, že se vytvoří nějaká nadtřída - už jsem ji vlastně pojmenoval na "geometrický útvar" a tato třída bude mít metodu pro vypočítání obvodu. Z této třídy budou dědit třídy kruh a obdélník, které si metodu pro výpočet obvodu implementují po svém. Konkrétně v C++ bude metoda float dejObvod(); deklarována jako virtuální a třídy kruh a obdélník si předefinují podle potřeby její tělo.

Abstraktní třídy, čiré metody

Zmiňované zavedení nadtřídy je vlastně abstrakcí tříd kruh a obdélník. Taková abstrakce se může na první pohled zdát zbytečná, proto se podívejme na výhody. V místech, kde budu používat instance tříd kruh, nebo obdélník, budu s těmito instancemi co nejvíce pracovat jako s instancemi jejich nadtřídy. Nebudu tím vlastně znát se kterou podtřídou zrovna pracuji a program bude algoritmus pracující s obecným geometrickým útvarem. Výhodu si uvědomíme, až když fungující program budu chtít rozšířit o další grafický útvar - třeba trojúhelník. V OOP se předpokládá, že při takové změně (rozšíření) se bude muset minimálně (nejlépe vůbec) zasahovat do napsaného zdrojového textu. Ve skutečně objektově orientovaném jazyce (Tím bohužel C++ není.) a při opravdu kvalitním návrhu je možné skutečně pří rozšiřování znovu použít hotové třídy beze změn. V C++ to s takovou ideální znovupoužitelností zdrojového textu není zase tak žhavé. Ale správně navržený program lze rozšiřovat s minimálními úpravami.

Abstrakce sebou nesou také určité problémy. Například v mém příkladě je otázka jaké tělo má mít metoda virtual float GrafickyUtvar::dejObvod(); . Nabízí se odpověď žádné, to by jsme ale měli nějak překladači dát najevo. K tomuto účelu existují tak zvané "pure" metody. Pure je v těchto souvislostech vhodné přeložil asi jako čirý. Nehodí se zde překlad prázdný, protože prázdná metoda je v C++ něco jiného. Čirou metodou může být jen metoda virtuální. Čirá metoda nemá tělo a nelze jí vyvolat. Čirá metoda se označí v deklaraci symboly = 0. Tedy například: virtual float dejObvod() = 0; . Aby se zabránilo vyvolání takové metody, nelze vytvořit instanci třídy, která má alespoň jednu čirou metodu. Třídu, která obsahuje alespoň jednu čirou metodu, nazýváme abstraktní třídou. Abstraktní třída nemůže mít své instance. ( Třídu nelze instanciovat - podivné slovo, které raději moc nepoužívám.) Vytvořím-li potomka, ve kterém nedefinuji tělo čiré metody, je metoda čirá i v potomkovi a potomek je také abstraktní třída.

Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h>  class GrafickyUtvar {  public:    virtual float dejObvod() = 0; };  class Kruh : public GrafickyUtvar {  private:    float r;  public:    Kruh():r(0){};    Kruh(float polomer):r(polomer){};    virtual float dejObvod(); };  class Obdelnik : public GrafickyUtvar {  private:    float a,b;  public:    Obdelnik():a(0),b(0){};    Obdelnik(float s1, float s2):a(s1),b(s2){};    virtual float dejObvod(); };  class Plocha {  private:    int Pocet;    GrafickyUtvar \*utvary[10];  public:    Plocha():Pocet(0){};    void pridej(GrafickyUtvar \*u)          { utvary[Pocet++] = u;};    float dejObvodyVsech();  };  float Plocha::dejObvodyVsech() {    float navrat = 0.0;     for(int p = 0; p<Pocet; p++)       {navrat += utvary[p]->dejObvod();}    return navrat; }  float Kruh::dejObvod() {    return 2\*3.14\*r; }  float Obdelnik::dejObvod() {    return 2\*(a+b); }  int main(void) {    Plocha p;    GrafickyUtvar \*a,\*b,\*c,\*d,\*e = NULL;    p.pridej(a = new Kruh(2));    p.pridej(b = new Kruh(10.3));    p.pridej(c = new Obdelnik(12,45));    p.pridej(d = new Obdelnik(1.4,100));    cout << p.dejObvodyVsech() << endl;     delete a;    delete b;    delete c;    delete d;    delete e; /\* Zkuste pro zkoušku odstranit // na následujících dvou řádcích. Nastane chyba. \*/    //GrafickyUtvar g;    //GrafickyUtvar \*gg = new GrafickyUtvar;    return 0; } | |

Třída plocha je napsána jen pro tento příklad, jinak není moc dobře implementována.

Nyní si představte situaci, že tento program chci rozšířit o třídu rovnostranný trojúhelník. Třída bude vypadat asi následovně:

|  |  |
| --- | --- |
| |  | | --- | | class RovnoStrannyTrojuhelnik : public GrafickyUtvar {  private:    float a;  public:    RovnoStrannyTrojuhelnik() : a(0){};    RovnoStrannyTrojuhelnik(float strana):a(strana) {};    virtual float dejObvod(); };  float RovnoStrannyTrojuhelnik::dejObvod() {    return 3\*a; } | |

Nyní ve funkci main před řádek cout << p.dejObvodyVsech() << endl; vepište řádek p.pridej(e = new RovnoStrannyTrojuhelnik(10)); . Přece jenom jsme museli již napsaný zdrojový text měnit - funkci main. Ale všimněte si, že jsem NIJAK nemusel měnit třídy Plocha, Grafický útvar, kruh, obdélník, nebo jejich metody! To není vše, ja jsem pro tohle rozšíření ani NEPOTŘEBOVAL ZDROJOVÉ TEXTY TĚCHTO TŘÍD!!!! Stačila mi pouze deklarace (V nějakém hlavičkovém souboru, který bych vložil pomocí include). Zdrojové texty těl metod tříd jsem nepotřeboval. Stačilo kdybych je měl zkompilovány v binárním tvaru (\*.obj, \*.dll, \*.o, atd...), které bych na závěr po kompilaci "přilinkoval".

Závěrem

Rozhodně Vám doporučuji používat polymorfismus v kombinaci s dědičností k vytváření abstraktních tříd. Je-li třída opravdu abstraktní, pro jistotu nějakou její metodu označte jako čirou. Nezapomeňte, že ne každá nadtřída musí být nutně abstraktní. Pro tento způsob abstrakcí musíte používat metody volané pozdní vazbou (virtuální). Má-li třída alespoň jednu virtuální metodu, měla by mít i virtuální destruktor - o tom jsem psal v minulém článku. Používání metod volaných pozdní vazbou je trochu pomalejší, než používání metod volaných časnou vazbou. Musí se totiž adresa podprogramu vypočítávat z TVM - viz předchozí článek. Přesto Vám doporučuji metody volané pozdní vazbou používat. Metody volané pozdní vazbou do OOP prostě patří. Existují dokonce jazyky (Java, atd...), které nemají časnou vazbu, vše je voláno pozdní vazbou. Časnou vazbu používejte, jen jste-li si jistí, ale opravdu jistí na 100%, že metodu v nějakém potomkovi nebudete měnit.

Ještě si Vás dovolím upozornit na poměrně častou chybu v přístupu k podobným problémům. Bylo by chybou dávat do třídy grafický útvar nějakou proměnnou, pomocí níž by jsem poznal, zda se jedná o obdélník kruh atd... Dědičnost bych nepoužil. Metoda na výpočet obvodu by byla vlastně jeden veliký switch a každý case by byl pro konkrétní útvar (kruh, atd...). Tím bych přišel o všechny výhody abstrakcí při rozšiřování programu. Není ani dobré dědičnost použít a v každé instanci nějaké podtřídy nastavit zmiňovanou proměnnou stejně. Cíl by byl identifikovat, o jakou podtřídu jde. Abstrakce spočívá právě v tom, že pracuji s nadtřídou a nevím jaká podtřída to vlastně je. Ale uznávám, že poměrně často je potřeba zjistit o jakou podtřídu se jedná. Kdybych náhodou já v uvedeném programu musel zjistit o jakou podtřídu jde, použil bych dynamickou identifikaci typu (O té si povíme později.), ale snažil bych se o to co nejméně, protože bych tím přišel o obecnost.

Tolik k polymorfismu. Příštích několik kapitol se budu věnovat zase dědičnosti, tentokrát vícenásobné a všem potížím, které může způsobit.

## [Vícenásobná dědičnost v C++](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c--155707cz)

**7. března 2001, 00.00 | V tomto článku si objasníme vícenásobnou dědičnost v C++ a předvedeme si některé problémy, které mohou nastat..**

Vícenásobná dědičnost je dědičnost, ve které je nová třída bezprostředním následovníkem více tříd. Narozdíl od jednoduché dědičnosti, kdy každá nová třída je bezprostředním následovníkem pouze jedné třídy.  
Vícenásobná dědičnost se chová podle stejných pravidel jako dědičnost jednoduchá, jen na rozdíl od jednoduché dědičnosti vzniká při vícenásobné dědičnosti několik "drobných" problémů. Vícenásobná dědičnost je tedy opět specializace. Nová třída "dědí" po svých předchůdcích vlastnosti, s tím, že další vlastnosti mohou být přidány. Přístup ke zděděným položkám při vícenásobné dědičnosti má stejná pravidla jako přístup ke zděděným položkám při dědičnosti jednoduché. Ve svém článku "[Jednoduchá dědičnost v C++](http://www.builder.cz/rubriky/c/c--/clanek-155692-jednoducha-dedicnost-v-c-/)" jsem uvedl tabulku přístupových práv, která beze změn platí i pro vícenásobnou dědičnost, proto ji zde nebudu uvádět znovu.

Syntaxe pro vícenásobnou dědičnost je velmi podobná jako syntaxe pro vyjádření dědičnosti jednoduché. Vlastně je zřejmé, že jednoduchá dědičnost je specielním případem dědičnosti vícenásobné, kdy počet předků je roven 1. Další předkové jsou oddělení čárkou. Tedy například:

class potomek : public Predek1, public Predek2, public Predek3 {};

Nyní jsem definoval třídu potomek, která má 3 předky, každý předek je děděn veřejnou dědičností. Nová třída nemá žádné nové atributy ani metody. Instanci třídy potomek v paměti počítače je dobré si představit jako-by byly "za sebou" 3 instance jednotlivých předků a dále nové vlastnosti třídy potomek.

Problémy při vícenásobné dědičnosti

Vse by bylo krásné nebýt několika velmi nepříjemných problémů, které vícenásobná dědičnost přináší. Problémy:

 Konflikt jmen

Při vícenásobné dědičnosti může nastat situace, že třída dědí z nadtříd, které mají stejné názvy atributů, nebo metod. Který potom zdědit?

 Opakovaná dědičnost

Nějaký atribut, nebo metoda mohou být zděděné "vícekrát". Opakovaná dědičnost nastává, jestliže v třídním diagramu existuje mezi 2 třídami více než jedna cesta. Například třída B dědí ze třídy A. Třída C dědí ze třídy A. Třída D dědí ze třídy C a B. Třída D vlastně dvakrát dědí ze třídy A. Má mít třída D položky třídy A dvakrát, nebo jen jednou?

 Volání konstruktorů a destruktorů

Jak jsem uvedl ve článku o jednoduché dědičnosti při vytváření instance se implicitně zavolají konstruktory předka, při rušení se volá implicitně destruktor potomka. Ve vícenásobné dědičnosti je ale bezprostředních předků více. V jakém pořadí se tedy mají volat konstruktory a destruktory?

Dalším problémem často se projevujícím u vícenásobné dědičnosti je správné (spíše špatné) přetypování potomka na předka. Jedná se ale spíše o problém s přetypováváním, proto se mu nebudu věnovat ve článcích o vícenásobné dědičnosti, ale až někdy v budoucnu ve článku věnovanému přetypování v C++.

Konflikt jmen

Tedy problém jaké atributy a metody dědit, jestliže se vyskytují ve více předcích se stejným názvem. Upozorňuji jen, že v tomto případě nesmí mít předci žádnou společnou nadtřídu. Mají-li, dojde sice také ke kolizi jmen, ale jedná se o problém opakovaného dědění. Problém kolize jmen se řeší tak, že třída zdědí všechny atributy a metody i v případě, že mají stejné jméno. Prostě ve třídě existuje více položek se stejným názvem. Protože přístup k položkám musí být jednoznačný, přistupuje se k položce pomocí tak zvaného úplného jména. Úplné jméno je jméno třídy následováno "čtyřtečkou" a poté následuje název atributu, nebo metody. Vše vysvětlím na příkladu:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> class A {   public:     int Atribut;     void nastav(int a) { Atribut = a; }     int vrat() { return Atribut; }     virtual void metoda(); };  class B {   public:     int Atribut;     void nastav(int a) { Atribut = a; }     int vrat() { return Atribut; }     virtual void metoda(); };  class C : public A, public B {   public:     void nuluj(); };  void A::metoda() {     cout << "A" << endl; } void B::metoda() {     cout << "B" << endl; } void C::nuluj() {     A::Atribut = 0;     B::Atribut = 0; /\* Odkomentujete-li následující řádek, překladač Vás chybou upozorní na nejednoznačnost. \*/     /\* Atribut = 0; \*/ }; | |

Dobře si všimněte jak v metodě C::nuluj přistupuji k atributu Atribut. Jedná se o úplné jméno atributu, čímž zamezím nejednoznačnosti. Kdybych chtěl zavolat metodu, jejichž jméno je v nějaké kolizi, musel bych také uvést úplné jméno. Každá z metod pracuje nad "svými" atributy. Tedy metoda A::nastav pracuje s atributem A::Atribut a metoda B::nastav pracuje s atributem B::Atribut. Pro pochopení přidávám následující funkci main:

|  |  |
| --- | --- |
| |  | | --- | | int main() {      C c;      c.A::nastav(10);      c.B::nastav(20); /\* Odkomentujete-li následující řádek, překladač Vás chybou upozorní na nejednoznačnost. \*/     /\* c.nastav(0); \*/      cout << c.A::vrat() << '\t' << c.B::vrat() << endl;      c.nuluj();      cout << c.A::vrat() << '\t' << c.B::vrat() << endl;      c.A::metoda();      c.B::metoda();      return 0; } | |

Jak vidíte každá metoda pracuje nad "svými" daty. U virtuálních metod je velkou výhodou, že je lze předefinovat. Proto by bylo vhodné metodu metoda ve třídě C předefinovat, aby se při jejím volání nemuselo psát úplné jméno. Nic mi nebrání tělo metody předefinovat například takto:

|  |  |
| --- | --- |
| |  | | --- | | void C::metoda() {     A::metoda(); } | |

nebo:

|  |  |
| --- | --- |
| |  | | --- | | void C::metoda() {     A::metoda();     B::metoda(); } | |

Nebo jakkoliv jinak.

Tolik tedy k jednomu z problému vícenásobné dědičnosti. Příště budu řešit problém opakovaného dědění a poté v dalším článku problém konstruktorů a destruktorů při vícenásobné dědičnosti.

## [Vícenásobná dědičnost v C++ - opakovaná dědičnost](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-opakovana-dedicnost-155717cz)

**14. března 2001, 00.00 | V tomto článku si povíme něco o dalším z problémů při vícenásobné dědičnosti - o opakovaném dědění.**

V minulém článku jsem popsal vícenásobnou dědičnost, vyjmenoval problémy, které přináší a ukázal problém konfliktu jmen. V tomto článku objasním druhý problém - opakovanou dědičnost. Jak jsem uvedl minule, opakovaná dědičnost nastane, jestliže v třídním diagramu existuje mezi 2 třídami více, než jedna cesta. Tedy například:

|  |  |
| --- | --- |
| |  | | --- | | class A {   public:     int A; };  class B : public A {   public:     int B; };  class C : public A {   public:     int C; }; | |

Třídy B, C mají tribut int A zděděný z třídy A. Nyní vytvořme třídu D následovně:

|  |  |
| --- | --- |
| |  | | --- | | class D : public B, public C {   public:     int D; }; | |

Třída D obsahuje dva atributy int A . Vše je v pořádku jestliže z nějakého důvodu chceme mít atribut int A ve třídě D dvakrát. K atributům B::A a C::A potom přistupujeme jako při běžném konfliktu jmen, který jsem popsal v minulém článku. Zde je ale spíše logické chtít, aby atribut int A byl ve třídy D jen jednou, protože se vlastně jedná o jeden atribut A::A, který byl opakovaně zděděn.

Virtuální nadtřída

Řešením je udělat třídu A virtuální nadtřídou tříd B, C. K vyjádření virtuální nadtřídy se používá klíčové slovo virtual , které známe jako slovo deklarující metodu volanou pozdní vazbou. "Virtuální dědění" nemá s pozdní vazbou nic společného. Opět vidíme v C++ situaci, kdy jedno klíčové slovo má více významů. Opravíme zdrojový text takto:

|  |  |
| --- | --- |
| |  | | --- | | class A {   public:     int A; };  class B : virtual public A {   public:     int B; };  class C : virtual public A {   public:     int C; };  class D : public B, public C {   public:     int D; }; | |

Nyní má třída D skutečně jen jeden atribut int A . Ještě jen upozorním, že všichni další potomci tříd B a C budou dědění virtuálně. Velkou nevýhodou virtuálního dědění je fakt, že pro virtuální dědičnost se musím rozhodnout dříve, než dojde k vícenásobnému dědění. Tedy když jsem v mém příkladě vytvářel třídy B a C, už jsem musel počítat s tím, že bude existovat nějaká třída D, u které nastane problém opakovaného dědění. Jinak bych u jednoduché dědičnosti nepoužil klíčové slovo virtual. Není příliš dobré preventivně u každé dědičnosti použít klíčové slovo virtual, protože třída, která má virtuální nadtřídu obsahuje navíc ještě jeden ukazatel. Takže třída class B1 : public A není stejná jako třída class B2 : virtual public A. Onen zmiňovaný ukazatel není ukazatelem na tabulku virtuálních metod. Jedná se o ukazatel, který bude nutný pro sdílení prvků třídy při opakované vícenásobné dědičnosti.

Právě problémy s virtuálním děděním jsou velmi silným argumentem odpůrců vícenásobné dědičnosti. Používáte-li vícenásobnou dědičnost, doporučuji Vám vyhnout se virtuálnímu dědění a všem problémům, které přináší. V příštím článku dokončíme téma vícenásobné dědičnosti přehledem pravidel pro vytváření a likvidování instancí tříd vyniklých vícenásobnou dědičností. Tedy povíme si, v jakém pořadí se volají konstruktory a destruktory při vícenásobné dědičnosti.

## [Vícenásobná dědičnost v C++ - volání konstruktorů a destruktorů](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-volani-konstruktoru-a-destruktoru-155723cz)

**20. března 2001, 00.00 | Dnes si dokončíme téma vícenásobné dědičnosti. Podíváme se jaké platí pravidla pro volání konstruktorů a destruktorů při vícenásobné dědičnosti.**

V tomto článku si ukážeme jak to je s voláním konstruktorů a destruktorů při vytváření a rušení instancí tříd vzniklých vícenásobnou dědičností. Stejně jako při jednoduché dědičnosti se i u vícenásobné dědičnosti nedědí konstruktory a destruktory, ale lze je vyvolat implicitně (stejně jako při jednoduché dědičnosti).

Konstruktory

Při vytváření instance třídy odvozené jednoduchou dědičností jsem napsal, že kromě konstruktoru této třídy je implicitně vyvolán bezparametrický konstruktor "nejvyšší" nadtřídy, potom konstruktor její podtřídy a tak dále až je vyvolán konstruktor dané třídy. U vícenásobné dědičnosti je situace obdobná. Není-li použito virtuální dědění (Viz předchozí článek [Vícenásobná dědičnost v C++ - opakovaná dědičnost](http://www.builder.cz/rubriky/c/c--/clanek-155717-vicenasobna-dedicnost-v-c-opakovana-dedicnost/)) je situace jednoduchá. Nejprve je volán konstruktor nejvyšší nadtřídy prvního předka (uvedeného v seznamu předků nejvíce vlevo), poté jeho potomka a tak dále, až je zavolán konstruktor předka naší třídy. Poté jsou stejným způsobem volány konstruktory ostatních předků naší třídy. Po vyvolání konstruktorů všech předků je vyvolán konstruktor naší třídy. Vše objasní příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> class A {   public:     A() { cout << "A" << endl; } };  class B : public A {   public:     B() { cout << "B" << endl; } };  class C : public A {   public:     C() { cout << "C" << endl; } }; class D : public B, public C {   public:     D() { cout << "D" << endl; } };  int main() {     D d;     cout << "Instance d je vytvorena" << endl;     return 0; } | |

Program vypíše:  
A  
B  
A  
C  
D  
Instance d je vytvořena  
  
Stejně jako u jednoduché dědičnosti lze i při vícenásobné dědičnosti volání bezparametrických konstruktorů potlačit a volat konstruktory s parametry.

Trochu problematičtější je použití virtuální dědičnosti. S virtuální dědičností jsou bohužel jen problémy. Rozdíly jsou vlastně dva:

 Ze všeho nejdříve budou vyvolány konstruktory virtuálních nadtříd a to bez ohledu na pořadí předků.

 Konstruktor virtuální nadtřídy bude volán jen jednou.

Pokud v mém předchozím příkladu upravíte třídy B a C tak, aby dědily virtuálně, program po spuštění vypíše:  
A  
B  
C  
D  
Instance d je vytvořena  
Tedy přesně podle pravidel, která jsem uvedl. Nyní můžeme trochu experimentovat a třídu deklarovat třeba takto: class D : public B, public E, public C { /\* To co tu bylo \*/ } . Před tím ale třídu E deklarovat jako nevirtuálního potomka třídy A. Vznikne zajímavá situace. Třída D má nadtřídy B a C, které mají virtuální nadtřídu A. Tedy D bude mít po B a C zděděné atributy a metody jen jedenkrát. Ale dále ještě přímo dědí z E, kde A není virtuální předek, takže zde budou atributy a metody třídy A podruhé. Stejně tak volání konstruktoru třídy A bude dvakrát. Jednou se A::A() zavolá jako virtuální předek tříd B a C (Platí pravidlo, že konstruktor virtuálního předka je volán jen jednou.) a poté se zavolá znovu A::A() jako nevirtuální přímý předek třídy E. Doporučuji Vám takto opravit zdrojový text a program spustit. Jak vidíte, použití jedné třídy jako virtuálního i nevirtuálního předka nám sem vneslo trochu zmatku.

Destruktory

Destruktory se chovají podle stejných pravidel jako konstruktory. S tím rozdílem, že jejich volání probíhá v opačném pořadí než volání konstruktorů. Tedy jejich volání neprobíhá směrem od "nadpředka" k potomkům ale od potomků k "nadpředkovi". Situace je obdobná jako při jednoduché dědičnosti.

Závěrem o vícenásobné dědičnosti

Používání vícenásobné dědičnosti (Hlavně použití virtuální dědičnosti při opakovaném dědění.) v C++ je složité a velmi nepřehledné. Představte si ne můj příklad, ale velký program, který má třeba desítky, nebo stovky tříd různě propojených jednoduchou a vícenásobnou dědičností. Bude asi radost orientovat se ve virtuálních a nevirtuálních nadtřídách a při volání konstruktorů a destruktorů virtuálních a nevirtuálních nadtříd. Některé programovací jazyky (Například Java) řeší tento problém velmi svérázným způsobem - vícenásobnou dědičnost vůbec nepodporují. O používání vícenásobné dědičnosti se vedou veliké spory. Vícenásobná dědičnost má své nadšence, kteří ji používají kde mohou, i velké odpůrce. Proti vícenásobné dědičnosti je silný argument. Není totiž znám způsob, kdy vícenásobnou dědičnost nelze korektně obejít. Další argumenty proti říkají, že vícenásobná dědičnost v C++ (hlavně virtuální dědění) je příliš složitá a nepřehledná. Ale podle mne je správné, že vícenásobná dědičnost v C++ je. Je-li tak špatná, jak tvrdí její odpůrci, nemusí ji programátor použít. Nedojde-li ke konfliktu jmen a opakovanému dědění, není vícenásobná dědičnost vlastně příliš složitá. S konfliktem jmen se při troše trpělivosti také dá pracovat. Jediné co Vám nedoporučuji, je pouštět se do opakovaného dědění a virtuální dědičnosti - to chce opravdu nervy :(

Tolik k vícenásobné dědičnosti. V příštím článku se podívám na přetěžování operátorů a poté v dalším článku na vstupní a výstupní operace v C++.

## [Přetěžování operátorů v C++ 1.díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-1-dil-155728cz)

**26. března 2001, 00.00 | Dnes se podíváme jak přetěžovat operátory v C++. Vysvětlíme si význam klíčového slova operator. Dále si ukážeme několik jednoduchých operátorů přetížených jako funkce, i jako členské metody.**

Nejprve se pokusím vysvětlit co pojem přetěžování operátorů znamená. Pro lepší představu považujme libovolný operátor v C++ (například operátor + , nebo \* ) jako normální funkci. Například operátor + může být považován za funkci, která má dva parametry a vrací výsledek. Oba parametry mohou být třeba int a návratová hodnota je potom také int. O přetěžování funkcí, nebo metod jsem se částečně již zmínil ve svém úvodním článku mého seriálu. Jedná se o to, že lze definovat i deklarovat metody, nebo funkce se stejným jménem ale s různými parametry. Stejně tak existují i přetížené operátory. Například již zmiňovaný operátor + je také přetížen. Lze jej totiž také použít například pro sčítání čísel float, atd... Fakt, že operátory lze použít pro operandy různých typů (Tedy že jsou přetížené) nebude asi pro nikoho, kdo zná nějaký programovací jazyk, žádnou novinkou. V C++ ale existuje způsob jak přetížit operátory pro programátorem definované typy.

Pravidla pro přetěžování operátorů

Nejprve se podíváme, jaké pravidla a omezení musíme používat pro přetěžování operátorů.

 Přetížené operátory se nemohou lišit pouze v typu návratové hodnoty, ale v typech parametrů. Toto omezení platí i pro přetěžování funkcí, nebo metod.

 Přetížíme-li binární operátor, musí být zase binární. Přetížíme-li unární operátor, musí být zase unární. U operátorů tedy nelze měnit počet parametrů. Například operátor / je binární (má dva operandy), nemůžeme jej přetížit na unární operátor.

 Nelze vytvářet nové operátory.

 Nelze přetěžovat operátory: . .\* :: ?: Ostatní přetěžovat lze. Lze dokonce přetěžovat operátor [] - indexování, () - volání funkce, i přetypování atd...

 Pro operátory platí stále stejná priorita, ať jsou přetížené jakkoliv. Prioritu operátorů v C++ nelze nijak změnit.

Jak přetěžovat operátory

Operátory v C++ lze přetěžovat dvěma způsoby. Jednak jako členské metody a jednak jako funkce. V mých předchozích článcích jsem příliš nedoporučoval používání funkcí, jen členských metod. Také jsem ale hned uznal, že v C++ existuje mnoho případů, kdy je použití funkcí výhodnější, nebo i nezbytné. Přetěžování operátorů je jedním z těchto případů. Mnohdy bude výhodné operátor přetížit jako funkci. Operátor se přetíží jako funkce, ne jako metoda, je-li jeho 1. parametrem primitivní datový typ, nebo třída, kterou nemůžeme měnit. Binární operátor přetížen jako funkce bude mít dva parametry, unární jen jeden. Binární operátor přetížený jako metoda bude mít jen jeden parametr, protože druhým parametrem bude implicitní parametr this. Unární operátor přetížený jako členská metoda nebude mít parametr, protože jeho jediným parametrem je this. K přetěžování operátorů slouží klíčové slovo operator. Jako příklad uvedu třídu pro tří rozměrný vektor a přetížím pro ni mnoho operátorů. Pro jednoduchost budou souřadnice vektoru celá čísla. Všechny operátory budou přetíženy jako členské metody kromě operátoru \*. Tento operátor přetížím jako funkci.

|  |  |
| --- | --- |
| |  | | --- | | class Vektor {   private:     int \*pole;   public:     Vektor():pole(new int[3]){}     Vektor(const Vektor &kopie);     ~Vektor() { delete[] pole;}     Vektor operator=(const Vektor &kopie);     bool operator==(const Vektor &druhy) const;     bool operator!=(const Vektor &druhy) const;     Vektor operator+(const Vektor &druhy) const;     int operator() (int x, int y) const;     int& operator[] (int i) const; }; Vektor operator\*(const int cislo, const Vektor &b); /\*Deklarace ("hlavička") funkce \*/  Vektor::Vektor(const Vektor &kopie) /\*Nutný kopírovací konstruktor \*/ :pole(new int[3]) {    int i;    for(i = 0; i < 3; i++)       this->pole[i] = kopie[i]; } | |

Jistě by se dalo najít více operátorů, které přetížit. Například += a podobné. Myslím, že pro příklad stačí tyto operátory. Nyní budu postupně implementovat těla metod. Nejprve operátor =. O významu tohoto operátoru jsem se již zmiňoval ve svém článku "Kopírovací konstruktor v C++". Nepřetížený operátor = vytváří pouze plytkou kopii, což v našem případě není výhodné. Proto jej musíme přetížit. U přetěžování všech operátorů nezapomeňte na jejich návratové hodnoty. Každý přetížený operátor by se měl chovat i vracet hodnoty tak, jak se od něj intuitivně očekává. Například operátor + by neměl odečítat a podobně. Stejně tak by operátory měly vracet hodnotu, která se intuitivně očekává. Například operátor = vrací svou pravou stranu. Mám-li například int a; a=3; , potom výraz a = 3 vrací 3. Normálně lze napsat b = a = 3; . Proto bude i náš operátor vracet pravou stranu.

|  |  |
| --- | --- |
| |  | | --- | | Vektor Vektor::operator=(const Vektor &kopie) {    for (int i = 0; i < 3; i++)    {       pole[i] = kopie[i];    }    return kopie; } | |

Operátor = není konstantní metoda, protože mění objekt this.

Dalším operátorem, který přetížíme bude operátor = , tedy operátor pro porovnání. Od tohoto operátoru se intuitivně očekává, že bude vracet proměnnou typu bool. V případě rovnosti objektů vrátí true, v opačném případě false. Při psaní této metody se dopustím veliké nepřesnosti. Pro jednoduchost budu považovat dva vektory za shodné, budou-li obsahovat stejné prvky. Tento příklad slouží pouze jako ukázka jak přetěžovat operátory, ne jak počítat s vektory. Matematicky správně je převést oba porovnávané vektory do normovaného tvaru a ty potom porovnat. Nechci Vás ale zatěžovat zdrojovým textem převádějícím vektory do normovaného tvaru, když téma článku je jiné.   
Tato metoda vlastně srovnává objekty this a druhy. Objekt this se nezmění, proto operátor ==může být deklarován jako konstantní metoda.

|  |  |
| --- | --- |
| |  | | --- | | bool Vektor::operator==(const Vektor &druhy) const {    for (int i = 0; i < 3; i++)    {       if (pole[i] != druhy[i])          return false;    }    return true; } | |

Operátor != lze nejjednodušeji přetížit takto:

|  |  |
| --- | --- |
| |  | | --- | | bool Vektor::operator!=(const Vektor &druhy) const {    return !(\*this == druhy); } | |

Dále přetížíme operátor + pro sečtení dvou vektorů. Objekt this se opět nemění, proto je metoda deklarována jako konstantní.

|  |  |
| --- | --- |
| |  | | --- | | Vektor Vektor::operator+(const Vektor &druhy) const {    Vektor navrat;    for (int i = 0; i < 3; i++)       navrat[i] = (\*this)[i] + druhy[i];    return navrat; } | |

Text článku je již příliš rozsáhlý a proto téma přetěžování operátorů dokončím příště. Další článek bude přímo navazovat na tento. Dokončím v něm přetěžování všech operátorů, které jsem deklaroval a ukážu i jednoduchý příklad ve funkci main, jak různě lze přetížené operátory použít.

## [Přetěžování operátorů v C++ 2. díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-2-dil-155729cz)

**29. března 2001, 00.00 | V tomto článku dokončíme téma přetěžování operátorů. Dokončíme příklad z minulého dílu a předvedeme si i použití přetížených operátorů na jednoduché funkci main.**

V tomto článku navazuji na svůj předchozí článek o přetěžování operátorů v C++. Všem, kteří můj předchozí článek ["Přetěžování operátorů v C++ 1. díl"](http://www.builder.cz/rubriky/c/c--/clanek-155728-pretezovani-operatoru-v-c-1-dil/) nečetli, doporučuji si jej přečíst. Všechny zdrojové texty, které napíši v tomto článku, přímo navazují na článek předchozí.

Dalším operátorem, který přetížím, je operátor volání funkce. Chtěl bych ukázat, že i tento operátor lze přetížit. Nenapadá mně ale žádný rozumný význam tohoto operátoru pro vektor. Tento operátor vrátí x-tý prvek vektoru zvětšený o y. Je to nesmyslný příklad, ale nic lepšího mě nenapadá. Díky přetížení tohoto operátoru se budu moci chovat k instancím třídy Vektor jako k funkcím s dvěma parametry typu int.

|  |  |
| --- | --- |
| |  | | --- | | int Vektor::operator()(int x, int y) const {    return ((x >= 0) && (x < 3))? (\*this)[x] + y : y; } | |

Dalším operátorem, který přetížím, je operátor indexování pole. Právě díky přetížení tohoto operátoru můžu pracovat s instancemi třídy vektor jako s jednorozměrným polem. Použil jsem jej již v těle operátoru ==, kde jsem napsal výraz druhy[i]. V těle této metody jednoduše ošetřím případnou chybu indexování. Bude-li požadavek mimo rozsah pole, vrátí metoda prvek s indexem 2.

|  |  |
| --- | --- |
| |  | | --- | | int& Vektor::operator[] (int i) const {    if ((i>=0) && (i<3))    {       return pole[i];    }    return pole[2]; } | |

Je dobré si všimnout, že metoda nevrací přímo prvek int, ale referenci na něj. Jak ukážu později, díky tomu může být výraz s operátorem indexování i na levé straně výrazu. Což se od něj také intuitivně očekává. Nevýhodou je, že tímto způsobem (Když operátor indexování bude ve výrazu vlevo) lze změnit i konstantní objekt, což začíná dělat zdrojový text nepřehledným.

Nyní ještě přetížím operátor \* jako funkci.

|  |  |
| --- | --- |
| |  | | --- | | Vektor operator\*(const int cislo, const Vektor &b) {    Vektor navrat;    for(int i = 0; i < 3; i++)       navrat[i] = b[i] \* cislo; /\* I zde je operátor [] vlevo \*/    return navrat; } | |

Tolik tedy přetížených operátorů. Nyní ukáži jednoduchou funkci main, ve které předvedu několik operací s instancemi mé třídy Vektor. Jsou-li například m, n vektory, a chci je sečíst pomocí mého operátoru, měl bych "správně" napsat: a.operator=(m.operator+(n)); . Ale právě proto přetěžuji operátory, abych je mohl zapisovat přirozeně. Tedy mohu klidně napsat a = m + n; .

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> int main(void) {    Vektor a,b,c;    for (int i = 0; i < 3; i++)    {       a[i] = i + 3; /\* Nebo také a.operator[](i) = i + 3; \*/    }    b = c = a;    b = 10 \* c;    if (a == c)       cout << "OK" << endl;    cout << b(1,2) << endl; /\* Operátor volání funkce \*/    a = b + c;    for (int i = 0; i < 3; i++)    {       cout << a[i] << endl;       cout << b[i] << endl;       cout << c.operator[](i) << endl;    }    return 0; } | |

Závěrem bych vám jen chtěl doporučit, abyste operátory přetěžovali s rozvahou. Je skutečně fantastické věc, mohu-li přetěžovat operátory +, \* , atd... pro nějaké matematické objekty jako třeba matice, vektory, body, souřadnice, zlomky, iracionální čísla a podobně. Co ale u jiných tříd objektů? Co je například součtem dvou oken? Je mnoho tříd, u kterých přetěžování operátorů vede jen k nepřehlednostem ve zdrojovém textu. Dále také přetěžujte operátory tak, aby se chovali, jak se od nich intuitivně očekává. Překladači nebude vadit, jestliže operátor +bude odčítat. Zmatete tím sice dokonale "západní rozvědky", ale hlavně také sebe, nebo své kolegy, s nimiž programujete.

Tolik tedy k přetěžování operátorů a v příštím článku se podíváme na vstupní a výstupní operace. Budeme při tom přetěžovat operátor << a operátor >>, jejichž původní význam jsou binární posuny. Jak si ale mnozí určitě všimli, používám je i já ve svých článcích v souvislosti s doposud "záhadným a tajemným" objektem cout k výpisu na stdout.

## [Vstupní a výstupní operace pomocí datových proudů v C++](http://www.builder.cz/rubriky/c/c--/vstupni-a-vystupni-operace-pomoci-datovych-proudu-v-c--155739cz)

**9. dubna 2001, 00.00 | V tomto článku si ukážeme jak pracovat se vstupním a výstupním zařízením v C++. Objasníme si význam objektů cin, cout, cerr. Podíváme se na datové proudy, třídu ios a její potomky. Ukážeme si také manipulátory a jak formátovat výstup.**

Vstupní a výstupní operace pomocí datových proudů v C++

V tomto článku si ukážeme jak pracovat se vstupem a výstupem v C++. V grafických operačních systémech se již standardní vstup a výstup příliš nepoužívá. Přesto je dobré tyto možnosti znát. Standardní vstup a výstup se používá například v CGI (O C++ a CGI v tomto seriálu také jednou budu psát.) a také mnohdy v Unixových OS pří psaní filtrů.

Každý, kdo čte mé články, si určitě všimnul, že pro výpis na stdout nepoužívám knihovnu jazyka C - stdio, ale používám iostream. V C++ lze také používat knihovnu stdio, já bych ale doporučoval používat iostream, na jejíž výhody bych chtěl v tomto článku poukázat.

Datový proud

Datový proud je reprezentován abstraktní (velmi abstraktní) třídou ios. Datový proud reprezentuje vždy nějaký "přesun" dat (plynutí dat, nebo proud dat) od zdroje k cíli. Ze třídy ios dědí další třídy jako například istream (vstupní proud), ostream (výstupní proud), fstreambase (datové proudy pro soubory), strstreambase (paměťové datové proudy pro řetězce). U tříd dědících z ios je naprosto běžná vícenásobná dědičnost. Třídy mají mnoho metod, které zde nebudu vypisovat. Lze je najít v každé dokumentaci, nebo i v hlavičkovém souboru iostream.h . V hlavičce iostream.h jsou definovány objekty cout, cin, cerr . Právě ty pro nás budou nyní důležité.

|  |  |  |  |
| --- | --- | --- | --- |
| název objektu | je instance třídy | datový prou pro | v jazyce C |
| cout | ostream | výstup | stdout |
| cin | istream | vstup | stdin |
| cerr | ostream | chybový výstup | stderr |

Třída ostream má přetížený operátor << (operátor bitového posunu) pro všechny primitivní datové typy a také pro pole char, tedy vlastně pro řetězce. Význam tohoto operátoru je poslat svůj pravý operand do datového proudu, který je levým operandem. Operátor << vrací referenci na instanci ostream, takže je možné operátory << dávat "za sebe". Vše si můžeme ukázat na následujícím příkladu, kdy pošleme nějaká data proudem na stdout a stderr.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> int main(void) {    cout << "Ahoj" << endl;    char a = 'A';    unsigned int c = 1000;    bool pravda = true;    cout << a << c << pravda << endl;    cerr << "Toto je chybový výstup:" << pravda << endl;    return 0; } | |

Výraz cout << a vrací opět cout (referenci). Proto je možné napsat cout << a << c. Důvody proč používat proudy místo funkcí z jazyka C mě napadají dva:

 Do proudu se dají poslat tak zvané manipulátory.

 Jednoduše se dá přetížit operátor << pro uživatelem definované typy.

Manipulátory

Manipulátory, jak již sám název napovídá, slouží k manipulaci s proudem. Jeden manipulátor již dlouho bez vysvětlení používám. Jedná se o manipulátor endl . Význam některých manipulátorů ukážu v tabulce:

|  |  |
| --- | --- |
| Manipulátor | Význam |
| endl | Vloží konec řádku a vyprázdní buffer (vyrovnávací paměť) proudu. |
| flush | Vyprázdní buffer proudu. |
| setw | Minimální počet znaků pro vypsání hodnoty. Tento manipulátor má jeden celočíselný parametr. |
| dec | Výpis čísel bude v desítkové soustavě. |
| oct | Výpis čísel bude v osmičkové soustavě. |
| hex | Výpis čísel bude v šestnáctkové soustavě. |
| setfill | Tento manipulátor má 1 parametr. Určuje jakým znakem bude vyplňováno volné místo, je-li nastaveno setw. |

Doporučuji Vám používat raději manipulátor endl, než posílat na proud znak '\n', protože endltaké vyprázdní buffer. Použijete-li nějaký manipulátor s parametrem, musíte vložit hlavičkový soubor iomanip.h . Použití manipulátorů si ukážeme na následujícím příkladě:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> #include <iomanip.h> int main(void) {    unsigned int c = 1000, b = 9;    cout << setw(3) << "b=" << b << endl << "c=" << c << endl;    cout << setw(3) << setfill('@') << hex << "c=" << c << endl << "b=" << b << endl;    return 0; } | |

Jak vidíme, manipulátory s parametrem "platí" jen do dalšího odeslání dat na cout. Pro každý manipulátor s parametrem je k dispozici metoda, která provede to samé. Například místo cout << setw(6); lze napsat cout.width(6); .

Opomenuli jsme objekt cin. Tento objekt reprezentuje datový proud "spojený" se standardním vstupem. Jak asi mnohé napadne třída istream má přetížený operátor >> pro vstup. Pravý operand operátoru je instance třídy istream a levý operand je proměnná, která má přijmout data. Operátor >> je přetížen pro všechny primitivní datové typy i pro pole znaků. U pole znaků může nastat jen jeden "malý" problém. Operátor >> čte znaky jen do prvního bílého znaku. Bílý znak je v tomto případě i mezera. Tedy stejný problém, jako když funkce scanf z jazyka C čte řetězec. Celý řádek lze přečíst pomocí metody istream& getline(char\*, int, char = '\n') , kde první parametr je řetězec, druhý parametr je maximální počet znaků a třetí parametr je ukončovač řádku, jehož implicitní hodnota je '\n'. Použití ukážu na příkladu:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream.h> int main(void) {    char a, s[100];    cout << "Napiš řetězec" << endl;    cin.getline(s,100);    cout << "Napsal jsi " << s <<endl;    cout << "Napiš znak" << endl;    cin >> a;    cout << "Napsal jsi " << a <<endl;    return 0; } | |

Tolik tedy k úvodu. Možná jsem Vás ještě nepřesvědčil o používání proudů místo funkci z stdio. Výhoda proudů se projeví hlavně při přetěžování operátorů << >> pro uživatelské datové typy a třídy. O tom jak tyto operátory přetěžovat, a také jak pracovat s textovými soubory, si povíme v příštím článku.

## [Přetěžování operátorů << a >> pro datové proudy v C++](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-lt-lt-a-gt-gt-pro-datove-proudy-v-c--155741cz)

**13. dubna 2001, 00.00 | Dnes si ukážeme jak přetížit operátory << a >> pro výstup a vstup do datových proudů. Také si řekneme něco o práci s textovými soubory pomocí datových proudů.**

Přetěžování operátorů << a >> pro datové proudy v C++

V minulém článku jsme si něco málo ukázali jak pracovat s datovými proudy a něco o objektech cin ,cout, cerr. Dnes si ukážeme jak poslat do (resp. přijmout z) datového proudu nejen primitivní datový typ, ale také instanci nějaké třídy. Také si ukážeme jak používat další stream - fstream, tedy datový proud, jehož "cílem" je soubor.

Jak posílat do datových proudů vlastní objekty?

Jedná se o vlastnost, kterou nám již knihovna stdio z jazyka C neposkytuje. Jde právě o jeden ze dvou minule zmiňovaných důvodů proč používat proudy místo funkcí s stdio. Opět použijeme přetížené operátory << a >>. Hlavní problém je, jak je přetížit. V první řadě musím říci, že operátory je dobré (vlastně i nutné!) přetěžovat jako obyčejné funkce, nikoliv jako členské metody. Důvod je jasný. Operátor přetížený jako členská metoda by musel být metodou nějakého proudu. To však autoři knihovny nemohli udělat, protože v době tvorby knihovny pro datové proudy nemohli znát náš datový typ, pro který budeme požadovat operátor <<, nebo >>. Stejně tak nemůžeme, jak se v OOP v podobných situacích dělá, vytvořit nějakou svou třídu mujstream, které zdědí vše například z ostream, a navíc třídu ostream rozšíří o operátor <<. Problém je v tom, že ze třídy ostream dědí mnoho tříd, které by dědily i nadále z ostream, ne z mujstream, takže by metodu << nedědily. Ve svém článku ["Přetěžování operátoru 1. díl"](http://www.builder.cz/rubriky/c/c--/clanek-155728-pretezovani-operatoru-v-c-1-dil/) jsem přesně uvedl, kdy přetěžovat operátory jako funkce a kdy jako metody. Nelze tedy operátory pro přístup k proudům přetížit jako metody. Smíříme se s tímto faktem a ukážeme si, jak přetížit operátor >> pro nějakou jednoduchou třídu:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> class MojeTrida {    private:       int A,B;    public:       int dejA() { return A; }       int dejB() { return B; }       void nastavA(int a) { A = a; }       void nastavB(int b) {B = b; } };  istream& operator>>(istream &is, MojeTrida &objekt) {    int cislo;    is >> cislo; /\* Využiji již přetížených operátorů pro primitivní datové typy.\*/    objekt.nastavA(cislo);    is >> cislo;    objekt.nastavB(cislo);    return is; /\* Nezapomeňte na tuto drobnost! Jinak nebudete moc dávat operátory za sebe. \*/ } | |

Operátory << , >> by měly vracet referenci na proud, na který byly volány. Je to nepsané pravidlo, aby šlo operátory dávat "za sebe".

Náš operátor přistupuje k soukromým atributům třídy MojeTřída pomocí veřejných metod. Operátor >> není ale zase tak "cizí" metoda, abychom ji nedovolili přistupovat k soukromým datům. Čtenář, který je zvyklý dodržovat principy OOP, nyní asi trochu pociťuje odpor k nějakému porušení zapouzdření, ale u operátorů << a >> se běžně používá. Musíme si uvědomit, že ne ke všem soukromým proměnným musí být přístup přes veřejné metody. řešením je označit funkci jako friend (přátelskou). Pro ukázku přetížím operátor << jako přátelskou funkci. Nejprve musím v definici mojí třídy uvést, že funkce << je přátelská. Vepište prosím do třídy MojeTřída na poslední řádek (před };) text: friend istream& operátor<<(ostream &os,MojeTrida &objekt);. Tedy deklaraci funkce s klíčovým slovem friend. Kromě funkcí mohou být "přátelské" také třídy. Přátelské funkce, nebo metody přátelských tříd mohou přistupovat k soukromým položkám třídy. Nyní stačí dopsat tělo funkce:

|  |  |
| --- | --- |
| |  | | --- | | ostream& operator<<(ostream &os, MojeTrida &objekt) {    return (os << objekt.A << objekt.B);    /\* Krátké že? \*/ } | |

A jednoduchá funkce main:

|  |  |
| --- | --- |
| |  | | --- | | int main(void) {    MojeTrida m;    cout << "Cekam objekt" << endl;    cin >> m;    cout << "Objekt byl:" << endl << m;    return 0; } | |

Asi Vás hned napadne, co se asi stane, jestliže místo čísla vložíte nějaký znak. Proudy mají tak zvané stavové bity (příznaky), které nastavují podle toho, zda došlo k nějaké chybě, nebo ne. Existují metody, kterými můžete zjistit, který bit je nastaven, tedy jestli se stalo něco špatného, případně co. Nejjednodušší způsob ale je využít přetíženého operátoru přetypování streamu na void\*. Je-li například (void\*) cin == NULL došlo k chybě. Samozřejmě operátor přetypování na void\* je k dispozici pro všechny proudy. Je-li proud v chybovém stavu, není schopen data posílat, nebo přijímat. Je nutné jej "resetovat" metodou clear(int = 0), která vyčistí chybové stavy. Pro příklad pozměníme funkci main:

|  |  |
| --- | --- |
| |  | | --- | | int main(void) {    MojeTrida m;    cout << "Cekam objekt" << endl;    cin >> m;    if (!cin)    {      cerr << "Objekt nenacten! " << endl;    }    else    {      cout << "Objekt byl:" << endl << m;    }    return 0; } | |

Nyní se podívejme na třídy ifstream (resp. ofstream), které dědí z istream (resp. ostream). Jedná se o datové proudy, jejichž zdrojem (resp. cílem) je soubor. Velkou výhodou je, že na ifstream lze volat operátor <<, protože ifstream je (dědí z) istream. Konstruktor těchto tříd může mít mnoho parametrů. Lze jej volat s 1 parametrem - jménem souboru. Otevřeme-li takto soubor a budeme-li používat operátory << a >>, bude se jednat o textový soubor a o tak zvaný formátovaný vstup a výstup. O jiných možnostech si povíme v dalším článku. Uvedu jednoduchý příklad. Opět změníme funkci main v předchozím příkladu:

|  |  |
| --- | --- |
| |  | | --- | | int main(void) {    MojeTrida m;    cout << "Cekam objekt" << endl;    cin >> m;    if (!cin)    {      cerr << "Objekt nenacten! " << endl;    }    else    {      ofstream f("data.txt");      if (!f)       {        cerr << "Nelze otevrit data.txt" << endl;        return -1;      }      f << "Objekt byl:" << endl << m;      cout << "Objekt byl:" << endl << m;    }    return 0; } | |

Třída ofstream je deklarována v hlavičkovém souboru fstream. Nezapomeňte proto na začátek našeho příkladu připsat: #include<fstream>

Soubor se uzavře buď při volání destruktoru streamu, nebo pomocí metody close().  
Tolik tedy dnes k datovým proudům a k přetěžování operátorů << a >>. Příště ukážu několik málo dalších příkladů a také se podíváme na neformátovaný přístup k datovým proudům - tedy na práci s binárními soubory. V následujícím článku potom k paměťovým proudům, čímž téma datových proudů dokončíme.

## [Neformátovaný vstup a výstup v C++](http://www.builder.cz/rubriky/c/c--/neformatovany-vstup-a-vystup-v-c--155754cz)

**2. května 2001, 00.00 | V tomto článku nejprve předvedu dva slíbené příklady na formátovaný vstup a výstup. Poté se podíváme na metody datových proudů read a write pro práci s bloky dat. Ukážeme si také, jak pracovat s binárními soubory pomocí proudů v C++.**

Neformátovaný vstup a výstup v C++

Ještě než se dostaneme k samotnému tématu článku, rád bych se vrátil ke článku minulému. Na konci jsem slíbil několik málo krátkých příkladů na formátovaný vstup a výstup. Jako první uvedu bez velkého komentáře program, který vše, co přečte ze vstupu, zapíše na výstup a do souboru:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> #include <fstream> int main(int argc, char \*argv[]) {    ofstream log("log.txt");    int znak;    znak = cin.get();    while (!cin.eof() && cout && log)    {       log << (char)znak;       cout << (char)znak;       znak = cin.get();    } } | |

Zde používám metodu get . Metoda get bez parametrů přečte jeden znak.

Druhý program přečte obsah parametrem zadaného souboru a zapíše jeho obsah v šestnáctkové soustavě.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> #include <fstream> #include <iomanip>  int main(int argc, char \*argv[]) {    if (argc != 3)    {       cerr << "Špatné parametry" << endl;       return -1;    }    ifstream in(argv[1]);    ofstream out(argv[2]);    out.setf(ios::hex | ios::uppercase);    char znak;    int pocet = 0;    in >> znak;    while (in && out)    {       if (pocet++ % 16 == 0)       {          out << endl << setfill('0') << setw(8) << (pocet-1) << ": " ;       }       out << (int)znak << " ";       in >> znak;    }    out << endl;    return 0; } | |

Tolik tedy k formátovanému vstupu a výstupu. Vše lze výhodně použít při práci s textovými soubory, nebo při tvorbě filtrů v Unixových OS. Stejně tak lze tyto možnosti použít při tvorbě CGI programů. K formátovanému vstupu a výstupu se ještě vrátíme u paměťových proudů. Nyní se ale konečně podívejme na neformátovaný vstup a výstup.

Neformátovaný vstup a výstup se používá hlavně při práci s binárními soubory. Vstup se provádí metodou read . Její první parametr je char\* a značí ukazatel na blok paměti, do kterého bude zapsán přečtený obsah souboru. Druhý parametr je int . Udává maximální počet přečtených bytů. Dojde-li k chybě (I v tomto případě je chybou také konec souboru.), můžeme skutečný počet přečtených bytů zjistit metodou int gcount(). K zapsání bloku dat slouží metoda write . Tato metoda má stejné parametry jako metoda read . Samozřejmě rozdíl je v tom, že metoda write blok paměti zapíše, ne přečte. Vše ukážu na následujícím příkladě. Vytvoříme něco jako příkaz copy, který nakonec vypíše, kolik bytů zkopíroval.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> #include <fstream> int main(int argc, char \*argv[]) {    if (argc != 3)    {       cerr << "Syntaxe: " << argv[0] << " zdroj cil" << endl;       return -1;    }    ifstream in(argv[1],ios::binary);    ofstream out(argv[2],ios::binary);    char buffer[1000];    int pocet = 0;    while (in && out)    {       in.read(buffer,1000);       out.write(buffer,in.gcount());       pocet++;    }    --pocet \*= 1000;    pocet += in.gcount();    cout << "Počet zkopírovaných bytů: " << pocet << endl;    return 0; } | |

Chtěl bych upozornit na konstruktory tříd ifstream a ofstream . Jako jejich druhý parametr je tak zvaný režim souboru. Mód ios::binary znamená, že soubor je otevřen v binárním módu. Existuje mnoho souborových módů. Například: ios::app - pokud soubor existoval, bude se připisovat na jeho konec, ios::ate - nastaví "ukazatel" souboru (Spíš bych měl říci pozici, nebo zapisovací hlavu.) na konec atd... Také bych chtěl upozornit na druhý parametr metody write . Zapíšu jen tolik, kolik jsem přečetl. Jinak soubory čtu i zapisuji po 1000 bytových blocích.

Tolik tedy k neformátovanému vstupu a výstupu. Jeho využití je hlavně při práci s binárními soubory. Příště se podíváme na paměťové proudy, které umožní formátovat řetězec v operační paměti.

Paměťové proudy v C++

**7. května 2001, 00.00 | V tomto článku si něco povíme o paměťových datových proudech. Paměťové datové proudy slouží hlavně k formátovaní řetězců v paměti. Práce s nimi je obdobou funkcí sprintf a sscanf z jazyka C.**

V tomto článku se podíváme na takzvané "paměťové proudy". Ve jednom z mých předchozích článků jménem ["Vstupní a výstupní operace pomocí datových proudů v C++ "](http://www.builder.cz/rubriky/c/c--/clanek-155739-vstupni-a-vystupni-operace-pomoci-datovych-proudu-v-c-/) jsem popsal datový proud jako proud dat od zdroje k cíli. Paměťový proud je proud, jehož cílem, nebo zdrojem je paměť (Nebo přesněji řetězec v paměti.). Paměťové proudy slouží k formátovaní řetězců v paměti počítače. Práce s paměťovými proudy v C++ je vlastně obdoba práce s funkcemi sprintf a sscanf v jazyce C.

Paměťový proud charakterizuje třída strstream. Třída strstream je potomkem tříd iostream a strstreambase. Pro nás je nyní důležitý hlavně fakt, že strstream je potomkem iostream, tedy že vlastně "strstream je iostream". Tedy všude, kde je očekáván vstupně výstupní proud lze použít paměťový proud. Z toho vyplývá, že pro paměťový proud máme k dispozici přetížené operátory << a >> pro všechny datové typy, pro které jsou tyto operátory přetížené u třídy iostream. Nyní si ukážeme jednoduchý příklad jak pracovat s paměťovým proudem.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> #include <strstream> int main(int argc, char \*argv[]) {    char buffer[1000]; /\* Paměť, nad kterou bude vše probíhat. \*/    strstream proud(buffer,1000,ios::out);    int cislo = 50;    proud << "Ahoj svete " << endl << cislo << ends;    /\* Nyní v poli buffer je řetězec "Ahoj světe \n50" \*/    cout << buffer << buffer << endl;    return 0; } | |

Nejprve jsem vytvořil paměťový blok o velikosti 1000 bytů. Jedná se vlastně o řetězec. Poté jsem na dalším řádku vytvořil paměťový proud jménem proud. Jako parametry jsem předal char \* jako adresu paměti, na kterou je proud "napojen". Tento řetězec bude cílem proudu. Dalším parametrem je velikost paměti, nad kterou bude proud pracovat. Posledním parametrem je mód otevření proudu, v tomto případě bude proud výstupní. O módech otevření jsem psal již ve svém minulém článku. Dalším možným je například ios::in - poté proud bude vstupní. Lze použít třídy istrstream, resp. ostrstream. Poté nemusíme zadávat zda je proud vstupní, nebo výstupní. (Tedy zda řetězec je zdroj, nebo cíl.) Použijeme-li zde mód ios::app a ios::ate, budou se data připisovat na konec řetězce. Konec řetězce je znak '\0', nikoliv konec bufferu. Ještě bych chtěl upozornit na fakt, že řetězec se neukončí manipulátorem endl, ale manipulátorem ends. Snažil jsem se to i demonstrovat. Manipulátor ends vloží jako další znak '\0', což je konec řetězce. Při používání paměťových proudů se nemusíme bát zapisování mimo vyznačený buffer. V konstruktoru proudu jsem zapsal velikost paměti 1000 bytů. Budu-li do proudu zapisovat data delší než označených 1000 znaků, proud se dostane do "chybového" stavu (viz můj článek ["Přetěžování operátorů << a >> pro datové proudy v C++"](http://www.builder.cz/rubriky/c/c--/clanek-155741-pretezovani-operatoru-lt-lt-a-gt-gt-pro-datove-proudy-v-c-/)) a další data nepřijme. Můžete udělat jednoduchý experiment a přepsat v předchozím příkladu obě čísla 1000 třeba na 5. Nemusíte se bát zapsání dat za rozsah pole. Možná ale narazíte na jiný problém. Řetězec končí znakem '\0', což buffer, na který byl napojen takový proud s chybou, končit nebude. Proto se nejspíš stane, že při výstupu uvidíte kromě prvních 5 znaků také nějaký "odpad", dokud nebude v paměti nalezen znak '\0'.

V dalším příkladu předvedu jak zapisovat za konec řetězce.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> #include <strstream> int main(int argc, char \*argv[]) {    char buffer[40]; /\* Paměť, nad kterou bude vše probíhat. \*/    ostrstream proud(buffer,40,ios::ate | ios::app);    cout << "Něco napište" << endl;    cin.get(buffer,40);    proud << "Můj přídavek" << ends;    if (proud)    {       cout << buffer << endl;    }    else    {       cerr << "Bohužel se to tam nevešlo " << endl;    }    return 0; } | |

Před vypsáním jsem zjistil, jestli je proud bez chyby.

Paměťové proudy jsou skvělým nástrojem na zpracovávání řetězců ještě před jejich zapsáním na výstup, nebo do souboru. Dá se také pomocí nic převádět řetězec na číslo a naopak. Stejně tak se nemusejí používat jen pro práci s řetězci. Lze pomocí metod read, nebo writepoužít paměťový proud k přenosu bloku dat. Myslím si, že tohle ale není zrovna nejlepší nápad a v tomto případě by bylo lepší použít jiný rychlejší způsob.

Tolik tedy k paměťovým proudům. Dokončili jsme téma datové proudy a příště se podíváme na tak zvané prostory jmen (namespace).

## [Prostory jmen v C++](http://www.builder.cz/rubriky/c/c--/prostory-jmen-v-c--155766cz)

**17. května 2001, 00.00 | V tomto článku si povíme něco o prostorech jmen (namespace). Prostory jsou prevencí konfliktů jmen (identifikátorů). Podíváme se kde výhodně použít prostory jmen. Vysvětlíme si význam klíčových slov using a namespace.**

Prostor jmen je oblast platnosti identifikátorů. Představme si situaci, kdy budeme chtít mít v jednom programu stejné identifikátory (názvy tříd, proměnných, metod, funkcí atd...). Nelze mít v jednom programu dvě různé třídy se stejným názvem. Překladači by se nelíbil konflikt jmen. Přesto taková situace může lehce nastat. Může třeba více programátorů, aniž by se domluvili na názvech identifikátorů, pracovat na jednom programu. To by byla zrovna velmi trapná situace, kdyby nebyli schopni se domluvit na názvech identifikátorů v programu. Může ale velmi často nastat situace, kdy je potřeba pracovat s více knihovnami, jejichž autoři se nejspíš nikdy neviděli, a nejspíše o sobě ani nevědí. Také může nastat situace, kdy píšeme svůj program, ve kterém používáme nějakou knihovnu. I zde může nastat konflikt našeho identifikátoru s jiným identifikátorem. Právě tyto konflikty jmen řeší tak zvané jmenné prostory.

Jmenný prostor je oblast, ve které nesmí dojít ke konfliktu jmen. Jméno identifikátoru v prostoru jmen ale může být v konfliktu s jakýmkoliv jménem identifikátoru mimo tento prostor jmen. K definici prostoru jmen slouží klíčové slovo namespace. Za namespace následuje jméno prostoru jmen a v závorkách {} se nachází ona oblast platnosti identifikátorů. Jako příklad nyní vytvořím prostor jmen "RadimuvProstor" a vněm deklaruji a definuji třídu a funkci:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream> namespace RadimuvProstor {    int secti(int a, int b);    class Trida    {       private:         int Atribut;       public:         void metoda();    };     void Trida::metoda()    {       cout << "Ahoj" << endl;    } }  int RadimuvProstor::secti(int a, int b) {    return a + b; } | |

V mém prostoru jmen jsem deklaroval funkci sečti a třídu Třída. Dále jsem definoval tělo metody metoda. Tělo funkce sečti jsem sice definoval "mimo" závorky, ale přesto patří do definovaného prostoru jmen. Vně závorky vymezující prostor jmen přistupujeme k identifikátorům z tohoto prostoru jmen jako by k prvkům třídy. Nyní zkusme definovat úplně jinou funkci sečti, která je mimo tento prostor jmen. K předchozímu zdrojovému textu dopište:

|  |  |
| --- | --- |
| |  | | --- | | int secti(int a, int b) {    return 2 \* a + 2 \* b; }  int main(void) {    cout << secti(2,3) << endl;    cout << RadimuvProstor::secti(2,3) << endl;    return 0; } | |

V tomto případě nedojde ke konfliktům jmen. Je to proto, že vlastně neexistují dvě funkce sečti, ale funkce sečti(int,int) a RadimuvProstor::sečti(int,int). Tím že jsem funkci zařadil do prostoru jmen, jsem vlastně rozšířil její název. Byla by veliká náhoda, kdyby se našly dva prostory jmen stejného názvu. U názvů tříd, funkcí, globálních proměnných atd... je to ale dost možné. Název prostoru jmen by měl být vybrán tak, aby byl jednoznačný. Identifikátory uvnitř jednoho prostoru jmen se nepoužívají s názvem tohoto prostoru jmen.

Implicitní prostor jmen

Dalo by se říci, že každý identifikátor je v nějakém prostoru jmen. Jestliže prostor jmen neuvedeme (Jako já u druhé funkce sečti, u funkce main a také ve všech svých příkladech z předchozích článků.), je tento identifikátor v implicitním prostoru jmen, někdy se také nepřesně říká "globální prostor jmen", který není pojmenován. Pro přístup k němu se před operátorem "čtyř-tečka" neudává jméno. V naší funkci main můžu tedy místo cout << secti(2,3) << endl; napsat cout << ::secti(2,3) << endl;. V tomto případě je tato čtyř-tečka samozřejmě zbytečná, ale mohla by se hodit, kdybychom někde uvnitř mého prostoru jmen chtěli vyvolat "globální" funkci sečti. Například přepíšu metodu metoda, ve které zavolám obě funkce sečti:

|  |  |
| --- | --- |
| |  | | --- | | void Trida::metoda()    {       cout << ::secti(1,1); << endl;/\* "globální" sečti \*/       cout << secti(1,1); << endl;/\* sečti z tohoto prostoru jmen\*/    } | |

Prostor jmen "std"

Podle ANSI C++ existuje standardní prostor jmen std. V tomto prostoru jmen je definováno mnoho identifikátorů ze standardní knihovny jazyka C++. V std jsou také, jak jsem byl správně ve svých předchozích článcích čtenáři upozorněn, objekty cin, cerr, cout a mnoho dalších. Konkrétně u těchto 3 objektů se dnes překladače chovají dost tolerantně, a přeloží je i když jsou používány jako by v implicitním prostoru jmen. Postupem času se překladače budou blížit normě, a takové nepřesnosti nebudou tolerovat. Je tedy dobré tyto objekty identifikovat jako std::cout, std::cin, std::cout, nebo si pomoci klíčovým slovem using.

Klíčové slovo using

Jak už jsem se zmínil, prostory jmen jsou proto, aby nedošlo ke konfliktům jmen. Pomáhají programátorovi v uspořádání zdrojových textů, ale používání prostoru jmen by se mohlo zdát jako "psaní něčeho navíc". Aby programátor nemusel vždy psát jméno prostoru společně i se jménem, které chce napsat, existuje klíčové slovo using. Napíši-li using namespace a jméno prostoru, bude v následujícím zdrojovém textu přístup k identifikátorům tohoto prostoru jako by jsem byl "uvnitř" tohoto prostoru. Tedy místo:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream> int main(void) {    std::cout << "Ahoj svete" << endl;    return 0; } | |

Lze napsat:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream> using namespace std; int main(void) {    cout << "Ahoj svete" << endl;    return 0; } | |

Musíme mít ale natolik nový překladač, aby měl objekt cout v prostoru jmen std.

Závěrem k prostorům jmen bych Vám chtěl doporučit prostory jmen používat. Narazíte-li na něco, co v prostoru jmen již je, nic jiného Vám ani nezbude. Je dobré také své identifikátory do prostoru jmen ukládat. Nemůže-li dojít ke konfliktu jmen, můžete použít slovo using, a v dalším zdrojovém textu si ani nevšimnete, že prostory jmen používáte. Naopak nebudete-li používat prostory jmen a ke konfliktu dojde, už Vám nezbude nic jiného, než přejmenovávat identifikátory. Já ve svých následujících článcích prostory jmen používat nebudu čistě z praktického hlediska. Moje příklady jsou natolik krátké a jednoduché, že prostory jmen by je udělaly jen méně přehledné.

Ohledně používání prostorů jmen bych chtěl ještě upozornit na jednu častou nepřesnost. Syntakticky je prostor jmen velice podobný třídě. Jedná se ale o něco úplně jiného. Třída je abstrakcí nějakých objektů, které mají podobné vlastnosti. U třídy se počítá s tím, že bude mít instance. Pokud možno, každá třída by měla být navržená tak, aby mohla mít více instancí. Naopak prostor jmen žádnou instanci nemá, ani mít nemůže. Jedná se jen o oddělení globálních proměnných, funkcí, uživatelem definovaných datových typů (tedy i tříd), které spolu nějak souvisejí od "zbytku" programu. Mnoho programátorů chybně prostory jmen nepoužívá, a místo prostoru jmen použije třídu.

Tolik tedy k prostorům jmen v C++. Příště se podíváme na řetězce v C++. V C++ existuje třída string. S instancemi této třídy se pracuje rozhodně pohodlněji, než s polem znaků jak tomu bylo u jazyka C. O tom ale až příště.

## [Řetězce v C++](http://www.builder.cz/rubriky/c/c--/retezce-v-c--155767cz)

**24. května 2001, 00.00 | V tomto článku si ukážeme jak pracovat s řetězci v C++. V C++ existuje třída string, se kterou se pracuje mnohem pohodlněji, než s polem znaků, jako v jazyce C.**

V tomto článku si ukážeme jak pracovat s řetězci v C++. V jazyce C je řetězec pole znaků a jako s polem se s ním musí pracovat. Některé pomocné funkce pro práci s řetězci jsou v hlavičkovém souboru string.h . V C++ je situace jiná. Standardní knihovna C++ obsahuje parametrizovanou třídu (šablonu) string. Něco o šablonách si povíme později.  
  
Práce s šablonou string je velice snadná. O šablonách nemusíme vědět nic, abychom mohli se stringem pracovat jako s jakýmkoliv jiným datovým typem (třídou). Práce s těmito řetězci je velice podobná práci s řetězci v jiných vyšších programovacích jazycích. String má mimo jiné přetíženy operátory == a != pro porovnávání řetězců, operátor = pro přirovnání, operátory +, += pro spojení řetězců. U tohoto řetězce se nemusíme starat o jeho velikost. Řetězec se automaticky zvětší vždy, kdy je to potřeba. Pro string jsou také přetíženy operátory << a >>pro vstup a výstup z (do) datových proudů.   
Třída string je deklarována v hlavičkovém souboru string. Ještě než ukážu první příklad chtěl bych trochu podrobněji rozvést, co vlastně musím pomocí include vložit. V souboru string.hjsou deklarace funkcí pro práci s polem znaků z jazyka C. Jsem přesvědčen, že budete-li používat třídu string, nebudete tyto funkce potřebovat. V souboru string je deklarace šablony string. Chcete-li pracovat i s šablonou string, i s funkcemi pro pole znaků, musíte vložit oba hlavičkové soubory. Nyní už jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include<string> #include<iostream> using namespace std; int main(void) {    string a = "Ahoj", b("svete"); /\* Inicializace pomocí operátoru = a pomocí konstruktoru. Jak jsem již dříve doporučoval, druhá možnost je lepší. \*/    string c,d;    cout << a << " " << c << endl;    c = a + string(" ") + c; /\* Operator + je přetížen pouze pro dva stringy. V uvozovkách je ale pole znaků. Proto jsem musel použít konstruktor. \*/    cout << c << endl;    cout << "Neco napis" << endl;    cin >> d;    cout << d; << endl;    cout << "3. znak:" << c[2] << " z poctu:" << c.length() << endl;    const char \*stary = c.c\_str();    cout << stary << endl;    /\* Nyní ukážu některé operátory \*/    if (a == b)    {       cout << "a == b" << endl;    }    if (a == string("Ahoj"))    {       cout << "a je Ahoj" << endl;       a += string(" ctenari");       cout << a << endl;    }    return 0; } | |

Metoda length vrací velikost řetězce. Metoda c\_str() vrací konstantní pole znaků. Tato metoda slouží pro kompatibilitu se starými řetězci z jazyka C. Nemusíte se tedy bát používat string. Budete-li někde potřebovat pole znaků, pomůžete si "konverzní" metodou c\_str().

Závěrem bych ještě chtěl jen dodat, že některé starší překladače nemusejí string znát, u jiných překladačů sice string je, ale jmenuje se jinak. Například Borland C++ Builder 1 má tuto třídu pod názvem AnsiString.

Tolik tedy k řetězcům v C++. Příště se podíváme na vyjímky v C++. Na klíčové slova catch, try, throw a vše, co s vyjímkami souvisí.

## [Výjimky v C++](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c--155798cz)

**14. června 2001, 00.00 | V tomto článku si povíme něco o výjimkách (exceptions) v C++. Vysvětlíme si mechanizmus výjimek, jejich vyvolání a ošetření. Podíváme se na význam klíčových slov try, catch, throw.**

Výjimky v C++

Dnes si povíme něco o výjimkách a o ošetření výjimek. Pod pojmem výjimka se rozumí nějaká výjimečná situace, která nastane ve volané metodě, nebo funkci. V jazyce C i C++ se často používá návratových hodnot funkcí, které vracejí úspěšnost provádění nějaké operace. Nejčastěji -1 se vrací v případě chyby, jinak se vrací 0. Chceme-li v posloupnosti několika řádků zdrojovém textu takto ošetřit všechny možné chyby, vznikne nám velice nepřehledný program plný vnořených příkazů if. Jinou možnost nám nabízí mechanizmus výjimek. Princip spočívá v tom, že označíme posloupnost příkazů do zvláštního bloku, říká se mu často hlídaný blok, ve kterém neošetřujeme žádné chyby. Právě v tomto bloku může vzniknout výjimka. Za tímto blokem označíme postupně jaké výjimky mohly v hlídaném bloku nastat, a jak je ošetřit. V C++ může být výjimkou proměnná jakéhokoliv primitivního datového typu, nebo instance jakékoliv třídy. Výjimka by v sobě měla nést nějakou informaci o situaci, která nastala a proč byla vyvolána.

Syntaxe výjimek v C++

Výjimka se vyvolá (vyvrhne) pomocí klíčového slova throw. Hlídaný blok se značí klíčovým slovem try. K odchycení vyvolaných výjimek slouží klíčové slovo catch. Vše uvedu na jednoduchém příkladu.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <string>  using namespace std;  class Vyjimka  { /\* Třída výjimek \*/  private:  string Duvod;  public:  void nastav(string d) { Duvod = d; }  string dej() { return Duvod; }  };  ostream &operator<< (ostream &os, Vyjimka &v)  {  return os << v.dej() << endl;  }    class Zlomek  {  private:  int C,J;  public:  void nastavCitatel(int c) { C = c;}  void nastavJmenovatel(int j) {J = j;}  double vydel() throw (Vyjimka);  /\*Z metody vyděl může být vyvržená výjimka - instance třídy Vyjimka\*/  };  double Zlomek::vydel() throw (Vyjimka)  { // začátek bloku 2  int \*i = new int;  if (J == 0)  { //začátek bloku 1  string s("Nejde");  Vyjimka v;  v.nastav(s);  throw v;  } // konec bloku 1  delete i;  return ((double)C / J);  } // konec bloku 2  int main(void)  {  Zlomek z1,z2;    z1.nastavCitatel(10);  z2.nastavCitatel(5);  for(int i = 5; i > -5; i--)  {  z1.nastavJmenovatel(i);  z2.nastavJmenovatel(i);  try  { /\* Zkusim: \*/  cout << "10 / " << i << " = " << z1.vydel() << endl;  cout << "5 / " << i << " = " << z2.vydel() << endl;  }  catch (Vyjimka v) // Zde je odchycení výjimky třídy Vyjimka  {  cout << v << endl;  }  /\*  catch (Jina\_vyjimka j)  {  .....  }  \*/  }  return 0;  } | |

Podívejme se podrobněji na tento příklad. Nejprve jsme vytvořili nějakou třídu s názvem Výjimka. Dále třídu Zlomek. V deklaraci jedné metody se zde objevilo další klíčové slovo throw. Za slovem throw následuje seznam typů výjimek, které mohou být z dané metody (nebo i funkce) vyvrženy. V případě naší metody vyděl se jedná o výjimky třídy Výjimka. Nedoporučuji používat jako výjimky primitivní datové typy, i když syntaxe jazyka do umožňuje. V případě, že atribut J třídy zlomek bude v momentě zavolání metody vyděl roven nule, dojde k vyvolání výjimky. V tomto případě nejprve vytvořím instanci třídy Výjimka, a poté ji příkazem throw vyvrhnu. Při vyvržení výjimky dojde k okamžitému opuštění aktuálního bloku daného závorkami { }. V našem příkladě jsem jej poznámkami označil jako blok 1. Výjimka byla z tohoto bloku vyvržena. Při opuštění tohoto bloku dojde k zavolání destruktorů všech lokálních instancí tak, jako by se jednalo o korektní opuštění bloku. V našem případě s a v. Nemusíte se obávat, že byla zlikvidována instance v, kterou bude v budoucnu používat. Budete totiž pracovat s její kopií. Po opuštění bloku se výjimka šíří dále. Dojde k vyvržení výjimky z "vnějšího" bloku, v naše případě označeného jako blok 2. V tomto bloku budou opět korektně zlikvidovány všechny lokální instance. V našem případě bude zlikvidován ukazatel, ale nedojde k uvolnění paměti, na kterou ukazuje. Později ukážu, jak tento problém řešit.  
Takto vše pokračuje, dokud nebude opuštěn hlídaný blok označený klíčovým slovem try. Při vyvržení výjimky z hlídaného bloku se zjistí, jestli za hlídaným blokem existuje pro tento typ výjimky odchycení catch. Jestliže ne, výjimka je vyvržená dále z aktuálního bloku. Jestliže ano, dojde k vykonání tohoto bloku označeného catch. Po odchycení a ošetření výjimky program normálně pokračuje příkazy za blokem catch. Jak jsem se již zmínil, při vyvolání výjimky dojde k likvidaci všech lokálních proměnných. Je-li ale lokální proměnnou ukazatel, nebo reference, nedojde k uvolnění dat, na které "ukazují". Do našeho příkladu jsem úmyslně vložil do metody vyděl ukazatel na int. Při vyvolání výjimky se řádek delete neprovede. Paměť na kterou ukazatel ukazuje zůstane neuvolněná. Bylo by vhodné vyvrženou výjimku ošetřit na více místech. (V mém jednoduchém příkladě bych mohl jednoduše před slovo throw napsat delete i, já ale chci ukázat jak ošetřit výjimku na více místech.) Opravme metodu vydělnásledovně:

|  |  |
| --- | --- |
| |  | | --- | | double Zlomek::vydel() throw (Vyjimka)  {  int \*i = new int;  try  {  if (J == 0)  { //začátek bloku 1  string s("Nejde");  Vyjimka v;  v.nastav(s);  throw v;  } // konec bloku 1  delete i;  }  catch (Vyjimka v)  { /\* Ošetřím, co můžu \*/  delete i;  throw; /\* V tomto případě má stejný význam jako throw v; \*/  }  return ((double)C / J);  } | |

Výjimku jsem odchytil ještě v metodě vyděl. Uvolnil jsem paměť, na kterou ukazuje ukazatel i, a poté jsem výjimku opět vyvrhl. Jsme-li v bloku catch a chceme-li odchycenou výjimku poslat dále, nemusíme uvádět její název.

Podívejme se ještě podrobněji na deklarace metod, ze kterých může být vyvržena výjimka. Metoda vyděl je deklarována: double vydel() throw (Vyjimka);. Znamená to, že z metody vydělmůže být vyvržena výjimka třídy Výjimka. Je-li více typů výjimek, které mohou být vyvrženy, oddělí se čárkou. Například double vydel() throw (Vyjimka,Jina1,Jina2); Není-li v deklaraci metody uvedeno klíčové slovo throw, znamená to, že z metody může být vyvržena JAKÁKOLIV výjimka. Pro ty, kteří znají Javu to může být trochu matoucí, protože v Javě je to přesně naopak (tedy žádná). Chceme-li v C++ deklarovat metodu, z níž nemůže být vyvržena výjimka, za deklaraci připíšeme throw (). Ještě jen zbývá dodat, že seznam výjimek, které mohou být vyvrženy je součástí názvu metody, nebo funkce. Musí tedy být uveden i v deklaraci, i v definici.

Pro dnešek by to o výjimkách mohlo stačit. Všem, kterým mechanizmus výjimek není jasný doporučuji můj příklad podrobně projít v debuggeru. Příště dokončíme téma výjimek. Podíváme se na situace, kdy výjimky vytvářejí dědičnou hierarchii, jak odchytit jakoukoliv výjimku a co se stane není-li výjimka ošetřená a opustí tělo funkce main.

## [Výjimky v C++ - výjimky tvoří dědičnou hierarchii](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-vyjimky-tvori-dedicnou-hierarchii-155809cz)

**20. června 2001, 00.00 | Dnes se podíváme na situaci, kdy výjimky v C++ tvoří dědičnou hierrachii. Tedy na situaci, kdy všechny výjimky mají společnou nadtřídu. Také si vysvětlíme jak zachytit jakoukoliv výjímku bez ohledu na její typ.**

V minulém článku jsme si ukázali základní princip výjimek. Dnes se podíváme na situaci, kdy výjimky jsou odvozeny z jedné společné nadtřídy.

Odchycení jakékoliv výjimky

Nejprve se ale podívejme, jak odchytit jakoukoliv výjimku. Za klíčovým slovem catch místo typu a lokálního názvu výjimky uvedeme 3 tečky. Například:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  class V1 {};  class V2 {};  int main()  {  for(int i = 0; i < 5; i++)  {  try  {  if (i % 2 == 0)  {  V1 v;  throw v;  }  else  {  V2 v;  throw v;  }  }  catch(...)  /\* Odchytím jak výjimky třídy V1, tak i výjimky třídy V2.\*/  {  cout << "Chyceno" << endl;  }  }  } | |

Tento příklad nedává moc smysl, ale jasně ukazuje odchycení jakékoliv výjimky. Protože při vyvržení, i odchycení výjimky dochází často ke kopírování vyvržené výjimky na zásobník, je určitě efektivnější pracovat s referencemi, nebo ukazateli na výjimky. Já v dalším textu budu používat ukazatele.

Výjimky tvoří dědičnou hierarchii

Je velmi výhodné používat výjimky, které tvoří dědičnou hierarchii. Představme si třídu výjimek. Může být i abstraktní. Z této třídy dědí jiné výjimky, které jsou speciálním případem výjimky. Vytvořme si jako názornou ukázku jednoduchou hierarchii výjimek.

|  |  |
| --- | --- |
| |  | | --- | | #include <string>  #include <iostream>  using namespace std;  class Vyjimka  {  private:  string Text;  public:  Vyjimka(string s):Text(s){}  string dejText() { return Text; }  };  class DeleniNulou : public Vyjimka  {  private:  int Cislo;  public:  DeleniNulou(string s, int i):Vyjimka(s),Cislo(i) {}  int dejCislo() { return Cislo; }  };  class Preteceni : public Vyjimka  {  public:  Preteceni(string s):Vyjimka(s) {}  }; | |

Vytvořili jsme tři výjimky. Nyní si ukážeme jak odchytávat výjimky, které mají společného předka. Problém je v tom, že musíme nejprve odchytit nejkonkrétnější výjimky a postupně odchytávat jejich nadtřídy. Dokončím příklad:

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  for(short int p = 5; p >= 0; p--)  {  try  {  if (p == 0) /\* Vyvrhnu ukazatel \*/  throw new DeleniNulou("Deleni nulou",10);  cout << "10 / " << p << " = " << 10.0 / p << endl;  if ((short int ) (p \* 10000) < p)  throw new Preteceni("Pretekl short int");  if (p == 3)  throw new Vyjimka("Jen tak, pro ukazku");  }  catch (Preteceni \*v) /\* Odchytávám ukazatel \*/  { /\* Došlo k přetečení \*/  cout << "Při násobení: " << v->dejText() << endl;  delete v;  }  catch (DeleniNulou \*v)  { /\* Došlo k dělení nulou \*/  cout << v->dejText() << " nelze " << v->dejCislo() << "/0" << endl;  delete v;  }  catch (Vyjimka \*v) /\* Nadtřídu odchytím až jako poslední \*/  { /\* Nějaká výjimka \*/  cout << v->dejText() << endl;  delete v;  }  }  return 0;  } | |

V příkladu jsme vyvrhli vždy ukazatel na třídu výjimka, nebo na nějakého potomka této třídy. Ukazatel na tuto třídu jsme museli odchytit až jako poslední. Stačí si uvědomit, že ukazatel na potomka je zároveň také ukazatelem na předka. Znamená to, že kdyby jsme odchytávali ukazatel na třídu výjimka jako první, došlo by zde vždy k odchycení a další výjimky by nebyly odchyceny nikdy. Můžete se o tom sami přesvědčit, když příklad upravíte tak, že nejprve bude blok catch (Vyjimka \*v) a potom ostatní odchytávací bloky. Tento problém nastává samozřejmě vždy, nejen při používání ukazatelů jako v našem příkladě.

Všimněte si v příkladě řádků delete v;. Zde vždy zlikviduji výjimku, která byla vyvržená. Často se místo vyvrhnutí ukazatelů na výjimky a odchycení ukazatelů pracuje s referencemi. Uvedu jen jednoduchý příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  class V  {};  int main()  {  try  {  V v;  throw v;  }  catch (V &v)  {  cout << "Chyceno" << endl;  }  return 0;  } | |

Pro dnešek by to mohlo stačit, příště se podíváme na některé doplňující informace k výjimkám. Podíváme se například co se stane, jestliže výjimka opustí tělo funkce main, když selže operátor new a podobně.

[Výjimky v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-dokonceni-155810cz)

**3. července 2001, 00.00 | V tomto článku dokončíme téma výjimek v C++. Podíváme se podrobněji na situaci, kdy výjimka opustí tělo funkce main, na vyvržení nepovolené výjimky a na výjimku bad\_alloc.**

Výjimka opustí tělo funkce main

Nyní se podívejme, co se stane, jestliže výjimka není odchycená a opustí tělo funkce main. K tomuto účelu si napíšeme velice jednoduchý program:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  int main()  {  throw 1; /\* Výjimka typu int \*/  cout << "Nestane se" << endl;  return 0;  } | |

V minulém článku jsem nedoporučoval používat jako výjimky primitivní datové typy, ale teď jsem to sám udělal. Snad mi to odpustíte, o typ výjimky teď vůbec nejde. Můžeme tento program zkompilovat a spustit. Uvidíme, že program se ukončí s nějakou chybovou hláskou na stderr. Neošetřená výjimka je dost závažná chyba, proto takto "tvrdý" konec. Výjimka bude vyvržená tak, jak jsme si ukázali minule. Opustí-li výjimka tělo naší funkce main, dojde k zavolání funkce terminate. Tato funkce, není-li předepsáno jinak, vypíše onu chybovou hlášku a ukončí program zavoláním funkce abort. Funkce abort se postará o to, aby volajícímu procesu byla vrácena návratová hodnota 3. Takové chování nemusí být žádoucí. Můžeme například chtít vrátit jinou návratovou hodnotu, vypsat jinou, nebo žádnou hlášku (stderr u programů pod OS Windows není). Toho lze docílit pomocí funkce set\_terminate. Funkce má jako parametr ukazatel na funkci, kterou má vyvolat volání funkce terminate. Funkce terminate i set\_terminate jsou deklarovány v hlavičkovém souboru exception. Ve starších překladačích se tento soubor může jmenovat jinak. Například except.h Uveďme příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <exception>  #include <iostream>  using namespace std;  void mojeTerminate()  {  cerr <<"Bohužel, programátor nebyl schopen ošetřit výjimky."<<endl;  exit(-10); /\* Doporučuji vždy ukončit program. \*/  }  int main()  {  set\_terminate(mojeTerminate);  throw 1; /\* Výjimka typu int \*/  cout << "Nestane se" << endl;  return 0;  } | |

Takto je chování programu v případě, že výjimka opustí funkci main, dáno ANSI normou. Problém nastane u aplikací pro Windows, které žádný stderr nemají. Chováni funkce terminate je jiné, a nejspíše se pro každý překladač liší. Například Borland C++ Builder vytvoří dialogové okno s hláškou: "Abnormal program termination". Budete-li ale chtít experimentovat s výjimkami v GUI aplikaci, zjistíte, že všechny výjimky vyvržené z metod formulářů jsou ošetřeny nějakým mechanizmem, který je pro programátora "neviditelný". Chcete-li si zkusit v GUI aplikaci v BCPPB vyvrhnout výjimku z funkce WinMain (obdoba main), musíte výjimku vyvrhnout přímo v této funkci. Pokud možno mimo blok try. U GUI aplikací v OS Linux tento problém nenastává, protože v Unixových systémech mají i GUI aplikace svůj standardní chybový výstup.

Vyvržení nedeklarované výjimky

Je-li z těla funkce, nebo metody vyvržená výjimka, která není v seznamu výjimek, které mohou být z těla funkce, nebo metody vyvrženy, dojde k zavolání funkce unexpected. Tato funkce implicitně zavolá funkci terminate. Chování funkce unexpected lze změnit pomocí funkce set\_unexpected, která má jako svůj parametr ukazatel na funkci bez parametrů a vracející void. Použití funkce set\_unexpected je obdobné jako použití funkce set\_terminate. Funkce unexpected je deklarována v souboru exception. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <exception>  #include <iostream>  using namespace std;  void mojeUnexpected()  {  cerr << "Bohužel, programátor špatně pracuje s výjimkami" << endl;  exit(-10);  }  void f() throw()  {  throw 1;  }  int main()  {  set\_unexpected(mojeUnexpected);  f();  return 0;  } | |

Standardní výjimky

V standardní knihovně C++ se výjimky příliš nepoužívají. Informace o chybě se často předává jako předem určená návratová hodnota z funkce, nebo metody, nebo pomocí chybového stavu - viz třeba objekt cin, nebo jiný objekt třídy istream, se kterým nelze pracovat v případě, že přečte chybná data. Několik standardních výjimek v C++ ale přece jenom je. Budeme se k nim postupně dostávat v dalších článcích. Všechny tyto výjimky jsou děděny ze třídy exception. Jedna výjimka, o které by jsme už ale měli vědět je výjimka typu bad\_alloc, nebo nějaký její potomek, kterou vyvrhne operátor new v případě, že selže. V některých starších překladačích, které neodpovídají normě, se setkáme s výjimkou typu xalloc. Výjimka bad\_alloc je definována v prostoru jmen std. Operátor new v případě selhání může buď vrátit NULL, nebo vyvrhnout tuto výjimku. Operátor new může selhat v situaci, kdy nelze alokovat požadovanou paměť. Úplné ošetření činnosti operátoru new je následující:

|  |  |
| --- | --- |
| |  | | --- | | {  int \*pointer;  try  {  if ((pointer = new int[3]) == NULL)  { /\* Přece jenom - pro jistotu \*/  cout << "Vrátil NULL" << endl;  }  }  catch (std::bad\_alloc &b)  {  cout << "Vyhodil výjimku" << endl;  }  } | |

Jak se chová new, když selže, se můžete přesvědčit například tak, že v tomto příkladě místo čísla 3 (velikost pole) dáte hodně velké číslo. Chcete-li možnost vyvržení výjimky u operátoru new potlačit, lze to pomoci parametru operátoru new, který je nothrow:

|  |  |
| --- | --- |
| |  | | --- | | {  int \*i;  i = new(nothrow) int[10]; /\* Určitě new nevyvrhne výjimku. \*/  } | |

Nevyvrhne-li výjimku operátor new, neznamená to samozřejmě, že nějaká nemůže být vyvržena z konstruktoru, který bude zavolán. Výjimka bad\_alloc, i parametr nothrow (Je to vlastně prázdná struktura.) se vztahují pouze na selhání alokace paměti.

Závěrem k výjimkám

Výjimky jsou určitě skvělá a pohodlná věc. Vše ale něco stojí. Při používání výjimek platíme tu největší daň za používání objektově orientovanému programování. Výsledný program používající výjimky bude jednak o mnoho větší a také mnohem pomalejší. Bude pomalejší dokonce i tehdy, nebude-li žádná výjimka vyvržená. Stačí pouze, že existuje blok try. Uvědomme si, co vše se musí v hlídacím bloku kontrolovat. To vše stojí čas a také jsou k tomu potřebné instrukce navíc. Záleží-li Vám opravdu na rychlosti programu, potom je používání výjimek na pováženou. Naopak ale například v programovacím jazyce Java se to výjimkami jen "hemží" a s rychlostí si nikdo nedělá starosti. Stejně tak podíváme-li se například na funkci WinMain v BCPPB, zjistíme, že celé její tělo je vlastně v hlídacím bloku. Používání výjimek je sice drahé, ale pro programátora velmi užitečné.

To by k výjimkám bylo asi tak vše. Příště se podíváme na dynamickou identifikaci typů.

## [Dynamická identifikace typů v C++](http://www.builder.cz/rubriky/c/c--/dynamicka-identifikace-typu-v-c--155828cz)

**13. července 2001, 00.00 | Dnes se podíváme na dynamickou identifikaci typů. Podívame se jak v C++ rozpoznat typ objektu, nebo proměnné. Seznámíme se s třídou type\_info, s operátorem typeid. Vysvětlíme si rozdíl mezi dynamickým a statickým rozpoznáváním typů.**

Dynamická identifikace typů v C++

Pod pojmem "Dynamická identifikace typů" rozumíme zjišťování typů proměnných, nebo objektů v době běhu programu. Identifikaci typů zajišťuje operátor typeid. Než se ale budeme zabývat tímto operátorem, podívejme se nejprve na třídu type\_info. Instance této třídy slouží k uchování informace o typech.

Třída type\_info

Třída je deklarována v hlavičkovém souboru typeinfo. Deklarace se nachází v prostoru jmen std. Instance této třídy v sobě mají informace o typech v době běhu programu. Třída má dvě veřejné metody: const char \*name() const a bool before(const type\_info& arg) const. Obě metody jsou konstantní, tedy nijak nemění vnitřní stav (hodnoty atributů) objektu. Metoda name vrací konstantní řetězec udávající název typu. Setkal jsem se s takovými překladači, které vytvářely programy, ve kterých metoda name sice vracela název typu, ale nikoliv "pěkně" čitelný pro programátora. Jednalo se asi o nějakou vnitřní reprezentaci typu, se kterou nejspíše pracuje linker. Dnes u "solidních" překladačů by se to stát snad už nemělo. Metoda before zjistí, zda parametr má, či nemá být umístěn před daným objektem při řazení typů. Jedná se vlastně o obdobu operátoru <. Vedle těchto metod jsou pro třídu type\_infopřetíženy operátory == a !=. Všechny veřejné metody jsou konstantní a operátor = i kopírovací konstruktor nelze použít. Oba jsou deklarovány jako soukromé metody. Neexistuje způsob, jak by mohl programátor změnit (korektní cestou) obsah objektu.

Operátor typeid

Operátor typeid vrací konstantní instanci třídy type\_info. Jeho argumentem může být název typu, nebo výraz. V prvním případě bude vrácená instance třídy type\_info udávat zadaný typ, ve druhém případě bude udávat typ návratové hodnoty výrazu. Uveďme jednoduchý příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <typeinfo>  using namespace std;  class Trida  {  private:  int A,B;  public:  void nastav(int a, int b);  };    void Trida::nastav(int a, int b)  {  A = a;  B = b;  }  Trida \*funkce()  {  cout << "Volani" << endl;  return new Trida;  }  int main()  {  Trida objekt;  const char \*nazev = typeid(objekt).name();  /\* Jen reference \*/  const type\_info &t1 = typeid(objekt), &t2 = typeid(Trida);  cout << "Nazev typu objekt: " << nazev << endl;  cout << "char < int == " << typeid(char).before(typeid(int)) << endl;  if (t2 == t1) /\* Nebo if ( typeid(objekt) == typeid(Trida) ) \*/  {  cout << "OK" << endl;  }  cout << typeid(funkce).name() << endl; /\* Typ ukazatel na funkci \*/  cout << typeid(funkce()).name() << endl; /\* Návratová hodnota funkce \*/  cout << typeid(\*funkce()).name() << endl;  return 0;  } | |

V tomto jednoduchém programu jsem předvedl jak pracovat s operátorem typeid a s instancemi třídy type\_info. Zajímavé jsou především poslední tři výpisy. V prvním z nich zjišťuji typ identifikátoru funkce, což je ve skutečnosti ukazatel na funkci bez parametrů vracející ukazatel na třídu Třída. V předposledním výpisu zjišťuji návratovou hodnotu funkce. K zavolání funkce nedojde. V mém příkladě k žádné DYNAMICKÉ IDENTIFIKACI NEDOŠLO. Všechny identifikace šlo vyhodnotit již v době překladu a také to při překladu překladač udělal. Chceme-li identifikovat typ nějaké instance v době běhu programu, musí se jednat o instanci polymorfní třídy. Tedy třída musí mít alespoň jednu metodu volanou pozdní vazbou ("virtuální" metodu). Tato metoda v ní samozřejmě nemusí být deklarovaná, třída ji muže i zdědit. Viz moje články [Časná versus pozdní vazba - úvod do polymorfismu v C++](http://www.builder.cz/rubriky/c/c--/clanek-155697-casna-versus-pozdni-vazba-uvod-do-polymorfismu-v-c-/)a [Polymorfismus - dokončení](http://www.builder.cz/rubriky/c/c--/clanek-155701-polymorfismus-dokonceni/). Upravme v našem příkladě deklaraci metody nastav takto: virtual void nastav(int a, int b);. Chování programu se nyní změní. Řádek cout << typeid(funkce()).name() << endl;se bude chovat stejně. Žádám vlastně o identifikaci ukazatele na třídu Třída. V době překladu nemůže být pochyb o tom, že se bude jednat o tento ukazatel. Jestliže ale tento ukazatel dereferencuji, již budu žádat o identifikaci typu instance polymorfního typu. Zde je situace jiná. Výraz musí být vyhodnocen (Funkce se zavolá.) a poté jej operátor typeid identifikuje pomocí tabulky virtuálních metod. Vše se provede v době, kdy program běží, nikoliv v době, kdy je kompilován. V TVM tedy nejsou jen adresy metod volaných pozdní vazbou, ale i informace o typu. Tento příklad neukazuje nejlépe rozdíly mezi identifikací typů v době kompilace a v době běhu programu. Snažil jsem se jen poukázat na fakt, že v tomto případě bude výraz vyhodnocen. Může se jednat o velmi častý "zdroj" chyb, protože výraz může mít nějaký "vedlejší efekt". Například může změnit globální proměnné, atd... V našem příkladě nastane jiný problém, že ve funkci bude vytvořena instance, která nebude nikdy zlikvidována. Je důležité nezapomenout, že v případě identifikace polymorfního typu vlastně dojde k vyhodnocení výrazu. Nyní vytvořme příklad, který lépe ukáže rozdíl mezi statickou a dynamickou identifikací.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <typeinfo>  using namespace std;  class NadTrida  {  private:  int Atribut;  public:  virtual void nastav(int a);  };    class PodTrida : public NadTrida  {};  void NadTrida::nastav(int a)  {  Atribut = a;  }  int main()  {  NadTrida \*a = new NadTrida;  NadTrida \*b = new PodTrida; /\* Ukazatel b "ukazuje" na PodTridu \*/  if (typeid(\*b) == typeid(NadTrida))  {  cout << "Typ identifikovan v dobe prekladu." << endl;  }  else  {  cout << "Typ identifikovan pri behu programu." << endl;  }  cout << "Ukazatel " << typeid(a).name() <<  " se odkazuje na " << typeid(\*a).name() << endl;  cout << "Ukazatel " << typeid(b).name() <<  " se odkazuje na " << typeid(\*b).name() << endl;  return 0;  } | |

Výraz typeid(b) bude vyhodnocen při překladu. Překladač jasně vidí, že b je deklarován jako ukazatel. Nemůže si ale být jistý, že tento ukazatel ukazuje na objekt typu Nadtřída. Nadtřídaje totiž polymorfní typ. Odebereme-li v deklaraci metod třídy Nadtřída klíčové slovo virtual(rozhodně si to zkuste), bude překladač předpokládat, že ukazatel na třídu Nadtřída bude ukazovat na instanci třídy Nadtřída. Což ale není pravda. V toto případě překladač identifikuje \*b jako instanci třídy Nadtřída a k žádné dynamické identifikaci nedojde. Vše bude rozpoznáno "staticky" při překladu programu.

Když dynamická identifikace selže

Nelze-li určit typ objektu, vyvrhne operátor typeid výjimku třídy bad\_typeid. Třída bad\_typeidje potomkem třídy exception. Viz moje předchozí tři články. Bezpečné zjištění typu by tedy vypadalo následovně: (Třídy jsou deklarovány v předchozím příkladu.)

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  NadTrida \*p = NULL;  try  {  cout << "Typ:" << typeid(\*p).name() << endl;  }  catch (std::bad\_typeid &e)  {  /\* Nějaké ošetření výjimky. Identifikace typu se nepovedla. \*/  cerr << "Odchycena vyjimka" << endl;  }  return 0;  } | |

V některých starších překladačích, které nevyhovují normě se může třída bad\_typeidjmenovat Bad\_typeid a také nemusí být potomkem třídy exception. Jak vidíme z příkladu, výjimka může být vyvržená například v případě, že jako argument operátoru typeid je NULL, který má být dereferencován.

Závěrem k identifikaci typů

Jak je vidět, C++ má poměrně jednoduchou identifikaci typů. Je proto chybný postup přidávat do tříd atribut, který nějakým způsobem udává typ, jak mnoho programátorů chybně dělá. Buď má instance tabulku virtuálních metod, kde je tato informace již uložená, nebo dojde k identifikaci v době překladu, potom je takový atribut stejně zbytečný. Při použití operátoru typeid si musíme jasně uvědomit, jestli chceme identifikovat polymorfní typ, nebo cokoliv jiného. Polymorfní typ bude identifikován až v době, kdy program poběží, a hlavně nesmíme zapomínat, že v tomto případě dojde k vyhodnocení argumentu (výrazu).

Tolik tedy k dynamické identifikaci typů. Příště se podíváme na téma, které s identifikací typů úzce souvisí. Příští článek bude o dynamickém přetypování v C++.

## [Přetypování v C++](http://www.builder.cz/rubriky/c/c--/pretypovani-v-c--155842cz)

**27. července 2001, 00.00 | Dnes se podíváme jak přetypovat výraz v jazyce C++. Podíváme se na operátory dynamic\_cast, static\_cast a reinterpret\_cast. Vysvětlíme si jejich význam a ukážeme použití.**

Dnes se podíváme na přetypování objektů a proměnných v C++. Jazyk C++ "zdědil" po jazyce C operátor přetypování. V jazyce C se přetypuje výraz například takto: char a; int b; a = (char) b;. Použít tento operátor přetypování v C++ není vhodné a někdy je dokonce i nebezpečné. Jazyk C++ nabízí nové operátory k přetypování: dynamic\_cast, static\_cast, reinterpret\_cast. Jejich syntaxe je operátor<cílový typ>(výraz). Proberme si jednotlivé operátory podrobněji.

Operátor reinterpret\_cast

Operátor reinterpret\_cast slouží k přetypování spolu nijak nesouvisejících datových typů. Převod ukazatelů na instance nijak nesouvisejících tříd, nebo struktur (bez společného předka). Převody ukazatele na celá čísla a podobně. Operátor reinterpret\_cast se používá především při programování na velmi nízké úrovni a jeho chování může být závislé na dané platformě. Uveďme si jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  struct Struktura1  {  short int a;  short int b;  };  struct Struktura2  {  int cislo;  };  int main()  {  Struktura2 \*s2 = new Struktura2;  s2->cislo = 255;  Struktura1 \*s1 = reinterpret\_cast<Struktura1\*> (s2);  cout << s1->a << " " << s1->b << endl;  s2->cislo = 1000000;  cout << s1->a << " " << s1->b << endl;  int &c = reinterpret\_cast<int&> (\*s2);  cout << c << endl;  return 0;  } | |

Jak je zřejmé v mém příkladě předpokládám, že 2\*sizeof(short int) == sizeof(int). To ale nemusí být na různých typech počítačů, nebo i na různých operačních systémech pravda. Jak napovídá název operátoru, dojde k změně interpretace nějakého místa v paměti. Je dobré si ještě všimnout, že v mém příkladě ukazatele s1, s2, i reference c se odkazují na stejné místo v paměti.

Operátor static\_cast

Operátor static\_cast je v podstatě lepší náhrada operátoru (typ) z jazyka C. Lze jej použít k různým konverzím mezi primitivními datovými typy, pro přetypování z potomka na předka, pro přetypování ukazatelů i referencí z potomka na předka, atd. Operátor static\_cast není vhodný pro přetypování z předka na potomka. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  int a = 65;  char A = static\_cast<char>(a);  cout << a << " " << A << endl;  return 0;  } | |

Převádím-li static\_cast<Cílový typ>(výraz), a u cílového typu existuje bezparametrický konstruktor s parametrem stejného typu jako výraz, bude výsledek vytvořen pomocí něj. Postará se o to operátor static\_cast. Bude-li výraz objektového typu (třída) s přetíženým operátorem přetypování na cílový typ, bude použita metoda operátor typ() (přetížený operátor přetypování). Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  class Trida  {  private:  int Atribut;  public:  Trida(){}  Trida(int i):Atribut(i) {cout << "Konstruktor" << endl; }  operator int() { cout << "Operator (int) " << endl; return Atribut; }  };  int main()  {  int a = 65;  Trida t = static\_cast<Trida>(a);  int b = static\_cast<int>(t);  cout << b << endl;  return 0;  } | |

Doporučuji všem tento program spustit a podívat se, co vše operátor static\_cast provádí.

Jak je vidět z příkladu, operátor přetypování z jazyka C je zbytečný, dokonce i když je přetížen. Operátor static\_cast přebírá veškerou práci za něj. Jak jsem se již zmínil, static\_castnení vhodný pro přetypování předka na potomka. K těmto účelům se hodí následující operátor.

Operátor dynamic\_cast

Operátor dynamic\_cast slouží výhradně k přetypovávání ukazatelů, nebo referencí. Z ukazatele lze vytvořit pouze ukazatel a naopak z reference lze vytvořit pouze reference. Operátor dynamic\_cast je vhodné použít pro přetypování ukazatele (nebo reference) na předka na ukazatel (referenci) na potomka. Operátor dynamic\_cast bezpečně přetypovává polymorfní třídy (Třída obsahující alespoň jednu virtuální metodu.), protože k přetypování dojde až za běhu programu s pomocí dynamické identifikace typů. Nelze-li přetypovat ukazatele, je výsledný ukazatel roven NULL. Nelze-li přetypovat reference je vyvržená výjimka typu bad\_cast. Příklad na operátor dynamic\_cast:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <typeinfo>  using namespace std;  class Predek  {  private:  int A;  public:  void nastavA(int a){ A = a; }  virtual int dejA() { return A; }  };  class Potomek : public Predek  {  private:  int B;  public:  void nastavB(int b) { B = b; }  };    int main()  {  Predek \*pr = new Predek, \*po = new Potomek;  Potomek \*p;  pr->nastavA(10);  po->nastavA(20);  /\* Ukazatel po ukazuje na potomka, lze s ním pracovat jako s potomkem! \*/  if ((p = dynamic\_cast<Potomek\*>(po)) != NULL)  {  cout << "OK" << endl;  p->nastavB(10);  }  /\* Ukazatel pr ukazuje na předka, nelze s ním pracovat jako s potomkem! \*/  if ((p = dynamic\_cast<Potomek\*>(pr)) != NULL)  {  cout << "Nestane se" << endl;  p->nastavB(10);  }  try  {  Potomek &ref = dynamic\_cast<Potomek&>(\*po);  cout << "Přetypováno" << endl;  ref = dynamic\_cast<Potomek&>(\*pr); /\* Bude vyvolána výjimka. \*/  cout << "Přetypováno" << endl;  }  catch (bad\_cast& e)  {/\* Odchycení výjimky chybného dynamického přetypování referencí.  (Chybné přetypování v době běhu programu.) \*/  cerr << "Nelze pretypovat" << endl;  }  return 0;  } | |

Je důležité v takových případech používat operátor dynamic\_cast. Zajistí přetypování až v době běhu programu, zajistí také bezpečnost přetypování. Znovu bych chtěl připomenout, že má-li být přetypování předka na potomka v dědičné hierarchii bezpečné, musí dynamic\_castpřetypovávat polymorfní typy. Mnoho programátorů dělá velkou chybu, když operátor dynamic\_cast ignorují, a používají místo něj přetypování z jazyka C. Takové přetypování není bezpečné. Nevěříte-li, zkuste v mém poslední příkladu všechny operátory dynamic\_castpřepsat na (typ), tedy přetypovat ukazatele tak, jak se to dělá v jazyce C. Program vypíše "Nestane se", a na řádku p->nastavB(10); zapíše 10 do nealokované paměti se všemi následky!

Tolik tedy pro začátek k přetypování. Předpokládali jsme, že dědičnost v posledním příkladě bude jednoduchá, nikoliv vícenásobná. Ve svých článcích o vícenásobné dědičnosti jsem slíbil, že se ještě vrátím k přetypovávání instancí tříd vzniklých vícenásobnou dědičností. Dnes jsme se seznámili s operátorem dynamic\_cast, proto se příště můžeme podívat na použití dynamic\_cast při vícenásobné dědičnosti. Tím téma přetypování skončíme a v dalším článku se začneme zabývat šablonami v C++.

## [Problémy s typy při vícenásobné dědičnosti](http://www.builder.cz/rubriky/c/c--/problemy-s-typy-pri-vicenasobne-dedicnosti-155865cz)

**28. srpna 2001, 00.00 | Dnes se podíváme na některé problémy s přetypováním u vícenásobné dědičnosti v C++.**

V [minulém článku](http://www.builder.cz/rubriky/c/c--/clanek-155842-pretypovani-v-c-/) jsme si ukázali použití operátorů dynamic\_cast, static\_cast a reinterpret\_cast. Dnes se podíváme na některé záludnosti při přetypování instancí tříd vzniklých vícenásobnou dědičností. Vytvořme si tři jednoduché třídy:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  class PrvniNadTrida  {  public:  virtual void prvni() { cout << "Prvni this=" << this << endl; }  };  class DruhaNadTrida  {  public:  virtual void druha() { cout << "Druha this=" << this << endl; }  };  class PodTrida : public PrvniNadTrida, public DruhaNadTrida  {}; | |

První dvě třídy mají po jedné metodě, které vypíšou adresu, na kterou se odkazuje this. Třetí třída zdědí obě tyto metody po svých předcích. Podívejme se v jednoduché funkci main na první problém.

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  PodTrida \*pod = new PodTrida;  pod->prvni();  pod->druha();  delete PodTrida;  return 0;  } | |

Při spuštění programu zjistíme, že jeden objekt, na který se odkazuje ukazatel pod, má pří volání každé své metody jinou hodnotu implicitního parametru this. Problém spočívá v tom, jak je v C++ implementována vícenásobná dědičnost. Objekt, na který se odkazuje ukazatel pod jsou vlastně dva objekty (Jeden třídy PrvníNadTřída, druhy třídy DruháNadTřída.) za sebou. Nebudu se zde zabývat tématem vícenásobné dědičnosti, protože jsem se jím již zabýval ve svých předchozích článcích. Zmíněný problém s this není zas tak velkým problémem. Zkrátka každá metoda má "svůj" this a vše funguje. S tímto ale úzce souvisí jiný problém, o kterém se zmíním později. Nejprve ale vytvořme ještě dvě další funkce a změňme funkci main:

|  |  |
| --- | --- |
| |  | | --- | | void funkce1(DruhaNadTrida \*objekt)  {  cout << "funkce1: parametr " << objekt << endl;  objekt->druha();  }  void funkce2(PrvniNadTrida \*objekt)  {  cout << "funkce2: parametr " << objekt << endl;  objekt->prvni();  }  int main()  {  PodTrida \*pod = new PodTrida;  funkce1(pod);  funkce2(pod);  delete PodTrida;  return 0;  } | |

Vytvořil jsem 2 funkce, které vypíšou adresu, na kterou se odkazuje ukazatel daný jako parametr, a dále v každé funkci se zavolá metoda objektu, na který se parametr funkce odkazuje. Funkce se liší jen typem parametru. Ve funkci main vytvořím instanci třídy PodTřídaa zavolám obě funkce. V souladu s principy dědičnosti "na místě, kde je očekáván předek může být potomek" předám oběma funkcím jako parametr ukazatel na instanci třídy PodTřída. Po spuštění zjistíme, že každá funkce dostane jako parametr jiný ukazatel. Překladač při překladu volání funkcí funkce1, funkce2 správně přetypoval parametr na předka. Při přetypování ukazatele (nebo i reference) na instanci třídy vzniklé vícenásobnou dědičností může dojít ke změně samotné adresy, na kterou se ukazatel (reference) odkazuje. Na tento fakt je dobré pamatovat. Je to taková zvláštnost, objekt vlastně ztrácí svou identitu. jak jsem se již v dřívějších článcích zmiňoval, každý objekt má svou identitu, pomocí níž jej lze jednoznačně odlišit od jakéhokoliv jiného objektu. V C++ je tato identita dána paměťovou adresou objektu, jejíž hodnotu máme v implicitním parametru this. Nemohou existovat dva různé objekty na stejné adrese. Stejně tak nemůže, i když v uvedených příkladech se děje opak, být jeden objekt na více adresách. Při vícenásobné dědičnosti v C++ tomu tak ale je. Při jednoduché dědičnosti žádný takový problém nenastane. Pozměňme pro ilustraci znovu funkci main takto:

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  PodTrida \*pod = new PodTrida;  PrvniNadTrida \*prvni = pod;  DruhaNadTrida \*druhy = pod;  if (prvni != druhy)  {  cout << "prvni neni druhy" << endl;  }  delete PodTrida;  return 0;  } | |

Vytvořil jsem instanci třídy PodTřída, na kterou se odkazuje ukazatel pod. Dále jsem vytvořil dva ukazatele, které "ukazují" na stejný objekt, na který "ukazuje" pod. Nyní chci porovnat, zda ukazatele první a druhy jsou stejné (To znamená, zda ukazují na stejný objekt, nebo-li "Je objekt, na který ukazuje ukazatel první identický s objektem na který ukazuje ukazatel druhý?". Z předchozích dvou řádků plyne, že ano. Přesto po spuštění programu zjistíme, že ne. Právě tato ztráta identity objektu je podle mne velikou nevýhodou vícenásobné dědičnosti v C++. Ještě jen podotknu, že některým překladačům se asi po právu nebude líbit, že porovnáváme ukazatele různých typů. Přesně by mělo porovnání vypadat if ( (void\*) prvni != (void\*) druhy). Na chování programu to ale nic nezmění. Jak tedy zjistit, zda jsou objekty identické?

K tomuto účelu nám slouží operátor dynamic\_cast. Používáme-li vícenásobnou dědičnost, můžeme identitu objektů porovnat pomocí "triku" - přetypování na void\* pomocí operátoru pro dynamické přetypování. Porovnání má správně vypadat

if (dynamic\_cast<void\*>(prvni) == dynamic\_cast<void\*>(druhy))

{

cout << "Jsou stejné" << endl;

}

else

{

cout << "Nejsou stejné" << endl;

}

Je třeba si uvědomit, že ve svém předchozím porovnání jsem použil chybné přetypování pomocí operátoru (typ), před kterým jsem v minulém článku varoval. Použití správného operátoru dynamic\_cast vyřeší náš problém. Při použití vícenásobné dědičnosti totiž nemusí platit rovnost dynamic\_cast<typ\*> (ukazatel) == ukazatel. Na tento fakt je nutné pamatovat. Při použití jednoduché dědičnosti žádný problém s identitou nevzniká.

Tímto jsme ukončili téma přetypování v C++. V příštím článku se budeme věnovat šablonám funkcí, čímž otevřeme velikou kapitolu o šablonách v C++.

## [Šablony funkcí v C++](http://www.builder.cz/rubriky/c/c--/sablony-funkci-v-c--155874cz)

**7. září 2001, 00.00 | V dnešním článku se podíváme na šablony funkcí. Vysvětlíme si k čemu šablony slouží, jak šablony deklarovat a jak je použít. Ukážeme si pro nás nové klíčové slovo template.**

Co jsou to šablony funkcí a proč je používat? Šablona funkce je konstrukce, která umožňuje vytvořit funkci na základě parametrů. Proto se také vytváření a používání šablon někdy nazývá parametrické programování, a v souvislosti s šablonami se hovoří o parametrismu. Šablony nemusí být jen šablony funkcí, ale také šablony datových typů (struktur - struct, tříd - class, unií - union). O těch si povíme něco příště, dnešní článek bude věnován jen šablonám funkcí. Význam šablon se často a nejlépe demonstruje na příkladu funkce swap. Swap je funkce, která vymění ("přehodí") obsah dvou proměnných. Představme si situaci, kdy chceme mít možnost vyměnit hodnoty proměnných mnoha typů, nebo i instancí mnoha tříd. Bez použití šablon bychom museli napsat mnoho funkcí swap, které by se od sebe lišili jen typem parametrů. Lepší řešení je vytvořit jednu šablonu funkce swap, podle které se dle potřeby vytvoří požadovaná funkce na základě parametru, kterým bude typ. Syntaxe šablony funkce je:

|  |  |
| --- | --- |
| |  | | --- | | template< parametry šablony > deklarace | |

Šablona funkce swap:

|  |  |
| --- | --- |
| |  | | --- | | template<class T> void swap(T &a, T &b)  {  T c(a);  a = b;  b = c;  } | |

Parametrem šablony je typ T. Klíčové slovo class značí, že parametrem bude datový typ. Nemusí se jednat pouze o třídu, ale o jakýkoliv datový typ, i primitivní datový typ. Dále následuje funkce, swap. Tato funkce bude mít dva parametry, jenž budou referencí na typ T. V těle funkce vytvoříme dočasnou proměnnou c, která je typu T. Dále provedeme záměnu hodnot proměnných pomocí třetí proměnné. Konkrétní funkci pro konkrétní typ (instanci šablony) vytvoříme dosazením typu za parametr T. Překladač ještě v době překladu vytvoří funkcí swap, která bude mít tělo jako šablona, ale za T bude dosazen typ int. Příklad instanciace a použití šablony swap:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  using namespace std;  int main()  {  int a(0),b(100);  swap<int>(a,b);  cout << a << " " << b << endl;  return 0;  } | |

Podívejme se na řádek swap<int>(a,b);. V závorkách < > je parametr šablony. v závorkách ( ) jsou parametry funkce. Je-li parametr šablony zřejmý z volání funkce, nemusí se parametry šablony vůbec uvádět. V našem případě je z typu proměnných a a b jasné, že parametrem šablony je int, proto stačí dokonce napsat jen swap(a,b). Naše šablona může být použitá pro jakýkoliv typ. Jen je dobrá si všimnout, že ve funkci swap bude vytvořena kopie proměnné (objektu) a, a že bude použít operátor = pro přiřazení hodnot. Proto by typ, který bude parametrem naší šablony měl mít kopírovací konstruktor, pokud to bude nutné (viz můj článek [Kopírovací konstruktor v C++](http://www.builder.cz/rubriky/c/c--/clanek-155685-kopirovaci-konstruktor-v-c-/)), a také, pokud to bude nutné, přetížený operátor = (viz můj článek [Přetěžování operátorů](http://www.builder.cz/rubriky/c/c--/clanek-155728-pretezovani-operatoru-v-c-1-dil/)). Ukažme si nyní využití šablony funkce swap. Spolu s deklarací šablony swap vytvořme program:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<string>  using namespace std;  class Trida  {  private:  int atribut;  public:  int dejAtribut() const {return atribut;}  void nastavAtribut(int atribut) { this->atribut = atribut;}  /\* Pro tuto třídu není potřeba  ani kopírovací konstruktor, ani operátor = \*/  };  int main()  {  int a(0),b(100);  cout << a << " " << b << endl;  swap(a,b);  cout << a << " " << b << endl;  char x('x'), y('Y');  cout << x << " " << y << endl;  swap(x,y);  cout << x << " " << y << endl;  string s1("Ahoj "), s2("svete");  cout << s1 << s2 << endl;  swap(s1,s2);  cout << s1 << s2 << endl;  Trida instance1, instance2;  instance1.nastavAtribut(100);  instance2.nastavAtribut(-100);  cout << instance1.dejAtribut() << " " << instance2.dejAtribut() << endl;  swap(instance1,instance2);  cout << instance1.dejAtribut() << " " << instance2.dejAtribut() << endl;  return 0;  } | |

Jak je vidět naše šablona funkce swap je použitelná pro mnoho typů. Je nutné upozornit, že v přeloženém programu šablona vlastně neexistuje. Existují v něm pouze instance šablony, které se vytvoří v době překladu. Nemusíme se tedy obávat nějakého zpomalení programu používáním šablon. Naopak šablony se mnohdy používají pro optimalizaci výsledného programu, protože mnoho činností přebírá překladač v době překladu, čímž může být výsledný program rychlejší. Šablony mohou někomu připomínat makra s parametry. Dalo by se skutečně říci, že šablony jsou vlastně lepší a "inteligentnější" makra.

Přetěžování šablon funkcí

Šablony funkce lze přetěžovat stejně tak jako samotné funkce. Pokusíme se přetížit naší šablonu funkce swap. Další šablona bude šablona funkce zaměňující hodnoty prvků dvou stejně dlouhých polí:

|  |  |
| --- | --- |
| |  | | --- | | template<class T, int N> void swap(T \*a, T \*b)  {  T temp;  for (register int p = 0; p < N; p++, a++, b++)  {  temp = \*a;  \*a = \*b;  \*b = temp;  }  } | |

Dalším parametrem šablony swap je celé číslo. Za tento parametr lze dosadit jen konstantu, nebo výraz, který lze vyhodnotit v době překladu. Společně s předchozí šablonou swap lze obě šablony použít například takto:

|  |  |
| --- | --- |
| |  | | --- | | int main()  {  int pole1[2] = {1, 2}, pole2[2] = {3,4};  swap<int,2>(pole1,pole2);  for(a = 0; a < 2; a++)  {  cout << pole1[a] << " " << pole2[a] << endl;  }  swap(pole1[0],pole2[0]);  for(a = 0; a < 2; a++)  {  cout << pole1[a] << " " << pole2[a] << endl;  }  return 0;  } | |

Priority volání

Vytvořme si do třetice obyčejnou funkci (Ne šablonu funkce!), která jako své parametry dostane ukazatele na první znak řetězce zakončeného znakem '\0'. Funkce přehodí samotné řetězce, ne pouze ukazatele na ně. Takto napsaná funkce je použitelné pouze pro řetězce stejné délky.

|  |  |
| --- | --- |
| |  | | --- | | void swap(char \*a, char \*b)  {  register char temp;  while ((\*a != '\0') && (\*b != '\0'))  {  temp = \*a;  \*a++ = \*b;  \*b++ = temp;  }  cout << "Není šablona" << endl;  /\* Výpis, abychom věděli, že funkce byla zavolána. \*/  } | |

Nyní společně se šablonami swap vytvořme jednoduchou funkci main:

|  |  |
| --- | --- |
| |  | | --- | | #include <string.h>  #include <iostream>  using namespace std;  int main()  {  char \*r1 = strdup("svete"), \*r2 = strdup("Ahoj ");  int i1(1), i2(2);  swap(r1,r2); /\* Bude volána void swap(char \*a, char \*b) \*/  cout << r1 << r2 << endl;  swap(i1,i2); /\* Bude volána template<int> void swap(int &a, int &b) \*/  cout << i1 << i2 << endl;  return 0;  } | |

Po spuštění programu se můžeme podle výpisu přesvědčit, že funkce swap budou skutečně volány tak, jak jsem uvedl v poznámkách. Pro dva řetězce nebude vytvořena instance šablony, a přeloženo její volání, ale bude přeloženo volání funkce swap(char \*a, char \*b). Priorita volání funkcí by šla shrnout do následujících pravidel:

1) Existuje-li pro daný typ parametrů funkce, přeloží se volání této funkce

.

2) Neexistuje-li pro daný typ parametrů funkce, překladač zjistí, jestli nemůže vytvořit instanci šablony.

3) Není-li k dispozici ani funkce, ani šablona, ze které lze vytvořit potřebnou instanci, jedná se o chybu.

Explicitní a implicitní vytváření instancí šablon.

Existují dva způsoby jak vytvářet instance šablon. První je implicitní vytvoření instance. Znamená to, že překladač vytvoří instanci a přeloží ji jen tehdy, jestliže ji potřebuje. Všechny instance šablony swap v předchozích příkladech jsou vytvořeny implicitně. Další možnost jak vytvořit instanci šablony explicitně. Budeme-li chtít, aby překladač vytvořil instanci šablony swap pro parametr float, i když nebude v daném modulu požadována, napíšeme:

|  |  |
| --- | --- |
| |  | | --- | | template void swap<float>(float &a, float &b); | |

Tolik tedy k šablonám funkcí. Příště se podíváme na šablony datových typů. Na jednoduché příkladu si vytvoříme šablonu třídy. Instance šablony třídy je třída. Také se podíváme na další, pro nás nové, klíčové slovo typename.

## [Šablony datových typů v C++](http://www.builder.cz/rubriky/c/c--/sablony-datovych-typu-v-c--155883cz)

**17. září 2001, 00.00 | Dnes si povíme něco o šablonách tříd v C++. Podíváme se také na úplné i částečné specializace šablon.**

V minulém článku jsme si pověděli něco o šablonách funkcí v C++. Dnes se zaměříme na šablony datových typů. V tomto článku se budu výhradně věnovat šablonám tříd, ale vše co zde uvedu je použitelné také na struktury a unie. Uveďme si příklad jednoduché šablony třídy.

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> class Obal  {  private:  Typ Promenna;  public:  Obal();  Obal(Typ p);  Obal(Obal &druhy);  inline Typ dejPromennou();  inline void nastavPromennou(Typ p);  }; | |

Šablona třídy Obal je jakýsi "objektový obal" pro jakoukoliv proměnnou. Možná by měl být ještě přetížen operátor =, případně i operátory == a !=, o to ale teď nejde. Vytvoříme instanci této šablony pro nějaký konkrétní typ, instance šablony třídy je třída. Šablona třídy by se dala považovat za něco jako metatřídu. Tedy třídu třídy. Není to ale přesné přirovnání, jazyk C++ metatřídy nemá. Problém je v tom, že šablony jsou v C++ instanciovány již v době překladu, tedy všechny parametry pro jakoukoliv šablonu (i šablonu funkce, i šablonu datového typu) musejí být známy již v době překladu. Nejprve si ukažme jak implementovat metody této šablony.

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> Obal<Typ>::Obal() : Promenna()  {}  template<class Typ> Obal<Typ>::Obal(Typ p) : Promenna(p)  {}  template<class Typ> Obal<Typ>::Obal(Obal &druhy) : Promenna(druhy.dejPromennou())  {}  template<class Typ> Typ Obal<Typ>::dejPromennou()  {  return Promenna;  }  template<class Typ> void Obal<Typ>::nastavPromennou(Typ p)  {  Promenna = p;  } | |

Metody šablony třídy se implementuji obdobně jako metody třídy. Jen je třeba si uvědomit, že i metoda šablony třídy je vlastně šablona. Proto musíme uvést klíčové slovo template i s parametry šablony. Tímto máme vytvořenou šablonu třídy obal. Je dobré si také všimnout, že o parametru šablony Typ v době kdy šablonu píšu vlastně nic nevím. Přesto předpokládám mnoho vlastností, které tento typ bude mít. Například musí mít bezparametrický konstruktor, kopírovací konstruktor a operátor =. Pokud si dobře prohlédne těla metod, zjistíte, že je používám. Samozřejmě je nemusím přetěžovat, jestli k tomu není důvod, ale je nutné, aby každý kdo bude šablonu používat tento fakt věděl. Nyní vytvoříme jednoduchý program, kde naši šablonu použijeme.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  int main(void)  {  Obal<int> a(10);  cout << a.dejPromennou() << endl;  a.nastavPromennou(3);  cout << a.dejPromennou() << endl;  Obal<char> b;  b.nastavPromennou('a');  cout << b.dejPromennou() << endl;  Obal<char> c(b);  cout << c.dejPromennou() << endl;  return 0;  } | |

Třída Obal<int> je instance šablony třídy Obal. Objekt a je instance třídy Obal<int>. Někdy je dobré pro přehlednost použít klíčové slovo typedef. Například typedef Obal<int> INT. Zde jsme vytvořili třídu INT, která je instancí šablony třídy Obal<int>. Dále lze INT použít normálním způsobem: INT a, \*b = new INT(2);. Zde jsou objekty a, \*b (objekt, na který ukazuje ukazatel b) instance třídy INT.

Co kdyby byl parametrem šablony Obal datový typ, který je instanci nějaké šablony? To není v praxi nic neobvyklého. Vytvoříme "obal obalu int": Obal<Obal<int> > DvojObal; . Důležité je, že mezi znaky > je mezera. Kdybychom napsali >>, překladač by tento token považoval za operátor binárního posunu, a nahlásil by chybu v syntaxi.

Specializace šablon tříd

Šablony tříd nelze přetěžovat tak, jako šablony funkcí. Lze ale vytvářet specializace šablon tříd. Naopak šablony funkcí nelze specializovat. Specializace šablony třídy může být částečná, nebo úplná. Nevím přesně proč, ale úplné specializaci se někdy také říká explicitní specializace. Specializace spočívá v tom, že vytvořím speciální případ šablony pro nějakou podmnožinu parametrů - částečná specializace, nebo specielní případ šablony pro konkrétní hodnoty parametrů - úplná specializace.

Částečná specializace

Pokusíme se pro ukázku částečně specializovat naší šablonu Obal. V případě, že parametrem bude ukazatel, vytvoříme kopii objektu, na který se ukazatel odkazuje. Nejprve je nutné uvést obecnou šablonu, která je napsána výše. Poté se uvede částečná specializace šablony:

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> class Obal<Typ\*>  {  private:  Typ \*Promenna;  public:  Obal() : Promenna(NULL) {}  Obal(Typ \*p) : Promenna(new Typ(\*p)) {}  Obal(Obal &druhy) : Promenna (new Typ(\*drhy.dejPromennou())) {}  ~Obal() { delete Promenna; }  Typ \*dejPromennou() { return Promenna; }  void nastavPromennou(Typ \*p) { Promenna = new Typ(\*p); }  }; | |

Specializace šablon nemá nic společného s dědičností tříd. Specializovaná šablona je úplně nový a nezávislý typ. Všechny metody i atributy je nutné znovu opsat, nic se nedědí po obecné šabloně. Všimněte si, že v prvním řádku specializované šablony uvádím za slovem "Obal" onu podmnožinu parametrů, pro které se má použít tato specializace šablony. Použití částečné specializace šablony si ukážeme v dalším odstavci společně s použitím úplné specializace.

Úplná specializace

Nyní vytvoříme úplnou specializaci naší šablony Obal. Pro případ, že by parametrem šablony byl ukazatel na char, jednalo by se o ukazatel na první znak klasického řetězce z C, který je ukončen nulou. V tomto případě si budeme chtít mít vytvořenou kopii tohoto řetězce v našem obalu. Nezapomeňte, že nejprve musí být deklarována obecná šablona. Potom částečná specializace, budeme-li chtít používat částečně specializovanou šablonu. A na konec nejkonkrétnější případ:

|  |  |
| --- | --- |
| |  | | --- | | #include<string.h>  // Úplná specializace  template<> class Obal<char\*>  {  private:  char \*Promenna;  public:  Obal();  Obal(char \*r);  Obal(Obal &druhy);  ~Obal();  inline char \*dejPromennou();  inline void nastavPromennou(char \*p);  };  Obal<char\*>::Obal() : Promenna(NULL)  {}  Obal<char\*>::Obal(char \*r) : Promenna(strdup(r))  {}  Obal<char\*>::Obal(Obal &druhy) : Promenna(strdup(druhy.dejPromennou()))  {}  Obal<char\*>::~Obal()  {  delete[] Promenna;  }  char \*Obal<char\*>::dejPromennou()  {  return Promenna;  }  void Obal<char\*>::nastavPromennou(char \*p)  {  delete[] Promenna;  Promenna = strdup(p);  } | |

Opět jsme z obecné šablony nic nezdědili, museli jsme všechny metody i atributy vytvořit znova. Uvedeme si příklad použití všech tří šablon:

|  |  |
| --- | --- |
| |  | | --- | | int main(void)  {  int cislo = 5;  Obal<int> a(10); // Použije se obecná šablona  cout << a.dejPromennou() << endl;  a.nastavPromennou(3);  cout << a.dejPromennou() << endl;  Obal<int\*> b(&cislo); // Použije se částečná specializace pro ukazatele.  cout << b.dejPromennou() << endl;  Obal<char\*> cc; // Použije se konkrétní specializace pro typ char\*  cc.nastavPromennou("Ahoj");  cout << cc.dejPromennou() << endl;  return 0;  } | |

Na závěr bych chtěl jen upozornit na fakt, který nemusí být zřejmý. Jak jsem již vysvětloval v minulém článku pojem šablony, upozornil jsem, že šablona existuje pouze v době překladu. Za běhu programu (dokonce i v době "linkování" (spojování) linkerem) existují jen instance šablony. Proto musí být celá šablona přímo deklarovaná v kompilovaném zdrojovém textu, nebo v hlavičkovém souboru, který bude vložen pomocí #include. Šablona je celá považována za deklaraci, i když třeba někomu šablona funkce čí metody připadá jako definice (implementace). U šablony třídy budou při instanciaci vytvořeny jen ty metody, které budou použity. Ostatní nechá překladač bez povšimnutí, i kdyby obsahovaly chyby. Jinou možností je vytvořit šablonu explicitně tak, jak jsem to popsal v minulém článku.

Článek je již dost dlouhý, a bohužel jsme se nedostali ke klíčovému slovu typename (V minulém článku jsem to sliboval.), které souvisí s vnořenými typy. Podíváme se na něj v příštím článku. Stejně tak se v příštím článku podíváme na vnořené šablony.

## [Vnitřní typy u parametrů šablon, vnořené šablony v C++](http://www.builder.cz/rubriky/c/c--/vnitrni-typy-u-parametru-sablon-vnorene-sablony-v-c--155889cz)

**25. září 2001, 00.00 | V dnešním článku se budeme věnovat dvěma nesouvisejícím tématům. Jednak vnitřním typům u parametrů šablon, s čímž souvisí pro nás nové klíčové slovo typename. Druhé téma budou vnořené šablony, tedy šablony, ve kterých je deklarována další šablona.**

V dnešním článku se budeme věnovat dvěma nesouvisejícím tématům. Prvním z nich bude práce s vnitřními typy u parametrů šablon, s čímž souvisí pro nás nové klíčové slovo typename. Jako druhé téma jsem vybral vnořené šablony, tedy šablony, ve kterých je deklarována další šablona.

Vnitřní typy u parametrů šablon v C++

V jazyce C++ může být v jednom typu deklarován typ jiný - vnitřní (vnořený) typ. Vnitřní typ může být jednak třída deklarovaná uvnitř třídy, nebo typ uvnitř třídy deklarovaný pomocí klíčového slova typedef. Opět, jako vždy ve svých článcích, se zabývám pouze třídou, vše co zde napíši je obdobně použitelné i pro struktury a unie. Má-li být schopen s vnitřním typem pracovat kdokoliv jiný, než objekt dané třídy, musí být vnitřní typ veřejný. Uveďme si jednoduchý příklad vnitřní třídy:

|  |  |
| --- | --- |
| |  | | --- | | class VnejsiTrida  { /\* Vější třída \*/  public:  class VnitrniTrida  { /\* Vnitřní třída \*/  private:  int Atribut;  public:  int dejAtribut() { return Atribut; }  void nastavAtribut(int a) { Atribut = a; }  };  VnitrniTrida &dejObsah(); /\* Vrátí instanci vnitřní třídy. \*/  private:  VnitrniTrida Obsah; /\* Vnější třída má jako atribut instanci  vnitřní třídy. \*/  };  VnejsiTrida::VnitrniTrida &VnejsiTrida::dejObsah()  {  return Obsah;  } | |

Nyní pro ukázku jak s vnitřní třídou pracovat předvedu velice jednoduchou funkci main.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  int main(void)  {  VnejsiTrida vnejsi;  vnejsi.dejObsah().nastavAtribut(10);  VnejsiTrida::VnitrniTrida vnitrni = vnejsi.dejObsah();  cout << vnitrni.dejAtribut() << endl;  return 0;  } | |

Nyní se vraťme zpět k šablonám. Představme si situaci, kdy chceme vytvořit šablonu, která by jako svůj parametr měla typ. Předpokládali by jsme, že za parametr dosadíme vždy třídu, která bude mít vnitřní typ pojmenovaný jako VnitřníTřída. Tedy například naší třídu. Překladač ale v době kdy "si prohlíží" šablonu neví, že parametr možná bude mít vnitřní třídu. Proto musíme použít klíčové slovo typename. Uveďme si jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> void vypisObsah(ostream &o, Typ parametr)  {  typename Typ::VnitrniTrida v; /\* v je objekt \*/  /\* Nestačilo by  VnitrniTrida v  protože VnitniTrida není globální typ, a překladač  jej proto pochopitelně nezná.  \*/  v = parametr.dejObsah();  o << v.dejAtribut() << endl;  } | |

Celý můj příklad je pouze ukázka použití typename. Jinak nedává žádný smysl. Snad jsem tím nikoho nezmátl. Vytvořil jsem šablonu funkce. Parametrem šablony je typ jménem Typ. Parametry funkce (parametry instance šablony) jsou výstupní proud a instance typu Typ. O typu Typ vím, že má vnitřní typ jménem VnitřníTřída. Bude-li šablona instanciována s parametrem, který nemá jako veřejný vnitřní typ se jménem VnitřníTřída, dojde k chybě. V praxi může existovat hodně tříd, které budou mít vnitřní třídu s názvem VnitřníTřída. Každá VnitřníTřída bude ale úplně jiná, bude mít jen stejně pojmenované metody, které ale mohou dělat úplně jinou činnost. Díky toho lze vytvářet velmi abstraktní šablony. Například až se budeme věnovat knihovně STL a datovým kontejnerům v C++ uvidíme, že každý kontejner má vnitřní typ jménem iterátor. Nyní ještě použití naší šablony.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <ofstream>  using namespace std;  int main(void)  {  VnejsiTrida v;  v.dejObsah().nastavAtribut(10);  vypisObsah(cout,v); /\* Nebo to samé: \*/  vypisObsah<VnejsiTrida>(cout,v);  fstream soubor("file.txt");  vypisObsah(soubor,v);  return 0;  } | |

Vnořené šablony

Nyní se podíváme na situaci, kdy v šabloně je deklarována šablona. Vytvořme si velice jednoduchou šablonu třídy, podobnou šabloně Obal z mého minulého článku. Tato třída bude mít přetížen operátor = jako svou metodu. Budeme ale chtít, aby náš operátor byl použitelný pro mnoho typů, které nemusíme v době psaní naší šablony ani znát.

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> class Obal  {  private:  Typ Atribut;  public:  void nastavAtribut(Typ a) { Atribut = a; }  Typ dejAtribut() { return Atribut; }  template<class Parametr> Obal<Typ> &operator=(Parametr druhy);  }; | |

Vytvořili jsme šablonu Obal. Šablona má jednu vnořenou šablonu - operátor =. Nyní vytvoříme implementaci tohoto operátoru.

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> template<class Parametr>  Obal<Typ> &Obal<Typ>::operator=(Parametr druhy)  /\* Pozor !! Nelze template<class Typ, class Parametr> \*/  {  Atribut = druhy; /\* Musí být operátor = pro typy Typ a Parametr \*/  return \*this;  }; | |

Jak jsem již upozornil ve zdrojovém textu, nelze napsat template<class Typ, class Parametr> , protože by se nejednalo o vnořenou šablonu, ale o šablonu s dvěma parametry. Nyní si ukážeme použití naší šablony.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  int main(void)  {  Obal<float> pi; /\* Přibližně 22/7 \*/  pi = 22.0 / 7; /\* Obal<float> operator=(float druhy) \*/  cout << pi.dejAtribut() << endl;  pi = 10; /\* Obal<float> operator=(int druhy) \*/  cout << pi.dejAtribut() << endl;  return 0;  } | |

Pro náš obal může být parametrem operátoru = cokoliv, co lze přiřadit pomocí operátoru = (implicitního, nebo přetíženého) k jeho atributu.

Pro dnešek je to všechno. V příštím článku se pokusíme vytvořit celkem i praktickou šablonu. Bude to asi i první zdrojový text v mém seriále, který bude mít snad i nějaké praktické využití. Pokusíme se totiž vytvořit pole, které nebude mít dolní index 0, ale libovolnou programátorem zadanou hodnotu. Například indexovat jej bude možné od -10 do +10, nebo od 50 do 100. Bude se jednat o pole, jaké známe například z programovacího jazyka Pascal. V dalších článcích se potom začneme věnovat knihovně STL.

## [Pole s libovolným intervalem indexování v C++](http://www.builder.cz/rubriky/c/c--/pole-s-libovolnym-intervalem-indexovani-v-c--155896cz)

**1. října 2001, 00.00 | Předvedeme si jak vytvořit pole, jehož indexy nemusí být v intervalu 0 až N-1 jak jsme zvyklí, ale v libovolném zadaném intervalu. Např. od -10 do 20. V podstatě pole jaké známe např. z Pascalu. Vše, co je k práci s takovým polem nutné, je zde ke stažení.**

Dnes si ukážeme jak vytvořit pole, jehož indexy nemusí být v intervalu 0 až N-1 jak jsme zvyklí, ale v libovolném programátorem zadaném intervalu. Například od -10 do +20. Tedy v podstatě pole jaké známe například z programovacího jazyka Pascal. Vše co je k práci s takovým polem potřeba je na konci tohoto článku k dispozici ke stažení.

Jak jsem již v minulém článku naznačoval, bude se jednat o šablonu. Ukážeme si vlastně na příkladu "větší" šablony jaké mají šablony v C++ nesmírné praktické využití. Všem, kteří nečetli mé předchozí 3 články o šablonách, a chtějí porozumět mému dnešnímu příkladu šablony, doporučuji si nejprve přečíst mé předchozí 3 články o šablonách.

Touto šablonou jsem se vlastně pokusil vytvořit něco jako datový kontejner z knihovny STL. Knihovnou STL se začneme zabývat od příštího článku. V STL je mnoho kontejnerů jako třeba zásobník, fronta, asociativní pole,"obyčejné" jednorozměrné pole, atd... Co ale v STL schází je právě takové pole, které se nyní budeme snažit vytvořit. Všechny kontejnery mají předepsané nějaké rozhraní (Veřejné metody a vnitřní veřejné typy.), které jsem se snažil dodržet. Existuje tedy mnoho metod, které jsou v příkladě podstatě jen z formality.

Vytváříme kontejner

Šablonu jsem nazval Array. Array je poměrně frekventované slovo v programovacích jazycích, přesto jsem nezjistil že by ve standardní knihovně C++ již bylo použito. Je ale možné, že jej používá nějaká jiná knihovna. Aby nedošlo ke konfliktu jmen, umístil jsem své pole do [prostoru jmen](http://www.builder.cz/rubriky/c/c--/clanek-155766-prostory-jmen-v-c-/), který jsem pojmenoval www\_builder\_cz. Bude se jednat o šablonu třídy. Parametrem šablony bude typ, který bude typem prvků v poli uložených. Interval indexů nebude parametrem šablony, protože by hranice intervalu musely být známy již v době kompilace, což se mi nezdá výhodné. Atributy třídy, která bude instancí šablony bude jednak ukazatel na první prvek "obyčejného" pole, které bude ve třídě zapouzdřeno. Bude obsahovat samotná data. Dále dva atributy, které budou udávat dolní hranici indexu a počet prvků v poli. Tyto dva atributy by měly stačit k výpočtu všech ostatních potřebných informací. Deklarace šablony třídy vypadá takto:

|  |  |
| --- | --- |
| |  | | --- | | template<class Typ> class Array  {  private:  Typ \*Data;  int Lower, Size; // Spodní hranice indexu, počet prvků v poli  public:  // Typy  typedef Typ value\_type;  typedef Typ &reference;  typedef const Typ &const\_reference;  typedef Typ \*iterator;  typedef const Typ \*const\_iterator;  typedef size\_t size\_type;  // size\_t je s největší pravděpodobností unsigned int  typedef ptrdiff\_t difference\_type;  // ptrdiff\_t je s největší pravděpodobností int  // Konstruktory, destruktor a operátor =  Array();  Array(const Array<Typ> &copy);  Array(difference\_type low, difference\_type up);  Array<Typ> &operator=(const Array<Typ> &copy);  ~Array();  // Iterátory  inline iterator begin();  inline iterator end();  inline const\_iterator begin() const;  inline const\_iterator end() const;  // Náhodný přístup k prvkům - operátor [], metoda at  inline const\_reference operator[] (difference\_type index) const;  inline reference operator[] (difference\_type index);  inline const\_reference at(difference\_type index) const;  inline reference at(difference\_type index);  // Relační operátory  bool operator==(const Array<Typ> &second) const;  inline bool operator!=(const Array<Typ> &second) const;  inline bool operator< (const Array<Typ> &second) const;  inline bool operator> (const Array<Typ> &second) const;  // swap  void swap(Array<Typ> &second);  // Metody pro informování o kapacitě, velikosti a indexech  inline size\_type size() const;  inline size\_type max\_size() const;  inline size\_type capacity() const;  inline difference\_type getLower() const;  inline difference\_type getUpper() const;  inline bool empty() const;  }; | |

Ve veřejném rozhraní třídy je mnoho vnitřních typů. Všechny standardní kontejnery mají takto pojmenované typy, proto jsem je zde i já rozhodl takto pojmenovat. Nejprve bych chtěl upozornit na typ size\_t, který jsem pojmenoval jako size\_type a typ ptrdiff\_t , který jsem pojmenoval jako difference\_type. Oba typy jsou jen přejmenované primitivní datové typy. Přejmenování závisí na jednotlivých překladačích, ale ve 32-bitovém prostředí se jedná s největší pravděpodobností o typy usigned int a int. Dalším důležitým typem je iterator. Iterátorům se budeme věnovat v jednom samostatném článku. Iterátory se chovají podobně jako ukazatele, a v našem případě to dokonce ukazatel je. Těm, kteří se s pojmem iterátor setkali poprvé se omlouvám, že používám pojem, který jsem nevysvětlil. Prozatím budeme iterátor považovat za ukazatel, v některém ze svých následujících článcích tento pojem vysvětlím blíže. Dále následují konstruktory, operátor = a destruktor. Následují metody vytvářející (vracející) iterátory (ukazatele) na začátek bloku dat a za poslední prvek. Metody begin a end by měly mít všechny kontejnery. Následuje operátor [] a metoda at. Metoda atdělá to samé, co operátor []. Rozhodl jsem se jí vytvořit jen pro to, protože stejnou metodu (i stejně zbytečnou) má jiný kontejner - vector. Poté jsou deklarovány relační operátory, metoda swap pro výměnu obsahů dvou polí, a některé informační metody o parametrech kontejneru. Metoda size vrací počet prvků v kontejneru, metoda capacity vrací velikost paměti (v bytech), které zabírají prvky v kontejneru. Metoda empty oznamuje, zda kontejner je, či není prázdný. Dále následuje metoda max\_size, která obvykle u kontejnerů vrací maximální možný počet prvků, které se do kontejneru dají přidat. Můj kontejner ale nedokáže měnit svou velikost. Schází zde metody insert a erase, které u jiných kontejneru jsou. Proto vlastně Array nemůže být považován za plnohodnotný kontejner. V mém kontejneru tedy metoda max\_size dělá tu samou činnost jako size. Dále jsem přidal dvě metody getLower pro vrácení dolního indexu a getUpper pro vrácení horního indexu. Tolik k deklaraci. Podívejme se na implementaci některých vybraných metod. Vše, s podle mne podrobným komentářem, je ke stažení na konci článku.

|  |  |
| --- | --- |
| |  | | --- | | /\* Kopírovací konstruktor \*/  template<class Typ> Array<Typ>::Array(const Array &copy) : Data(NULL)  ,Lower(copy.getLower()),Size(copy.size())  {  register iterator d;  Data = new Typ[this->size()];  d = begin();  for (register const\_iterator p = copy.begin(); p < copy.end(); p++, d++)  {  \*d = \*p;  }  }  /\* Operátor = \*/  template<class Typ> Array<Typ> &Array<Typ>::operator=(const Array<Typ> &copy)  {  delete[] Data;  Size = copy.size();  Lower = copy.getLower();  Data = new Typ[size()];  register iterator d = this->begin();  for(register const\_iterator t = copy.begin(); t < copy.end(); t++, d++)  {  \*d = \*t;  }  return \*this;  }  /\* Metoda end \*/  template<class Typ> inline Array<Typ>::iterator Array<Typ>::end()  {  return &Data[size()];  }  /\* Operátor [] \*/  template<class Typ> inline Array<Typ>::const\_reference Array<Typ>::operator[]  (difference\_type index) const  {  return const\_cast<const\_reference>(Data[index - getLower()]);  }  /\* Operátor < \*/  template<class Typ>  inline bool Array<Typ>::operator< (const Array<Typ> &second) const  {  return std::lexicographical\_compare(this->begin(), this->end(),  second.begin(), second.end());  } | |

První dvě metody ukazují, jak se naše pole kopíruje. Je dobré si všimnou práce s iterátory (ukazateli). Znovu připomínám, že begin vrací iterátor (ukazatel) na začátek pole, end vrací iterátor ZA poslední prvek v poli. Ještě bych chtěl upozornit na operátory < a >. Používají v sobě funkci lexicographical\_compare. Jedná se o instanci jedné z mnoha standardních šablon funkcí. Jde o další věc, kterou předkládám bez vysvětlení, za což se omlouvám. Nejen standardním kontejnerům, iterátorům, ale i standardním funkcím se budeme v budoucnu věnovat podrobněji.

Důležité je upozornit, že typ, který bude parametrem šablony. Tedy typ prvků, které budou uloženy v poli, musí mít k dispozici bezparametrický konstruktor. Dále musíme vzít v úvahu, že bude pro proměnné (instance) tohoto typu používán operátor =. Proto nebude-li použitelný implicitní operátor, bude se muset přetížit. Dále, pokud budete chtít porovnávat dvě pole pomocí operátorů < ,nebo >, musejí být tyto operátory k dispozici i pro prvky v poli.

Nyní si ukažme několik příkladů jak naše pole použít ve svých programech. Velice jednoduchý příklad je:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include "array.h"  using namespace www\_builder\_cz; //Nezapomeňte na prostor jmen  int main(int argc, char \*\*argv)  {  Array<int> Inty(10,20);  Inty[10] = 10;  cout << Inty[10] << endl;  for(int p = 10; p <= 20; p++)  {  Inty[p] = p \* 10;  }  for(int p = 10; p <= 20; p++)  {  cout << Inty[p] << endl;  }  cout << "Dolni " << Inty.getLower() << endl  << "Horni " << Inty.getUpper() << endl  << "Velikost " << Inty.size() << endl  << "Rozloha " << Inty.capacity() << endl;  Array<int> I = Inty;  if (I == Inty)  {  cout << "==" << endl;  }  else  {  cout << "!=" << endl;  }  I[15] = 100;  I.swap(Inty);  cout << I[15] << endl;  if (I != Inty)  {  cout << "!=" << endl;  }  else  {  cout << "==" << endl;  }  return 0;  } | |

Nyní se bez větších komentářů podíváme na příklad, kdy do našeho pole chceme vkládat instance nějaké vlastní třídy, ne proměnné primitivního datového typu.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <string>  #include "pole.h"  using namespace std;  class TOsoba  {  private:  string Jmeno;  string Prijmeni;  int Vek;  public:  TOsoba() : Vek(0) {}  TOsoba(string prijmeni, string jmeno, int vek)  : Prijmeni(prijmeni), Jmeno(jmeno), Vek(vek) {}  string dejJmeno() const { return Jmeno; }  string dejPrijmeni() const { return Prijmeni; }  int dejVek() const { return Vek; }  void nastavJmeno(string jmeno) { Jmeno = jmeno; }  void nastavPrijmeni(string prijmeni) { Prijmeni = prijmeni; }  void nastavVek(int vek) { Vek = vek; }  }; /\* Není v tomto případě nutné přetěžovat operátor = \*/  ostream &operator<<(ostream &os, const TOsoba &clovek)  {  if (clovek.dejPrijmeni() == "")  {  return os;  }  os << clovek.dejPrijmeni() << " " << clovek.dejJmeno()  << " " << clovek.dejVek() << endl;  return os;  }  /\* Definujeme si novy typ: \*/  typedef www\_builder\_cz::Array<TOsoba> TSeznamLidi;  int main(int argc, char \*\*argv)  {  TSeznamLidi abecedniSeznam('A','Z');  TOsoba osoba("Dostal","Radim",23);  abecedniSeznam['D'] = osoba;  osoba.nastavJmeno("Karel");  osoba.nastavPrijmeni("Novak");  osoba.nastavVek(55);  abecedniSeznam['N'] = osoba;  // Iterátory:  for(TSeznamLidi::iterator i = abecedniSeznam.begin();  i < abecedniSeznam.end(); i++)  {  cout << \*i;  }  return 0;  } | |

A na závěr celá šablona [array.h](http://www.builder.cz/data/array.h) ke stažení. Měla by být univerzálně použitelná na všech překladačích podporujících alespoň trochu ANSI normu. Jak je asi zřejmé psaní šablon je poněkud náročnější, než psaní běžných tříd, nebo funkcí. Hlavně je velice obtížné je ladit, protože mnoho dnešních ladících nástrojů má s šablonami mnoho problémů. Na druhou stranu používání šablon je velice výhodné. Ideální stav je tedy takový, kdy nám někdo šablonu již napíše, a dodá nám ji již funkční a hotovou. My ji jenom používáme. Autoři jazyka C++ si toho byli vědomi, proto zahrnuli do standardu jazyka knihovnu STL (Standard Template Library). Tato knihovna nabízí k použití mnoho již vytvořených šablon jednak datových kontejnerů, a jednak algoritmů pro práci s nimi. STL knihovna je (měla by být) k dispozici u každého překladače C++. Od příštího článku se začneme knihovně STL věnovat.

## [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/datove-kontejnery-v-c-uvod-do-stl-155907cz)

**9. října 2001, 00.00 | Dnes se podíváme na datové kontejnéry z knihovny STL jazyka C++. Datový kontejnér je objekt, který je schopen "skladovat" jiné objekty. Ukážeme si rozdělení a vlastnosti kontejnérů. Na příkladu si ukážeme použití kontejnéru stack a bitset.**

STL je zkratka slov Standard Template Library. Jedná se o standardní knihovnu šablon jazyka C++. Knihovna by měla být dodávána s každým překladačem C++. V STL můžeme najít mnoho různých a užitečných šablon. Mimo jiné zde jsou šablony datových kontejnerů, iterátory a šablony algoritmů, které pomocí iterátorů pracují s datovými kontejnery. Kompletní knihovnou STL se zabývat nebudeme. V několika článcích si uděláme něco jako přehled podstatných částí STL.

Datové kontejnery v C++

Pod pojmem datový kontejner si představme něco, co má schopnost "zapamatovat si" nějaké data. Data jsou v kontejneru nějakým způsobem uloženy. kontejnery v STL se dělí na posloupnosti a asociativní kontejnery. Posloupnosti jsou logické posloupnosti dat, které ale nemusejí být v paměti fyzicky za sebou. Asociativní kontejnery jsou kontejnery, kde k datům přistupujeme pomocí nějakého klíče. Způsob uložení dat, přístupu k nim je dán typem kontejneru. V knihovně STL jsem jich napočítal celkem 11. Snad jsem na žádný nezapomněl. Přehled datových kontejnerů je v následující tabulce.

|  |  |  |  |
| --- | --- | --- | --- |
| Název kontejneru | Typ kontejneru | Hlavičkový soubor | Popis kontejneru |
| bitset | posloupnost | bitset | Posloupnost bitů pevné délky. |
| deque | posloupnost | deque | Oboustranná fronta. Prvky lze vkládat, nebo odebírat z obou konců. Sice lze rovněž odebírat, nebo vkládat prvky na libovolné místo ve frontě (kontejner deque to umožňuje), ale tato operace není příliš efektivní. |
| list | posloupnost | list | Oboustranně zřetězený seznam. |
| map | asociativní kontejner | map | Asociativní pole. pole, které nemusí být indexováno celočíselným typem, ale čímkoliv. Třeba řetězcem. Pro daný klíč může existovat pouze 1 asociovaná hodnota. Tomuto kontejneru se budeme v budoucnu zabývat v samostatném článku. |
| multimap | asociativní kontejner | map | Asociativní pole. Pro daný klíč (index) může existovat více asociovaných hodnot. Tomuto kontejneru se budeme v budoucnu zabývat v samostatném článku. |
| multiset | asociativní kontejner | set | Multimnožina. množina, ve které se mohou prvky opakovat. Tomuto kontejneru se budeme věnovat později v samostatném článku. |
| priority\_queue | posloupnost | queue | Prioritní fronta. Fronta, ve které neplatí pravidlo "první dovnitř, první ven". Prvky, které se do fronty uloží jsou uspořádány podle nějaké relace. Dalo by se říci, že předbíhají ve frontě podle nějaké předem dané priority. |
| queue | posloupnost | queue | Klasické fronta. platí pravidlo, že prvek, který byl jako první vložen do fronty, z ní bude také první vybrán. |
| set | asociativní kontejner | set | Množina. Daná hodnota může být v množině obsažena jen jednou. Tomuto kontejneru se budeme věnovat později v samostatném článku. |
| stack | posloupnost | stack | Klasický zásobník. Platí pravidlo, že prvek, který byl vložen do zásobníku jako poslední bude vybrán jako první. |
| vector | posloupnost | vector | Obdoba jednorozměrného pole. Tomuto kontejneru se budeme věnovat později v samostatném článku. |

Každý kontejner je šablona třídy. Parametrem této šablony je typ prvků, které mají být v kontejneru uloženy. Pole s libovolným intervalem indexování, které jsem předvedl v minulém článku byl vlastně takový pokus o napsání vlastního datového kontejneru. Každý kontejner v STL obsahuje následující veřejné typy. "Typ" je zde parametr šablony, tedy typ prvků, které jsou v kontejneru uloženy.

 value\_type, což je přejmenování (pomocí typedef) typu Typ.

 reference, což je přejmenování (pomocí typedef) typu Typ&.

 const\_reference, což je přejmenování typu const Typ&.

 iterator, což je typ iterátor na prvky daného kontejneru. Iterátorům se budeme věnovat později.

 const\_iterator

 size\_type, což je přejmenování typu size\_t.

 difference\_type, což je přejmenování typu ptrdiff\_t.

Dále každý kontejner má přetížené operátory = == !- < >, má k dispozici bezparametrický a kopírovací konstruktor. Nemusíme se tedy ničeho bát, a klidně jakkoliv kontejner kopírovat, nebo porovnávat. Dále každý kontejner má následující veřejné metody:

 begin - touto metodou se budeme podrobně zabývat až ve článku o iterátorech.

 empty - vrací true, jestliže je kontejner prázdný.

 end - touto metodou se budeme podrobně zabývat až ve článku o iterátorech.

 max\_size - vrací maximální možnou velikost kontejneru.

 size - vrací aktuální velikost kontejneru.

 swap - zajistí výměnu prvků s jiným kontejnerem.

Další metody se v různých kontejnerech liší.

U typu, který má být parametrem kontejneru, musíme počítat s tím, že jeho instance budou kopírovány pomocí kopírovacího konstruktoru, případně pomocí operátoru =. U některých kontejnerů, mohou být prvky také porovnávány pomocí relačních operátorů. Proto nemohou-li být použity implicitní operátory je nutné je přetížit.

Dále je nutno upozornit, že všechny kontejnery (Nejen kontejnery, v podstatě vše z STL.) jsou deklarovány v prostoru jmen std.

Chce-li někdo proniknout do "tajů" kontejnerů, může si podrobně prostudovat hlavičkové soubory, ve kterých jsou deklarovány. Je v nich k dispozici úplný zdrojový text šablon. Rovněž se můžete podívat na můj velice jednoduchý ukázkový kontejner [array](http://www.builder.cz/rubriky/c/c--/clanek-155896-pole-s-libovolnym-intervalem-indexovani-v-c-/) z minulého článku.

Tolik tedy k přehledu o datových kontejnerech z STL jazyka C++. Některými kontejnery se budeme věnovat v samostatných článcích, dnes si letmo jen pro názornou ukázku ukážeme práci s dalšími kontejnery.

Musím ještě pro úplnost upozornit na tak zvané adaptéry. Některé kontejnery, které jsem zde uvedl, nejsou vlastně "skladiště dat". Jsou to třídy, které zapouzdří nějaký jiný kontejner a přizpůsobí (adoptují - adaptéry) ho jiným požadavkům. Například adaptér je stack, který jako svůj druhý parametr šablony vyžaduje nějaký kontejner. Implicitní hodnota tohoto parametru je deque. Adaptér kontejneru poznáme tak, že je u něj možnost jako další parametr šablony zadat jiný kontejner.

A nyní již jednoduchý příklad. Vytvoříme zásobník čísel int. Nějaké čísla do něj vložíme, a potom zase vybereme.

|  |  |
| --- | --- |
| |  | | --- | | #include <stack>  #include <iostream>  int main()  {  std::stack<int> zasobnik;  for(int i = 0; i < 10; i++)  {  zasobnik.push(i); // Metoda push uloží prvek na vrchol zásobníku.  }  std::cout << "Počet prvků je " << zasobnik.size() << endl;  while (!zasobnik.empty())  {  std::cout << zasobnik.top() << endl;  // Metoda top vrátí prvek na vrcholu zásobníku.  zasobnik.pop();  // Metoda pop odstraní prvek na vrcholu zásobníku.  }  return 0;  } | |

Kdybychom z nějakých důvodů nechtěli mít jako zásobník (stack) adoptovánu deque, ale například vector, potom by řádek definice objektu zásobník vypadal takto:std::stack<int,std::vector<int> >.

Dalším příkladem bude bitset, tedy posloupnost bitů. Tento kontejner má oproti všem ostatním jednu zvláštnost. Parametrem šablony není typ, ale počet bitů, které budou v kontejneru uloženy. Podívejme se na jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include <bitset>  #include <iostream>  int main()  {  std::bitset<32> bity;  for(int i = 0; i < 32; i++)  {  bity.set(i, (i % 2));  /\* Metoda set nastaví bit (jeho pořadí je dáno 1.parametrem)  na hodnotu druhého parametru. My zde máme nastaven každý lichý  bit na 1, sudý na 0. \*/  }  std::cout << bity.to\_string() << endl;  std::cout << (~bity).to\_string() << endl; /\* Operátor ~ provádí negaci \*/  std::cout << bity.to\_ulong() << endl;  /\*Metoda to\_ulong vrátí unsigned long, které reprezentují bity v poli. \*/  for(int i = 0; i < 32; i++)  {  if (bity[i])  { /\* Vypíši pořadí bitů s hodnotou 1 \*/  std::cout << i << endl;  }  }  try  {  std::cout << bity[1000] << endl;  /\*Pokud indexuji mimo zadanou horni mez indexovaní, bude vyvržena  vyjimka typu out\_of\_range. \*/  }  catch (std::out\_of\_range &vyjimka)  {  std::cerr << vyjimka.why() << endl;  }  return 0;  } | |

Tolik tedy pro úvod ke kontejnerům. V následujících článcích se budu některými kontejnery zabývat podrobněji. Nebudu zde ale uvádět výčet všech metod kontejnerů. Bylo by to naprosto zbytečné. Pro tyto účely slouží dokumentace ke knihovně. Jedna podle mého názoru pěkná dokumentace je na adrese <http://www.sgi.com/tech/stl/stl_index_cat.html>. Příště se podrobně budeme zabývat kontejnerem vector, což je jednorozměrné pole. V dalších článcích potom asociativním polem, a množinou.

## [Vector - datový kontejner v C++](http://www.builder.cz/rubriky/c/c--/vector-datovy-kontejner-v-c--155916cz)

**17. října 2001, 00.00 | Dnes se podíváme na šablonu třídy vector. Jedná se o datový kontejner z knihovny STL jazyka C++. Vektor je šablona jednorozměrného pole. Na rozdíl od "klasického" pole má vector, mnoho užitečných vlastností a služeb.**

Dnes se podíváme na šablonu třídy vector. Jedná se o datový kontejner z knihovny STL jazyka C++. Vektor je šablona jednorozměrného pole. Na rozdíl od "klasického" pole má vector, mnoho užitečných vlastností a služeb. Lze do něj například pomocí různých metod vložit prvek, a tím zvětšit jeho velikost.

Šablona vector je datový kontejner - posloupnost. Má tedy všechny prvky, které jsem ve svém minulém článku vyjmenoval pro datové kontejnery posloupnosti. Tento kontejner je deklarován v hlavičkovém souboru vector v prostoru jmen std. Podívejme se na velice jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  #include <iostream>  int main()  {  std::vector<int> a(7); // Pole 7 čísel  std::vector<int> b; // Pole 0 čísel  std::cout << "Počet prvků v a je " << a.size() << " ";  std::cout << "Počet prvků v b je " << b.size() << std::endl;  for(int i = 0; i < a.size(); i++)  { // "Normálně lze používat operátor []  a[i] = i + 1;  std::cout << "a[" << i << "] = " << a[i] << std::endl;  }  b = a; //Bez problémů lze použít operátor =, lze použít  //také kopírovací konstruktor  a.resize(15); // a má nyní 15 prvků  std::cout << "Počet prvků v a je " << a.size() << " ";  std::cout << a[13] << std::endl; // Náhodný  a.resize(3); // v a zůstalo jen prvních 3 prvků  for(int i = 0; i < a.size(); i++)  {  std::cout << "a[" << i << "] = " << a[i] << std::endl;  }  for(int i = 0; i < b.size(); i++)  {  std::cout << "b[" << i << "] = " << b[i] << std::endl;  }  if (a != b)  {  std::cout << "a != b " << std::endl;  }  return 0;  } | |

V příkladu je vidět základní práci s vektorem. Další zajímavé metody jsou back, push\_back, pop\_back. Metoda back vrátí referenci na poslední prvek. Metoda push\_back vloží nový prvek na konec. Ne tak, že by jím "přemazala" poslední prvek, ale zvýší velikost vektoru o 1 prvek (na konci). Metoda pop\_back naopak odstraní poslední prvek, čímž sníží velikost vektoru o 1. Například:

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  #include <iostream>  int main()  {  std::vector<int> a;  for(int i = 0; i < 5; i++)  {  a.push\_back(i+1);  std::cout << "Poslední prvek je: " << a.back() << std::endl;  }  for(int i = 0; i < a.size(); i++)  {  std::cout << a[i] << '\t';  }  std::cout << std::endl;  while (!a.empty()) // Opakuj, dokud není prázdný (Dokud nemá 0 prvků )  {  a.pop\_back(); // Odeberu poslední prvek  }  std::cout << "Velikost " << a.size() << std::endl;  return 0;  } | |

Další velmi užitečné metody pro vkládání a likvidování prvků jsou insert a erase. K jejich používání musíme ale vědět něco o iterátorech, kterým se budeme věnovat příště. Zatím se tedy těmto metodám nebudeme věnovat.

Podívejme se ještě jak udělat pomocí vektoru vícerozměrné pole. Vytvoříme vlastně vektor vektorů.

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  #include <iostream>  using namespace std;  int main()  {  vector<vector<int> > matice(3); //Matice 3 x 0  /\*Mezera mezi > a > je důležitá!  Viz moje předchozí články o šablonách. \*/  for(int a = 0; a < 3; a++)  {  matice[a].push\_back(a+1);  matice[a].push\_back(a+2);  matice[a].push\_back(a+3);  }  /\* Nyní máme matici 3 x 3 \*/  for(int y = 0; y < 3; y++)  {  for(int x = 0; x < 3; x++)  {  cout << matice[x][y] << '\t';  }  cout << endl;  }  return 0;  } | |

Chtěl bych je upozornit, že podobným způsobem lze vytvořit vícerozměrné pole i z mé šablony array, kterou jsem vytvořil v článku "[Pole s libovolným intervalem indexování v C++](http://www.builder.cz/rubriky/c/c--/clanek-155896-pole-s-libovolnym-intervalem-indexovani-v-c-/)".

Závěrem bych chtěl ještě připomenout, že každý typ, jehož proměnné chceme vkládat do vektoru musí být schopen vytvořit svou kopii pomocí kopírovacího konstruktoru a operátoru=. Nebudou-li vyhovovat implicitní kopírovací konstruktor a operátor =, musíme je vytvořit, resp. přetížit. Vyhneme se tím zbytečným a na první pohled nepochopitelným pádům programu. Dále typ, jehož prvky budeme vkládat do vektoru musí mít k dispozici bezparametrický konstruktor. Já jsem používal ve svých příkladech typ int, nebo v posledním vector. U typu intje vše potřebné již implicitně dáno. Instance šablony vector<> je třída, která má všechny potřebné konstruktory k dispozici a operátor = je přetížen.

V příštím článku se podrobněji podíváme na iterátory. Ukážeme si také další metody šablony vector. V dalších článcích se podíváme na podle mne také důležité kontejnery. Budou to mapa a multimapa (asociativní pole).

## [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/iteratory-v-c--155928cz)

**25. října 2001, 00.00 | Dnes si povíme něco o iterátorech. Iterátor je zobecněný ukazatel. Používá se při práci s STL kontejnery. Používáme-li iterátory, můžeme pracovat s kontejnerem, aniž by jsme věděli o jaký kontejnér se jedná. Ukážeme si také iterátory v datových proudech.**

Iterátor v C++ je vlastně taková obdoba ukazatelů pro kontejnery. Dříve, než se dostaneme k bližšímu vysvětlení pojmu iterátor, ukážeme si jednoduchý příklad, jak pracovat s obyčejným polem pomocí ukazatelů.

|  |  |
| --- | --- |
| |  | | --- | | int pole[20];  for (int \*temp = pole, temp != &pole[20]; temp++)  {  \*temp = 0;  } | |

Tento způsob není v C++ žádná novinka, známe ho z jazyka C. Druhá možnost je v každém kroku cyklu zvýšit nějakou celočíselnou hodnotu, která bude použita jako index v operátoru []. Uvedený příklad by ale měl být přeložen efektivněji, protože zde odpadá výpočet adresy prvku při vyhodnocení operátoru [].

V minulém článku jsme si ukázali šablonu jednorozměrného pole z STL - vector. Kdybychom chtěli s nějakým kontejnerem pracovat pomocí ukazatelů jako v našem příkladě s polem, neuspěli by jsme. Máme ale k dispozici iterátory. Iterátor je vlastně něco jako zobecněný ukazatel. U ukazatele operátor ++ způsobí, že ukazatel ukazuje na paměť bezprostředně následující za pamětí, na kterou ukazoval před vyhodnocením tohoto operátoru. V případě pole v našem příkladě se jedná vždy o následující prvek. Naopak operátor ++ u iterátoru způsobí vždy "přechod" na další prvek v kontejneru, nemusí se ale jednat o paměťové místo bezprostředně za předchozím místem. Chování iterátoru je tedy závislé na druhu kontejneru, který procházíme. Pro různé kontejnery jsou iterátory různé (různě implementovány), mají ale všechny stejné rozhraní (se všemi se pracuje stejně). V kontejnerech jsou metody begin a end. Metoda begin vrátí iterátor na první prvek. Metoda end vrátí iterátor za poslední prvek. Prvek na který ukazuje iterátor vrácený metodou end je neplatný (nealokován), stejně jako je neplatný v příkladu prvek pole[20]. Uveďme si obdobu našeho příkladu pro vector.

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  std::vector<int> vektor(20);  for (std::vector<int>::iterator temp = vektor.begin(), temp != vektor.end; temp++)  {  \*temp = 0;  } | |

Iterátor je tedy typ (třída), který má přetíženy operátory tak, aby se s ním pracovalo stejně jako s ukazatelem. Slouží k procházení kontejnerů. Nemusí se ale vždy jednat o nějakou třídu. Například ve velice jednoduchém příkladu kontejneru Array v mém článku ["Pole s libovolým intervalem indexování"](http://www.builder.cz/rubriky/c/c--/clanek-155896-pole-s-libovolnym-intervalem-indexovani-v-c-/) je iterátor vlastně ukazatel. Mohl jsem si dovolit použít ukazatel jako iterátor, protože jsem měl jistotu, že prvky v Array budou vždy fyzicky za sebou.

Výhoda iterátorů spočívá v tom, že pomocí iterátorů mohu pracovat s libovolným kontejnerem, aniž bych musel vědět o jaký kontejner se jedná. Lze tedy napsat obecný algoritmus použitelný pro libovolný kontejner. V době kdy píši tento algoritmus nemusím vědět, pro jaký kontejner bude použit. Může být také použit pro více typů kontejnerů bez úpravy. Až probereme kontejnery, podíváme se na standardní algoritmy v STL, které umějí spoustu operací s kontejnery. Přístup ke kontejnerům je pomocí iterátorů.

Kategorie iterátorů

Existuje několik kategorií iterátorů. Zde je jejich přehled:

|  |  |  |
| --- | --- | --- |
| Název | Český název | Charakteristika |
| InputIterator, OutputIterator | vstupní (čtecí) a výstupní (zapisovací) iterátory | operátory: -> , \* , = , ==, != , ++ Dereferencovaný vstupní iterátor nemůže být l-hodnota. tedy nemůže být "vlevo" při použití operátoru =. Dereferencovaný výstupní iterátor nemůže být p-hodnota. Algoritmy používající tyto iterátory mohou pomocí jednoho iterátoru procházet kontejner pouze jedenkrát. |
| ForwardIterator | dopředný iterátor | Kombinace vstupního a výstupního iterátoru. Pomocí jednoho iterátoru lze kontejner procházet vícekrát, ale jen v jednom směru. |
| BidirectionalIterator | obousměrný iterátor | Iterátor, který může kontejner procházet i v opačném směru. Tedy vlastně dopředný iterátor rozšířený o operátor -- |
| RandomaccessIterator | Iterátor s náhodným přístupem | Má v podstatě všechny možnosti jako ukazatel. Jedná se tedy o obousměrný iterátor rozšířený o operátory: + - < > <= >= [] |

Všechny iterátory dále mají bezparametrický, kopírovací konstruktor a destruktor. Na místě, kde je očekáván iterátor nějaké kategorie, lze dosadit iterátor požadované kategorie, nebo iterátor který je v této tabulce níže. Tedy například místo vstupního iterátoru lze dosadit také iterátor obousměrný bez potřeby nějakých konverzí, nebo přetypování. Iterátor s náhodným přístupem lze tedy dosadit kamkoliv. Za iterátor s náhodným přístupem lze považovat ukazatel na nějaký prvek v poli. Ukazatel lze tedy dosadit místo jakéhokoliv iterátoru. Iterátory kontejneru STL jsou alespoň vstupní, nebo výstupní. Ukážeme si jednoduchý příklad šablony funkce pro výstup do datového proudu:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <fstream>  #include <vector>  using namespace std;  template<class InputIterator>  int vypis(ostream &os, InputIterator zacatek, InputIterator konec)  {  register int pocetVypsanych = 0;  for(InputIterator i = zacatek; i != konec; i++)  {  os << \*i << endl; /\* Pro prvek, který bude v kontejneru  musí být přetížen operátor << \*/  pocetVypsanych++;  }  return pocetVypsanych;  }  /\* K napsání této funkce jsem nepotřeboval vědět  z čeho budu zapisovat - výhoda iterátorů  kam budu zapisovat - výhoda datových proudů  \*/  int main()  {  vector<int> vektor(20);  int pole[20], \*uk;  vector<int>::iterator iterator it;    for(it = vektor.begin(); it != vektor.end(); it++)  {  \*it = 20;  }  for( uk = pole; uk != &pole[20]; uk++)  {  \*uk = 10;  }  fstream souborovyProud("vystup.txt");  cout << "Do souboru jsem zapsal:"  << vypis(souborovyProud,vektor.begin(), vektor.end()) << endl;  vypis(cout,pole,&pole[20]);  return 0;  } | |

Parametrem šablony je nějaký typ. Já jsem tento typ nazval InputIterator. Jedná se ale pouze o název typu. Mohl jsem pochopitelně použít úplně jiný název (např. SchvalneSpatneOutputIterator) a vše by fungovalo. Já ale jako parametr šablony očekávám iterátor kategorie vstupní iterátor. Je nepsaným pravidlem, iterátory pojmenovávat tak, jak je uvedeno v tabulce ve sloupci název. Až budeme pracovat se standardními algoritmy z STL , nebo s metodami kontejnerů z STL budeme se setkávat s názvy uvedených v této tabulce. Pro přehlednost je dobré dodržovat tyto konvence i u vlastních šablon funkcí.

Dále je nutno také upozornit na existencí konstantních iterátorů. Konstantní iterátor je obdoba ukazatele na konstantu. Například:

|  |  |
| --- | --- |
| |  | | --- | | vector<int> a(100);  vector<int>::const\_iterator = a.begin(); | |

Dále také existují adaptéry iterátorů. Adaptér iterátoru přizpůsobí existující iterátor pro trochu jiné použití. Asi nejzajímavějším adaptérem iterátoru je tak zvaný reverzní iterátor. Reverzní iterátor je vlastně adaptován iterátor s náhodným přístupem. Operátor ++ u reverzního iterátoru má stejný význam jako operátor -- u iterátoru s náhodným přístupem a naopak. Reverzní iterátory jsou vhodné především tehdy, chceme-li aby již existující algoritmus pracující s kontejnerem v pořadí od prvního prvku k poslednímu pracoval beze změny s prvky v opačném pořadí. Kontejner vektor vrací reverzní iterátor metodou rbegin resp. rend.

Další zajímavou možností, o které je vhodné se v souvislosti s iterátory zmínit, jsou iterátory v datových proudech. V datových proudech existují vstupní (ve vstupních datových proudech) a výstupní (ve výstupních datových proudech) iterátory. Znamená to, že některé algoritmy napsané pro kontejnery z STL jsou použitelné i pro datové proudy. Jak je vidět, tak iterátory umožňují opravdu velmi obecnou práci s daty. například šablona funkce vypiš v mém příkladu dokáže pomocí proudových iterátorů také pracovat s datovými proudy. Uvedu příklad jak pomocí této šablony vypsat na stdout soubor, který předchozí příklad vytvořil, a také jak tento soubor kopírovat pomocí vypiš.

|  |  |
| --- | --- |
| |  | | --- | | // Kromě funkce main je vše stejné jako v mém příkladě s funkcí vypiš  int main()  {  ifstream vstup("vystup.txt");  /\* Asi trochu nevhodné pojmenování - vystup.txt  je výstup minulého programu. Je nutné předchozí program nejprve spustit.  Tak vytvoříme tento soubor. Zde pro zjednodušení nebudu ošetřovat případ,  že soubor neexistuje. \*/  ofstream kopie("kopie.txt");  istream\_iterator<int,ptrdiff\_t> proudovyIterator1(vstup), konec1;  /\* Vytvořen konstruktorem bez parametru se jedna o konec souboru, tedy  iterátor "za poslední prvek v souboru". \*/  vypis(cout,proudovyIterator1,konec1);  vstup.close();  vstup.open("vystup.txt");  istream\_iterator<int,ptrdiff\_t> proudovyIterator2(vstup), konec2;  vypis(kopie,proudovyIterator2,konec2);  /\* istream\_iterator je vstupní iterátor - jeden iterátor = jeden průchod \*/  return 0;  } | |

Parametrem šablony istream\_iterator je jednak typ prvků, které se budou načítat, a také typ pro vzdálenost mezi prvky. Typ pro proměnnou udávající vzdálenost mezi prvky je většinou ptrdiff\_t, což je s největší pravděpodobností signed int. Existuje také výstupní proudový iterátor. Jeho použití je obdobné. V jeho konstruktoru je jako další parametr možnost zadat řetězec, který bude oddělovat jednotlivé prvky v proudu. Při dereferencování proudových iterátorů se použijí operátory << resp. >>.

Tolik tedy k iterátorům. Původně jsem měl v úmyslu v tomto článku dokončit téma šablony vector. Bylo by dobré se podívat na metody erase a insert. Nezmínil jsem se o nich minule, protože jejich pochopení je potřeba vědět něco o iterátorech. Článek by byl ale příliš dlouhý, proto téma vector dokončíme příště.

## [Šablona vector v C++ a iterátory](http://www.builder.cz/rubriky/c/c--/sablona-vector-v-c-a-iteratory-155932cz)

**2. listopadu 2001, 00.00 | Dříve jsme zabývali šablonou vector. Neukázali jsme si ale metody insert a erase. Těmto metodám se dnes budeme věnovat. Ukážeme si jak efektivně kopírovat do kontejneru pole. A jak do kontejnereru ukládat objekty, které pro ukládání nesplńují podmínky.**

V předminulém článku jsme se zabývali šablonou vector. Neukázali jsme si ale metody inserta erase. Právě těmto metodám se dnes budeme věnovat. Také si ukážeme jak efektivně nakopírovat do kontejneru obyčejné pole, nebo obsah jiného kontejneru. A jak přimět kontejner aby pracoval i s instancemi tříd, které nesplňují podmínky pro uložení v kontejneru.

Metoda insert

Metoda insert slouží k vkládání prvků do vektoru. V žádném případě se nejedná o nějakou obdobu operátoru [], nebo metody at. Metoda insert žádný prvek vektoru nepřepíše. Vkládání prvku pomocí insert vypadá tak, že všechny prvky od pozice na kterou chci zapisovat až po konec se posunou "doprava", tedy jejich index bude zvýšen o 1. Poté se nový prvek zapíše na novou pozici. Vektor se při této operaci zvětší o jeden prvek. Kdybychom chtěli něco takového provést v poli, museli by jsme velikost pole zvětšit (nejspíše pomocí realloc), poté překopírovat část prvků na nové pozice, a nakonec zapsat nový prvek. V kontejneru vector je všechna tato činnost již naprogramovaná a ukrytá v metodě insert. Metoda insert je 3 krát přetížena (existují 3 varianty této metody).

    void insert(iterator pozice, const Typ& novyPrvek = Typ());    První možnost je předat metodě insert dva parametry. První je iterátor určující pozici pro vložení prvku. Druhý parametr je vkládaný prvek. Implicitní hodnota pro druhý parametr je prvek vytvořený bezparametrickým konstruktorem. Znamená to, že zavoláme-li metodu insert pouze s parametrem iterátoru, bude na danou pozici vložen prvek vytvořený bezparametrickým konstruktorem. Ještě jen zbývá podotknout, že identifikátor Typ je typ prvků, které jsou ve vektoru vloženy. Identifikátor iterator je veřejný vnitřní typ daného kontejneru. V našem případě se jedná o kontejner vector, ale i další kontejnery mají vnitřní typ iterator. Viz moje předchozí články.

    void insert(iterator pozice, size\_type n, const Typ& novyPrvek = Typ());    Další možností je předat metodě insert navíc číslo udávající počet nových prvků. V tomto případě nebude vložen jeden prvek (novyPrvek), ale n jeho kopií.

    template<class InputIterator> void insert(iterator pozice, InputIterator zacatek, InputIterator konec);     Zde na pozici danou prvním parametrem vložíme prvky, které jsou v jiném kontejneru mezi iterátory zacatek a konec. Prvek, na který se odkazuje iterátor zacatek bude vložen, prvek na který se odkazuje iterátor konec již vložen nebude. Pojem vstupní iterátor (InputIterator) jsem vysvětlil ve svém předchozím článku [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/).  
  
Metoda erase

Oproti metodě insert metoda erase prvky z kontejneru odebírá. Existují dvě varianty:

    erase(iterator pozice);    Zruší prvek daný iterátorem.

    erase(iterator zacatek, iterator konec);    Zruší všechny prvky v oblasti dané iterátory zacatek a konec. Prvek, na který se odkazuje iterátor zacatek bude odstraněn, prvek na který se odkazuje iterátor konec již odstraněn nebude.

Další věc, o které bych se chtěl v souvislosti s vektorem zmínit je konstruktor, který má dva parametry - vstupní iterátory. Tento konstruktor vytvoří vektor, který bude obsahovat kopie prvků mezi zadanými iterátory jiného kontejneru. Často se tento konstruktor používá pro vytvoření vektoru z obyčejného pole. Vše si podrobně ukážeme v příkladě.

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  #include <iostream>  using namespace std;  int main()  {  int pole[10];  for (int p = 0; p < 10; p++)  {  pole[p] = p;  }  vector<int> v1(pole,&pole[10]);  /\* Misto iterátoru ukazatel - viz předchozí článek \*/  vector<int> v2(v1.begin()+6,v1.end());  cout << "Velikost v1: " << v1.size() << " velikost v2: "  << v2.size() << endl;  for (vector<int>::iterator p = v1.begin(); p != v1.end(); p++)  {  cout << \*p << " ";  }  cout << endl;  for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++)  {  cout << \*p << " ";  }  cout << endl;  v1.insert(v1.begin(),-100); //Vložím před první prvek -100  v1.insert(v1.end(),3,500); // Vložím na konec 3 krát 500  v2.insert(v2.begin()+2,v1.begin(),v1.end());//Vložím celý vektor v1 do v2  cout << "Velikost v1: " << v1.size() << " velikost v2: "  << v2.size() << endl;  for (vector<int>::iterator p = v1.begin(); p != v1.end(); p++)  {  cout << \*p << " ";  }  cout << endl;  for (vector<int>::iterator p = v2.begin(); p != v2.end(); p++)  {  cout << \*p << " ";  }  cout << endl;  v2.erase(v2.begin(), v2.end()); // Mažu všechny prvky ve v2  cout << "Velikost v2: " << v2.size() << endl;  while (!v1.empty())  {  cout << "Mažu " << \*v1.begin() << " z v1" << endl;  v1.erase(v1.begin());  }  cout << "Velikost v1: " << v1.size() << endl;  return 0;  } | |

Kontejner a typ prvků, který neodpovídá podmínkám pro vkládání do kontejneru

Ve svém článku [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/clanek-155907-datove-kontejnery-v-c-uvod-do-stl/) jsem uvedl podmínky pro typ, jehož instance chceme do kontejneru ukládat. Právě u metod insert a erase vidíme, proč je u vkládaného objektu důležitá schopnost sebe kopírovat. Mnohdy ale můžeme pracovat s třídou, která nemá kopírovací konstruktor, resp. operátor = přetížen a implicitní použít nelze. Viz můj článek [Kopírovací konstruktor v C++](http://www.builder.cz/rubriky/c/c--/clanek-155685-kopirovaci-konstruktor-v-c-/). Také je možné, že kopírovací konstruktor, či operátor = je deklarován jako soukromá položka. Tím autor třídy dal jasně najevo, že kopírování instancí je zakázáno. Jak pracovat s takovými typy v kontejnerech? Řešení ukážu na kontejneru vector, protože ten již známe. Tento postup je ale použitelný pro libovolný kontejner.

Celá finta spočívá v tom, že do kontejneru nebudeme ukládat samotné prvky, ale pouze ukazatele na ně. Vytvoříme si ukázkovou třídu, která nesplňuje podmínky pro to, aby její instance byly ukládány do kontejneru.

|  |  |
| --- | --- |
| |  | | --- | | class TridaINT  {  private:  int \*prvek;  public:  TridaINT(): prvek(new int) {};  TridaINT(int cislo): prvek(new int) {\*prvek=cislo;};  ~TridaINT() { delete prvek; };  int dejPrvek() const { return \*prvek; };  void nastavPrvek(const int novy) { \*prvek = novy; };  }; | |

U této třídy nelze korektně použít implicitní kopírovací konstruktor ani operátor= - povede to k pádu programu. Také si můžete vyzkoušet, že k nekorektnímu chování dojde i tehdy, jsou-li instance této třídy vkládány, či odebírány z libovolného kontejneru.

Kontejner ukazatelů

Ukazatel vždy splňuje podmínky pro uložení v kontejneru. Můžeme vytvořit kontejner ukazatelů, musíme mít ale neustále na mysli dvě věci. Jednak nesmíme do kontejneru uložit ukazatel na lokální objekt (lokální zásobník). Všechny objekty, na které ukazující ukazatelé budou v nějakém kontejneru by měly být vytvořeny pomocí new. Dále si také musíme uvědomit, že při likvidaci kontejneru (volání jeho destruktoru) budou zlikvidovány jeho prvky - ukazatele, nebudou ale zlikvidovány samotné objekty, na které bylo z kontejneru odkazováno. O zavolání jejich destruktorů se musíme postarat sami. Například:

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <vector>  using namespace std;  int main()  {  vector<TridaINT\*> cisla(3);  cisla[0] = new TridaINT(1);  cisla[1] = new TridaINT(2);  cisla[2] = new TridaINT;  cisla[2]->nastavPrvek(3); /\* cisla[2] je ukazatel, proto -> \*/  for(vector<TridaINT\*>::iterator i = cisla.begin(); i != cisla.end(); i++)  {  cout << (\*i)->dejPrvek() << endl;  /\*  i je iterátor.  \*i je ukazatel - prvek kontejneru na který se odkazuje iterátor.  \*\*i je objekt na který ukazuje ukazatel daný iterátorem  (Obdoba ukazatele na ukazatel). Vlastně je to \*(i->operator\*()).  (\*i)-> je přístup k metodám a atributům objektu, na  který se odkazuje ukazatel \*i daný iterátorem i.  &\*i je skutečná adresa ukazatele \*i (ukazatel na ukazatel).  Vlastně je to &(i->operator\*()).  Nedoporučuji nikde používat!  \*/  }  for(vector<TridaINT\*>::iterator i = cisla.begin(); i != cisla.end(); i++)  {  delete \*i; /\* V tomto případě musíme provést sami! \*/  }  return 0;  } | |

Tolik tedy k vektoru. V mnoha případech je použití vektoru pohodlnější, než použití pole. Program pracující s vektorem je sice trochu pomalejší (oproti programu pracujícím s polem), ale velkého zpoždění se bát nemusíme. Žádná metoda žádného kontejneru není volána pozdní vazbou, a některé jsou dokonce překládány jako inline. Také je fakt, že kontejnery jsou šablony, tedy značná část operací se provede již v době kompilace, při běhu programu se na rychlosti nijak neprojeví. Příště se podíváme na kontejnery map a multimap, což jsou asociativní pole.

[Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/asociativni-pole-v-c--155938cz)

**9. listopadu 2001, 00.00 | Dnes se podíváme na šablony map a multimap. Jedná se o asociativní pole. V asociativním poli jsou uloženy hodnoty ve tvaru (klíč,hodnota). Klíčem může být libovolný objekt, například string. Ukážeme si jak udělat jednoduchou hash tabulku v C++.**

Dnes se podíváme na šablony map a multimap. Jedná se o asociativní pole. V asociativním poli jsou uloženy hodnoty ve tvaru (klíč,hodnota), kde klíč je vlastně "index" prvku. Klíčem může být libovolný objekt, například string.

Šablony map i multimap jsou kontejnery z knihovny STL (Viz můj článek [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/clanek-155907-datove-kontejnery-v-c-uvod-do-stl/)). Jedná se o asociativní kontejnery. Jak ukládané prvky, tak i klíče musejí mít k dispozici:

* Bezparametrický konstruktor
* Musejí být schopny se kopírovat pomocí kopírovacího konstruktoru a operátoru =. Nemohou-li být použity implicitní, musí se operátor = přetížit, a vytvořit kopírovací konstruktor.
* Destruktor

Oba kontejnery mají vnitřní typ value\_type. Jedná se o šablonu struktury jménem pair. Tato šablona je přejmenovaná pomocí typedef. Šablona struktury pair je struktura parametrizovaná dvěma typy - klíčem a hodnotou. K vytvoření "páru" můžeme použít konstruktor této šablony, nebo pomocí šablony funkce make\_pair. Identifikátory pair a make\_pair jsou deklarovány v prostoru jmen std. Jak v šabloně map, tak i v multimap jsou vlastně uloženy tyto "páry" (klíč,hodnota). Ke každé hodnotě se přistupuje pomocí klíče. Rozdíl mezi map a multimap je v tom, že map umožňuje mít pro jeden klíč pouze jednu hodnotu, kdežto multimap může mít pro jeden klíč více asociovaných hodnot. Oba kontejnery jsou deklarovány v hlavičkovém souboru map.h v prostoru jmen std.

Šablona map

Šablona map má tři parametry. Jedná se o typ klíče, typ prvku, a "něco", co porovnává klíče. Třetí parametr má svou implicitní hodnotu, a my jej zatím nebudeme používat. Kdyby někdo chtěl zajistit, aby klíče nebyly porovnávány pomocí relačního operátoru <, zadal by třetí parametr. Třetí parametr může být funkce (ukazatel na funkci), nebo tak zvaný funkční objekt. Funkčními objekty se chci zabývat v budoucnu v samostatném článku. K pochopení šablony map nejsou důležité, proto se jimi dnes nebudeme zabývat. V dnešním článku budeme používat pouze první dva parametry šablon map a multimap. Šablona map má k dispozici konstruktory a destruktor jako všechny ostatní kontejnery (Viz moje předchozí články). K jednotlivým prvkům se přistupuje pomocí klíčů. Operátor indexování [] má jako svůj parametr také hodnotu klíče. Všechny páry v kontejneru jsou vždy seřazeny podle hodnoty svého klíče. Podívejme se nyní na některé metody:

* begin, empty, size, max\_size, swap - metody společné pro všechny kontejnery z STL. Viz můj článek [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/clanek-155907-datove-kontejnery-v-c-uvod-do-stl/)
* find - parametrem je klíč, metoda vrátí pár (klíč,hodnota), ve ktérém klíč odpovídá klíči, který byl zadán jako parametr. Není-li takový pár v kontejneru, metoda vrátí iterátor za poslední prvek (iterátor, který vrací metoda end()).
* count - parametrem je klíč. Metoda vrátí 1, jestliže v kontejneru existuje hodnota asociována k danému klíči, jinak vrátí 0. Vlastně vrací počet párů s daným klíčem.
* insert - existují opět 3 varianty této metody (tak jako u šablony vector. Jedna možnost je předat metodě insert jako parametr objekt typu value\_type. Můžeme jej například vytvořit pomocí šablony funkce std::make\_pair. V případě, že v kontejneru neexistuje pár s klíčem daným jako parametr, bude zadaný pár do kontejneru vložen. Jestliže daný klíč již v kontejneru existuje, nestane se nic. Poloha pro nový prvek bude bude vybrána tak, aby kontejner zůstal uspořádán pomocí hodnot klíčů. Další dvě varianty metody insert jsou obdoné jako u šablony vector (Viz můj článek [Šablona vector v C++ a iterátory](http://www.builder.cz/rubriky/c/c--/clanek-155932-sablona-vector-v-c-a-iteratory/)). Lze pomocí iterátor upřesnit polohu, nebo vložit prvky s jiného kontejneru. Upřesníme-li polohu iterátorem, kontejner si zkontroluje, zda je poloha správná. Kontejner musí být uspořádán podle klíčů. Je-li správná, potom dojde ihned k vložení páru. Ušetří se čas, který by kontejner potřeboval k nalezení správného místa. V opačném případě kontejner nalezne vhodné místo sám. Zachová se tedy, jako by jsme polohu iterátorem vůbec nezadali.
* erase - existují dvě varianty této metody. Mohu zadat klíč nějaké asociované hodnoty, nebo iterátor udávající polohu páru. Metoda erase odstraní prvek (i s klíčem, tedy celý pár) z kontejneru.
* Operátor [] - slouží k přístupu k hodnotám pomocí klíčů. Není-li v kontejneru pár s daným klíčem, bude vložen pár s daným klíčem, a hodnotou vytvořenou bezparametrickým konstruktorem. Takové chování nemusí být vždy žádoucí (viz příklad).

To je výčet asi nejdůležitějších metod. Nyní si vše osvětlíme na příkladu

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <map>  #include <string>  int main(int argc, char \*\*argv)  {  std::map<std::string,int> TelefoniSeznam;  /\* Klíč je string, hodnota je int \*/  TelefoniSeznam.insert(std::make\_pair(std::string("Dostal"),4578963));  TelefoniSeznam.insert(std::make\_pair(std::string("Novak"),12458632));  TelefoniSeznam.insert(std::make\_pair(std::string("Hasici"),150));  TelefoniSeznam.insert(std::make\_pair(std::string("Policie"),158));  std::cout << "Hasici " << TelefoniSeznam[std::string("Hasici")] << std::endl;  std::cout << "Dostal Radim " << TelefoniSeznam[std::string("Dostal")]  << std::endl;  std::string jmeno("Bond");  std::cout << "Bond James ";  if (!TelefoniSeznam.count(jmeno))  std::cout << "neni v seznamu." << std::endl;  else  std::cout << TelefoniSeznam[jmeno] << std::endl;  std::cout << "Muzete se o tom presvedcit: " << TelefoniSeznam[jmeno]  << std::endl;  /\* Právě jsem zapsal Bonda do seznamu!  Takové chování operátoru [] nemusí být vždy žádoucí.  Může být lepší raději použít metodu find. \*/  std::map<std::string,int>::iterator i;  for(i = TelefoniSeznam.begin(); i != TelefoniSeznam.end(); i++)  {  /\* Bond je tady taky! \*/  std::cout << i->first << " " << i->second << std::endl;  /\* Šablona struktury pair má dva atributy: first a second \*/  }  TelefoniSeznam.erase(TelefoniSeznam.begin());  TelefoniSeznam.erase(std::string("Novak"));  std::cout << std::endl;  for(i = TelefoniSeznam.begin(); i != TelefoniSeznam.end(); i++)  {  std::cout << i->first << " " << i->second << std::endl;  }  return 0;  } | |

Doufám, že tentokrát jsem nikde nezapoměl na prostor jmen std. Dobře si všimněte, jak se v kontejneru objevil Bond. Použil jsem ho jako parametr pro operátor [].

Šablona multimap

Šablona multimap je asociativní pole, které umožní mít k jednomu klíči asociováno několik hodnot. Práce se šablonou multimap je podobná jako práce se šablonou map. Zaměřím se pouze na rozdíly. První důležitá věc je, že šablona třídy multimap již nemá operátor []. Prvky s daným klíčem lze nalézt pomocí metody find. Tím ale nalezneme pouze jeden (ten první) prvek s daným klíčem. Chceme-li nalézt všechny prvky s daným klíčem, musíme použít metody lower\_bound, upper\_bound, nebo equal\_range. Tyto metody byly i v šabloně map, ale tam neměly snad žádné využití.

* lower\_bound - parametrem je klíč. Vrací iterátor na prvek s nejmenším klíčem, který je větší, nebo roven zadanému klíči. Neexistuje-li takový prvek, vrací iterátor za konec kontejneru ( end() ).
* upper\_bound - parametrem je klíč. Vrací iterátor na prvek s největším klíčem, který je menší, nebo roven zadanému klíči. Neexistuje-li takový prvek, vrací iterátor za konec kontejneru ( end() ).
* equal\_range - parametrem je klíč. Vrací dvojci iterátorů (pair - viz příklad níže). První ve dvojci je výsledek volání metody lower\_bound. Druhý ve dvojci je výsledek volání metody upper\_bound. Metoda equal\_range za nás vlastně zavolá tyto dvě metody.

Nyní si vytvoříme ukázkový příklad. Bude se jednat o jednoduchou ukázkovou hash tabulku s velice jednoduchou hash funkcí - mod 13. V tabulce budou uloženy čísla int. Jejich klíče budou také typu int. Klíče jsou int pouze pro přehlednost tohoto ukázkového zdrojového textu. Pro úsporu místa by bylo asi lepší mít klíče short int, nebo i char. Hodnota klíče totiž nebude nikdy větší než 12.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <map>  using namespace std;  typedef multimap<int,int> THashTabulka;  int main(int argc, char \*\*argv)  {  THashTabulka Tabulka;  int p;  THashTabulka::iterator i;  for(p = 39; p >= 0; p--)  {  Tabulka.insert(make\_pair(p % 13, p));  }  cout << "Pocet prvku v tabulce " << Tabulka.size() << endl;  for(i = Tabulka.begin(); i != Tabulka.end(); i++)  {  cout << "(" << i->first << "," << i->second << ")\t";  }  cout << endl;  cout << "PocetPrvku s klicem 10 je " << Tabulka.count(10)  << ". Prvni je: " << (\*Tabulka.find(10)).second << endl;  i = Tabulka.find(5);  cout << "PocetPrvku s klicem 5 je " << Tabulka.count(5) << endl;  for(p = 0; p < Tabulka.count(5); p++,i++)  {  cout << "(" << i->first << "," << i->second << ")\t";  }  cout << endl;  /\* Lepší způsob je: \*/  THashTabulka::iterator j = Tabulka.upper\_bound(0);  cout << "Prvky s indexem 0:" << endl;  for(i = Tabulka.lower\_bound(0); i != j; i++)  {  cout << "(" << i->first << "," << i->second << ")\t";  }  cout << endl;  /\* Asi nejlepší způsob je: \*/  pair<THashTabulka::iterator,THashTabulka::iterator>  par=Tabulka.equal\_range(7);  cout << "Klici s indexem 7 odpovida:" << endl;  for(i = par.first; i != par.second; i++)  {  cout << i->second << "\t";  }  cout << endl;  return 0;  } | |

Tolik tedy k asociativním polím. Až se budeme zabývat tématem funkčních objektů v C++, připomeneme si, že i v šablonách map a multimap se dají funkční objekty použít. Je to v případě, kdy nechcete, aby kontejner třídil klíče podle relačního operátoru <, ale podle jiné, Vámi zadané relace. Příště se podíváme na množinu a multimnožinu v C++.

## [Množina v C++](http://www.builder.cz/rubriky/c/c--/mnozina-v-c--155952cz)

**26. listopadu 2001, 00.00 | Dnes si povíme něco o množině a multimnožině v C++. Množina je datová struktura, ve které jsou uloženy prvky. V množině, na rozdíl od multimnožiny, nesmí být dva stejné prvky. Nad množinou lze provádět různé operace. Vše je součástí STL jazyka C++.**

Dnes si povíme něco o množině a multimnožině v C++. Množina je datová struktura, ve které jsou uloženy nějaké prvky. V množině nesmí být dva stejné prvky. Naopak multimnožina může obsahovat i stejné prvky. Nad množinou lze provádět množinové operace jako jsou například sjednocení a průnik. Pro množinu a multimnožinu existují v C++ šablony set a multiset.

Šablony set a multiset jsou setříděné asociativní kontejnery. Na rozdíl od šablon map amultimap v nich nejsou uloženy dvojce (klíč, hodnota), ale jsou zde uloženy pouze klíče. O kontejnerech map a multimap jsem se zmiňoval ve svém předchozím článku [Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/clanek-155938-asociativni-pole-v-c-/).

Množina

Množina je datová struktura ve které jsou uloženy jednotlivé prvky. Prvky v množině nesmí být stejné. Množina je v C++ reprezentována šablonou set. Šablona set má jeden povinný parametr, který udává typ prvků v množině, a jeden nepovinný parametr, který udává způsob, jakým se budou prvky v množině porovnávat. Nezadáme-li druhý parametr, bude se používat operátor <. Druhým parametrem může být ukazatel na funkci, nebo tak zvaný funkční objekt. Prozatím nebudeme druhý parametr používat. Příští článek bude na téma funkční objekty v C++, potom si povíme něco více o těchto možnostech. Jak je zatím vidět, šablona set se přece jenom trochu liší od množiny, jakou známe z matematiky. Prvním rozdílem je, že v šabloně set mohou být uloženy pouze prvky stejného typu. Mohou zde být i instance různých tříd, ale tyto třídy musejí mít stejného předka (musejí být odvozeny ze stejné třídy). Druhý zásadní rozdíl je v tom, že pro prvky v množině jsou vždy seřazené. Musí být tedy definováno porovnání pomocí operátoru <, nebo jiné porovnání (které se zadá právě tím druhým, nepovinným parametrem). Důležitý je fakt, že je nutné, aby prvky v kontejneru byly seřazeny podle nějaké relace. Kontejner totiž setřídění prvků využívá při vyhledávání a ukládání, kdy použije rychlejší algoritmus.

Šablona set má metody velmi podobné jako šablona map z minulého článku. Pro vkládaní prvků zde existují metody insert. Pro odstranění prvků zase erase. Existují zde metody begina end pro práci s iterátory atd... Myslím, že nemá smysl zde význam těchto metod znovu vypisovat. Všechny jsou popsány v mých předchozích článcích. Prvky, které se budou do množiny ukládat musejí být schopny vytvářet kopie pomocí kopírovacího konstruktoru, a operátoru =. Budeme-li chtít zjistit, jestli se nějaký prvek nachází v množině použijeme metodu count, která vrátí 0, jestliže prvek v množině není, jinak vrátí 1. Tato metoda vlastně vrací počet prvků v množině, což je 0, nebo 1. U multimnožiny může vrátit 0, nebo celé kladné číslo. V souvislosti s množinou bych ale chtěl upozornit na některé množinové operace, které jsou v STL implementovány. V STL jsou tak zvané standardní algoritmy, což jsou šablony funkcí pracujících s kontejnery. Standardním algoritmům se chci věnovat v, samostatném článku. Dnes si v souvislosti s množinou ukážeme pouze 4 algoritmy - množinové operace.

Operace nad množinou

Průnik množin A a B je množina, jejíž prvky jsou v A a zároveň v B. Sjednocení množin A a B je množina, jejíž prvky jsou v A, nebo v B. Množinový rozdíl množin A a B je množina, jejíž prvky jsou v A, a zároveň nejsou v B. Symetrická diference dvou množin je množina, která je Sjednocení(A,B) - Průnik(A,B), kde symbol - značí množinový rozdíl. Snad není tento zápis moc matoucí, ale já si vůbec nevím rady s tím, jak napsat v HTML symboly pro množinové operace.

|  |  |
| --- | --- |
| **operace** | **algoritmus** |
| průnik | set\_intersection |
| sjednocení | set\_union |
| množinový rozdíl | set\_difference |
| symetrická diference | set\_symmetric\_difference |

Všechny algoritmy jsou šablony funkcí. Všechny šablony mají 3 parametry. První dva jsou typy vstupních iterátorů obou množin. Třetí parametr je typ výstupního iterátoru výsledku. Každá funkce má 5 parametrů. Nejlépe bude napsat deklaraci jedné z nich:  
template<class InputIterator\_1, class InputIterator\_2, class OutputIterator > OutputIterator set\_union(InputIterator\_1 zacatekPrvniho, InputIterator\_1 konecPrvniho, InputIterator\_2 zacatekPrvniho, InputIterator\_2 konecPrvniho, OutputIterator zacatekVysledku);  
První dva parametry funkce udávají začátek a konec první množiny. Třetí a čtvrtý parametr udává začátek a konec druhé množiny. První a druhá dvojce parametrů nejsou stejného typu. Znamená to, že nemusíme provádět sjednocení pouze množin (set), ale i jiných kontejnerů. Ale pokud použijeme jiné kontejnery, musejí být setříděné. Posledním parametrem je výstupní iterátor výsledku. Dalším důležitým algoritmem je algoritmus includes, který určí, zda jedna množina je podmnožinou druhé. Šablona includes má 2 parametry. Jsou to typy vstupních iterátorů první a druhé množiny. Parametry funkce jsou 4. Začátek a konec obou množin. Všechny tyto algoritmy jsou deklarovány v hlavičkovém souboru algorithm v prostoru jmen std. Nejlépe bude, ukážeme-li si příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include <string>  #include <set>  #include <algorithm>  #include <iterator>  using namespace std;  template<class InputIterator> void vypis(ostream &os, InputIterator zacatek,  InputIterator konec)  {  /\* Funkci jsem si vypůjčil ze svého článku o iterátorech \*/  for(InputIterator i = zacatek; i != konec; i++)  {  os << \*i << " ";  }  cout << endl;  }  int main(int argc, char \*\*argv)  {  set<string> ctyruhelniky, stejneDlouhe, vysledek;  ctyruhelniky.insert(string("ctverec"));  ctyruhelniky.insert(string("obdelnik"));  ctyruhelniky.insert(string("kosoctverec"));  ctyruhelniky.insert(string("lichobeznik"));  ctyruhelniky.insert(string("rovnobeznik"));  stejneDlouhe.insert(string("ctverec"));  stejneDlouhe.insert(string("kosoctverec"));  stejneDlouhe.insert(string("rovnostranny trojuhelnik"));  cout << "Vypis ctyruhelniku:" << endl;  vypis(cout,ctyruhelniky.begin(),ctyruhelniky.end());  cout << "Vypis rovinnych obrazcu, ktere maji stejne velikosti svych stran:"  << endl;  vypis(cout,stejneDlouhe.begin(),stejneDlouhe.end());  insert\_iterator<set<string> > iter(vysledek,vysledek.begin());  /\* Sjednocení: \*/  set\_union(ctyruhelniky.begin(),ctyruhelniky.end(),stejneDlouhe.begin()  ,stejneDlouhe.end(),iter);  cout << "Vypis \"vsech\" rovinnych obrazcu:" << endl;  vypis(cout,vysledek.begin(),vysledek.end());  vysledek.erase(vysledek.begin(),vysledek.end());  /\* Průnik: \*/  set\_intersection(ctyruhelniky.begin(),ctyruhelniky.end(),stejneDlouhe.begin()  ,stejneDlouhe.end(),iter);  cout << "Vypis rovnostrannych ctyruhelniku:" << endl;  vypis(cout,vysledek.begin(),vysledek.end());  vysledek.erase(vysledek.begin(),vysledek.end());  /\* Rozdíl: \*/  set\_difference(ctyruhelniky.begin(),ctyruhelniky.end(),stejneDlouhe.begin()  ,stejneDlouhe.end(),iter);  cout << "Vypis nerovnostranných ctyruhelniku:" << endl;  vypis(cout,vysledek.begin(),vysledek.end());  vysledek.erase(vysledek.begin(),vysledek.end());  /\* Symetrická diference: \*/  set\_symmetric\_difference(ctyruhelniky.begin(),ctyruhelniky.end(),  stejneDlouhe.begin(),stejneDlouhe.end(),iter);  cout << "Vypis \"vsech\" rovinnych obrazcu,  krome rovnostranych ctyruhelniku:" << endl;  vypis(cout,vysledek.begin(),vysledek.end());  vysledek.erase(vysledek.begin(),vysledek.end());  /\* Je čtverec čtyřúhelník? (Náleží čtverec množině čtyřúhelníků?) \*/  if (ctyruhelniky.count(string("ctverec")) == 1)  {  cout << "Ano" << endl;  }  /\* Je autor článku čtyřúhelník ? \*/  if (ctyruhelniky.count(string("Radim Dostál")) == 0)  {  cout << "Neni" << endl;  }  if (!includes(ctyruhelniky.begin(),ctyruhelniky.end(), stejneDlouhe.begin()  , stejneDlouhe.end()))  {  cout << "Rovnostranne plosne obrazce nejsou podmnozinou ctyruhelniku"  << endl;  }  return 0;  } | |

V článku jsem použil zmiňované množinové operace. Také jsem ukázal jak zjistit, zda nějaký prvek náleží do množiny, nebo ne. Šablona set má metody begin a end vracející konstantní iterátory. Konstantní iterátor nelze použít jako výstupní, proto nemůže být pátým parametrem funkcí set\_.... parametr vysledek.begin(). Kdybych pracoval s kontejnerem, který nemá metodu begin pouze konstantní (viz další příklad) musel bych řešit další problém. Musel bych mít ve výsledném kontejneru vytvořeno místo pro výsledné prvky. Abych se vyhnul těmto problémům použil jsem adaptér jménem insert\_iterator. Jedná se o šablonu třídy. Parametrem šablony je typ kontejneru, do kterého se bude iterátor "odkazovat". Parametry konstruktoru jsou konkrétní instance kontejneru, a pozice v kontejneru, na kterou se bude zapisovat. Takto vytvořený objekt se pro své okolí jeví jako výstupní iterátor. Může se kdykoliv jako výstupní iterátor použít. Zavolá-li se pro něj operátor = ,tedy zapisuje-li se na danou pozici, objekt třídy insert\_iterator vloží do "svého" kontejneru nový prvek pomocí metody kontejneru insert. Tím vytvoří v kontejneru místo pro nový prvek, a na toto místo nový prvek zapíše. Všechna tato činnost je zapouzdřena v metodě insert\_iterator::operator=(const typename Container::value\_type& novaHodnota), a uživatel třídy se o ní nemusí starat. Container je parametr šablony, value\_type je typ prvků v kontejneru.  
V mém druhém příkladě tento adaptér schválně nepoužiji, abych ukázal jeho výhody. Ještě jen musím dodat, že existují i další, podobné adaptéry, které jsem zde mohl použít. Adaptér insert\_iterator vkládá do kontejneru vždy na pozici udanou parametrem v konstruktoru. Dále existují back\_insert\_iterator, který vkládá vždy na konec kontejneru, nebo front\_insert\_iterator, který vkládá vždy na začátek kontejneru. Ale u setříděných kontejnerů nehraje žádnou roli poloha, na kterou se ukládá, protože prvek bude vložen vždy na své místo dané uspořádáním. Proto je jedno jaký adaptér jsem zvolil pro náš příklad. Všechny zmiňované adaptéry jsou deklarovány v hlavičkovém souboru iterator v prostoru jmen std.

Multimnožina

Práce s multimnožinou je obdobná jako práce s množinou. Multimnožina může obsahovat více stejných prvků. Pomocí metody count zjistíme nejen to, zda prvek v množině je, ale také kolikrát v dané množině je. Přejděme rovnou na příklad. Jeden ze způsobu jak najít nejvyššího společného dělitele dvou čísel je rozložit obě čísla na součin prvočísel. Každý rozklad je vlastně multimnožina, a součin prvků jejich průniků je výsledek. Všimněte si, jaké problémy přináší, když se pokusím nepoužít adaptér vkládací iterátor.

|  |  |
| --- | --- |
| |  | | --- | | #include <set>  #include <algorithm>  #include <vector>  using namespace std;  void rozloz(int cislo, multiset<int> &vysledek)  {  register int i = 2;  vysledek.insert(1);  while(cislo != 1)  {  if (cislo % i == 0)  {  cislo /= i;  vysledek.insert(i);  }  else  {  i++;  }  }  }  int main()  {  multiset<int> prvni,druha;  /\* Množinové operace nemusíme provádět pouze se set, nebo multiset\*/  vector<int> vyslednyRozklad(100,0);  /\*  MUSÍM zadat velikost vektoru!! Jinak by v těle funkce set\_intersection  bylo zapsáno do nealokované paměti. Protože výraz vyslednyRozklad.begin()  by se odkazoval ZA poslední prvek kontejneru. Daleko vhodnější by bylo  použití nějakého "vkládacího" iterátoru. Tento příklad je pouze  demonstrační. Krom toho, že musím zadávat velikost kontejneru, jsem také  touto velikostí limitován. Kdyby měl výsledný rozklad více než 100 cifer,  program by havaroval. Proto bylo lepší použít nějaký adaptér jako v  minulém příkladě. Protože vector není seřazený kontejner, bylo by již  podstatné, jaký adaptér zvolíme.  \*/  int vysledek = 1;  rozloz(120,prvni);  rozloz(740,druha);  set\_intersection(prvni.begin(),prvni.end(),druha.begin(),druha.end()  ,vyslednyRozklad.begin());  for(vector<int>::iterator i = vyslednyRozklad.begin(); \*i != 0; i++)  {  vysledek \*= \*i;  }  cout << vysledek << endl;  return 0;  } | |

Příště se podíváme na funkční objekty v C++.

## [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/funkcni-objekty-v-c--155955cz)

**4. prosince 2001, 00.00 | Funkční objekt je instance třídy, která má jako svou veřejnou metodu operátor (), tedy operátor pro volání funkce. V dnešním článku si ukážeme jak zobecnit funkci, a jak používat funkční objekt na místě, kde se očekává ukazatel na funkci.**

Funkční objekt je instance třídy, která má jako svou veřejnou metodu operátor (), tedy operátor pro volání funkce. V dnešním článku si ukážeme jak zobecnit funkci, jak používat funkční objekt na místě, kde se očekává ukazatel na funkci. Než se začneme věnovat využití funkčních objektů, zopakujeme si, jak přetížit operátor (). Ukážeme si vytvoření jednoduchého funkčního objektu.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  using namespace std;  class pokus  {  public:  int operator()(int parametr);  };  pokus::operator()(int parametr)  {  cout << "Je volan operator () s parametrem " << parametr << endl;  return parametr \* 2;  }  int main()  {  pokus objekt;  objekt(0); // Nebo: objekt.operator()(0);  cout << objekt(10) << endl;  return 0;  } | |

Vytvořili jsme si jednoduchou třídu funkčních objektů, třída ma operátor () pro parametr int. Instanci této třídy tedy lze použít (zapsat) jako funkci (volání funkce) s parametrem int. Jedná se pouze o ukázkový příklad, který nemá snad žádné využití. Než se dostaneme k poněkud užitečnějším funkčním objektům, podívejme se nejprve na jeden problém, který nemá s funkčními objekty nic společného.

Představme si, že chceme vytvořit funkci, která zjistí, kolik prvků v nějakém kontejneru, nebo poli vyhovuje nějaké podmínce. Podmínku v době psaní naší funkce neznáme. Naše funkce by měla být univerzálně použitelná, pro jakoukoliv podmínku. Jedno z možných řešení je mít jako parametr naší funkce ukazatel na funkci vracející bool, která vrátí, zda její parametr odpovídá podmínce. Ukažme si to na příkladu.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <vector>  using namespace std;  bool podminka(int a)  { /\* Je číslo menší, než 3 ? \*/  return a < 3;  }  template<class InputIterator>  int pocetPlatnych(InputIterator zacatek, InputIterator konec, bool (\*fce)(int))  {  register int pocet = 0;  for(InputIterator i = zacatek; i != konec; i++)  {  if (fce(\*i))  {  pocet++;  }  }  return pocet;  }  int main()  {  vector<int> v;  int pole[5] = { 1, 2, 3, 4, 5} ;  v.push\_back(1);  v.push\_back(2);  v.push\_back(3);  v.push\_back(4);  v.push\_back(5);  v.push\_back(-20);  cout << pocetPlatnych(v.begin(), v.end(), podminka) << endl;  cout << pocetPlatnych(pole, &pole[5], podminka) << endl;  return 0;  } | |

Naše funkce početPlatných může mít jako svůj třetí parametr jakoukoliv funkci (ukazatel na funkci), která má jako parametr int a vrací bool. Toto řešení má několik nedostatků. Za prvé v momentě, kdy budeme chtít, použít jinou podmínku (Například <5 místo <3.), nebudeme moci použít funkci podmínka, budeme muset napsat novou podmínku. To by se snad dalo ještě obejít pomocí globální proměnné, ale problém by nastal, kdyby jsme chtěli například podmínku 3 < x < 5. Naše funkce početPlatných by se už vůbec nedala použít pro pole, nebo kontejner obsahující jiné prvky než int. Chceme-li napsat opravdu obecnou funkci početPlatných, musíme použít funkční objekty.

Kdyby jsme si chtěli vytvořit nějakou třídu s definovaným operátorem (), a její instanci použít jako parametr naší funkce početPlatných, překladač by to odmítl jako nekompatibilitu typů. (Instance třídy není ukazatel na funkci.) Musíme naší šablonu funkce početPlatných poněkud zobecnit. Parametrem šablony nebude jenom typ iterátoru, ale další typ, který jsem nazval Funkce. Třetí parametr funkce početPlatných bude parametr typu Funkce. Tím docílíme toho, že parametrem může být "cokoliv". Tělo funkce početPlatných nijak měnit nemusíme. Podíváme-li se podrobně na vzniklou šablonu funkce zjistíme, že za typ Funkce může být dosazen buď typ funkce s jedním parametrem, která vrací bool (V našem případě může vracet i int.), nebo třída, která má jako veřejnou metodu operátor () s jedním parametrem, vracejícím bool, nebo int. Podívejme se na příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <vector>  using namespace std;  class TridaFunkcnichObjektu  {  private:  int HorniMez;  public:  TridaFunkcnichObjektu() : HorniMez(0) {}  TridaFunkcnichObjektu(int mez) : HorniMez(mez) {}  bool operator()(int i) { return i < HorniMez; }  void nastavMez(int mez) { HorniMez = mez; }  };  bool podminka(int a)  {  return a < 3;  }  template<class InputIterator, class Funkce>  int pocetPlatnych(InputIterator zacatek, InputIterator konec, Funkce fce)  {  register int pocet = 0;  for(InputIterator i = zacatek; i != konec; i++)  {  /\*  fce je typu Funkce. Právě kvůli následujícího řádku jsou  omezení na typ Funkce. Buď se musí jedna o ukazatel na funkci, nebo  o třídu funkčních objektů - viz text nad příkladem.  \*/  if (fce(\*i))  {  pocet++;  }  }  return pocet;  }  int main(int argc, char \*\*argv)  {  vector<int> v;  int pole[5] = { 1, 2, 3, 4, 5} ;  v.push\_back(1);  v.push\_back(2);  v.push\_back(3);  v.push\_back(4);  v.push\_back(5);  v.push\_back(-20);  cout << pocetPlatnych(v.begin(), v.end(), podminka) << endl;  cout << pocetPlatnych(pole, &pole[5], podminka) << endl;  cout << "To same:" << endl;  TridaFunkcnichObjektu funkcniObjekt(3);  cout << pocetPlatnych(v.begin(), v.end(), funkcniObjekt) << endl;  cout << pocetPlatnych(pole, &pole[5], funkcniObjekt) << endl;  funkcniObjekt.nastavMez(5);  cout << endl << pocetPlatnych(v.begin(), v.end(), funkcniObjekt) << endl;  cout << pocetPlatnych(pole, &pole[5], funkcniObjekt) << endl;  return 0;  } | |

Šablona funkce početPlatných je nyní hodně obecná. Lze ji použít na jakýkoliv kontejner, obsahující instance jakéhokoliv typu. Podmínka může být libovolná, kterou lze v C++ zapsat. Použití šablony funkce početPlatných je znázorněno v jednoduché funkci main. Vytvořili jsme vlastně něco jako obecný algoritmus pracující s kontejnery. V knihovně STL je implementováno mnoho algoritmů pracujících s kontejnery. Až se budeme algoritmy v STL zabývat (což by mělo být v článku po následujícím článku), uvidíme, že v STL je i obdoba naší šablony funkce početPlatných.

Dnes jsme si předvedli princip funkčních objektů. Ještě než se podíváme na standardní algoritmy v STL, budeme se v příštím článku zabývat funkčními objekty v STL. V STL je také několik standardních funkčních objektů. Funkční objekty se velmi často používají pro práci se standardními algoritmy, ale je možné je používat i u vlastních šablon funkcí, či metod.

## [Standardní funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/standardni-funkcni-objekty-v-c--155969cz)

**10. prosince 2001, 00.00 | V minulém článku jsme se seznámili s pojmem funkční objekt. Dnes si ukážeme jaké funkční objekty (třídy funkčních objektů) obsahuje standardní knihovna STL. Podíváme se na unární a binární funkční objekty, a také na šablony binder1st a binder2nd.**

V minulém článku ["Funkční objekty v C++"](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/) jsme se seznámili s pojmem funkční objekt. Dnes si ukážeme jaké funkční objekty (třídy funkčních objektů) obsahuje standardní knihovna STL. Jestliže chceme pracovat se standardními algoritmy, je dobré znát standardní funkční objekty, nebo si umět vytvořit vlastní. Standardním algoritmům v STL se budeme věnovat v příštím článku.

Šablony unary\_function a binary\_function

V STL existují třídy unárních a binárních funkčních objektů. Unární funkční objekt je funkční objekt, který má jen jeden parametr operátoru (). Binární funkční objekt je funkční objekt, který má dva parametry operátoru (). Šablony unary\_function a binary\_function jsou šablony struktur. Obě šablony slouží jako nadtřídy (Jedná se sice o struktury, ale budu používat raději pojem třída. Slovo "nadstruktura" se mi zdá nějaké divné.) pro třídy funkčních objektů. Nemají žádnou metodu, ani atribut. Obsahují pouze definici vnitřních typů. Neobsahují dokonce ani operátor (), proto se vlastně nejedná ani o třídy funkčních objektů. Každá standardní třída funkčních objektů je šablonou třídy (struktury), a dědí z jedné z těchto dvou šablon tříd (struktur). Každá standardní třída funkčních objektů rozšíří tyto šablony minimálně o operátor (). Šablona unary\_function má dva parametry, první udává typ parametru, druhý udává typ návratové hodnoty operátoru(). Šablona binary\_function má 3 parametry. První dva udávají typy prvního a druhého parametru operátoru(). Poslední parametr udává typ návratové hodnoty parametru (). Programátor, který používá funkční objekty s šablonami binary\_functiona unary\_function nepříjde nijak do styku. Bude pracovat s šablonami odvozenými. Přesto si myslím, že je dobré o binary\_function a unary\_function vědět. Rovněž je dobré použít je jako nadtřídu při psaní vlastních tříd funkčních objektů. (Což jsem ve svém [minulém článku](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/)neudělal, protože jsme takové šablony ještě neznali.)

Třídy unárních funkčních objektů

Šablony tříd unárních funkčních objektů z STL jsou potomky šablony třídy (struktury)unary\_function. Seznam některých tříd unárních funkčních objektů:

|  |  |
| --- | --- |
| Název šablony | Činnost |
| negate | Šablona má jeden parametr. Udává typ návratové hodnoty, i typ parametru operátoru (). Funkční objekt neguje svůj parametr. Je-li parametrem operátoru () proměnná (objekt) x, výsledek bude -x. |
| logical\_not | Šablona má jeden parametr. Udává typ parametru operátoru (). Návratová hodnota operátoru () je vždy typu bool. Funkční objekt neguje svůj parametr. Je-li parametrem operátoru () proměnná (objekt) x, výsledek bude !x. |

Třídy binárních funkčních objektů

Šablony tříd binárních funkčních objektů z STL jsou potomky šablony třídy (struktury)binary\_function. Seznam některých tříd binárních funkčních objektů:

|  |  |
| --- | --- |
| Název šablony | Činnost |
| divides | Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vydělí své dva parametry. |
| equalto | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr == druhý parametr. |
| greater | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr > druhý parametr. |
| greater\_equal | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr >= druhý parametr. |
| less | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr < druhý parametr. |
| less\_equal | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr <= druhý parametr. |
| logical\_and | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Operátor () vrací první parametr && druhý parametr. |
| logical\_or | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Operátor () vrací první parametr || druhý parametr. |
| minus | Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vrátí rozdíl svých dvou parametrů. |
| modulus | Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vrátí zbytek po celočíselném dělení. |
| not\_equal | Šablona má 1 parametr. Parametr udává typ obou parametrů operátoru (). Operátor () vrací proměnnou typu bool. Vrací true v případě, že první parametr != druhý parametr. |
| plus | Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () sečte své dva parametry. |
| times | Šablona má 1 parametr. Parametr udává typ obou parametrů a návratové hodnoty operátoru (). Operátor () vynásobí své dva parametry. |

Všechny tyto třídy jsou deklarovány v hlavičkovém souboru function v prostoru jmen std. Omezení na parametr šablony vyplývá z její činnosti. Například šablona plus může mít jako svůj parametr typ, který má definován operátor + (implicitně, nebo definován operator+).

Nyní si ukážeme jednoduchý příklad. Ve svých článcích [Množina a multimnožina v C++](http://www.builder.cz/rubriky/c/c--/clanek-155952-mnozina-v-c-/) a [Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/clanek-155938-asociativni-pole-v-c-/) jsem uvedl, že tyto kontejnery mají poslední, nepovinný parametr, kterým je funkční objekt. Implicitní hodnota posledního parametru je less<T>, kde T je první parametr kontejneru.

|  |  |
| --- | --- |
| |  | | --- | | #include <map>  #include <string>  #include <iostream>  #include <functional>  using namespace std;  int main()  {  map<string,string,greater<string> > pole;  pole.insert(make\_pair(string("Klic1"),string("Hodnota1")));  pole.insert(make\_pair(string("Klic2"),string("Hodnota2")));  cout << pole[string("Klic1")] << endl;  /\* Klíče jsou seřazeny pomocí relačního operátoru >,  nikoliv < jako v předchozích článcích.\*/  return 0;  } | |

Tento příklad zatím asi nikoho nepřesvědčil o důležitosti funkčních objektů. Jejich význam se ukáže až v mých dalších článcích.

Šablony binder1st a binder2nd

V souvislosti se standardními třídami funkčních objektů je dobré se také seznámit s šablonami binder1st a binder2nd. Jedná se o šablony tříd s operátorem (), který má 1 parametr. Obě třídy zapouzdřují binární funkční objekt (operátor () má 2 parametry) a jednu hodnotu. Tato hodnota bude daná jako první parametr u šablony binder1st nebo jako druhý parametr u šablony binder2nd. Zbývající parametr bude předán operátoru () jako parametr. Obě třídy tedy vlastně slouží k převedení binárního funkčního objektu na unární s pevně danou hodnotou jednoho ze svých parametrů. Zní to možná složitě, ale vše si ukážeme na jednoduchém příkladu. K vytvoření instancí tříd binder1st a binder2nd můžeme použít jednak konstruktory, nebo pomocné funkce jménem bind1st a bind2nd.

Ve svém [minulém článku](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/) jsem ve svém posledním příkladu vytvořil šablonu funkce pocetPlatnych, která vrací počet prvků v kontejneru vyhovujících nějaké podmínce, která je také předána jako parametr. Podmínka je buď ukazatel na funkci s jedním parametrem, nebo funkční objekt, jehož operátor () má jeden parametr. Vytvořil jsem třídu funkčních objektů, která má jeden atribut. Takto jsem se snažil řešit problém, kdy operátor () má jen jeden parametr, já bych potřeboval aby měl dva. Nyní se podíváme jak napsat stejný příklad jen pomocí standardních šablon z STL. V STL existuje algoritmus count\_if (Algoritmy podrobně probereme příště.), který dělá stejnou činnost jako moje funkce pocetPlatnych. S tím rozdílem, že vrací void, protože výsledek je uložen v posledním parametru, který je předáván jako reference. Zatímco moje funkce umožňovala vrátit výsledek pouze jako int, funkcecount\_if z STL má jako parametr i typ výsledku, což umožňuje ještě větší abstrakce. Můj poslední příklad z minulého článku napsaný jen pomocí standardních šablon vypadá takto:

|  |  |
| --- | --- |
| |  | | --- | | #include <vector>  #include <iostream>  #include <functional>  #include <algorithm>  using namespace std;  int main(int argc, char \*\*argv)  {  vector<int> v;  int vysledek = 0;  int pole[5] = { 1, 2, 3, 4, 5} ;  v.push\_back(1);  v.push\_back(2);  v.push\_back(3);  v.push\_back(4);  v.push\_back(5);  v.push\_back(-20);  binder2nd<less<int> > funkcniObjekt = bind2nd(less<int>(),3);  /\* Nyní volání funkcniObjekt(x) je stejne jako volání less<int>(x,3) \*/  count\_if(v.begin(),v.end(),funkcniObjekt,vysledek);  cout << vysledek << endl;  vysledek = 0;  count\_if(pole,&pole[5],funkcniObjekt,vysledek);  cout << vysledek << endl;  vysledek = 0;  count\_if(pole,&pole[5],bind2nd(less<int>(), 5) ,vysledek);  cout << vysledek << endl;  vysledek = 0;  count\_if(v.begin(),v.end(),bind2nd(less<int>(), 5) ,vysledek);  cout << vysledek << endl;  return 0;  } | |

Ve svých příkladech používám vždy kontejnery s primitivními datovými typy, nebo s řetězci. Musím ale zdůraznit, že vše lze použít i na kontejnery obsahující instance libovolných tříd, které mají definovány příslušné operátory.

Příště se podíváme na standardní algoritmy v STL.

[Úvod do standardních algoritmů v C++](http://www.builder.cz/rubriky/c/c--/uvod-do-standardnich-algoritmu-v-c--155983cz)

**7. ledna 2002, 00.00 | Dnes se seznámíme se standardními algoritmy v jazyce C++. Standardní algoritmy jsou součástí STL. Jedná se   
o spoustu užitečných šablon funkcí, které za nás naprogramoval někdo jiný. Na nás je jen, abychom je používali.**

Dnes se seznámíme se standardními algoritmy v jazyce C++. Standardní algoritmy jsou součástí STL. Jedná se o spoustu užitečných šablon funkcí, které za nás naprogramoval někdo jiný. Na nás je jen, abychom je používali. Také si ukážeme příklady algoritmů pro naplnění kontejneru.

Standardní algoritmy jsou šablony funkcí. Všechny šablony jsou deklarovány v hlavičkovém souboru algorithm v prostoru jmen std. Algoritmy ve většině případů slouží k práci s kontejnery. Žádný s algoritmů ale nepracuje s kontejnery přímo. K prvkům kontejneru přistupují algoritmy zásadně pomocí iterátorů. Viz můj článek [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/). To umožňuje, aby algoritmy byly obecné a jednou napsaný algoritmus byl použitelný pro jakýkoliv kontejner. Díky toho lze algoritmy aplikovat také na obyčejné pole.

S jistým typem algoritmů jsme se již setkali. Jednalo se o množinové operace pro průnik, sjednocení, množinový rozdíl, symetrickou diferenci a určení zda se jedná o podmnožinu. Těmito operacemi se dále nebudeme zabývat. Probrali jsme je v článku [Množina a multimnožina v C++](http://www.builder.cz/rubriky/c/c--/clanek-155952-mnozina-v-c-/).

Algoritmy nepracující s kontejnery

Jak jsem uvedl, většina algoritmů je pro manipulaci s daty v kontejnerech. Jako příklad algoritmů, které s kontejnery nemanipulují uvedu algoritmy min, max, swap. Algoritmus swap"vymění" obsah obou prvků. Šablona swap má parametr udávající typ obou prvků. Parametrem funkce jsou dvě reference na proměnné, jejíž hodnoty chceme vyměnit. Typ, který je parametrem šablony musí být schopen se kopírovat pomocí kopírovacího konstruktoru a operátoru =. Nelze-li použít implicitní, musí je definovat programátor. Algoritmymin a max vracejí minimální a maximální prvek. Pro každý algoritmus existují dvě varianty:

* template <class T>const T& min(const T& a, const T& b); - parametrem šablony je typ obou prvků a typ návratové hodnoty. Algoritmus vrátí minimální prvek z a,b. Použije při tom operátor <.
* template <class T, class Compare>const T& min(const T& a, const T& b, Compare comp); - obdobně jako minulá možnost. K výběru minima nebude použit operátor <, ale funkční objekt zadaný jako poslední parametr. Viz můj článek [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Tím zajistíme, aby objekt a a objekt b byly porovnány pomocí jiné, námi vybrané relace.

Pro algoritmus max je situace obdobná. Použití min, max a swap je velmi jednoduché.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include <algorithm>  #include <functional>  using namespace std;  less<int> l;  int main()  {  int a = 9, b = 10, c = 11;  cout << min(a,b) << endl;  cout << max(b,c) << endl;  swap(c,a);  cout << min(a,b,l) << endl;  /\*  Jenom ukázka možností. V tomto konkrétním  případě jsem chování min pomocí objektu l  nijak nezměnil.  \*/  return 0;  } | |

Algoritmy pro vyplňování kontejnerů

V C++ existují 4 algoritmy, kterými lze vyplňovat kontejnery nějakými hodnotami. Pro vyplnění kontejneru konstantní hodnotou slouží algoritmy fill a fill\_n. Šablona fill má dva parametry. Typ iterátoru (Očekává se dopřední iterátor. Viz můj článek [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/).) Druhým parametrem je typ vkládaného prvku. Parametry funkce jsou dva iterátory (začátek a konec) a prvek, který se má vložit mezi prvky dané iterátory začátek a konec. Šablona fill\_n má 3 parametry. Typ iterátoru (očekává se výstupní iterátor), typ parametru udávající počet vložených prvků a typ vkládaného prvku. Parametry funkce jsou iterátor, udávající začátek, počet prvků a vkládaný prvek. Použití je jednoduché. Ukážeme si jej na jednoduchém příkladu.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<fstream>  #include<algorithm>  #include<vector>  using namespace std;  int main()  {  vector<int> vektor(10,3);  /\* vektor má 10 prvků - samé trojky. \*/  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  fill(vektor.begin(),vektor.end(),20);  /\* V celém vektoru jsou dvacítky. \*/  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  fill\_n(vektor.begin() + 2,5,100);  /\* Do prvku vektor[2] a 4 následujících (celkem 5)  vloží číslo 100. \*/  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  ofstream soubor("Pokus.txt");  fill\_n(ostream\_iterator<int>(soubor,","),10,0);  /\*  Nezapomeňme, že také proudy mají své iterátory.  Předchozí řádek zapíše do textového souboru "Pokus.txt"  deset nul oddělených čárkou.  \*/  soubor << endl;  return 0;  } | |

Pro vyplnění kontejneru nekonstantní hodnotou jsou k dispozici algoritmy generate a generate\_n. Parametry jsou obdobné jako u fill, resp. fill\_n jen s tím rozdílem, že místo parametru šablon udávající typ vkládaného prvku je parametrem třída funkčního objektu (generátoru). A jako parametr funkce místo hodnoty, která se má vkládat, je parametrem funkční objekt (generátor), nebo ukazatel na funkci. Očekává se ukazatel na funkci, která nemá parametry a vrací typ prvku v kontejneru, nebo funkční objekt jehož třída má přetížen operátor () tak, aby vracel typ prvku v kontejneru a neměl parametry. Viz. můj článek [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Uveďme si příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<fstream>  #include<algorithm>  #include<vector>  #include<stdlib.h>  using namespace std;  class Faktorial  {  private:  int I, Vysledek;  public:  Faktorial() : I(0), Vysledek(1) {}  int operator() () { return Vysledek \*=++I; }  /\*  N-tý prvek ve vektoru bude n! (n - faktoriál).  Upozorňuji, že prvek s indexem 0 je první,  nikoliv nultý.  \*/  };  class Rada  {  private:  int A,B;  public:  Rada() : A(0),B(1) {}  int operator() () { swap(A,B); return A += B; }  /\*  První dva prvky jsou jedničky, každý další prvek  je součet předchozích dvou. Může se to zdát trochu  zvláštně napsané, ale pracuje to tak, jak má.  \*/  };  int main()  {  vector<int> vektor(10); /\* vektor má 10 prvků\*/  Faktorial f;  Rada r;  generate(vektor.begin(),vektor.end(),rand);  /\*  Jako generátor je ukazatel na funkci rand  vracející náhodné číslo. Vyplnil jsem vektor  náhodnými čísly.  \*/  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  generate(vektor.begin(),vektor.end(),f);  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  generate(vektor.begin(),vektor.end(),r);  for(vector<int>::iterator i = vektor.begin();  i != vektor.end(); i++)  {  cout << \*i << '\t';  }  cout << endl;  /\* Nesmíme zapomenout, že vše lze použít i na pole. \*/  int pole[20];  generate\_n(pole,20,r);  for(int \*p = pole; p != &pole[20]; p++)  {  cout << \*p << '\t';  }  cout << endl;  return 0;  } | |

Jako poslední parametr funkce generate je možné zadat ukazatel na funkci (V mém příkladě rand.), nebo funkční objekt (V mém příkladě objekt f třídy Faktorial, nebo objekt r třídy Rada.).

Příště se podíváme na kopírovací a přesouvací algoritmy.

[Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-a-presouvaci-algoritmy-v-c--156005cz)

**18. ledna 2002, 00.00 | V dnešním článku si ukážeme jak lze jednoduše v C++ odstranit data, nebo kopírovat data mezi libovolnými kontejnery z knihovny STL. Vše lze i mezi obyčejnými poli, nebo datovými proudy. Pro tuto činnost je v STL dispozici několik šablon funkcí.**

V minulém článku jsme se dověděli, že standardní knihovna jazyka C++ má k dispozici mnoho užitečných šablon funkcí. Mnoho z nich manipuluje s kontejnery pomocí iterátorů (Viz můj článek "[Iterátory v C++](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/)"). Dnes se podrobněji podíváme na kopírovací a přesouvací algoritmy.

Kopírovací algoritmy v C++

Ke kopírování dat slouží algoritmy copy a copy\_backward. Parametry šablony copy jsou typy vstupních a výstupních iterátorů. Parametry funkce jsou iterátory na začátek a konec zdrojové oblasti. Třetím parametrem je iterátor na začátek cílové oblasti. Deklarace je následující:  
template <class InputIterator, class OutputIterator>  
OutputIterator copy(InputIterator zacatek, InputIterator konec, OutputIterator cil)  
Algoritmus zkopíruje data počínaje prvkem, na který se odkazuje zacatek a konče prvkem daným iterátorem konec na pozici danou iterátorem cil. Funkce vrací iterátor na poslední zkopírovaný prvek. Algoritmus copy\_backward je obdobný. S tím rozdílem, že parametry šablony jsou obousměrné iterátory (Viz můj článek [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/).), a data se kopírují "od zadu". Je dobré si uvědomit, že všude, kde je možné použit iterátor, lze použít také ukazatel. A také datové proudy mají iterátory. Nyní jednoduchý příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<vector>  #include<algorithm>  #include<stdlib.h>  #include<fstream>  #include<set>  using namespace std;  int main()  {  vector<int> vektor(10);  set<int> m; /\* Viz článek Množina v C++ \*/  int pole[10];  ofstream f("Pokus.txt");  generate(vektor.begin(),vektor.end(),rand); /\* Viz předchozí článek \*/  copy(vektor.begin(),vektor.end(),pole);  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,","));  cout << endl;  fill(pole,&pole[10],0); /\* Viz předchozí článek \*/  copy(pole,&pole[10],ostream\_iterator<int>(cout,","));  cout << endl;  copy\_backward(vektor.begin(),vektor.end(),&pole[10]);  copy(pole,&pole[10],ostream\_iterator<int>(cout,","));  copy(pole,&pole[10],ostream\_iterator<int>(f,"\n"));  if (f)  {  cout << endl << "Byl vytvořen soubor Pokus.txt" << endl;  }  else  {  cout << endl << "Problémy se souborem Pokus.txt" << endl;  }  insert\_iterator<set<int> > i(m,m.begin());  copy(pole,&pole[10],i);  copy(m.begin(),m.end(),ostream\_iterator<int>(cout," "));  return 0;  } | |

Postupně kopírujeme data mezi vektorem a polem. Mezi vektorem a datovým proudem cout, mezi polem a proudem cout atd... Tímto příkladem se můžeme také přesvědčit, že kopírovat lze mezi libovolnými kontejnery jako vektor, nebo množina (Viz můj článek "[Množina a multimnožina v C++](http://www.builder.cz/rubriky/c/c--/clanek-155952-mnozina-v-c-/)"). Ale také mezi datovými proudy, nebo poli. Pokud budeme kopírovat prvky na standardní výstup pomocí iterátoru datového proudu, narazíme na jednu nepříjemnost. Oddělovací znak bude vložen i za poslední číslo. To může být někdy nežádoucí (například zde, když na stdout je za posledním číslem znak ','), jindy zase ano (například v souboru Pokus.txt je dobré mít za posledním číslem nový řádek).

Přesouvací algoritmy v C++

Pod pojmem přesun si každý představí smazání dat v jednom kontejneru, a vytvoření těchto dat v kontejneru jiném. Problém je v tom, že takto se algoritmy, které chci popsat nechovají. Slovo "přesouvací" je můj dosti nevhodný překlad slova remove. Nenapadá mne ale žádné jiné slovo, kterým by se dala činnost "remove" algoritmů popsat. Algoritmy remove a remove\_if prvky z kontejneru odstraní. Data nebudou nikam přesunuta. Zde bych chápal význam slova remove jako odstranit. Ale algoritmy remove\_copy a remove\_copy\_if ani nepřesouvají ani neodstraňují. Nezabývejme se ale názvem článku, a pojďme se podívat na činnost těchto algoritmů. Deklarace šablon:

* template <class InputIterator, class OutputIterator, class Typ> OutputIterator remove\_copy (InputIterator zacatek, InputIterator konec, OutputIterator vysledek, const Typ& hodnota) - V oblasti dané iterátory zacatek a konec vybere všechny prvky, které NEjsou rovny s prvkem hodnota. Porovnání bude provedeno operátorem != , který je buď implicitní, nebo jej musíme přetížit. Tyto prvky zkopíruje do jiného kontejneru na pozici danou iterátorem vysledek. Původní kontejner se nijak nezmění.
* template <class InputIterator, class OutputIterator, class TFunkcniObjekt> OutputIterator remove\_copy\_if (InputIterator zacatek, InputIterator konec, OutputIterator vysledek, TFunkcniObjekt podminka) - Vybere v oblasti dané iterátory zacatek a konec všechny prvky, které NEvyhovují podmínce (NEplatí podminka(\*i) == true, kde i je iterátor v rozsahu zacatek až konec). Tyto prvky zkopíruje do kontejneru na pozici danou iterátorem vysledek. Původní kontejner se nijak nezmění.
* template <class ForwardIterator, class Typ> ForwardIterator remove (ForwardIterator zacatek, ForwardIterator konec, const Typ &hodnota); - Obdobný algoritmus jako remove\_copy s tím rozdílem, že vybrané prvky budou z kontejneru odstraněny. Nebudou nikam přesunuty ani zkopírovány. Odstraněny budou prvky, které jsou si rovny s daným prvkem.
* template <class ForwardIterator, class TFunkcniObjekt> ForwardIterator remove\_if (ForwardIterator zacatek, ForwardIterator konec, TFunkcniObjekt podminka); - Obdobný algoritmus jako remove\_copy\_if s tím rozdílem, že vybrané prvky budou z kontejneru odstraněny. Nebudou nikam přesunuty ani zkopírovány. Odstraněny budou prvky, pro které platí podmínka.

Algoritmy remove a remove\_if nijak nemění velikost kontejneru. Jejich návratová hodnota je iterátor na poslední platný prvek v kontejneru po provedení algoritmu. Je vhodné zbytek kontejneru (od tohoto iterátoru až po konec) smazat. NEZAPOMEŇTE na to! Je to zdrojem častých chyb. Vše je znázorněno v příkladu. V příkladu je použita šablona třídyunary\_compose, která slouží ke "skládání" funkcí, nebo [funkčních objektů](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Parametry šablony jsou dva typy unárních funkcí, nebo třídy unárních [funkčních objektů](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Konstruktoru předám dva ukazatele na unární funkci, nebo unární funkční objekty. Výsledkem volání operátoru () instanci třídy unary\_compose je volání O1(O2(x)), kde x je parametr operátoru (), O1 a O2 jsou funkce, nebo funkční objekty dané konstruktoru. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<vector>  #include<algorithm>  #include<iostream>  #include<stdlib.h>  #include<functional>  using namespace std;  int main(int argc, char \*\*argv)  {  vector<int> v1(10), v2;  generate(v1.begin(), v1.end(), rand);  cout << "Původní:" << endl;  copy(v1.begin(),v1.end(),ostream\_iterator<int>(cout,","));  cout << endl;  cout << "Jen čísla menší než 5000:" << endl;  vector<int>::iterator i =  remove\_if(v1.begin(),v1.end(),bind1st(less<int>(),5000));  /\* Odstranil jsem prvky x, pro které platí 5000 < x \*/  v1.erase(i,v1.end()); /\* NEZAPOMEŇTE ! \*/  copy(v1.begin(),v1.end(),ostream\_iterator<int>(cout,","));  /\*  Všimněte si, jak jsem "poskládal" podmínku dělitelnosti  deseti jen ze standardních prostředků.  \*/  unary\_compose<binder1st<not\_equal\_to<int> >, binder2nd<modulus<int> > >  delitelne\_10(bind1st(not\_equal\_to<int>(),0),bind2nd(modulus<int>(),10));  remove\_copy\_if(v1.begin(),v1.end(),inserter(v2,v2.begin()),delitelne\_10);  cout << endl << "Nic jsem neodstranil:" << endl;  copy(v1.begin(),v1.end(),ostream\_iterator<int>(cout,","));  cout << endl << "Jen čísla dělitelná deseti:" << endl;  copy(v2.begin(),v2.end(),ostream\_iterator<int>(cout,","));  cout << endl;  if (!v2.empty())  {  int c = v2[0];  cout << "Odeberu všechny čísla " << c << endl;  i = remove(v2.begin(),v2.end(),c);  v2.erase(i,v2.end()); /\* NEZAPOMEŇTE ! \*/  copy(v2.begin(),v2.end(),ostream\_iterator<int>(cout,","));  cout << endl;  }  return 0;  } | |

Všimněte si, jak jsem pouze pomocí standardních prostředků C++ "složil" funkční objekt delitelne\_10. Použil jsem při tom [standardní funkční objekty](http://www.builder.cz/rubriky/c/c--/clanek-155969-standardni-funkcni-objekty-v-c-/).

Příště se podíváme na vyhledávácí algoritmy.

[Vyhledávací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/vyhledavaci-algoritmy-v-c--156011cz)

**25. ledna 2002, 00.00 | Dnes si povíme o standardních algoritmech v C++, které nám umožńují vyhledávat prvky v datových kontejnerech podle různých kritérií. Také se podíváme na vyhledávání metodou půlením intervalu (binární vyhledávání). Vše je v STL již naprogramováno.**

K vyhledávání dat v datových kontejnerech existuje celá řada standardních algoritmů. Ukažme si některé z nich:

* template <class InputIterator, class Typ> InputIterator find(InputIterator zacatek, InputIterator konec, const Typ& hledanaHodnota); Algoritmus bude prohledávat interval daný iterátory začátek a konec. Vrátí iterátor na první výskyt prvku hledanáHodnota. Není-li v daném intervalu hledaný prvek, algoritmus vrátí iterátor konec. K porovnání bude použit operátor ==.
* template <class InputIterator, class Podminka> InputIterator find\_if(InputIterator zacatek, InputIterator konec, Podminka p); Najde první prvek v intervalu začátek a konec, který odpovídá dané podmínce. Podmínka je [funkční objekt](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Neexistuje-li takový prvek, algoritmus vrátí iterátor konec.
* template <class ForwardIterator1, class ForwardIterator2> ForwardIterator1 search (ForwardIterator1 zacatek1, ForwardIterator1 konec1, ForwardIterator2 zacatek2, ForwardIterator2 konec2); V intervalu začátek1 a konec1 nalezne první výskyt posloupnosti danou prvky začátek2 a konec2. Algoritmus vrátí iterátor na prvního výskytu druhé posloupností v první. V případě, že taková posloupnost neexistuje, vrátí iterátor konec1. K porovnání prvků je použit operátor ==.
* template <class ForwardIterator1, class ForwardIterator2, class TFunkcniObjekt> ForwardIterator1 search (ForwardIterator1 zacatek1, ForwardIterator1 konec1, ForwardIterator2 zacatek2, ForwardIterator2 konec2, TFunkcniObjekt o); Obdobné chování jako předchozí algoritmus. Rozdíl je v tom, že místo operátoru == bude použit [funkční objekt](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/) o.

Mezi vyhledávací algoritmy lze zařadit i algoritmy:

* template <class ForwardIterator> InputIterator max\_element(ForwardIterator zacatek, ForwardIterator konec); Vrátí iterátor na největší prvek v intervalu začátek a konec. K porovnání bude použit operátor <.
* template <class ForwardIterator> InputIterator min\_element(ForwardIterator zacatek, ForwardIterator konec); Vrátí iterátor na nejmenší prvek v intervalu začátek a konec. K porovnání bude použit operátor <.

Oba tyto algoritmy mají i svou druhou variantu, která má jako další parametr funkční objekt. V této možnosti není použit operátor <, ale zadaný funkční objekt. Podívejme se na příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<algorithm>  #include<functional>  #include<string.h>  using namespace std;  int main()  {  char \*retezec = "http://www.builder.cz/serial24.html";  char \*podretezec = "/serial";  int pole[10] = { 1, 20, 45, 34, 23, 45, 87, 56, 55, 100} ;  int \*i1 = find(pole, &pole[10], 23); // i1 ukazuje na 23  int \*i2 = find\_if(pole, &pole[10], bind1st(equal\_to<int>(),45));  // i2 ukazuje na první 45  int \*i3 = find\_if(i2 + 1, &pole[10], bind1st(equal\_to<int>(),45));  // i3 ukazuje na druhou 45  char \*c = search(retezec, &retezec[35], podretezec, &podretezec[7]);  // c ukzuje na první výskyt podretezce v retezci  cout << \*i1 << endl << \*i2 << endl << \*i3 << endl << c << endl;  int \*max = max\_element(pole, &pole[10]);  int \*min = min\_element(pole, &pole[10]);  cout << \*max << endl << \*min << endl;  return 0;  } | |

Binární vyhledávání

Princip binárního vyhledávání, nebo-li vyhledávání půlením intervalu si vysvětlíme poněkud populární formou. Představme si hru dvou hráčů. Jeden si myslí číslo z předem dohodnutého intervalu. Druhý má za úkol toto číslo uhádnout. Hledající hráč předloží číslo, o kterém si myslí, že je hledané. Druhý hráč mu odpoví, zda se trefil, nebo zda hledané číslo je, či není vyšší (nebo nižší), než hledané číslo. Hráč hledající číslo má za úkol najít hledané číslo na co nejméně špatných pokusů. Jakou má hledající hráč zvolit strategii?

Jedna ne příliš dobrá strategie by byla zkoušet postupně všechny čísla. Představme si, že hledané číslo je v intervalu 1 až 100. Potom by hledající hráč procházel čísla postupně. Tedy zkoušel by 1, potom 2, potom 3, atd... Pokud by měl smůlu, a protivník by si myslel číslo 100, našel by jej hledající hráč až na stý pokus. Daleko lepší strategie by byla rozpůlit interval na polovinu. V našem případě je polovina intervalu 50. Zeptat se protivníka na číslo 50. Ten buď odpoví, že hledané číslo je opravdu 50 (velké štěstí), nebo nám řekne, že hledané číslo je větší, nebo menší. Teď už víme, že hledané číslo je buď v intervalu 1 až 50 (v případě, že hledané číslo je menší než 50), nebo 50 až 100 v případě, že hledané číslo je větší než padesát. Zúžili jsme tak interval. Touto vhodnou otázkou jsme si ušetřili 50 pokusů. Nyní si opět vybereme polovinu našeho nového intervalu. Touto druhou otázkou si ušetříme 25 pokusů. Takto pokračujeme, dokud nenalezneme hledané číslo. Ukážeme si algoritmus pro hledajícího hráče zapsaný v přirozeném jazyce, který je "velice" podobný C:

Vstupem do algoritmu jsou čísla začátek a konec udávající začátek

a konec intervalu, ve kterém se hledaná hodnota nachází.

Opakuj

{

pokus = začátek + (začátek + konec) / 2

Když je pokus menší než hledané číslo, tak konec = pokus

Když je pokus větší než hledané číslo, tak začátek = pokus

} Dokud pokus není hledané číslo.

Nyní si pozměňme pojem hledané číslo je nižší, nebo vyšší na pojem hledané číslo je před, nebo za daným číslem. Na naší hře to v podstatě nic nezmění.

Jiná hra, pro hledajícího hráče trochu složitější by měla podobné pravidla. Rozdíl by byl v tom, že protivník by měl posloupnost čísel neuspořádanou. Hledající hráč by jeho posloupnost neznal. Zde by metodu půlení intervalu nebylo možné použít. Protivník by sice řekl, zda hledané číslo je před, nebo za daným číslem, ale pro hledajícího hráče by tato informace byla k ničemu. Nezbývalo by mu nic jiného, než zkoušet postupně čísla od 1 do 100. Tomuto způsobu hledání se také říká sekvenční vyhledávání. Z toho plyne, že metodu půlení intervalu lze použít pouze tehdy, jestliže je posloupnost uspořádaná podle relace <. Je-li posloupnost uspořádaná, potom je výhodnější metodu binárního vyhledávaní použít, protože oproti sekvenčnímu vyhledávání má menší složitost (což jsme si ukázali na naší první hře).

Pro člověka, který se učí programovat, je určitě užitečné zkusit si alespoň jednou v životě algoritmus binárního vyhledávání implementovat. Pro ty, kteří algoritmus dobře znají, a nechtějí ztrácet čas jeho neustálým opisováním jsou v knihovně STL k dispozici šablony binary\_search a equal\_range. Existují dvě varianty šablony binary\_search:

* template <class ForwardIterator, class Typ> bool binary\_search(ForwardIterator zacatek, ForwardIterator konec, const Typ& hledanaHodnota); Parametrem šablony je typ iterátoru a typ hledané hodnoty (vlastně typ prvků v kontejneru). Parametrem funkce jsou iterátory začátek a konec intervalu, a hledaná hodnota. K porovnání hodnot slouží operátory <, a ==. Existuje-li hledaný prvek v daném intervalu, šablona funkce vrací true, jinak false. Musí být zajištěno, aby prvky kontejneru v intervalu daném iterátory začátek a konec byly uspořádány pomocí operátoru <. V opačném případě nelze brát výsledek funkce vážně.
* template <class ForwardIterator, class Typ, class TFunkcniObjekt> bool binary\_search(ForwardIterator zacatek, ForwardIterator konec, const Typ& hledanaHodnota, TFunkcniObjekt o); Obdobné chování jako první verze šablony. Jediný rozdíl je v relaci uspořádání. Tentokrát nebude použit operátor <, ale funkční objekt o třídy TFunkcniObjekt. Je očekáván binární funkční objekt vracející bool. Viz článek [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/).

Algoritmus binary\_search nám pouze oznámí, zda hledaný prvek v intervalu je, nebo vůbec není. Pokud jej ale chceme nalézt použijeme šablonu equal\_range. Varianty equal\_range:

* template <class ForwardIterator, class Typ> pair<ForwardIterator, ForwardIterator> equal\_range(ForwardIterator zacatek, ForwardIterator konec, const Typ& hledanaHodnota); Vrátí dvojici (s šablonou pair jsme se již setkali v článku [Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/clanek-155938-asociativni-pole-v-c-/)) iterátorů, která udává interval, ve kterém se nacházejí hledané hodnoty. Algoritmus vrátí dvojici iterátorů - začátek a konec oblasti, kde se nachází hledané hodnoty. Atribut first se odkazuje na první výskyt hledané hodnoty. Atribut second se odkazuje za poslední výskyt dané hodnoty. V případě, že hledaná hodnota v intervalu začátek a konec není, budou first i second ukazovat na stejný prvek. Bude to nejmenší prvek, který je větší než prvek s hledanou hodnotou. V případě, že takový prvek neexistuje, budou mít oba atributy hodnotu konec.
* template <class ForwardIterator, class Typ, class TFunkcniObjekt> pair<ForwardIterator, ForwardIterator> equal\_range(ForwardIterator zacatek, ForwardIterator konec, const Typ& hledanaHodnota, TFunkcniObjekt o); V podstatě stejná činnost algoritmu jako v předchozím příkladě. Pouze místo operátoru < bude použit funkční objekt o.

Nyní si binární vyhledávání ukážeme na příkladě.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<algorithm>  using namespace std;  int main()  {  int pole[10] = { 2, 5, 23, 45, 76, 76, 76, 100, 200, 1000};  /\* Pole je setříděno \*/  cout << "Číslo 25 v poli ";  if (binary\_search(pole,&pole[10],25))  {  cout << "je." << endl;  }  else  {  cout << "není." << endl;  }  pair<int\*,int\*> dvojce = equal\_range(pole,&pole[10],23);  cout << \*dvojce.first << " " << \*dvojce.second << endl;  cout << "Počet prvků 23: " << dvojce.second - dvojce.first << endl;  dvojce = equal\_range(pole,&pole[10],76);  cout << \*dvojce.first << " " << \*dvojce.second << endl;  cout << "Počet prvků 76: " << dvojce.second - dvojce.first << endl;  dvojce = equal\_range(pole,&pole[10],25);  cout << \*dvojce.first << " " << \*dvojce.second << endl;  cout << "Počet prvků 25: " << dvojce.second - dvojce.first << endl;  return 0;  } | |

Rád se pokusím odpovědět na všechny Vaše dotazy, které máte k mému seriálu. Chtěl bych Vás ale poprosit, aby jste své dotazy psali jako komentář pod článkem, a neposílali mi soukromé dopisy. Často musím odpovídat na stejné, nebo podobné otázky několikrát. Také se mi hromadí dopisy, na které bych rád odpověděl, ale nemohu, protože adresa odesílatele je špatná. Pište prosím své dotazy, nebo komentáře do diskuze pod článkem.

Příště se podíváme na "skenovací" (prohlížecí) algoritmy.

[Skenovací (prohlížecí) algoritmy v C++](http://www.builder.cz/rubriky/c/c--/skenovaci-prohlizeci-algoritmy-v-c--156016cz)

**1. února 2002, 00.00 | Dnes si ukážeme, že v C++ existuje for\_each. Algoritmus, který pro zadaný kontejner, nebo jeho část provede určenou činnost. Pro každý prvek zavolá funkci, nebo operátor () funkčního objektu. Dále si také povíme o algoritmech count, count\_if, accumulate.**

Skenovací algoritmy jsou algoritmy, které prohlédnou kontejner prvek po prvku, a po každé provedou nějakou operaci. Mezi tyto algoritmy patří: accumulate a for\_each. Také by se do této skupiny mohl zařadit algoritmus count, resp. count\_if. Deklarace jsou:

* template <class InputIterator, class Typ, class TypVysledek> void count(InputIterator zacatek, InputIterator konec, const Typ &hodnota, TypVysledek &n); - Parametry šablony jsou 3. První udává typ iterátoru, druhý udává typ prvků v kontejneru a poslední je typ výsledku. Jako typ výsledku bude asi v 99% použit nějaký celočíselný typ. Není to ale podmínkou. Důležité pouze je, aby typ výsledku měl k dispozici operátor ++. Parametry funkce jsou iterátory udávající začátek a konec oblasti. Dále následuje hledaná hodnota a proměnná pro výsledek. Po každé, co je mezi prvky danými iterátory začátek a konec nalezena hledaná hodnota (Použije se operátor == implicitní, nebo přetížený.), bude zvýšena hodnota výsledku (V našem případě n) pomocí operátoru++.
* template <class InputIterator, class TPodminka, class TypVysledek> void count(InputIterator zacatek, InputIterator konec, const TPodminka podminka, TypVysledek &n); - Obdobně jako minule. S rozdílem, že nyní nebude hledána konkrétní hodnota, ale hodnoty, které vyhovují zadané podmínce.
* template <class InputIterator, class Typ> Typ accumulate (InputIterator zacatek, InputIterator konec, Typ pocatecniHodnota); - Parametry šablony jsou typ iterátoru a typ prvků v kontejneru. Parametry funkce jsou iterátory udávající začátek a koneckontejneru. Dále počáteční hodnota výsledku. Funkce vrací součet všech prvků. Použije operátor +, který je buď implicitní, nebo přetížený. Algoritmus vlastně sečte počáteční hodnotu a všechny prvky v kontejneru. Zadáte-li jako počáteční hodnotu 0 (Což uděláte asi v 99%), vrátí funkce sumu prvků v oblasti dané iterátory začátek a konec.
* template <class InputIterator, class Typ, class BinaryOperation operace> Typ accumulate (InputIterator zacatek, InputIterator konec, Typ pocatecniHodnota, BinaryOperation operace); - Obdobně jako předchozí. Rozdíl je v tom, že tato varianta accumulate nebude sčítat, ale provede binární operaci zadanou programátorem.

U algoritmu accumulate musím upozornit, že jeho deklarace se nenachází v hlavičkovém souboru algorithm, ale v numeric. Vše si ukážeme na jednoduchém příkladu. Úmyslně zde střídám pole s vektorem (kontejnerem), aby bylo zřejmé, že algoritmy lze použít i na obyčejné pole.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<numeric>  #include<vector>  using namespace std;  bool podminka(int a)  {  return (20 < a) && (a < 80);  }  int main()  {  int pole[10] = { 12, 80, 34, 5, 23, 60, 82, 80, 9, 10 };  vector<int> vektor(pole,&pole[10]);  int pocet = 0;  cout << "Počet výskytů čísla 80: ";  count(vektor.begin(), vektor.end(), 80, pocet);  cout << pocet << endl << "Počet čísel 20 < x < 80: ";  pocet = 0; // Nezapomeňte na to!  count\_if(pole,&pole[10],podminka,pocet);  cout << pocet << endl << "Součet všech prvků je: ";  int soucet = accumulate(vektor.begin(), vektor.end(), 0);  cout << soucet << endl << "Součin prvních 3 prvků :";  int soucin = accumulate(pole, &pole[3], 1, times<int>());  cout << soucin << endl;  return 0;  } | |

Algoritmus for\_each v C++

Často jsem se setkal s mylným tvrzením, že v C++ neexistuje for\_each, nebo nějaká jeho obdoba. Hlavně velmi často v článcích (tištěných i na internetu) o jiných programovacích jazycích autoři srovnávají daný jazyk s C++. Vždy při tom prohlásí, že jednou z nevýhod C++ oproti jazyku, o kterém píšou, je absence for\_each. Snad se mi podaří tuto nepravdu vyvrátit. Algoritmus for\_each provede pro všechny prvky v zadaném rozmezí nějakou zadanou operaci. Deklarace:  
template void for\_each(InputIterator zacatek, InputIterator konec, TFunkce f); - Parametry šablony jsou typ iterátoru a typ funktoru (typ funkce, nebo třída funkčních objektů). Parametry funkce jsou začátek a konec oblasti a ukazatel na funkci, nebo funkční objekt. Vrací-li funkce, či operátor() funkčního objektu nějakou hodnotu, bude ignorována. Funkce, nebo operátor() funkčního objektu mohou při svém volání udělat nějaký "vedlejší efekt" jako třeba změna globální proměnné, atributu objektu, nebo poslání dat do datového proudu. Ukážeme si na příkladu použití for\_each. Vytvoříme si nějaké body v 2D, které dáme do kontejneru. Tyto body bude přesouvat.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<algorithm>  #include<vector>  using namespace std;  void funkce(int a)  {  cout << "Je volána funkce s parametrem " << a << endl;  }  class Bod  {  private:  int X,Y;  public:  Bod() : X(0), Y(0) {}  Bod(int x, int y) : X(x), Y(y) {}  void nastavX(int x) { X = x; }  void nastavY(int y) { Y = y; }  int dejX() const { return X; }  int dejY() const { return Y; }  };  class Posun  {  private:  int X,Y;  public:  Posun() : X(0), Y(0) {}  Posun(int x, int y) : X(x), Y(y) {}  void nastavX(int x) { X = x; }  void nastavY(int y) { Y = y; }  int dejX() const { return X; }  int dejY() const { return Y; }  void operator()(Bod &b) const  {  b.nastavX(b.dejX() + this->dejX());  b.nastavY(b.dejY() + this->dejY());  }  };  ostream &operator<<(ostream &o, const Bod &b)  {  o << "Bod: << b.dejX() << " Y = " << b.dejY();  return o;  }  int main()  {  int pole[7] = {1 , 2, 100, 23, 43, 56, 75 };  for\_each(pole,&pole[7],funkce);  vector<Bod> body;  Bod b1(0,0), b2(10,10), b3(-100, 1000), b4(10,7);  body.push\_back(b1);  body.push\_back(b2);  body.push\_back(b3);  body.push\_back(b4);  copy(body.begin(),body.end(),ostream\_iterator<Bod>(cout,"\n"));  cout << endl;  /\* Teď posuneme body o 10 jednotek na ose x i y \*/  Posun posun(10,10);  for\_each(body.begin(),body.end(),posun);  copy(body.begin(),body.end(),ostream\_iterator<Bod>(cout,"\n"));  cout << endl;  /\* Teď posuneme body o -10 jednotek na ose x i y. Vrátíme je zpět\*/  posun.nastavX(-10);  posun.nastavY(-10);  for\_each(body.begin(),body.end(),posun);  copy(body.begin(),body.end(),ostream\_iterator<Bod>(cout,"\n"));  cout << endl;  return 0;  } | |

Určitě je dobré si pro procvičení tento příklad rozšířit. Pomocí znalostí, které již máme s tohoto a předchozích článků můžeme například zjistit kolik bodů je ve kterém kvadrantu. Vyhledat body v daném kvadrantu, nebo v dané oblasti, atd... Všem, kteří nemají příliš zkušenosti s používáním standardních algoritmů toto doporučuji jako cvičení. Konečně nemáme v kontejneru int, char, nebo jiné primitivní datové typy.  
Funkce, nebo operátor () funkčního objektu mohou ve svém těle volat i nekonstantní metody. V mém příkladě volám metody nastavX a nastavY. Pomocí for\_each tedy lze měnit prvky v kontejneru. Někdy by bylo ale lepší, kdyby algoritmus na základě svého argumentu vytvořil nový objekt (přetransformoval parametr), který by byl vložen do kontejneru místo původního. Takový algoritmus v C++ existuje. Jmenuje se transform. Transformačních algoritmů je v C++ více. Budeme se jim věnovat v příštím článku.

[Transformační algoritmy v C++](http://www.builder.cz/rubriky/c/c--/transformacni-algoritmy-v-c--156022cz)

**5. února 2002, 00.00 | Dnes si ukážeme algoritmy pro transformace kontejneru. Podíváme se na algoritmy replace, reverse a transform. Algoritmy nám umožňují měnit prvky v kontejneru podle zadaných kritérií. Vše je součástí STL.**

Transformační algoritmy jsou algoritmy, které nějakým způsobem mění (přetransformují) kontejner, nebo jeho část. Mohou změnit hodnoty prvků (algoritmy replace), mohou obrátit pořadí prvků (algoritmy reverse), nebo mohou kontejner přetransformovat tak, že na každý prvek zavolají funkci, nebo operátor () funkčního objektu. Některé varianty algoritmů mění přímo jim daný kontejner, jiné vytvoří nový kontejner, do kterého uloží výsledek. Originální kontejner pak zůstane nezměněn.

Nejprve se podívejme na algoritmus transform. O něm jsem se zmínil již [minule](http://www.builder.cz/rubriky/c/c--/clanek-156016-skenovaci-prohlizeci-algoritmy-v-c-/), když jsme se zabývali algoritmem for\_each. Algoritmus transform je tak jako každý jiný algoritmus z STL šablona funkce. Existují dvě varianty transform. Deklarace jsou:

* template <class InputIterator,class OutputIterator,class TUnarniOperace> OutputIterator transform (InputIterator zacatek, InputIterator konec, OutputIterator zacatekVysledku, TUnarniOperace operace); - Parametry šablony jsou typy vstupních a výstupních [iterátorů](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/). Dále je parametrem typ unární operace. Může se jednat buď o typ ukazatele na funkci, nebo o třídu [funkčních objektů](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/). Parametry funkce jsou začáteka konec kontejneru, nebo jeho části, která má být transformována. Dalším parametrem je začátek oblasti výsledku. Posledním parametrem je unární operace. Činnost algoritmu se dá popsat takto. Pro všechny prvky počínaje prvkem který je dán iterátorem začátek a konče prvkem před prvkem daným iterátorem konec bude postupně provedena daná operace, která má prvek jako svůj parametr. Návratová hodnota této operace bude vložena na pozici danou iterátorem zacatekVysledku. Chceme-li navíc transformovat kontejner, jehož iterátory mají vlastnosti i vstupních i výstupních iterátorů, lze výsledek zapisovat rovnou do originálního kontejneru. Vše je uvedeno v příkladu.
* template <class InputIterator1, class InputIterator2, class OutputIterator, class TBinarniOperace> OutputIterator transform (InputIterator1 zacatek1, InputIterator1 konec1, InputIterator2 zacatek2, OutputIterator zacatekVysledku, TBinarniOperace operace); - Parametry šablony jsou dva typy vstupních iterátorů. Bude se pracovat s dvěma vstupními kontejnery, proto pro každý jeden. Dále typ výstupního iterátoru a typ binární operace. Parametry funkce jsou začátek a konec oblasti v prvním kontejneru. Začátek oblasti v druhém kontejneru, následuje začátek oblasti pro výsledek a binární operace. Algoritmus postupně provede binární operaci nad prvky prvního a druhého kontejneru, a výsledek uloží na pozici pro výsledek. Tedy:
* \*začátekVýsledku = operace(\*začátek1,\*začátek2);
* \*(začátekVýsledku + 1) = operace(\*(začátek1 + 1),\*(začátek2 + 1));
* atd...

Předpokládá se, že je k dispozici potřebný počet prvků za prvkem, který je dán iterátorem začátek2. Stejně tak musí být zajištěno, že je dostatek místa v kontejneru od pozice dané iterátorem zacatekVysledku.

Další transformující algoritmy jsou reverse a reverse\_copy. Deklarace jsou:

* template <class BidirectionalIterator> void reverse (BidirectionalIterator zacatek, BidirectionalIterator konec); - V oblasti dané iterátory začátek a konec obrátí pořadí prvků. Tedy první prvek bude poslední a naopak, druhý předposlední a naopak, atd...
* template <class BidirectionalIterator, class OutputIterator> OutputIterator reverse\_copy (BidirectionalIterator zacatek, BidirectionalIterator konec, OutputIterator zacatekVysledku); - V podstatě stejná činnost jako výše uvedený algoritmus. Rozdíl je jen v tom, že originální kontejner nebude nijak pozměněn. Prvky v obráceném pořadí budou dány do jiného kontejneru na pozici začátekVýsledku. Bude tedy vlastně vytvořena kopie dat.

Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<vector>  #include<algorithm>  #include<functional>  #include<stdlib.h>  using namespace std;  int main(int, char\*\*)  {  int pole[10] = {-1, 30, 43, -20, 80, -76, 100, 193, -456, 354};  vector<int> vektor(pole,&pole[10]);  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Všechny prvky v poli vynásobím deseti.  \*/  transform(pole,&pole[10],pole,bind1st(times<int>(),10));  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Všechny prvky v poli nahradím jejich absolutní hodnotou.  \*/  transform(pole,&pole[10],vektor.begin(),abs);  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Sečtu prvky v poli s prvky ve vektoru.  Výsledek uložím opět do vektoru.  \*/  transform(pole,&pole[10],vektor.begin(),vektor.begin(),plus<int>());  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Obrátím pořadí prvků v poli.  \*/  reverse(pole,&pole[10]);  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Do vektoru uložím prvky z pole v obráceném pořadí.  \*/  reverse\_copy(pole,&pole[10],vektor.begin());  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  return 0;  } | |

Další užitečné šablony funkcí jsou šablony replace. Jejich deklarace:

* template <class ForwardIterator, class Typ> void replace (ForwardIterator zacatek, ForwardIterator konec, const Typ& puvodniHodnota, const Typ& novaHodnota); - Parametry šablony jsou typ iterátoru a typ prvků v kontejneru. Parametrem funkce jsou iterátory začátek a konec. Dále původní hodnota a nová hodnota. Algoritmus v úseku daným iterátory začátek a konec nahradí všechny původní hodnoty za hodnoty nové. K porovnání bude použit operátor ==.
* template <class ForwardIterator, class TPodminka, class Typ> void replace\_if (ForwardIterator zacatek, ForwardIterator konec, TPodminka podminka, const Typ& novaHodnota); - Obdobně jako předchozí algoritmus. Pouze hodnoty, které mají být nahrazeny nejsou dány konstantní hodnotou, ale podmínkou. K nahrazení dojde, je-li podmínka pravdivá.
* template <class InputIterator, class OutputIterator, class Typ> OutputIterator replace\_copy (InputIterator zacatek, InputIterator konec, OutputIterator zacatekVysledku, const Typ& puvodniHodnota, const Typ& novaHodnota); - V podstatě stejná činnost jako u algoritmu replace s tím rozdílem, že původní kontejner zůstane nezměněn. Jeho "změněná kopie" bude v jiném kontejneru na pozici dané iterátorem začátek výsledku.
* template <class InputIterator, class OutputIterator, class TPodminka, class Typ> OutputIterator replace\_copy\_if (InputIterator zacatek, InputIterator konec, OutputIterator zacatekVysledku, TPodminka podminka, const Typ& novaHodnota); - Kombinace algoritmů replace\_if a replace\_copy.

Srozumitelnější bude příklad.

|  |  |
| --- | --- |
| |  | | --- | | #include<algorithm>  #include<iostream>  #include<vector>  using namespace std;  int main(int, char\*\*)  {  int pole[10] = {1, 2, 3, -10, -20, -30, 2, 54, -78, 10};  vector<int> vektor;  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Nahradím všechny čísla 2 čísly -2  \*/  replace(pole,&pole[10],2,-2);  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Nahradím všechny záporné čísla číslem 0  \*/  replace\_if(pole,&pole[10],bind2nd(less<int>(),0),0);  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  insert\_iterator<vector<int> > ins(vektor,vektor.begin());  /\*  Do vektoru uložím všechny čísla z pole, a při tom nahradím všechny  čísla 10 čísly 100.  \*/  replace\_copy(pole,&pole[10],ins,10,100);  copy(pole,&pole[10],ostream\_iterator<int>(cout,"\t"));  cout << endl;  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  return 0;  } | |

Doporučuji porovnat dnešní algoritmy, které mají ve svém názvu copy s podobnými algoritmy, se kterými jsme se již setkali. Bylo to v článku [Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156005-kopirovaci-a-presouvaci-algoritmy-v-c-/).

Příště se podíváme na řadící algoritmy. I řadící algoritmy jsou v C++ již implementovány. Nemusíte žádný quick sort sami psát. Vše je již naprogramováno, stačí jen dané algoritmy používat.

[Řadící algoritmy v C++](http://www.builder.cz/rubriky/c/c--/radici-algoritmy-v-c--156023cz)

**14. února 2002, 00.00 | Dnes se podiváme na řadící algoritmy v C++. V STL existují již hotové šablony. Podíváme se také, jaké jsou možnosti řadit v jayzce C. Řadící algoritmy slouží k seřazení pole, nebo kontejneru. Někdy jsou řadící algoritmy nepřesně nazývány třídicí.**

Řadící algoritmy jsou algoritmy, které seřadí prvky v kontejneru, nebo jeho části. Řadícím algoritmům se často říká třídicí algoritmy. Není to ale přesné označení, protože pod pojmem třídicí algoritmus by jsme si měli představit spíše něco, co rozděluje objekty podle toho, jaké jsou třídy. Prvky lze seřadit podle nějaké relace. Algoritmy z STL k řazení používají buď operátor <, který může být implicitní, nebo přetížený. Dále také umožňují programátorovi zadat funkci vracející bool, která definuje relaci, podle níž mají být prvky seřazeny.

Řazení v C

Ještě než se začneme věnovat řadícím šablonám z STL chtěl bych připomenout, že existuje možnost standardního řazení i v ANSI C (nejenom v C++). Jazyk C nezná šablony. K řazení je v C k dispozici funkce qsort deklarovaná v hlavičkovém souboru stdlib.h. Její deklarace je:void qsort(void \*pole, size\_t pocet, size\_t velikost, int (\*porovnani)(const void \*prvni, const void \*druhy));. Prvním parametrem je ukazatel na pole, které chceme seřadit. Jedná se vlastně o ukazatel na první prvek pole (prvek s indexem 0). Druhým parametrem je počet prvků, které chceme seřadit. Následuje velikost jednoho prvku a ukazatel na porovnávací funkci. Mluvíme o jazyku C, nikoliv C++, proto žádné přetěžování operátoru < není možné. Ukazatel na funkci ukazuje na funkci vracející int mající dva parametry, což jsou ukazatele na void. Tato funkce by měla vracet libovolné záporné číslo, je-li \*a < \*b, měla by vracet 0, jestliže \*a == \*b, a vracet libovolné kladné číslo, jestliže \*a > \*b. Tím, že zadáváme porovnávací funkci, i velikost prvku, je funkce qsort univerzální. Je použitelná pro pole jakýchkoliv prvků. Uvedeme si příklad. Nejprve seřadíme pole int, potom pole bodů v 2D podle jejich vzdálenosti k bodu 0,0. Úmyslně program napíši v ANSI C, nepoužiji nic z ANSI C++.

|  |  |
| --- | --- |
| |  | | --- | | #include<stdio.h>  #include<stdlib.h>  /\*  Pozor! Tento příklad je C, ne C++! Žádné jednořádkové komentáře.  Žádné class, count, atd... :-)  \*/  struct TBod  {  int X,Y;  };  int porovnejInt(const void \*a, const void \*b)  {  /\*  Budu vracet kladné, záporné číslo, nebo 0, podle pravidel,  která jsem uvedl.  \*/  if ( \*((int\*)a) < \*((int\*)b) )  {  return -1;  }  if ( \*((int\*)a) > \*((int\*)b) )  {  return 1;  }  return 0;  }  int porovnejBody(const void \*a, const void \*b)  {  /\*  Nejprve vytvořím ukazatele typu TBod, abych  nemusel pořád přetypovávavat.  \*/  struct TBod \*bodA = (struct TBod\*)a, \*bodB = (struct TBod\*)b;  /\* Nyní spočítám čtverce vzdáleností. \*/  int mocninaVzdalenostiA = bodA->X \* bodA->X + bodA->Y \* bodA->Y;  int mocninaVzdalenostiB = bodB->X \* bodB->X + bodB->Y \* bodB->Y;  if (mocninaVzdalenostiA < mocninaVzdalenostiB)  {  return -1;  }  if (mocninaVzdalenostiA > mocninaVzdalenostiB)  {  return 1;  }  return 0;  }  int main(int argc, char\*\* argv)  {  int p,pole[10] = {12, 35, 100, -14, 45, -89, 8, 87, 100, -100};  struct TBod body[5] = { {0,100} , {20 , 50}, {-98,45}, {0,0}, {100,1000} };  qsort((void\*)pole,10,sizeof(int),porovnejInt);  qsort((void\*)body,5,sizeof(struct TBod),porovnejBody);  for(p = 0; p < 10; p++)  {  printf("%d ",pole[p]);  }  for(p = 0; p < 5; p++)  {  printf("\n(%d,%d)",body[p].X,body[p].Y);  }  return 0;  } | |

Tolik k funkci qsort z jazyka C. Uvedl jsem ji, protože se na ni často zapomíná.

Řazení v C++

Jazyk C++ po C "zdědil" funkci qsort. Můžeme ji tedy používat i v C++. Já to ale nedoporučuji. Funkce qsort umí uspořádat pouze pole. Neuspořádáme s ní žádný kontejner z STL. V C++ máme k dispozici šablony funkcí, které umí uspořádat i pole, i datové kontejnery. Než se jim budeme věnovat, podívejme se na trochu teorie o "stabilním" řazení.

Stabilní řazení je řazení, které garantuje, že prvky, které mají stejný klíč (podle kterého se prvky řadí) vůči sobě nezmění pořadí. U "nestabilního" řazení může být tato vlastnost také splněna, ale nemáme jistotu, že tomu tak bude vždy.

K řazení v C++ existují algoritmy sort a stable\_sort. Deklarace:

* template <class RandomAccessIterator> void sort (RandomAccessIterator zacatek, RandomAccessIterator konec); - Seřadí oblast danou iterátory začátek a konec podle operátoru <. Operátor < může být implicitní, nebo přetížený. Parametrem šablony je typ iterátoru, parametry funkce jsou iterátory udávající oblast pro seřazení.
* template <class RandomAccessIterator, class TPorovnani> void sort (RandomAccessIterator zacatek, RandomAccessIterator konec, TPorovnani porovnani); - Stejná činnost jako u předchozího algoritmu. Rozdíl je jen v tom, že k porovnání dvou prvků nebude použit operátor <, ale funkce, nebo funkční objekt zadaný programátorem. TPorovnani je buď třída funkčních objektů, nebo ukazatel na funkci. Musí mít dva parametry stejného typu (prvky), a vracet typ bool, případně celé číslo.
* template <class RandomAccessIterator> void stable\_sort (RandomAccessIterator zacatek, RandomAccessIterator konec); - Význam parametrů je stejný jako u algoritmu sort. Rozdíl je jen v tom, že stable\_sort řadí stabilně.
* template <class RandomAccessIterator, class TPorovnani> void stable\_sort (RandomAccessIterator zacatek, RandomAccessIterator konec, TPorovnani porovnani); - Význam parametrů je stejný jako u algoritmu sort. Rozdíl je jen v tom, že stable\_sort řadí stabilně.

Ukážeme si vše v příkladu.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<vector>  #include<algorithm>  #include<functional>  #include<string>  using namespace std;  int main(int, char\*\*)  {  int pole[5] = {1, 80, -87, 25, 0 };  vector<int> vektor(pole,&pole[5]);  /\* Seřadím pole vzestupně. \*/  sort(pole,&pole[5]);  copy(pole,&pole[5],ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\*  Teď prvky vektoru seřadím sestupně. Použiji stabilní řazení,  i když zde se to nemá jak projevit!  \*/  sort(vektor.begin(),vektor.end(),greater<int>());  copy(vektor.begin(),vektor.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  return 0;  } | |

Příště se podíváme haldu, povíme si co to je, a ukážeme si standadní operace nad haldou, které nám C++ poskytuje.

[Halda v C++](http://www.builder.cz/rubriky/c/c--/halda-v-c--156036cz)

**21. února 2002, 00.00 | Dnes si ukážeme jak v C++ pracovat s haldou. Ukážeme si standardní funkce z STL pro práci s haldou. Také si zjednodušeně vysvětlíme, co pojem halda vlastně znamená.**

Než se začneme věnovat algoritmům pro práci s haldou, které jsou v STL, vysvětlíme si, co vlastně halda je. Nejprve pár matematických pojmů.

Graf

Graf je uspořádaná dvojce (V,H), kde V je množina vrcholů (uzlů) a H je množina hran. Hrana je dvojce vrcholů. Za touto definicí se nesmí hledat žádná magie. Pro názornost uvedu příklad. Představme si 4 počítače. Označme si je A, B, C, D. Počítače jsou propojeny v síti. Počítač A je propojen s B a s C. Počítač B je tedy spojen s A a dále s D. Počítač C je propojen s počítači A, D. Počítač D je propojen s B a C. Je dobré si tuto síť nakreslit. Jednotlivé počítače budeme považovat za vrcholy. Spoje mezi počítači jsou "dráty" se dvěma konci. Každý "drát" lze přesně určit pomocí počítačů na jeho dvou koncích. Jedná se o hrany. Naše počítačová síť je graf, tedy uspořádaná dvojce ({A,B,C,D} , { (A,B) , (A,C) , (B,D) , (C,D) } ). Stejně tak graf je například i železnice. Vrcholy jsou nádraží a hrany jsou koleje spojující nádraží. Graf je v informatice poměrně častý pojem. V grafu je definováno mnoho algoritmů, například nalezení nejlevnější cesty, atd...

Cesta v grafu

Cesta v grafu z počátečního vrcholu do koncového je posloupnost vrcholů, které jsou spojeny hranou. Tato posloupnost začíná počátečním vrcholem, a končí koncovým. Definice by se dala uvést exaktněji, ale pro pochopení myslím stačí. Například v naší počítačové síti je cesta z A do D třeba posloupnost (A,B,D), nebo (A,C,D), ale také třeba (A,B,D,C,A,C,D).

Souvislý graf

Souvislý graf je graf, ve kterém existuje alespoň jedna cesta mezi libovolnou dvojicí vrcholů. Naše počítačová síť je souvislá. Kdyby jsme ale přidali další počítač (nazvěme ho E), který by nebyl propojen s žádným jiným počítačem, graf by už nebyl souvislý. Protože např z A do E nevede žádná cesta. Když přidáme počítač F, který bude propojen s E, stále bude graf nesouvislý. Pořád nevede cesta z A do E.

Strom

Každý, kdo programuje už asi o stromu něco slyšel, zvláště o binárním stromu. Strom je specielní případ grafu. Strom je souvislý graf, ve kterém mezi libovolnými vrcholy vede právě jedna cesta. Strom je souvislý graf, protože mezi každými vrcholy existuje jedna cesta. Náš příklad počítačové sítě není strom. Jak jsem uvedl u definice cesty, například z A do D vede více než jedna cesta. Zrušíte-li v našem grafu (poč. síti) jednu libovolnou hranu, vznikne strom. Můžete si to nakreslit a přesvědčit se o tom.

Binární strom

Binární strom je strom, kde každý vrchol má maximálně 2 následovníky. Hodně populární formou lze říci, že jednou hranou jsem do vrcholu přišel, a dále mám maximálně dvě možné hrany, po kterých jít dál.

Halda

Halda je binární strom, který má ohodnocené vrcholy. Ohodnocení vrcholů znamená, že ke každému vrcholu je přiřazena nějaká jeho hodnota. Například číslo. Aby byl ohodnocený binární strom haldou, musí platit, že každý vrchol NEMÁ menší hodnotu než kterýkoliv z jeho následovníků. Často se halda definuje tak, že ohodnocení vrcholu není větší, než ohodnocení jeho následovníků. Standardní halda v C++ to má ale naopak. Není to příliš důležité. Důležitý je spíše fakt, že existuje nějaká relace, která dokáže prvky seřadit. I v C++ budeme mít možnost tuto relaci zadat. Pro haldu ale platí ještě jedna podmínka ohledně uspořádání vrcholů a hran. Vysvětlím to opět velmi zjednodušeně. Mám graf o jednom vrcholu - což je halda. Mám tím pádem první řadu vrcholů. Přidám nový vrchol do druhé řady zleva. Další vrchol přidám zase zleva vedle již existujících vrcholů. Až dokončím druhou řadu, začnu vytvářet novou řadu opět zleva. Novou řadu dokončím tehdy, když každý vrchol předchozí řady má dva následníky. V haldě má tedy každý vrchol, který není v poslední a předposlední řadě dva následníky. V poslední řadě jsou vrcholy bez následovníků. V předposlední řadě mohou mít vrcholy 2, 1, nebo žádného následníka. V předposlední řadě jsou z leva vrcholy s dvěma následníky, jestliže existují, potom je jeden, nebo žádný vrchol s jedním následovníkem a následují vrcholy bez následovníků, pokud existují.

Implementace haldy

Haldu lze velmi efektivně uchovávat v poli. Kořen haldy by byl první prvek v poli. Jeho dva následovníci by byli druhý a třetí prvek v poli. Následovníci druhého prvku v poli by byli 4. a 5. prvek v poli atd... Tedy i-tý prvek v poli by měl následovníky na pozicích 2 \* i, 2 \* i + 1. Indexy 2 \* i, nebo 2 \* i + 1 mohou být mimo rozsah pole. V takovém případě následníci neexistují. Protože v C i v C++ se indexuje pole od nuly, nikoliv od jedné musíme si naše dva vzorce pro polohu následovníků upravit takto: i-tý prvek má za své následovníky prvky s indexem 2 \* i + 1, 2 \* i + 2, nebo prvky nemá, jeli výsledný index mimo rozsah pole. To bylo jen tak pro zajímavost, pro práci se standardní haldou v C++ nic takového vědět nemusíme.

Halda v C++

V C++ neexistuje pro haldu nějaký kontejner, nebo adaptér kontejneru. Haldu můžeme vytvořit v libovolném kontejneru mezi zadanými iterátory. Pro práci s haldou existují standardní algoritmy make\_heap, pop\_heap, push\_heap, sort\_heap.

* template <class RandomAccessIterator> void make\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec); - vytvoří v úseku daném iterátory začátek a konec haldu. Každý prvek haldy není menší, než jeho následovníci. Halda bude v posloupnosti uložena tak, jak jsem uvedl v odstavci o implementaci haldy. Haldu lze vytvořit v kontejneru, kterým lze procházet pomocí iterátorů s náhodným přístupem. Já vám doporučuji používat vector, ale není to podmínka.
* template <class RandomAccessIterator, class TRelace> void make\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRelace relace); - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
* template <class RandomAccessIterator> void push\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec); - slouží k vložení prvku na haldu. Předpokládá se, že v rozmezí daném iterátory začátek a konec - 2 prvky je halda. Poslední prvek (konec - 1 prvek ; prvek hned před prvkem na který ukazuje iterátor konec) je nový prvek, který do haldy vkládám. Vkládání (stejně jako odebírání) prvku na haldu je trochu zvláštní. Uvedl jsem jej v příkladu.
* template <class RandomAccessIterator, class TRelace> void push\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRealce realce); - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
* template <class RandomAccessIterator> void pop\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec); - slouží k odebrání prvku z haldy.
* template <class RandomAccessIterator, class TRelace> void pop\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRealce realce); - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude halda organizována. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.
* template <class RandomAccessIterator> void sort\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec); - převede haldu v oblasti dané iterátory začátek a konec na seřazenou posloupnost.
* template <class RandomAccessIterator, class TRelace> void sort\_heap(RandomAccessIterator zacatek, RandomAccessIterator konec, TRelace relace); - obdobně jako minulá funkce. Jen je nový parametr udávající relaci, podle které bude posloupnost seřazena. Může se opět jednat o funkční objert, nebo o ukazatel na funkci.

Metody pop\_heap, push\_heap vlastně nevloží, případně neodeberou prvek s haldy. Slouží pouze k přeorganizování haldy. Chci-li vložit nový prvek, vložím nový prvek do kontejneru a poté zavolám push\_heap. Chci-li odebrat prvek s haldy (odebírá se vždy kořen), přečtu si první prvek na haldě, zavolám pop\_heap a odeberu poslední prvek z kontejneru. Funkce pop\_heap vlastně přehodí první prvek s posledním a přeorganizuje haldu bez posledního prvku. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<algorithm>  #include<iostream>  #include<vector>  using namespace std;  int main()  {  int pole[10] = {7, 3, 2, 1, 1, 0, 8, 9, 4, 5};  vector<int> v(pole,&pole[10]);  // Vytvořím haldu  make\_heap(v.begin(), v.end());  cout << "Halda:" << endl;  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  // Vložím prvek  v.push\_back(5);  cout << "Halda pred operaci push\_heap:" << endl;  copy(v.begin(),v.end(),ostream\_iterator(cout,"\t"));  cout << endl;  push\_heap(v.begin(),v.end());  cout << "Halda po operaci push\_heap:" << endl;  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  // Vyberu prvek:  pop\_heap(v.begin(),v.end());  cout << "Halda po operaci pop\_heap:" << endl;  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  cout << "Odebral jsem: " << v.back() << endl;  v.pop\_back();  // Setřídím haldu  sort\_heap(v.begin(),v.end());  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  /\* Použiji jinou relaci: \*/  make\_heap(v.begin(),v.end(),greater<int>());  cout << "Pouziji jinou relaci:" << endl;  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  v.push\_back(2);  push\_heap(v.begin(),v.end(),greater<int>());  copy(v.begin(),v.end(),ostream\_iterator<int>(cout,"\t"));  cout << endl;  return 0;  } | |

Halda má mnoho využití. Na haldě je založeno řazení heap-sort. Pomocí haldy lze implementovat prioritní frontu atd...

V úvodu článku jsem některé matematické pojmy definoval velmi zjednodušenou, populární formou. Snad se skuteční matematici nebudou zlobit. Příště si shrneme vše, co jsme si o standardních algoritmech řekli. Tím téma standardních algoritmů ukončíme.

[Standardní algoritmy v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/standardni-algoritmy-v-c-dokonceni-156045cz)

**27. února 2002, 00.00 | Dnes dokončíme téma standardních algoritmů z STL. Vše si shrneme v přehledné tabulce. Navíc se podíváme na algoritmus adjacent\_find.**

Dnes dokončíme téma standardních algoritmů z STL. Ve svých poslední několika článcích jsem se zabýval některými šablonami funkcí z STL. Dnes se pokusím vše shrnout do přehledné tabulky, ze které by mělo být zřejmé, kdy jaký algoritmus použít. Ještě než začneme přehledem musím jen upozornit, že algoritmy o kterých jsem psal nejsou zdaleka všechny.

|  |  |  |  |
| --- | --- | --- | --- |
| Použití | Popis | Jména šablon | V článku |
| Algoritmy pro vyplňování kontejnerů. | Algoritmy vyplní kontejner nebo jeho část určenou hodnotou. Hodnota může být konstantní, nebo proměnná. | fill, fill\_n, generate, generate\_n | [Úvod do standardních algoritmů v C++](http://www.builder.cz/rubriky/c/c--/clanek-155983-uvod-do-standardnich-algoritmu-v-c-/) |
| Algoritmy pro kopírování dat | Algoritmy kopírují data mezi datovými kontejnery. Data lze kopírovat "ze předu", nebo "ze zadu". | copy, copy\_backward | [Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156005-kopirovaci-a-presouvaci-algoritmy-v-c-/) |
| "Přesouvací" algoritmy | Algoritmy ve skutečnosti data nepřesunují. Některé algoritmy data z kontejneru smažou, jiné vytvoří kopie určených prvků. Slovo přesouvací je nepřesný překlad slova remove. | remove, remove\_copy, remove\_if, remove\_copy\_if | [Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156005-kopirovaci-a-presouvaci-algoritmy-v-c-/) |
| Algoritmy pro vyhledávání dat | Algoritmy vyhledávají data v kontejnerech podle různých kritérií. Existují jak sekvenční vyhledávací algoritmy, tak i binární vyhledávací algoritmy. | find, find\_if, search, binary\_search | [Vyhledávací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156011-vyhledavaci-algoritmy-v-c-/) |
| Prohledávací ("skenovací") algoritmy | Algoritmy prohlédnou kontejner, ve kterém provedou pro každý prvek nějakou operaci. Může se jednat přímo o zavolání funkce, kde je daný prvek parametrem (algoritmus for\_each), nebo může třeba jenom zvýšit čítač, tedy počítat počet nějakých prvků. | for\_each, count, count\_if, accumulate | [Skenovací (prohlížecí) algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156016-skenovaci-prohlizeci-algoritmy-v-c-/) |
| Algoritmy pro transformaci | Algoritmy změní (přetransformují) data v kontejneru. Transformace se provádějí podle různých kritérií. | transform, reverse, reverse\_copy, replace, replace\_if, replace\_copy, replace\_copy\_if | [Transformační algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156022-transformacni-algoritmy-v-c-/) |
| Řadící algoritmy | Algoritmy seřadí (nepřesně řečeno třídí) data v kontejneru. | sort, stable\_sort | [Řadící algoritmy v C++](http://www.builder.cz/rubriky/c/c--/clanek-156023-radici-algoritmy-v-c-/) |
| Algoritmy pro práci s haldou | Algoritmy pracují se standardní haldou. Jsou zde algoritmy pro vytvoření hady, přidání a odebrání prvku z haldy, převedení haldy na seřazenou posloupnost. | make\_heap, pop\_heap, push\_heap, sort\_heap | [Halda v C++](http://www.builder.cz/rubriky/c/c--/clanek-156036-halda-v-c-/) |
| Množinové operace | Algoritmy pro práci s množinou a multimnožinou. Nalezneme zde algoritmy pro průnik, sjednocení, podmnožinu, množinový rozdíl, symetrickou diferenci. | set\_intersection, set\_union, set\_difference, set\_symmetric\_difference, includes | [Množina v C++](http://www.builder.cz/rubriky/c/c--/clanek-155952-mnozina-v-c-/) |
| Algoritmy, které nepracují s kontejnery | Algoritmy, které nepracují s kontejnery. Provádí většinou nějakou jednoduchou činnost | min, max, swap | [Úvod do standardních algoritmů v C++](http://www.builder.cz/rubriky/c/c--/clanek-155983-uvod-do-standardnich-algoritmu-v-c-/) |

Znovu připomínám, že se nejedná o všechny algoritmy z STL.

Až na pár vyjímek jsou všechny algoritmy deklarovány v hlavičkovém souboru algorithm. Všechny jsou deklarované v [prostoru jmen](http://www.builder.cz/rubriky/c/c--/clanek-155766-prostory-jmen-v-c-/) std. Všechny jsou nezávislé na typu kontejneru. Ke kontejneru přistupují pomocí [iterátorů](http://www.builder.cz/rubriky/c/c--/clanek-155928-iteratory-v-c-/). Velmi často se při používání standardních algoritmů používají [funkční objekty](http://www.builder.cz/rubriky/c/c--/clanek-155955-funkcni-objekty-v-c-/).

V mnoha algoritmech se jako parametr udávají iterátory začátek a konec. Chtěl bych upozornit, že algoritmus proběhne vždy od prvku daného iterátorem začátek a konče prvkem před prvkem daným iterátorem konec. Prvek, na který se odkazuje iterátor konec již nebude použit. Pracujeme-li s celým kontejnerem, zadáváme jako konec iterátor za poslední prvek. Tedy iterátor, který vrací metoda end(). Pracujeme-li s polem, zadáme jako konec ukazatel na prvek za posledním prvkem. Můžete si být jisti, že ukazatel, nebo iterátor konec nebudou nikdy dereferencováni.

Algoritmus adjacent\_find

Na závěr bych se chtěl zmínit o algoritmu adjanced\_find. Tento algoritmus by se dal zařadit mezi vyhledávací algoritmy, ale já jsem na něj ve článku o vyhledávacích algoritmech nějak zapomněl. Jak jsem již napsal, neuváděl jsem všechny algoritmy, jen ty, které se mi zdály užitečné. Myslím, že adjanced\_find mezi užitečné patří. Deklarace:

* template <class ForwardIterator> ForwardIterator adjacent\_find(ForwardIterator zacatek, ForwardIterator konec); Algoritmus v oblasti dané iterátory nalezne dva po sobě jdoucí prvky, které jsou si rovny. K porovnání použije operátor ==. Existuje-li dvojce, funkce vrátí iterátor na první prvek dvojce. Neexistuje-li funkce vrátí iterátor konec. Parametry šablony jsou typy iterátorů.
* template <class ForwardIterator, class Podminka> ForwardIterator adjacent\_find(ForwardIterator zacatek, ForwardIterator konec, Podminka pod);Obdobná činnost jako předchozí. Jen s tím rozdílem, že nebude použit operátor ==, ale binární funkční objekt. Tedy nalezne první dvojici, pro které je pod(\*i,\*(i+1)) == true. Identifikátor i je iterátor.

Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<algorithm>  #include<vector>  #include<functional>  using namespace std;  int main()  {  int pole[5] = { 90, 30, 10, 10, 7 };  /\* Naleznu první dva prvky v poli, které jsou stejné \*/  int \*i = adjacent\_find(pole,&pole[5]);  if (i != &pole[5])  {  cout << "Stejná dvojce: " << \*i << " " << \*(i+1) << endl;  }  /\* Naleznu první dva prvky v poli, které nejsou dělitelné \*/  int \*ii = adjacent\_find(pole,&pole[5],modulus<int>());  /\*  Modulus je zbytek po celočíselném dělení.  Modulus vrací 0 (false) pro čísla dělitelná.  \*/  if (ii != &pole[5])  {  cout << "První dvě čísla, která nejsou beze zbytku dělitelná "  << \*ii << " " << \*(ii+1) << endl;  }  /\* Nyní to samé pro vektor \*/  vector<int> v(pole,&pole[5]);  vector<int>::iterator a = adjacent\_find(v.begin(),v.end());  vector<int>::iterator b = adjacent\_find(v.begin(),v.end(),modulus<int>());  if (a != v.end())  {  cout << "Stejná dvojce: " << \*a << " " << \*(a+1) << endl;  }  if (b != v.end())  {  cout << "První dvě čísla, která nejsou beze zbytku dělitelná "  << \*b << " " << \*(b+1) << endl;  }  return 0;  } | |

Tímto jsme téma standardních algoritmů z STL ukončili. U STL ale ještě zůstaneme. Příště se podíváme na tak zvané automatické ukazatele.

[Automatické ukazatele v C++](http://www.builder.cz/rubriky/c/c--/automaticke-ukazatele-v-c--156054cz)

**7. března 2002, 00.00 | Dnes si povíme něco o tak zvaném automatickém ukazateli. Automatický ukazatel je šablona třídy,   
která "zapouzdřuje" ukazatel na libovolný objekt. Objekt vytvoříme, a dále se již nemusíme starat o jeho dealkoci.**

Dnes si povíme něco o tak zvaném automatickém ukazateli. Automatický ukazatel je šablona třídy, která "zapouzdřuje" ukazatel na libovolný objekt. Objekt jednou vytvoříme, zapouzdříme jej v automatickém ukazateli, a již se nemusíme starat o jeho dealokaci. Automatický ukazatel jej odstraní (pomocí delete) automaticky sám ve svém destruktoru.

Automatický ukazatel je šablona z STL. Je deklarována v hlavičkovém souboru memory v prostoru jmen std. Šablona se jmenuje auto\_ptr. Má přetížené operátory \* a -> tak, aby nám práce s šablonou auto\_ptr co nejvíce připomínala práci s obyčejným ukazatelem. Parametrem šablony je typ, na který se bude odkazovat zapouzdřený ukazatel. Pozor parametrem tedy je samotný typ objektu, nikoliv typ ukazatel na objekt. Metody a přetížené operátory (Typ je zde parametr šablony):

* auto\_ptr(Typ \*pointer = NULL); - konstruktor. Jeho parametr je ukazatel, který chceme zapouzdřit. Nezadáme-li ukazatel, jeho implicitní hodnota je NULL.
* auto\_ptr(auto\_ptr<Typ> &original); - kopírovací konstruktor. Vytvoří kopii automatického ukazatele. Chování tohoto konstruktoru je asi jiné než by jsme mohli na první pohled čekat. Originální objekt totiž "předá" zapouzdřený ukazatel a sám si "svůj" zapouzdřený ukazatel nastaví na NULL. Obdobně se chová také operátor =. V jednom okamžiku tedy existuje pouze jeden zapouzdřený ukazatel, který si automatické ukazatele nekopírují, ale "předávají".
* ~auto\_ptr(); - destruktor. Zlikviduje automatický ukazatel, pokud je v automatickém ukazateli zapouzdřen ukazatel na jiný objekt, dojde k likvidaci tohoto objektu pomocí delete.
* operator=(auto\_ptr<Typ> &original); - obdobné chování jako kopírovací konstruktor. Originál předá, nikoliv zkopíruje zapouzdřený ukazatel.
* Typ &operator\* () const; - vrátí referenci na objekt, na který se odkazuje zapouzdřený ukazatel.
* Typ \*get() const; - vrátí zapouzdřený ukazatel.
* X \*operator-> () const; - umožní přístup k prvkům objektu, na který se odkazuje zapouzdřený ukazatel. V podstatě také vrací zapouzdřený ukazatel jako get(), ale v případě, že budeme přistupovat k prvkům objektu, nabízí přehlednější zápis.
* void reset(Typ \*pointer = NULL); - nastaví zapouzdření nového ukazatele. Objekt, na který se odkazoval starý ukazatel bude zlikvidován operátorem delete. Místo něj bude nyní nastaven nový zadaný ukazatel. Metoda reset vlastně přenastaví zapouzdřený ukazatel. Implicitní hodnota nového ukazatele je NULL.
* X \*release(); - vrátí zapouzdřený ukazatel. Zapouzdřený ukazatel bude nyní NULL. Objekt, na který se odkazoval "starý" ukazatel, který je vrácen, nezlikviduje. Metoda v podstatě slouží k "vysvobození" zapouzdřeného ukazatele. Použijeme ji tehdy, jestliže chceme, aby se automatický ukazatel přestal starat o náš normální ukazatel.

Vše si uvedeme na jednoduchém příkladě. Vytvoříme instanci naší pokusné třídy. Ukazatel na ni předáme automatickému ukazateli jménem a. Ukážeme si jak zavolat metodu pomocí operátoru ->. Potom vytvoříme automatický ukazatel b, který bude "kopie" a. Až skončí aktuální blok, automatický ukazatel b bude zlikvidován. S ním bude také zlikvidován objekt, na který se odkazoval zapouzdřený ukazatel. Když jsme vytvářeli b, přepsali jsme zapouzdřený ukazatel v a na NULL. O všem se přesvědčíte, jestliže program spustíte.

|  |  |
| --- | --- |
| |  | | --- | | #include<memory>  #include<iostream>  using namespace std;  class Trida  {  public:  Trida() { cout << "Konstruktor" << endl; }  ~Trida() { cout << "Destruktor" << endl; }  void metoda() { cout << "Metoda" << endl; }  };  int main()  {  auto\_ptr<Trida> a(new Trida);  a->metoda();  {  auto\_ptr<Trida> b(a); // Od teď je a.get() == NULL !  (\*b).metoda();  b.get()->metoda();  cout << "Objekt b konci svou platnost " << b.get() << endl;  }  cout << "Objekt b uz neexistuje" << endl;  /\* Nyní uvidíte, že a skutečně zapouzdřuje NULL. \*/  cout << "Objekt a konci svou platnost " << a.get() << endl;  return 0;  } | |

Já osobně automatickými ukazateli nejsem příliš nadšen, a nepoužívám je. Původně jsem se o nich nechtěl ani zmiňovat. Je ale dobré vědět, že existují. Hlavně jsem tento článek napsal jako úvod k mému příštímu článku.

Na automatických ukazatelích mi nejvíce vadí, že při jejich kopírování si zapouzdřený ukazatel pouze předávají. Daleko lepší by bylo, kdyby ukazatele kopírovali. Objekt by byl pouze jeden, na který by se odkazovalo více ukazatelů. Každý z těchto ukazatelů by byl zapouzdřen v automatickém ukazateli. Bylo by to krásné do té doby, než by jeden z automatických ukazatelů byl zlikvidován. Ve svém destruktoru by totiž dealokoval onen objekt. Ostatní ukazatele by se odkazovali do nealokované paměti. Kdyby ale někde existovala informace o tom, kolik takových ukazatelů se na objekt odkazuje, mohl by být objekt zničen až v době, kdy na něj neexistuje ukazatel. Tím by jsme udělali něco skvělého. Vytvořili by jsme objekt, se kterým by jsme pracovali pomocí našich "inteligentních" ukazatelů. O likvidaci tohoto objektu by jsme se již nestarali. Objekt by byl sám zlikvidován v momentě, kdy by na něj již neexistoval žádný odkaz. Myslíte, že něco takového v C++ nejde? V příštím článku něco takového uděláme.

[Inteligentní ukazatel - čítač referencí v C++](http://www.builder.cz/rubriky/c/c--/inteligentni-ukazatel-citac-referenci-v-c--156071cz)

**22. března 2002, 00.00 | Dnes implementujeme čítač referencí v C++. Představte si, že v C++ vytvoříte objekt, o který se již dále nemusíte starat. Objekt automaticky zanikne, v momentě, kdy na něj již neexistuje reference. Vše potřebné je na konci článku ke stažení.**

Ve svém předchozím článku o [automatických ukazatelích](http://www.builder.cz/rubriky/c/c--/clanek-156054-automaticke-ukazatele-v-c-/) jsme si povídali o šabloně auto\_ptr. V závěru článku jsme si pověděli o nevýhodách této šablony. Dnes se ji pokusíme vylepšit.

Našim úkolem je vytvořit něco (inteligentní ukazatel), čemu předáme ukazatel na již existující objekt. Tento ukazatel "zapomeneme" a k objektu budeme přistupovat jen pomocí "inteligentního" ukazatele. Takových ukazatelů na jeden objekt může být více. Nemusíme volat destruktor pro objekt. Destruktor bude automaticky zavolán v momentě, kdy na objekt nebude existovat již žádný ukazatel. Vše zajistíme implementací čítače referencí. Každý objekt bude vědět kolik je na něj ukazatelů. Nejprve si vytvoříme "obal" objektu, na který se budou odkazovat "inteligentní" ukazatele. Bude se jednat o šablonu třídy jménem \_ObjectWRC (Object with references counter). Parametrem šablony bude třída, na jejíž objekty se budeme odkazovat. Šablona \_ObjectWRC bude zapouzdřovat normální ukazatel na požadovaný objekt a čítač referencí. Čítač referencí je číslo udávající počet odkazů na objekt. Šablona \_ObjectWRC je pouze pomocná šablona. Uživatel by s ní neměl nijak přijít do styku. (Proto také podtržítko na začátku.) Implementace této šablony je jednoduchá.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> **class** \_ObjectWRC  {  **private**:  **unsigned** **int** ReferenceCount; *// Počet odkazů na objekt*  Type \*Data; *// Zapouzdřený ukazatel*  *// Metody, které nesmí být nikdy zavolány.*  \_ObjectWRC(**const** \_ObjectWRC<Type>&);  \_ObjectWRC &**operator**=(**const** \_ObjectWRC<Type>&);  **public**:  \_ObjectWRC(Type \*normalPointer)  : ReferenceCount(0), Data(normalPointer) {}  ~\_ObjectWRC() { **delete** Data; }  *// Metody pro manipulaci s daty:*  **const** Type \*getData() **const** { **return** Data; }  Type \*getData() { **return** Data; }  **void** setData(Type \*normalPointer) { Data = normalPointer; }  *// Metody pro práci s čítačem odkazů*  **unsigned** **int** getReferenceCount() **const** { **return** ReferenceCount; }  **unsigned** **int** increment() { **return** ++ReferenceCount; }  **unsigned** **int** decrement() { **return** --ReferenceCount; }  }; | |

Instance třídy \_ObjectWRC<něco> se nesmí nijak kopírovat. Proto jsou kopírovací konstruktor a operátor = soukromé. Jako parametr konstruktoru bude "normální" ukazatel na existující objekt. V konstruktoru musíme také vynulovat počet referencí. Dále jsme vytvořili metody set a get pro manipulaci s objektem. K dispozici je také metoda getReferenceCount, která vrátí počet referencí na objekt. Tuto službu poskytnou naše "inteligentní" ukazatele programátorovi. Programátor tedy bude vědět, kolik ukazatelů ukazuje na objekt, se kterým pomocí jednoho ukazatele pracuje. Poslední dvě metody slouží ke zvýšení a snížení počtu referencí na objekt.

Inteligentní ukazatel

Nyní vytvoříme šablonu Pointer, což bude náš "inteligentní" ukazatel. Šablona bude mít přetížené operátory tak, aby práce s ní nám co nejvíce připomínala práci s normálními ukazateli. Popíšu zde jen ty nejdůležitější metody. Obě šablony jsou kompletní k dispozici na konci článku. Pro implementaci platí pravidla:

* Má-li ukazatel ukazovat na nový objekt, sníží počet referencí na objekt, na který ukazoval dříve o 1.
* Má-li ukazatel ukazovat na nový objekt, zvýší počet referencí na nový objekt o 1.
* Je-li ukazatel likvidován (Například na konci aktuálního bloku.), sníží počet referencí na objekt o 1.
* Vzniká-li ukazatel, zvýší počet referencí na objekt, na který se bude odkazovat o 1.
* Pokaždé, když se snižuje počet referencí, ukazatel zkontroluje, jestli počet referencí již není 0. Jestliže ano, potom objekt zlikviduje destruktorem.

Implementace:

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> **class** Pointer  {  **private**:  \_ObjectWRC<Type> \*MyObject; *// Objekt s čítačem referencí*  **public**:  *// Konstruktory*  Pointer() : MyObject(NULL) {}  Pointer(\_ObjectWRC<Type> \*object);  Pointer(**const** Pointer<Type> &pointer);  Pointer(Type \*normalPointer);  *// Destruktor*  ~Pointer();  *// Operátory =*  Pointer<Type> &**operator**=(**const** Pointer<Type> &pointer);  Pointer<Type> &**operator**=(Type \*normalPointer);    *// Relační operátory*  **bool** **operator**==(**const** Pointer<Type> &pointer)  {  **return** **this**->MyObject == pointer.MyObject;  }  **bool** **operator**!=(**const** Pointer<Type> &pointer)  {  **return** !(**this**->MyObject == pointer.MyObject);  }  *// Operátory dereference*  **const** Type &**operator**\*() **const** **throw** (std::runtime\_error);  Type &**operator**\*() **throw** (std::runtime\_error);  *// Operátory -> (Operátory pro přístup k prvkům objektu)*  **const** Type \***operator**->() **const** **throw** (std::runtime\_error);  Type \***operator**->() **throw** (std::runtime\_error);  *// Metoda vracející počet ukazatelů na objekt, na který*  *// se odkazuje tento ukazatel. Vždy alespoň 1*  **unsigned** **int** getReferenceCount() **const**  {  **return** MyObject == NULL ? 0 : MyObject->getReferenceCount();  }  *// Metoda nastaví ukazatel na NULL*  **void** setNULL();    *// Metoda zjistí, zda-li je ukazatel NULL*  **bool** isNULL() { **return** MyObject == NULL; }  *// Metoda pro přetypováni ukazatele*  **template**<**class** NewType> Pointer<NewType> cast()  **const** **throw**(std::bad\_cast);  }; | |

Parametrem šablony je opět typ prvku, na který se bude odkazovat ukazatel. Atributem třídy je ukazatel na objekt s čítačem referencí. Poměrně jednoduché metody, které jsou inline jsou relační operátory != a ==, které porovnávají, zda ukazatele ukazují na stejný objekt. Dále metoda getReferenceCount vracející počet ukazatelů na objekt. Protože minimálně jeden ukazatel (ten, jehož metodu voláme) na objekt ukazuje, je výsledek vždy alespoň 1. Podívejme se na kopírovací konstruktor. Při volání kopírovacího konstruktoru vlastně vytváříme nový ukazatel. Musíme zvýšit počet referencí.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type>  Pointer<Type>::Pointer(**const** Pointer<Type> &pointer)  : MyObject(pointer.MyObject)  {  **if** (MyObject != NULL)  {  MyObject->increment();  }  } | |

Nastavíme si ukazatel a zvýšíme počet referencí. Obdobně vypadají oba operátory =. S tím rozdílem, že ještě také sníží počet referencí u starého objektu. Viz. zdrojový text na konci článku. Odečítání počtu referencí lze nejlépe vidět v destruktoru.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> Pointer<Type>::~Pointer()  {  **if** ( (MyObject != NULL) && (MyObject->decrement() == 0) )  {  **delete** MyObject;  }  MyObject = NULL;  } | |

Nemůžu zlikvidovat objekt, je-li na něj alespoň jeden odkaz. Objekt musím zlikvidovat, není-li již na něj žádný odkaz. Jinak by jej už nikdy nešlo zlikvidovat.

Nyní si vysvětlíme jeden z operátorů \*. Druhý je v podstatě stejný. Obdobně také vypadají operátory ->. Budeme-li se snažit dereferencovat ukazatel NULL, bude vyvržena vyjimka typu runtime\_error z prostoru jmen std.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type>  **const** Type &Pointer<Type>::**operator**\*() **const** **throw**(std::runtime\_error)  {  **if** (MyObject == NULL)  {  **throw** std::runtime\_error("Null pointer is dereferenced");  }  **register** Type \*ret = MyObject->getData();  **if** (ret == NULL)  {  **throw** std::runtime\_error("Null pointer is dereferenced");  }  **return** \*ret;  } | |

Nejprve jsme zjistili, jestli náhodnou není ukazatel NULL. Jestliže ano, je vyvržena vyjimka. Potom jsme pro jistotu zjistili, jestli instance \_ObjectWRC<něco> náhodou nezapouzdřuje NULL. Jestliže ano vyvrhneme vyjimku. Jestliže ne, vrátíme referenci na skutečný objekt.

Asi nejzajímavější a nejkontroverznější je vnořená šablona cast. Slouží k přetypování ukazatele. Nebýt vnořené šablony cast, měla by šablona Pointer problémy s přetypováním i s dědičnosti typů. Například máme-li nadtřídu, ze které dědí podtřída, při práci s obyčejnými ukazateli můžeme na místo ukazatele na nadtřídu kdykoliv dosadit ukazatel na podtřídu. Ale místo Pointer<nadtřída> stěží dosadíme Pointer<podtřída>. Budeme muset použít šablonu cast.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> **template**<**class** NewType>  Pointer<NewType> Pointer<Type>::cast() **const** **throw**(std::bad\_cast)  {  **if** (MyObject == NULL)  {  **return** Pointer<NewType>(**static\_cast**<NewType\*>(NULL));  }  **if** (**dynamic\_cast**<NewType\*> (MyObject->getData()) == NULL)  {  **throw** std::bad\_cast();  }  **return** Pointer<NewType>((\_ObjectWRC<NewType>\*)(MyObject));  } | |

Pomocí dynamic\_cast zjistím, jestli je přetypování možné. Jestliže ne, vyvrhnu vyjimku std::bad\_cast. Nakonec použiji úplně obyčejné přetypování ukazatelů. Podíváme-li se podrobněji na cast, zjistíme dva nedostatky.

* Budeme-li používat "inteligentní" ukazatele na primitivní datové typy, nebudeme moc naše ukazatele přetypovat. Tento problém by se asi dal vyřešit specializacemi.
* Vnořená šablona cast je naprosto nepoužitelná pro přetypování instancí tříd vzniklých vícenásobnou dědičností. Je to velký problém celého čítače referencí. Vůbec nevím jak ho řešit. Nápady přivítám v diskusi pod článkem.

Myslím ale, že tyto dva nedostatky nejsou zas tak zásadní. I s primitivními datovými typy, i s objekty tříd vzniklých vícenásobnou dědičností se pomocí našich "inteligentních" ukazatelů pracovat dá. Nejdou pouze přetypovávat. O různých řešeních těchto problému si povíme v příštím článku.

Dalším velkým problémem čítače odkazů jsou vazby mezi ukazateli, které tvoří kruh. Představme si graf, ve kterém jsou vrcholy (uzly) objekty, a hrany jsou naše ukazatele. Vznikne-li někde v grafu kruh (cyklus, smyčka), je s čítačem referencí trochu problém. Každý objekt v kruhu má jeden odkaz, proto nebude zničen. Ve skutečnosti ale mají být zničeny všechny, protože na celý kruh už odkaz není. O takovém kruhu musíme vědět, a včas (před ztrátou reference na něj) jej přerušit.

Článek je již příliš dlouhý, proto téma dokončíme příště. Povíme si, jak s našimi ukazateli pracovat. Jak je vytvářet a používat. Jak psát třídy, funkce, či metody nezávisle na tom, zda v nich budou používány obyčejné ukazatele, nebo naše "inteligentní" ukazatele. Povíme si také, jak použít naše ukazatele tam, kde se očekávají pouze ukazatele "normální". Ukážeme si praktické použití naší šablony a poukážeme na možné chyby při práci s ní. Zaujal-li Vás nápad čítače referencí, rozhodně si nenechte ujít příští článek. V tom dnešním je jen polovina informací.

Na závěr je tady šablona Pointer ke stažení. Jedná se o soubor [pointer.h](http://www.builder.cz/data/pointer.h). Šablona je deklarována v prostoru jmen www\_builder\_cz.

[Použití čítače referencí v C++](http://www.builder.cz/rubriky/c/c--/pouziti-citace-referenci-v-c--156074cz)

**28. března 2002, 00.00 | Dnes si ukážeme jak použít náš jednoduchý "garbage collector" v C++. Minule jsme pro tyto účely vytvořili šablonu Pointer. Dnes si ukážeme jak ji používat.**

V minulém článku [Inteligentní ukazatel - čítač referencí v C++](http://www./) jsme si implementovali šablonu Pointer. Jedná se o "inteligentní" ukazatel, který se odkazuje na objekt s čítačem referencí. Není-li na objekt žádná reference, objekt bude automaticky zlikvidován. Dnes si ukážeme jak s náš Pointer používat.

Nejprve si vytvoříme jednoduchou třídu. Instance této třídy budeme používat v následujících příkladech.

|  |  |
| --- | --- |
| |  | | --- | | **class** Trida  {  **private**:  **int** Pocet;  **public**:  Trida() : Pocet(0) { cout << "konstruktor" << endl; }  ~Trida() { cout << "destruktor" << endl; }  **void** metoda() { cout << "metoda " << ++Pocet << endl; }  **virtual** **void** constmetoda()  **const** { cout << "konstantni metoda" << endl; }  }; | |

Nyní vytvoříme "inteligentní" ukazatel na instanci třídy Třída. Parametr šablony je typ prvku, na který "inteligentní" ukazatel ukazuje. Parametrem konstruktoru může být jednak jiný "inteligentní" ukazatel (kopírovací konstruktor), nebo normální ukazatel na vytvořený objekt. Všimněte si také na příkladu, že lze použít i operátory == a !=. V příkladech nesmíme zapomenout, že Pointer je deklarován v souboru pointer.h v prostoru jmen www\_builder\_cz.

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  #include<exception>  #include"pointer.h"  **using** **namespace** std;  *// Tady musí být deklarace tříd uvedená na začátku článku !*  **int** main()  {  */\**  *Vytvořím objekt, který předám "inteligentnímu" ukazateli.*  *Více se o tento objekt nemusím starat. Bude automaticky*  *zlikvidován v momentě, kdy na něj již nebude existovat žádný*  *"inteligentní" ukazatel.*  *\*/*  www\_builder\_cz::Pointer<Trida> ptr(**new** Trida);  */\* S instancí ptr pracuji jako s normálním ukazatelem. \*/*  ptr->metoda();  (\*ptr).metoda();  www\_builder\_cz::Pointer<Trida> ptr2, ptr3;  **if** (ptr2.isNULL())  {  cout << "ptr2 je NULL" << endl;  }  **if** (ptr2 == ptr3)  {  cout << "Správně" << endl;  }  **else**  {  cout << "Špatně" << endl;  }  ptr2 = ptr;  ptr->metoda();  ptr2->metoda();  ptr2->constmetoda();  */\*Vytvářím nový objekt. Celkem už jsou vytvořeny 2.\*/*  ptr3 = **new** Trida;  **if** (ptr3 != ptr2)  {  cout << "Správně" << endl;  }  ptr3 = ptr2;  */\**  *Tímto je přepsán poslední odkaz na druhý vytvořený objekt.*  *Objekt je likvidován destruktorem.*  *\*/*  ptr2.setNULL();  **try**  {  ptr2->metoda();  }  **catch** (std::exception &e)  {  cout << e.what() << endl;  }  cout << "Konec" << endl;  **return** 0;  */\* Nyní bude automaticky volán destruktor prvního objektu. \*/*  } | |

K nastavení ukazatele na NULL slouží metoda setNULL(). Předáme-li jednou našemu "inteligentnímu" ukazateli normální ukazatel, je dobré odstranit všechny normální ukazatele na tento objekt. Inteligentní ukazatelé jej totiž zlikvidují až se jim to bude zdát vhodné. Na normální ukazatele neberou ohled. Buď budeme s ukazatelem pracovat pomocí normálních ukazatelů nebo pomocí šablony Pointer. Kombinovat obě možnosti je velmi nebezpečné.

Inteligentní ukazatel jako parametr funkce, či metody

Instance šablony Pointer může být bez problému použita jako typ parametru funkce.

|  |  |
| --- | --- |
| |  | | --- | | **void** fce(www\_builder\_cz::Pointer<Trida> poi)  {  cout << "Ve funkci: " << endl;  poi->metoda();  (\*poi).constmetoda();  */\**  *Nezapomeňte, že počet referencí je nyní o 1 více.*  *Nová reference je "poi".*  *\*/*  cout << "Počet referencí " << poi.getReferenceCount() << endl;  } | |

Možná někoho od používání Pointer odradí fakt, že má k dispozici již mnoho funkcí, či metod, které jako své parametry vyžadují normální ukazatele. I tento problém lze řešit. Zde ale již musíme dávat trochu pozor. Začínáme totiž kombinovat přístup k objektu pomocí normálních a našich "inteligentních" ukazatelů. Předání skutečného ukazatele na objekt jako parametru funkce či metody by to nevadilo pokud:

* Ve funkci či metodě nedojde k dealokaci objektu, na který se odkazuje ukazatel předán jako parametr. Důvod je jasný. Funkce, nebo metoda by zničila objekt, se kterým by "inteligentní" ukazatel dále pracoval.
* Funkce, nebo metoda neudělá s ukazatelem nějaký "vedlejší efekt". Například nezapíše ukazatel na náš objekt jako globální proměnnou, nebo jako atribut objektu, jedná-li se o metodu. Vznikl by tím odkaz na něco, co může být kdykoliv dealokováno.

Představme si například, že máme nějakou funkci void f(Trida \*t);. Je-li ptr typu Pointer<Trida>, lze funkci volat f(&\*ptr);. Tím získáme skutečný (normální) ukazatel na objekt třídy Třída.

Funkce a metody nezávislé na typu ukazatele

Existuje také možnost vytvořit funkce či metody nezávislé na tom, zda se jako parametr použije normální, nebo "inteligentní" ukazatel. jak asi každého napadne, jedná se o šablony, jejichž parametrem je typ ukazatele. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Pointer> **void** sablona(**const** Pointer point)  {  point->constmetoda();  (\*point).constmetoda();  } | |

Použití:

|  |  |
| --- | --- |
| |  | | --- | | www\_builder\_cz::Pointer<Trida> ptr(**new** Trida);  Trida \*tr = **new** Trida;  sablona(ptr); *// Lze i náš ukazatel*  sablona(tr); *// Lze i normální ukazatel*  **delete** tr; *// tr musíme sami zničit :-)* | |

Přetypování

Problémy s přetypováním jsem nastínil ve svém předchozím článku. Dnes si ukážeme jak metodu cast použít. Znovu jen zopakuji, že cast je naprosto nepoužitelná pro přetypování "inteligentního" ukazatele na primitivní datový typ. Také není vhodná na přetypování instancí třídy vzniklých vícenásobnou dědičností. Metoda cast je vnořená šablona. Typ této šablony je novým typem, na který chceme přetypovat. Ukážeme si vše na příkladu.

|  |  |
| --- | --- |
| |  | | --- | | *// "Trida" je deklarována výše*  *// Také budeme používat "funkce" a "sablona",*  *// které jsou již deklarovány*  **class** PodTrida : **public** Trida  {  **public**:  **virtual** **void** constmetoda() **const**  { cout << "konstantni metoda podtridy" << endl; }  };  **void** neco()  {  www\_builder\_cz::Pointer<PodTrida> pod(**new** PodTrida);  sablona(pod); *// Zde se přetypovat nemusí*  */\**  *Nelze fce(pod) , protože Pointer<PodTrida>*  *není potomkem Pointer<Trida>.*  *\*/*  fce(pod.cast<Trida>());  www\_builder\_cz::Pointer<Trida> ptr = pod.cast<Trida>();  ptr->constmetoda(); *// Polymorfismus funguje.*  } | |

Pointer NENÍ garbage collector!

Šablona pointer není žádný garbage collector, jaký známe z jiných programovacích jazyků (např. Java). Není to dokonce ani náhrada za garbage collector. Když jsem se v upoutávce na článek o garbage collectoru zmínil, jednalo se spíše o nadsázku. Jak jsem již v minulém článku upozorňoval čítač referencí je bezmocný na kruhové vazby mezi objekty. Máme-li například objekty A,B. V A je ukazatel ukazující na B, v B je ukazatel ukazující na A. Nebude-li existovat žádný jiný ukazatel na A, nebo B, objekty by měly být oba zničeny. Nic takového se ale nestane, protože každý z objektů bude mít čítač referencí nastaven na 1. Takové "kruhy" musíme sami "přetrhnout". Například metodou setNULL, kterou zavoláme pro jeden z ukazatelů tvořících kruh. Tento problém by pro skutečné GC například z Javy nebyl žádným problémem. Skutečné GC totiž pracují trochu jinak.

Shrnutí

Šablonu Pointer rozhodně doporučuji k prostudování, ale při praktickém použití v programech má nějaké ty nevýhody. Pokud o nich ale víme, a nepřekvapí nás, myslím že je možné "inteligentní" ukazatele používat. Zvlášť za velkou nevýhodu považuji problémy s přetypováním. Ty z Vás, kteří nečtou mé články pravidelně, asi překvapilo, co to vlastně ten identifikátor Pointer je. Jedná se o šablonu, která je ke stažení [zde](http://www./). Šablonou Pointer jsem se pokusil vylepšit standardní čablonu [auto\_ptr](http://www.builder.cz/rubriky/c/c--/clanek-156054-automaticke-ukazatele-v-c-/).

Tímto téma čítače odkazů ukončíme. Příště se podíváme na problém, který s počítáním odkazů jen trochu souvisí. Podíváme se na potlačení kopírování velkých objektů. Představme si, že předáváme jako parametr funkce či metody velký objekt. Tento parametr je nutné zkopírovat, což je u velkých objektů neefektivní. Z nějakých důvodů ale nechceme předávat pouze referenci, nebo ukazatel. Jak to řešit? Uvidíme příště.

[Kopírování velkých objektů v C++](http://www.builder.cz/rubriky/c/c--/kopirovani-velkych-objektu-v-c--156115cz)

**27. května 2002, 00.00 | Po menší odmlce je tady opět další díl seriálu objektově orientované programování. Dnes se podíváme na problematiku kopírování velkých objektů. Kopírování lze někdy potlačit, aniž bychom použili referenci, nebo ukazatel.**

Dnes si ukážeme jak mít plně pod kontrolou kopírovaní instancí. Představme si situaci, kdy máme veliký objekt. Objekt je veliký například proto, že obsahuje veliké pole. Můžeme se dostat do situace, kdy je nutné vytvořit jeho kopii. Například jej musíme předat funkci, nebo metodě jako parametr. Kopírovat objekt je neefektivní. Ale předávat funkci, či metodě pouze ukazatel, nebo referenci není vždy možné. V těle funkce nebo metody můžeme chtít provádět s objektem nekonstantní operace, tedy operace které změní vnitřní stav objektu. Kdyby byla funkci nebo metodě předána jako parametr reference (nebo ukazatel), došlo by ke změně stavu i u originálu. Což může být nežádoucí.  
  
Ideální by bylo, kdyby se objekt kopíroval pouze v případě, kdy je to potřeba, ne hned při předávání objektu jako parametru. A úplně nejideálnější by bylo, kdyby se objekt kopíroval jen v případě, kdy je to potřeba, a jen ta část objektu, kterou je potřeba zkopírovat. Jak na to?

Pro přesnost musím jen podotknout, že ke kopii vlastně dojde vždy. Řeší se zde, jestli má být kopie hluboká, nebo plytká. Pojmy hluboká a plytká kopie objektu jsme si vysvětlili ve článku [Kopírovací konstruktor v C++](http://www.builder.cz/art/cpp). Je-li objekt velký (Například obsahuje velké pole), vytváření hluboké kopie je velmi náročné na čas (a také na paměť). Naopak plytká kopie je někdy nepoužitelná. Vytvoříme tedy třídu, jejíž objekty budou vždy kopírovány jako plytká kopie, ale v případě potřeby dojde dodatečně k vytvoření hluboké kopie. Budeme mít vlastně vytvořený objekt, který se bude kopírovat vždy jako plytká kopie. Až v případě potřeby se automaticky dodatečně vytvoří hluboká kopie.

Obvykle se volí postup, kdy se objekt vlastně rozdělí na dvě části. Na část "obsahovou" a "přístupovou". Je tedy nutné vytvořit dvě třídy. Třída "obsahových" objektů má jako své atributy data, jejichž kopírování chceme mít pod kontrolou. Instance této třídy obsahují "velká" data. Programátor by neměl mít na tyto objekty žádné reference, nebo ukazatele. Pracovat s nimi by měl pouze pomocí "přístupových" objektů. Obsahový objekt v sobě musí mít zapouzdřen čítač referencí. Musí vědět, kolik "přístupových" objektů se na něj odkazuje. "Obsahový" objekt se bude kopírovat pouze v nejnutnějším případě. Naproti tomu třída "přístupových" objektů bude jako svůj atribut nutně obsahovat ukazatel na jeden "obsahový" objekt. Přístupový objekt bude možné libovolně kopírovat, protože je malý. Programátor bude pracovat s "přístupovým" objektem.   
Mějme úplně obyčejnou třídu, jejíž instance budou pravděpodobně zabírat velkou část paměti. Chceme ji předělat tak, aby jsme ušetřili zbytečné kopírování takové instance. Obecně lze doporučit postup:

* 1) Vyjmeme z naší třídy všechny atributy, které dělají její instanci velikou. Dáme je do jiné třídy, kterou můžeme například nazvat stejně, jenom s dvěmi podtržítky na začátku. Dejme tomu, že jsme měli třídu Třída, nyní máme třídy Třída a \_\_Třída. Třída je třída přístupových objektů, \_\_Třída je třída obsahových objektů. O existenci třídy "\_\_Třída" a o jejích instancích nemusí programátor používající přístupovou třídu vůbec vědět.
* 2) Třídě \_\_Třída přidáme soukromý atribut udávající počet existujících referencí na instanci. Nazvěme si jej například ReferenceCount. Bude typu unsigned int. Dále přidáme veřejné metody, které zvýší, sníží, vrátí počet referencí. Pojmenujme si je například incrementReferenceCount, decrementReferenceCount, getReferenceCount.
* 3) Třídě \_\_Třída vytvoříme kopírovací konstruktor a operátor = tak, aby vytvářeli hlubokou kopii. Dále by měl být k dispozici pochopitelně destruktor, který uvolní paměť a také nějaké jiné konstruktory. Ve všech konstruktorech nastavíme výchozí počet referencí na 1.
* 4) Třídě Třída přidáme ukazatel na instanci typu \_\_Třída. Nazvěme jej napříkladObjekt. Tedy Třída má atribut \_\_Třída \*Objekt. Měl by být soukromý.
* 5) Třídě Třída přidáme metodu, která odregistruje objekt. Nejprve sníží počet referencí na objekt, na který se odkazuje ukazatel Objekt. Provede to pomocí metody decrementReferenceCount. Je-li po zavolání metody decrementReferenceCount počet odkazů na objekt 0, potom jej zničí destruktorem. Metodu nazveme například free. Neměla by být veřejná. Měla by být soukromá, nebo chráněná.
* 6) Třídě Třída přidáme metodu, která dodatečně provede kopírování instance Objekt do hloubky. Metodu můžeme nazvat například copy. Je-li počet referencí 1, kopie není potřeba a metoda se ukončí. V opačném případě zavoláme metodu free (uvolnění starého objektu) a poté vytvoříme hlubokou kopii objektu Objekt například pomocí kopírovacího konstruktoru třídy \_\_Třída.
* 5) Třídě Třída vytvoříme kopírovací konstruktor a operátor =, které vytvoří jen plytkou kopii instance Objekt. Navíc zavolají objektu Objekt metodu incrementReferenceCount.
* 6) Třídě Třída vytvoříme destruktor, ve kterém zavoláme metodu free.
* 7) Všechny metody (kromě konstruktorů, destruktoru, operátoru =, metod copy a free) ze třídy Třída "přesuneme" do třídy \_\_Třída.
* 8) Pro všechny metody, které jsme v bodě 7 přesunuli vytvoříme ve třídě Třída metody, které "přesměrují" volání na objekt Objekt. Jedná-li se navíc o metodu, která mění vnitřní stav objektu, zavoláme na jejím začátku metodu copy.  
  Ukázka "přesměrování" metody, která nemění vnitřní stav objektu:

|  |  |
| --- | --- |
| |  | | --- | | návratová\_hodnota Trida::metoda(parametry) {  return Objekt->metoda(parametry); } | |

* Ukázka "přesměrování" metody, která mění vnitřní stav objektu:

|  |  |
| --- | --- |
| |  | | --- | | návratová\_hodnota Trida::metoda(parametry) {  copy();  return Objekt->metoda(parametry); } | |

Uveďme si velice jednoduchý příklad. Vytvoříme velice jednoduchý příklad třídy, která bude zapouzdřovat velmi rozsáhlé pole.

|  |  |
| --- | --- |
| |  | | --- | | #include<algorithm>  #include<numeric>  **class** Trida  {  **private**:  **int** \*Pole;  **public**:  Trida() : Pole(**new** **int**[10000]) {}  Trida(**const** Trida &original);  ~Trida() { **delete**[] Pole; }  Trida &**operator**=(**const** Trida &original);  **void** nastavPrvek(**int** index,**int** hodnota){Pole[index]=hodnota;}  **int** dejPrvek(**int** index) { **return** Pole[index]; }  **int** secti() { **return** std::accumulate(Pole,&Pole[10000],0); }  };  Trida::Trida(**const** Trida &original):Pole(**new** **int**[10000])  {  std::copy((**int** \*)original.Pole,&original.Pole[10000],Pole);  }  Trida &Trida::**operator**=(**const** Trida &original)  {  std::copy((**int** \*)original.Pole,&original.Pole[10000],Pole);  **return** \***this**;  } | |

Nyní si představme hodně zvláštní funkci, která jako svůj parametr bude mít objekt naší třídy a příznak. Je-li příznak nulový, funkce vrátí součet prvků v poli objektu naší třídy. Je-li příznak nenulový, přičte k prvním deseti prvkům příznak a vrátí součet prvků v poli.

|  |  |
| --- | --- |
| |  | | --- | | **int** fce(Trida objekt, **int** priznak)  {  **if** (priznak != 0)  {  **for**(**int** p = 0; p < 10; p++)  {  objekt.nastavPrvek(p,objekt.dejPrvek(p) + priznak);  }  }  **return** objekt.secti();  } | |

Je zřejmé, že v této funkci nemůže být parametr objekt třídy Třída předáván referencí, nebo ukazatelem. Došlo by ke změně prvních deseti prvků i u originálního objektu. Na druhou stranu je zbytečné vytvářet na zásobníku kopii objektu v případě, že parametr příznak bude 0. Řešením je potlačit kopírování objektu. Dodržme postup, který jsem uvedl v osmi bodech a vytvořme nové dvě třídy. Výsledek bude vypadat takto:

|  |  |
| --- | --- |
| |  | | --- | | #include<algorithm>  #include<numeric>  **class** \_\_Trida  {  **private**:  **int** \*Pole;  **unsigned** **int** ReferenceCount;  **public**:  \_\_Trida() : Pole(**new** **int**[10000]),ReferenceCount(1) {}  \_\_Trida(**const** \_\_Trida &original):Pole(**new** **int**[10000]),ReferenceCount(1)  {  std::copy((**int** \*)original.Pole,&original.Pole[10000],Pole);  }  ~\_\_Trida() { **delete**[] Pole; }  \_\_Trida &**operator**=(**const** \_\_Trida &original)  {  std::copy((**int** \*)original.Pole,&original.Pole[10000],Pole);  **return** \***this**;  }  **inline** **void** incrementReferenceCount() { ReferenceCount++; }  **inline** **void** decrementReferenceCount() { ReferenceCount--; }  **inline** **int** getReferenceCount() **const** { **return** ReferenceCount; }  **void** nastavPrvek(**int** index, **int** hodnota) { Pole[index] = hodnota; }  **int** dejPrvek(**int** index) { **return** Pole[index]; }  **int** secti() { **return** std::accumulate(Pole,&Pole[10000],0); }  };  **class** Trida  {  **private**:  \_\_Trida \*Objekt;  **void** free();  **void** copy();  **public**:  Trida() : Objekt(**new** \_\_Trida) {}  Trida(**const** Trida &original):Objekt(original.Objekt)  {  Objekt->incrementReferenceCount();  }  ~Trida() { free(); }  Trida &**operator**=(**const** Trida &original)  {  free();  Objekt = original.Objekt;  Objekt->incrementReferenceCount();  **return** \***this**;  }  **void** nastavPrvek(**int** index, **int** hodnota)  {  copy();  Objekt->nastavPrvek(index,hodnota);  }  **int** dejPrvek(**int** index) { **return** Objekt->dejPrvek(index); }  **int** secti() { **return** Objekt->secti(); }  };  **void** Trida::free()  {  Objekt->decrementReferenceCount();  **if** (Objekt->getReferenceCount() == 0)  {  **delete** Objekt;  }  }  **void** Trida::copy()  {  **if** (Objekt->getReferenceCount() == 1)  { *// Kopie není potřeba*  **return**;  }  free();  \_\_Trida \*temp = **new** \_\_Trida(\*Objekt);  Objekt = temp;  } | |

Nyní je zřejmé, že k hluboké kopii, tedy ke kopírování velkého pole dojde pouze v těle metody copy. Ve zmiňované funkci fce nedojde ke kopírování, jestliže je parametr příznak roven 0.

V druhém příkladě bude programátor s třídou Třída zacházet normálně jako-by zacházel se třídou Třída v prvním příkladě.

V mnoha knihovnách se tato "technika řízeného kopírování" používá a programátor používající danou knihovnu o tom možná ani neví. Jen někde v dokumentaci může být třeba napsáno, že objekty nějaké třídy používají řízené kopírování, kopírování až při potřebě nebo něco v tom smyslu. Mezi nevýhody patří zejména fakt, že se musí už při tvorbě třídy brát v úvahu fakt, že kopírování bude řízené. Je třeba vytvářet pomocné třídy a tím se zvyšuje množství zdrojového textu. Nelze vytvořit nějaký obecný program, který by dokázal předloženou třídu "přetransformovat" podle zmiňovaných osmi bodů na požadovaný výsledek. Stejně tak není možné vytvořit obecnou šablonu, které by jsme předložili "normální" třídu jako parametr a ona by nám vytvořila typ, který by měl řízené kopírování. Prostě vše musí udělat programátor ručně.

Velmi často se ale vyskytují ve třídách atributy, které jsou jednorozměrná pole (i v ukázkovém příkladu v tomto článku). Taková pole mohou být velice objemná. Tím se kopírování objektů ,obsahujících taková pole, stává velmi náročné. Zde by se dala vytvořit šablona, jejíž parametr by byl typ prvku v poli. Pole by se kopírovalo pouze v případě, že by se do pole zapisovalo. Pro čtení z pole není nutné provádět kopii. Takovou šablonu vytvořím a dám k dispozici ve svém příštím článku. Pomocí šablony, kterou předložím lze například pole rozdělit na několik částí. V případě zápisu do pole se provede kopie pouze té části, které se zápis týká. V příkladu, kde si ukážeme použití šablony vytvoříme matici, která bude při zápisu kopírovat pouze řádky, do kterých bude vepisováno.

Všem, které tento článek zaujal doporučuji přečíst si článek následující. Bude bezprostředně navazovat na tento článek.

[Řízené kopírování prvků v poli v C++](http://www.builder.cz/rubriky/c/c--/rizene-kopirovani-prvku-v-poli-v-c--156130cz)

cpp oop

**13. června 2002, 00.00 | Dnes dokončíme téma řízené kopírování objektů v C++. Podíváme se na řízené kopírování prvků pole. Vytvoříme si šablonu pole, které bude kopírovat své prvky jen v případě, že to bude potřeba. Šablona je v článku k dispozici ke stažení.**

V [minulém článku](http://www.builder.cz/rubriky/c/c--/clanek-156115-kopirovani-velkych-objektu-v-c-/)jsme se seznámili se způsobem řízení kopírování velkých objektů. Uvedli jsme si jednu velkou nevýhodu celé techniky. Obecně ji nelze nijak parametrizovat. Velice často ale nastává případ, že objekt je veliký, protože jeho atributem je veliké pole. Taková byla i situace v ukázkovém příkladě v [minulém článku](http://www.builder.cz/rubriky/c/c--/clanek-156115-kopirovani-velkych-objektu-v-c-/). Takový speciální případ parametrizovat lze. Nejprve ale všem, kteří můj minulý článek ["Kopírování velkých objektů v C++"](http://www.builder.cz/rubriky/c/c--/clanek-156115-kopirovani-velkych-objektu-v-c-/) nečetli, doporučuji jej přečíst. V tomto článku budu na předchozí článek navazovat.

Představme si velké pole. Podle návodu uvedeného v minulém článku jej rozdělíme na dvě části. Na část "obsahovou" a "přístupovou". Přístupovou část můžeme nazývat také "prezentační". Při kopírování pole vytvoříme pouze plytkou kopii. Pokud budeme z plytké kopie pouze číst, není důvod vytvářet hlubokou kopii. Hluboká kopie se vytvoří automaticky až při zápisu do prvku pole. Pro jednoduchost budeme při zápisu do jednoho prvku pole v případě potřeby vždy kopírovat celé pole, nejen daný prvek. O výhodách a nevýhodách si povíme v závěru.

Mohli by jsme si takhle vytvořit pole prvků nějakého typu. Poté jiné pole prvků jiného typu. Je zřejmé, že všechny třídy si budou podobné. Budou se lišit pouze typem uložených prvků. Jak asi mnohé napadne, vhodným řešením je šablona. Šablona se napíše jednou a podle potřeby se budou vytvářet její instance - třídy.

Nejprve vytvoříme šablonu obsahových tříd. Nazveme ji \_\_Atribut. Šablona přístupových tříd se bude jmenovat Atribut. To proto, že instance šablony bude typ, jehož objekty budou atributy jiných tříd. Například v minulém článku jsme měli třídu s atributem pole typu int.Atribut možná není nejvhodnější název, ale mne žádný jiný nenapadl. Šablona \_\_Atribut bude podle rad z minulého článku obsahovat čítač referencí, samotná data (pole) a délku pole. Dále metody, které manipulují s čítačem referencí, metody vracející ukazatel na první prvek v poli, konstruktory a destruktor.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> **class** \_\_Atribut  {  **private**:  *// Počet referencí a délka pole*  **unsigned** **int** ReferenceCount, Lenght;  *// Zapouzdřená data. Data jsou "velká".*  Type \*Data;  **public**:  *// Konstruktory*  \_\_Atribut():ReferenceCount(1),Lenght(0), Data(NULL) {}  \_\_Atribut(**unsigned** **int** n):ReferenceCount(1),Lenght(n),  Data(**new** Type[n]) {}  \_\_Atribut(**const** \_\_Atribut<Type> &original)  :ReferenceCount(1),Lenght(original.Lenght)  {  Data = **new** Type[Lenght];  std::copy(Data,&Data[Lenght],original.Data);  original.incrementReferenceCount();  }  *// Destruktor*  ~\_\_Atribut()  {  **delete**[] Data;  }  *// Operátor =*  \_\_Atribut<Type> &**operator**=(**const** \_\_Atribut<Type> &original)  {  **if** (Data != NULL)  {  **delete**[] Data;  }  Lenght = original.getLenght();  Data = **new** Type[Lenght];  std::copy(original.Data,&original.Data[Lenght],Data);  **return** \***this**;  }  *// Metoda getReferenceCount vrátí počet referencí*  **inline** **unsigned** **int** getReferenceCount() **const** {  **return** ReferenceCount;  }  *// Metoda decrementReferenceCount() sníží počet referencí*  *a vrátí nový počet referencí*  **inline** **unsigned** **int** decrementReferenceCount() {  **return** --ReferenceCount;  }  *// Metoda incrementReferenceCount() zvýší počet referencí*  *a vrátí nový počet referencí*  **inline** **unsigned** **int** incrementReferenceCount() {  **return** ++ReferenceCount;  }  *// Metoda getLenght() vrátí délku pole*  **inline** **unsigned** **int** getLenght() **const** { **return** Lenght; }  *// Metoda getData vrátí ukazatel na data*  **inline** **const** Type \*getData() **const** { **return** Data; }  **inline** Type \*getData() { **return** Data; }  }; | |

Parametrem šablony je typ prvků, které budou v poli uloženy.

Přistupme k implementaci šablony přístupových tříd.

|  |  |
| --- | --- |
| |  | | --- | | **template**<**class** Type> **class** Atribut  {  **private**:  *// Object je skutečné pole s čítačem referencí.*  \_\_Atribut<Type> \*Object;  *// Soukromé metody pro řízení kopírování*  **void** copy();  **void** free();  **public**:  *// Konstruktory*  Atribut():Object(**new** \_\_Atribut<Type>){}  Atribut(**unsigned** **int** n):Object(**new** \_\_Atribut<Type>(n)){}  Atribut(**const** Atribut<Type> &original);  *// Destruktor*  ~Atribut();  *// Operátor =*  Atribut<Type> &**operator**=(**const** Atribut<Type> &original);  *//Operátory [] a metoda at pro přístup k prvkům*  **inline** **const** Type &**operator**[](**unsigned** **int** index) **const**;  Type &**operator**[](**unsigned** **int** index);  Type at(**unsigned** **int** index) **const** ;  *// Relační operátory*  **bool** **operator**==(**const** Atribut<Type> &atribut) **const** ;  **bool** **operator**!=(**const** Atribut<Type> &atribut) **const** ;  *// Metoda vracející počet ukazatelů na objekt, na který*  *// se odkazuje tento ukazatel. Vždy alespoň 1*  **inline** **unsigned** **int** getReferenceCount() **const** ;  *// Metoda getPointer slouží k získání ukazatele na*  *// první prvek pole.*  **inline** **const** Type \*getPointer() **const** { **return** Object->getData();  **inline** Type \*getPointer() { copy(); **return** Object->getData();  *// Vytvoří nové pole. Slouží k změně rozměru pole.*  *// Staré prvky v poli smaže. Nevytváří kopii!*  **void** newArray(**unsigned** **int** size);  }; | |

Šablona třídy je v principu stejná jako přístupová třída z minulého článku. Obdobným způsobem zvyšuje a snižuje počet referencí. Soukromá metoda copy je zavolána vždy, je-li potřeba vytvořit kopii. Tedy jestliže je prováděna operace, která by změnila hodnotu nějakého prvku v poli. Naopak metoda free je zavolána v případě, že přístupový objekt "zahazuje" referenci na "obsahový" objekt.

Metoda at slouží k získání prvku v poli. Metoda at vrací samotný prvek, nikoliv referenci na něj. Proto by měla být použita jen na levé straně výrazu. Oproti operátoru [] neprovádí hlubokou kopii. Při použití operátoru [] (vrací referenci) je nutné provést hlubokou kopii. Nevíme totiž, jestli bude použit na pravé, či levé straně výrazu. Pod pojmy "na pravé straně výrazu" a "na levé straně výrazu" mám na mysli to, jestli je ve výrazu něco napravo nebo nalevo od operátoru =. Například a = b; je a na levé straně výrazu a b na pravé straně výrazu.

Kompletní implementace šablony třídy Atribut a \_\_Atribut jsou v souboru [atribut.h](http://www.builder.cz/data/atribut.h). Šablona je opět deklarovaná v prostoru jmen www\_builder\_cz.

Nyní si vytvoříme jednoduchý příklad použití šablony Atribut. Bude se jednat o matici, která bude mít řízené kopírování svých prvků. Bude kopírován pouze řádek, ve kterém došlo ke změně. Matice může mít mnoho prvků. Kopírovat se ale bude jen několik řádků, podle toho jaké prvky budeme měnit. Při velmi velkých rozměrech matice to bude výhodné. Ještě než se podíváme na zdrojový text třídy Matice, musíme si ujasnit, že šablona Atribut (resp. \_\_Atribut) má jako svůj parametr typ , který má k dispozici bezparametrický konstruktor, kopírovací konstruktor a operátor =.

Kromě matice vytvoříme funkce vypis a fce. Funkce vypis vypíše prvky matice. Je dobré si všimnout, že matice předávaná této funkci jako parametr nebude vlastně nikdy kopírována. Při předávání parametru dojde k vytvoření pouze mělké kopie. Druhá funkce fce je ukázková a asi nemá žádné využití. Vynuluje zadaný řádek a ve vzniklé matici sečte zadaný sloupec.

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream>  #include "atribut.h"  using namespace std;  **class** Matice  {  **private**:  www\_builder\_cz::Atribut<www\_builder\_cz::Atribut<**int**> > Data;  **unsigned** **int** Radky, Sloupce;  **public**:  Matice(**unsigned** **int** r, **unsigned** **int** s):Data(r),Radky(r),Sloupce(s)  {  **for**(**unsigned** **int** p = 0; p < r; p++)  {  Data[p].newArray(s);  }  }  Matice(**const** Matice &druha):Data(druha.Data),Radky(druha.Radky),  Sloupce(druha.Sloupce){}  Matice &**operator**=(**const** Matice &druha)  {  Data = druha.Data;  Sloupce = druha.Sloupce;  Radky = druha.Radky;  **return** \***this**;  }  **inline** **unsigned** **int** dejSloupec() **const** { **return** Sloupce; }  **inline** **unsigned** **int** dejRadek() **const** { **return** Radky; }  **inline** **void** nastavPrvek(**unsigned** **int** r, **unsigned** **int** s, **int** prvek)  { Data[r][s] = prvek; }  **inline** **const** **int** dejPrvek(**unsigned** **int** r, **unsigned** **int** s)  **const** { **return** Data.at(r).at(s); }  **int** sectiSloupec(**unsigned** **int** index);  };  **int** Matice::sectiSloupec(**unsigned** **int** index)  {  **int** navrat = 0;  **for**(**unsigned** **int** p = 0; p < Radky; p++)  {  navrat += Data.at(p).at(index);  }  **return** navrat;  }    **int** fce(Matice matice, **int** radek, **int** sloupec)  {  **for**(**unsigned** **int** p = 0; p < matice.dejSloupec(); p++)  {  matice.nastavPrvek(radek,p,0);  }  **return** matice.sectiSloupec(sloupec);  }  **void** vypis(Matice m)  {  **for**(**unsigned** **int** r = 0; r < m.dejRadek(); r++)  {  **for**(**unsigned** **int** s = 0; s < m.dejSloupec(); s++)  {  cout << m.dejPrvek(r,s) << "\t";  }  cout << endl;  }  cout << endl;  }  **int** main()  {  Matice m1(5,8),m2(m1);  m1.nastavPrvek(1,1,1) | |

[Dokončení seriálu objektově orientované programování v C++](http://www.builder.cz/rubriky/c/c--/dokonceni-serialu-objektove-orientovane-programovani-v-c--156137cz)

**4. července 2002, 00.00 | Dnes ukončíme seriál objektově orientované programování v C++. Podíváme se na klíčové slova mutable a static, na které jsem v úvodu seriálu poněkud pozapomněl.**

Dnešním článkem ukončím seriál [Objektově orientované programování v C++](http://www.builder.cz/serial24.html). Nejprve se ale musím přiznat, že jsem v úvodu seriálu nějak pozapomněl na dvě dost podstatné klíčové slova. Jedná se o static a mutable. Jedná se vlastně o základy jazyka C++. Správně jsem se o nich měl zmínit tak maximálně v pátém článku seriálu, nikoliv v posledním díle.

Klíčové slovo mutable

Jak jsem již dříve napsal, v C++ lze metodu označit klíčovým slovem const. Tím říkáme překladači, že metoda nijak nemění vnitřní stav objektu. Lze ji tedy zavolat na konstantní objekt. Znamená to, že v těle takové metody nesmí dojít ke změně atributů objektu. Změníme-li atributy objektu v těle "konstantní" metody, překladač ohlásí syntaktickou chybu. Přece ale existují atributy, které lze měnit konstantní metodou. Takový atribut musí být označen klíčovým slovem mutable. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  **using** **namespace** std;  **class** Trida  {  **private**:  **mutable** **int** MAtribut;  **int** Atribut;  **public**:  Trida() : MAtribut(0), Atribut(0) {}  **void** zvetsi() { MAtribut++; Atribut++; }  **int** vrat() **const**;  **int** vratMutableAtribut() **const** { **return** ++MAtribut; }  };  **int** Trida::vrat() **const**  {  MAtribut++;  *// Odkomentování následujícího řádku je syntaktická chyba*  *// Atribut++;*  **return** Atribut;  }  **int** main(**int**, **char**\*\*)  {  Trida objekt1;  objekt1.zvetsi();  objekt1.zvetsi();  cout << objekt1.vrat() << endl;  cout << objekt1.vratMutableAtribut() << endl;  **const** Trida objekt2;  cout << objekt2.vrat() << endl;  cout << objekt2.vratMutableAtribut() << endl;  **return** 0;  } | |

Není dobré klíčové slovo mutable psát neuváženě jen proto aby šlo obejít omezení konstantních objektů. Slovo mutable slouží hlavně k označení atributů, které přímo nesouvisejí s vnitřním stavem objektu. Lze je tedy měnit i v případě, že objekt je konstantní.

Klíčové slovo static

Z jazyka C "zdědil" jazyk C++ klíčové slovo static, které lze použít u globálních nebo lokálních proměnných. Je-li například static int promenna; globální proměnná, je viditelná pouze v daném souboru zdrojového textu (modulu). Z jiného modulu k ní nelze přistoupit. Obdobně může být static použito i u funkcí. Funkce je potom viditelná pouze v daném modulu. Je-li static int promenna; lokální proměnná funkce, potom její hodnota je uchovávána mezi jednotlivým voláním. Proměnná vzniká prvním zavoláním funkce a zaniká při ukončení programu. Tolik (stručně) k použití static v jazyce C. V jazyce C++ máme ale navíc dvě další možnosti jak použít klíčové slovo static. Můžeme ho použít u členské proměnné ve třídě, nebo u metody.

Klíčové slovo static a členská proměnná

Je-li klíčovým slovem static označena členská proměnná, jedná se o atribut, který všechny instance dané třídy sdílejí. Atribut existuje (lze s ním pracovat) i v případě, že neexistuje žádná instance třídy. Často se používá k počítání instancí dané třídy. Lze k ní přistupovat pomocí identifikátoru třídy, symbolu :: a názvu proměnné. Vše si ukážeme níže v příkladu.

Klíčové slovo static a členská metoda

Je-li klíčovým slovem static označena členská metoda, jedná se o metodu, která nemá jako svůj první implicitní parametr this. Není tedy těsně spojená s nějakou konkrétní instancí. V těle metody nelze pracovat s "nestatickými" atributy třídy a nelze volat "nestatické" metody třídy. Je to logické, protože není jasné pro který objekt je volána (není parametr this). Lze ji také zavolat, aniž by existovala nějaká instance dané třídy. Příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  **using** **namespace** std;  **class** Trida  {  **private**:  **static** **int** PocetInstanci = 0;  **int** Atribut;  **public**:  Trida() : Atribut(0) { PocetInstanci++; }  ~Trida() { PocetInstanci--; }  **static** **int** dejPocetInstanci() { **return** PocetInstanci; }  **int** operaceSNestatickymAtributem(**int** a)  { **int** temp = Atribut; Atribut = a; **return** temp; }  };  **int** main(**int**, **char**\*\*)  {  Trida objekt1, objekt2, \*objekt3 = **new** Trida();  cout << Trida::dejPocetInstanci() << endl;  cout << objekt2.operaceSNestatickymAtributem(10) << endl;  cout << objekt1.operaceSNestatickymAtributem(10) << endl;  cout << objekt3->operaceSNestatickymAtributem(10) << endl;  cout << objekt3->operaceSNestatickymAtributem(5) << endl;  **delete** objekt3;  cout << Trida::dejPocetInstanci() << endl;  **return** 0;  } | |

V "čistě" objektově orientovaných jazycích (mezi které určitě nepatří C++) se třída považuje za objekt. Je to objekt, který má svou identitu, stav a svému okolí poskytuje nějaké služby. Může se například jednat o službu "vytvoření své instance". (Vznikají zde zajímavé problémy. Je-li třída objekt, co je její třídou? Tedy co je třídou třídy? Je to metatřída, což je opět objekt. Co je třídou metatřídy? atd... S podobnými "žertíky" se člověk může setkat například ve Smalltalku.) V této souvislosti lze atributy dělit na instanční atributy (atributy objektu třídy - každý objekt má své) a třídní atributy (atributy, které má třída). Díky klíčovému slovu staticmůžeme považovat atributy a metody označené jako static za třídní atributy a metody, naopak atributy a metody neoznačené jako static můžeme považovat za instanční atributy a metody. Není to ale přesná analogie, protože v C++ NENÍ třída objektem. Pro představu, jak jsem to myslel, uvádím poslední příklad:

|  |  |
| --- | --- |
| |  | | --- | | #include<iostream>  **using** **namespace** std;  **class** Trida  {  **private**:  **static** **unsigned** **int** PocetInstanci = 0; *// "Třídní" proměnná*  **int** Atribut; *// "InstanÈní" proměnná*  **public**:  Trida() : Atribut(0) {PocetInstanci++;}  ~Trida() { PocetInstanci--; }  **static** Trida \*vytvorInstanci() { **return** **new** Trida(); }  **static** dejPocetInstanci() { **return** PocetInstanci; }  **int** vrat() **const** { **return** Atribut; }  **void** nastav(**unsigned** **int** a) { Atribut = a; }  };  **int** main(**int**, **char**\*\*)  {  cout << "Pocet instanci:" << Trida::dejPocetInstanci() << endl;  Trida objekt1;  cout << "Pocet instanci:" << Trida::dejPocetInstanci() << endl;  objekt1.nastav(3);  cout << objekt1.vrat() << endl;  Trida \*objekt2 = Trida::vytvorInstanci(),  \*objekt3 = Trida::vytvorInstanci();  cout << objekt2->vrat() << objekt3->vrat() << endl;  cout << "Pocet instanci:" << Trida::dejPocetInstanci() << endl;  **delete** objekt2;  **delete** objekt3;  cout << "Pocet instanci:" << Trida::dejPocetInstanci() << endl;  **return** 0;  } | |

Tímto bych seriál ukončil. Snažil jsem se podrobněji popsat jazyk C++ pro ty, kteří znají C. Chtěl jsem ukázat vymoženosti jazyka C++ oproti C. Snad se mi to alespoň trochu povedlo. Mezi vymoženosti C++ oproti C patří především principy OOP, přetěžování operátorů, parametrizace pomocí šablon a STL. Původně jsem ani nepočítal s tím, že seriál bude mít tak mnoho dílů. Už je skoro jako Esmeralda :-) Děkuji všem, kteří seriál dočetli až sem. Také děkuji všem, kteří mě v komentářích pod článkem upozornili na nějaké chyby a měli se mnou trpělivost. Chyb bylo opravdu dost.

Na úplný závěr ještě uvedu zdroje, ze kterých jsem čerpal. Jedná se spíše o typy pro Vás, pokud se o jazyk C++ více zajímáte. O některých jsem věděl již dávno, na některé mě upozornili čtenáři v komentářích pod články.

* Cay S. Horstman: Vyšší škola objektově orientovaného návrhu v C++, Science 1997
* Stanislav Racek, Martin Kvoch: Třídy a objekty v C++, Kopp 1988
* Pavel Herout: Učebnice jazyka C, Kopp 1996
* Chip ročníky 1999 - 2001, články zaměřené na C++. Autoři jsou převážně pánové J. Franěk a M. Virius.
* Dokumentace k STL - <http://www.sgi.com/tech/stl/stl_index_cat.html>
* [www.cplusplus.com](http://www.cplusplus.com/)
* Manuálové stránky OS LINUX
* Elektronická dokumentace a nápověda k překladači Borland C++ Builder®
* Elektronická dokumentace a nápověda k překladači GNU C/C++
* Elektronická dokumentace a nápověda k překladači [Mingw](http://www.mingw.org/)

**Obsah seriálu (**[**více o seriálu**](http://www./)**):**

* [Základy OOP v C++: Od C k C++](http://www.builder.cz/rubriky/c/c--/zaklady-oop-v-c-od-c-k-c--155657cz)
* [Základní pojmy objektově orientovaného programování](http://www.builder.cz/rubriky/c/c--/zakladni-pojmy-objektove-orientovaneho-programovani-155665cz)
* [Vytváření tříd, instance třídy, zasílání zpráv v C++](http://www.builder.cz/rubriky/c/c--/vytvareni-trid-instance-tridy-zasilani-zprav-v-c--155667cz)
* [Vytváření instancí - konstruktory, destruktory](http://www.builder.cz/rubriky/c/c--/vytvareni-instanci-konstruktory-destruktory-155678cz)
* [Kopírovací konstruktor v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-konstruktor-v-c--155685cz)
* [Jednoduchá dědičnost v C++](http://www.builder.cz/rubriky/c/c--/jednoducha-dedicnost-v-c--155692cz)
* [Časná versus pozdní vazba - úvod do polymorfismu v C++](http://www.builder.cz/rubriky/c/c--/casna-versus-pozdni-vazba-uvod-do-polymorfismu-v-c--155697cz)
* [Polymorfismus - dokončení](http://www.builder.cz/rubriky/c/c--/polymorfismus-dokonceni-155701cz)
* [Vícenásobná dědičnost v C++](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c--155707cz)
* [Vícenásobná dědičnost v C++ - opakovaná dědičnost](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-opakovana-dedicnost-155717cz)
* [Vícenásobná dědičnost v C++ - volání konstruktorů a destruktorů](http://www.builder.cz/rubriky/c/c--/vicenasobna-dedicnost-v-c-volani-konstruktoru-a-destruktoru-155723cz)
* [Přetěžování operátorů v C++ 1.díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-1-dil-155728cz)
* [Přetěžování operátorů v C++ 2. díl](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-v-c-2-dil-155729cz)
* [Vstupní a výstupní operace pomocí datových proudů v C++](http://www.builder.cz/rubriky/c/c--/vstupni-a-vystupni-operace-pomoci-datovych-proudu-v-c--155739cz)
* [Přetěžování operátorů << a >> pro datové proudy v C++](http://www.builder.cz/rubriky/c/c--/pretezovani-operatoru-lt-lt-a-gt-gt-pro-datove-proudy-v-c--155741cz)
* [Neformátovaný vstup a výstup v C++](http://www.builder.cz/rubriky/c/c--/neformatovany-vstup-a-vystup-v-c--155754cz)
* [Paměťové proudy v C++](http://www.builder.cz/rubriky/c/c--/pametove-proudy-v-c--155755cz)
* [Prostory jmen v C++](http://www.builder.cz/rubriky/c/c--/prostory-jmen-v-c--155766cz)
* [Řetězce v C++](http://www.builder.cz/rubriky/c/c--/retezce-v-c--155767cz)
* [Výjimky v C++](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c--155798cz)
* [Výjimky v C++ - výjimky tvoří dědičnou hierarchii](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-vyjimky-tvori-dedicnou-hierarchii-155809cz)
* [Výjimky v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/vyjimky-v-c-dokonceni-155810cz)
* [Dynamická identifikace typů v C++](http://www.builder.cz/rubriky/c/c--/dynamicka-identifikace-typu-v-c--155828cz)
* [Přetypování v C++](http://www.builder.cz/rubriky/c/c--/pretypovani-v-c--155842cz)
* [Problémy s typy při vícenásobné dědičnosti](http://www.builder.cz/rubriky/c/c--/problemy-s-typy-pri-vicenasobne-dedicnosti-155865cz)
* [Šablony funkcí v C++](http://www.builder.cz/rubriky/c/c--/sablony-funkci-v-c--155874cz)
* [Šablony datových typů v C++](http://www.builder.cz/rubriky/c/c--/sablony-datovych-typu-v-c--155883cz)
* [Vnitřní typy u parametrů šablon, vnořené šablony v C++](http://www.builder.cz/rubriky/c/c--/vnitrni-typy-u-parametru-sablon-vnorene-sablony-v-c--155889cz)
* [Pole s libovolným intervalem indexování v C++](http://www.builder.cz/rubriky/c/c--/pole-s-libovolnym-intervalem-indexovani-v-c--155896cz)
* [Datové kontejnery v C++ - Úvod do STL](http://www.builder.cz/rubriky/c/c--/datove-kontejnery-v-c-uvod-do-stl-155907cz)
* [Vector - datový kontejner v C++](http://www.builder.cz/rubriky/c/c--/vector-datovy-kontejner-v-c--155916cz)
* [Iterátory v C++](http://www.builder.cz/rubriky/c/c--/iteratory-v-c--155928cz)
* [Šablona vector v C++ a iterátory](http://www.builder.cz/rubriky/c/c--/sablona-vector-v-c-a-iteratory-155932cz)
* [Asociativní pole v C++](http://www.builder.cz/rubriky/c/c--/asociativni-pole-v-c--155938cz)
* [Množina v C++](http://www.builder.cz/rubriky/c/c--/mnozina-v-c--155952cz)
* [Funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/funkcni-objekty-v-c--155955cz)
* [Standardní funkční objekty v C++](http://www.builder.cz/rubriky/c/c--/standardni-funkcni-objekty-v-c--155969cz)
* [Úvod do standardních algoritmů v C++](http://www.builder.cz/rubriky/c/c--/uvod-do-standardnich-algoritmu-v-c--155983cz)
* [Kopírovací a přesouvací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/kopirovaci-a-presouvaci-algoritmy-v-c--156005cz)
* [Vyhledávací algoritmy v C++](http://www.builder.cz/rubriky/c/c--/vyhledavaci-algoritmy-v-c--156011cz)
* [Skenovací (prohlížecí) algoritmy v C++](http://www.builder.cz/rubriky/c/c--/skenovaci-prohlizeci-algoritmy-v-c--156016cz)
* [Transformační algoritmy v C++](http://www.builder.cz/rubriky/c/c--/transformacni-algoritmy-v-c--156022cz)
* [Řadící algoritmy v C++](http://www.builder.cz/rubriky/c/c--/radici-algoritmy-v-c--156023cz)
* [Halda v C++](http://www.builder.cz/rubriky/c/c--/halda-v-c--156036cz)
* [Standardní algoritmy v C++ - dokončení](http://www.builder.cz/rubriky/c/c--/standardni-algoritmy-v-c-dokonceni-156045cz)
* [Automatické ukazatele v C++](http://www.builder.cz/rubriky/c/c--/automaticke-ukazatele-v-c--156054cz)
* [Inteligentní ukazatel - čítač referencí v C++](http://www.builder.cz/rubriky/c/c--/inteligentni-ukazatel-citac-referenci-v-c--156071cz)
* [Použití čítače referencí v C++](http://www.builder.cz/rubriky/c/c--/pouziti-citace-referenci-v-c--156074cz)
* [Kopírování velkých objektů v C++](http://www.builder.cz/rubriky/c/c--/kopirovani-velkych-objektu-v-c--156115cz)
* [Řízené kopírování prvků v poli v C++](http://www.builder.cz/rubriky/c/c--/rizene-kopirovani-prvku-v-poli-v-c--156130cz)
* [**Dokončení seriálu objektově orientované programování v C++**](http://www.builder.cz/rubriky/c/c--/dokonceni-serialu-objektove-orientovane-programovani-v-c--156137cz)