***Builder.cz - http://www.builder.cz***
**Autor: Radim Dostál**
**Rubrika: C/C++**
**Datum vydání: 01.10. 2001**
**Url: http://www.builder.cz/art/cpp/array.html**

## Pole s libovolným intervalem indexování v C++

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, 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 `at` dělá 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 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.