Builder.cz - http://www.builder.cz
Autor: Radim Dostál
Rubrika: C/C++
Datum vydání: 25.01. 2002
Url: http://www.builder.cz/art/cpp/vyhledcpp.html

Vyhledávací algoritmy v C++

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. 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 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++ .

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++) 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.

------ http://www.builder.cz ------