

Algoritmy ve standardní knihovně

1131 Predikáty a funktory



Při práci s algoritmy budeme často používat *predikáty* nebo *funktory*. Obojí je buď objektový typ s přetíženým operátorem volání funkce, nebo ukazatel na funkci. (V této souvislosti občas hovoříme o *volatelných typech*.)

- Predikát označuje funkci, jejímž výsledkem je logická hodnota rozhodující o pořadí prvků, o tom, zda má být odstraněn atd.
- Funktor označuje funkci, jež provede např. nějakou transformaci uložené hodnoty.

Obojí předáme jako jeden z parametrů funkci, která implementuje knihovní algoritmus, a tento predikát nebo funktor se postará o potřebnou akci – porovnání prvků kontejneru, jejich transformaci atd. (Připomeňme si návrhový vzor *strategie*, tip 831.)

Poznamenejme, že z hlediska efektivity může být předání objektu s přetíženým operátorem volání funkce výhodnější než předání ukazatele, neboť přetížený operátor je zpravidla definován jako vložená metoda (inline), zatímco ukazatel na funkci ukazuje vždy na funkci, kterou je třeba volat – se vši režii, která se k tomu váže.

1132 Třídění neboli řazení v jazyce C



Máme pole celých čísel `pole` a chceme ho seřadit podle velikosti. Standardní knihovna jazyka C obsahuje funkci `qsort()`, deklarovanou v hlavičkovém souboru `<stdlib.h>`, která implementuje známý algoritmus rychlého třídění *quicksort*. Jejím prvním parametrem je ukazatel na první prvek řazeného pole, předávaný jako `void*`. Druhým je počet prvků pole a třetím je velikost jednoho prvku pole.

Čtvrtým parametrem je ukazatel na funkci, která obstará porovnání dvou prvků pole. Musí to být funkce se dvěma parametry typu `const void*`, která vrátí zápornou hodnotu, je-li první parametr menší než druhý, nulu, jsou-li si rovny, a kladnou hodnotu, je-li první parametr větší než druhý.

V případě řazení pole celých čísel může porovnávací funkce vypadat takto:

```
int porovnej(const void* prvek1, const void* prvek2)
{
    return *(int*)prvek1 - *(int*)prvek2;
}
```

Pole `pole` pak seřadíme příkazem

```
qsort(pole, sizeof(pole)/sizeof(int), sizeof(int), porovnej);
```



Poznámka: Nahlédneme-li do slovníku, zjistíme, že *třídění* znamená rozřazování do tříd (například krmené brambory – potravinářské brambory), zatímco řazení znamená uspořádání podle hodnoty nějakého znaku, například podle velikosti. I když v uvedeném algoritmu – i v několika následujících – nám jde o *řazení*, hovoří programátoři tradičně o *třídění*. Poznamenejme, že tento omyl není výsadou češtiny; anglické *sorting* znamená třídění, řazení by bylo *ordering*.

1133 Třídění neboli řazení v C++



pokročilý

Standardní šablonová knihovna jazyka C++ obsahuje šablonu funkce `sort<>`, která implementuje algoritmus quicksort. Její použití je jednodušší a je i efektivnější než funkce `sort`, neboť nevolá pomocnou funkci. Tato funkce očekává dvojici iterátorů pro náhodný přístup, z nichž první ukazuje *na první prvek* řazeného úseku pole a druhý ukazuje *na první prvek za* tímto úsekem. Jak víme, na místě iterátorů pro náhodný přístup lze použít i ukazatele. To znamená, že pole deklarované příkazem

```
int pole[N];
```

seřadíme příkazem

```
sort(pole, &pole[N]);
```

Při použití této šablonové funkce se předpokládá, že řazené pole obsahuje prvky jakéhokolli typu, které lze porovnávat běžnými relačními operátory.



Poznámka: Připomeňme si, že výraz `&pole[N]` je korektní, nebudeme-li takto získaný ukazatel dereferencovat. V jazycích C a C++ vždy smíme získat adresu prvního prvku za polem, použijeme-li ji pouze pro účely porovnávání.

1134 Řazení vektoru



pokročilý

Nyní úlohu z předchozího tipu lehce pozměníme: Nechceme řadit prvky pole, ale prvky vektoru. Máme vektor `vek` obsahující celá čísla, tedy typu `vector<int>`, a chceme jeho obsah seřadit podle velikosti. I zde použijeme algoritmus `sort` z STL, předáme mu však dvojici iterátorů určujících řazený úsek. Jako vždy musí první z nich ukazovat *na první prvek* tohoto úseku a druhý *za* poslední prvek tohoto úseku. V našem případě jde o celý vektor, takže napíšeme

```
sort(vek.begin(), vek.end());
```

1135 Řazení typů, pro které není definována relace <



pokročilý

Představte si, že máme vektor `oni` obsahující záznamy o zaměstnancích uložené v proměnných typu `zaměstnanec`,

```
struct zaměstnanec {
    string jmeno, prijmeni;
```

```
int plat;
};
```

Potřebujeme je seřadit podle výše platu. Pro tyto datové typy není definován implicitní operátor <; to však nevadí. Použijeme algoritmus `sort` se třemi parametry, kde třetím parametrem bude funkční objekt – predikát – s přetíženým operátorem volání funkce, který bude vracet výsledek porovnání:

```
class porovnej
{
public:
    bool operator()(zamestnanec ten, zamestnanec onen)
    { // Testuje podmínku, podle které mají být data řazena
      return ten.plat < onen.plat;
    }
};
```

Kontejner pak seřadíme příkazem

```
sort(oni.begin(), oni.end(), porovnej());
```



Poznámka: Zatím se záměrně vyhýbáme abecednímu řazení. K tomu se vrátíme v kapitole Lokální nastavení v C a v C++.

1136 Stabilní řazení



pokročilý

Funkce `sort<>()` i `qsort` použité v předchozích tipech, implementují algoritmus quicksort, který není stabilní, tj. nezachovává pořadí prvků se stejnou hodnotou (říkáme také *prvků se stejným klíčem*). U celých čísel a v mnoha dalších situacích je to jedno; představte si ale, že máme kontejner obsahující záznamy o zaměstnancích, který je seřazený podle jejich křestního jména, a my ho poté seřadíme podle příjmení. Použijeme-li stabilní algoritmus, zachová se pořadí hodnot se stejným klíčem. To znamená, že např. všichni Novákové budou seřazeni podle křestního jména. Přesněji: Zaměstnanci se stejným příjmením budou seřazeni podle křestního jména.

Standardní knihovna jazyka C++ nabízí šablonu funkce `stable_sort<>()`, která implementuje stabilní třídění. Její použití je stejné jako použití funkce `sort<>()` popsané v tipech 1133 a 1134, vyžaduje však pouze obousměrné iterátory.

1137 Částečné řazení



pokročilý

Někdy potřebujeme seřadit pouze úvodní část kontejneru a na pořadí dalších prvků už nezáleží. Pak oceníme algoritmus `partial_sort<>()`, který má v základní podobě tři parametry (všechny tři jsou iterátory s náhodným přístupem): První z nich určuje počáteční prvek řazeného úseku, druhý z nich určuje prvek, u kterého se řazení zastaví, a třetí určuje první prvek za zpracovávaným úsekem.

Máme-li např. pole `A` celých čísel o `N` prvcích a chceme-li seřadit pouze první 4 prvky, napíšeme

```
partial_sort(A, A+4, &A[N]);
```

Jako čtvrtý parametr můžeme opět zadat predikát určující způsob porovnávání (viz též tip 1135).



Poznámka: Funkce `partial_sort<>()` představuje implementaci známého algoritmu třídění haldou (heapsort).

1138 Částečné řazení kopie posloupnosti



pokročilý

Ne vždy chceme posloupnost řadit v původním kontejneru. Někdy chceme původní posloupnost zachovat a její část po seřazení překopírovat do jiného kontejneru.

K tomu nám C++ nabízí šablonovou funkci `partial_sort_copy<>()`, která očekává dvojici vstupních iterátorů určujících po řadě první prvek zpracovávaného úseku výchozího kontejneru a první prvek za zpracovávaným úsekem, a pak dva iterátory pro náhodný přístup, z nichž první určuje místo, kde se má uložit první z překopírovaných a seřazených prvků, a druhý určuje první prvek za posledním uloženým.

Máme např. pole celých čísel `A`

```
int A[9] = {9,8,7,6,5,4,3,2,1};
```

a chceme jeho první tři prvky po seřazení překopírovat do vektoru celých čísel `B`. Vektor ovšem musí mít dostatečnou kapacitu, neboť prvky se do něj nebudou ukládat pomocí metody `push_back()`, ale pomocí iterátorů. Požadovanou kapacitu zadáme při vytvoření jako parametr konstrukturu:

```
vector<int> B(3);
```

Po provedení příkazu

```
partial_sort_copy(A, A+9, B.begin(), B.begin()+3);
```

se obsah pole `A` nezmění a vektor `B` bude obsahovat hodnoty 1, 2, 3.

Poznamenejme, že přetížené verzi této funkce můžeme jako pátý parametr zadat predikát určující způsob porovnávání prvků (viz tip 1135).

1139 Řazení seznamu



pokročilý

Chceme-li seřadit prvky seznamu, můžeme – na rozdíl od jiných kontejnerů – použít jeho vlastní metodu. Máme-li např. seznam celých čísel.

```
list<int> seznam;
```

seřadíme jeho prvky příkazem

```
seznam.sort();
```

Budeme-li chtít předepsat způsob porovnávání, použijeme přetíženou verzi této metody, které zadáme jako parametr binární predikát určující způsob porovnávání.



Poznámka: Na seznam můžeme také použít funkci `stable_sort<>()`, o níž jsme hovořili v tipu 1136. Nelze však použít `sort<>()` ani `partial_sort<>()`, neboť ty vyžadují iterátory pro náhodný přístup, ale seznam poskytuje pouze obousměrné iterátory.

1140 k-tý prvek podle velikosti



Máme posloupnost a zajímá nás hodnota k -tého prvku podle velikosti. Jednoduchým řešením je posloupnost seřadit a pak vzít prvek na k -tém místě, existují však rychlejší algoritmy, jak takovýto prvek najít. Implementací jednoho z nich je šablonová funkce `nth_element<>()`. Tato funkce očekává tři iterátory s náhodným přístupem: První z nich určuje počátek zpracovávaného úseku kontejneru, druhý pozici, která nás zajímá, a třetí ukazuje na první prvek za zpracovávaným úsekem.

Tato funkce rozdělí posloupnost na tři úseky. První, o délce $k - 1$ prvků, obsahuje všechny prvky, které jsou menší než hledaný k -tý (nebo jsou mu rovny). Druhý úsek o délce 1 obsahuje prvek, který je k -tý podle velikosti, a poslední úsek obsahuje všechny prvky, které jsou větší než hledaný k -tý nebo jsou mu rovny. Pořadí prvků uvnitř prvního a třetího úseku je libovolné.

Máme např. pole celých čísel,

```
int A[9] = {9,1,8,2,7,3,6,4,5};
```

a zajímá nás čtvrtý prvek podle velikosti. Napíšeme tedy

```
nth_element(A, A+3, A+9);
```

a po provedení tohoto příkazu bude `A[3]` obsahovat hledaný prvek. (Nezapomeňte, že indexujeme od nuly.)

Přetížené verzi této funkce můžeme jako čtvrtý parametr zadat predikát určující způsob porovnávání prvků (viz tip 1135).

1141 Otočení obsahu posloupnosti



Chceme-li otočit obsah posloupnosti, tedy přerovnat prvky v daném kontejneru nebo v jeho části v opačném pořadí, než v jakém právě je, použijeme šablonovou funkci `reverse<>()`. Tato funkce očekává dva obousměrné iterátory; první z nich jako obvykle určuje první prvek zpracovávaného úseku, druhý ukazuje na první prvek za ním.

Máme například pole znaků `text` obsahující řetězec "9876.1-", který je výsledkem převodu číselné hodnoty do znakové podoby, ale je zapsaný obráceně. To znamená, že ho potřebujeme otočit. K tomu nám poslouží algoritmus `reverse()`, kterému předáme obousměrný iterátor ukazující na první prvek otáčeného úseku a obousměrný iterátor ukazující za poslední prvek tohoto úseku:

```
reverse(text, text+strlen(text));
```

Kdybychom tento řetězec měli v instanci `text` typu `string`, napsali bychom

```
reverse(text.begin(), text.end());
```

1142 Náhodné promíchání prvků kontejneru



Při simulacích náhodných dějů potřebujeme někdy náhodně promíchat obsah kontejneru. K tomu slouží funkce `random_shuffle<>()`, které předáme dvojici iterátorů pro náhodný přístup určující rozmezí v kontejneru, které chceme promíchat.

Máme pole

```
int A[9] = {1,2,3,4,5,6,7,8,9};
```

a chceme promíchat všechny prvky kromě prvních tří a posledního – ty zůstanou na svém místě:

```
random_shuffle(A+3, A+8);
```

Výsledkem může být pole s prvky 1, 2, 3, 8, 5, 7, 4, 9.

1143 První prvek s danou hodnotou



Potřebujeme najít v kontejneru první prvek s danou hodnotou. K tomu použijeme funkci `find<>()`, které předáme vstupní iterátory ukazující na první a za poslední prvek prohledávaného úseku a požadovanou hodnotu. Tato funkce vrátí vstupní iterátor ukazující na nalezenou hodnotu nebo za poslední prvek prohledávaného úseku.

Máme např. znakový řetězec

```
string text = "Žluťoučký kůň příšerně úpěl";
```

a chceme najít první výskyt znaku 'č'. K tomu nám poslouží příkaz

```
string::iterator it = find(text.begin(), text.end(), 'č');
if(it != text.end()) { // nějaké zpracování, byl-li nalezen
```

1144 První prvek splňující danou podmínku



Máme kontejner a chceme v něm najít první prvek, který splňuje danou podmínku. Pak nám pomůže funkce `find_if<>()`, které předáme vstupní iterátory ukazující na první a za poslední prvek prohledávaného úseku a predikát – funkční objekt – implementující danou podmínku. Tato funkce vrátí vstupní iterátor ukazující na nalezenou hodnotu nebo za poslední prvek prohledávaného úseku.

Vezměme opět řetězec

```
string text = "Žluťoučký kůň příšerně úpěl";
```

a chceme v něm najít první velké písmeno. Definujeme proto funkční objekt

```
locale loc("Czech_Czech Republic");
struct podm {
    bool operator()(int c)
    { // Vrátí true, vyhovuje-li znak podmínce
```

```
        return isupper(c, loc);  
    }  
};
```

Příkaz pro vyhledání prvního velkého písmene bude mít tvar

```
string::iterator it = find_if(text.begin(), text.end(), podm());
```

V našem případě bude `it` ukazovat hned na první znak.

1145 První prvek z dané množiny



Chceme v kontejneru najít první prvek, který je v dané množině; tato množina je dána obsahem jiného kontejneru. V takovém případě použijeme funkci `find_first_of<>()`, která očekává dvojici dopředných iterátorů určujících prohledávaný úsek a další dvojici dopředných iterátorů určujících množinu, jejíž prvek hledáme. Tato funkce vrátí dopředný iterátor ukazující na nalezený prvek nebo – v případě neúspěchu – za poslední prvek prohledávaného úseku.

Vezmeme ještě jednu řetězec

```
string text = "Žluťoučký kůň příšerně úpěl";
```

Naším úkolem je tentokrát najít první malé písmeno s diakritickým znaménkem. Budeme-li uvažovat pouze česká diakritická znaménka, stačí nám jako definice množiny hledaných znaků řetězec

```
string diakritika = "áéěíóúůýžščřďťň";
```

Vlastní hledání obstará příkaz

```
string::iterator it = find_first_of(text.begin(), text.end(),  
                                   diakritika.begin(), diakritika.end());
```

1146 Počet prvků se zadanou hodnotou



Chceme zjistit, kolik prvků se zadanou hodnotou kontejner obsahuje. K tomu nám pomůže funkce `count<>()`, které předáme dvojici dopředných iterátorů určujících prohledávaný rozsah a hledanou hodnotu. Chceme-li např. zjistit, kolikrát se vyskytuje znak `'k'` v řetězci

```
string text = "Žluťoučký kůň příšerně úpěl";
```

použijeme příkaz

```
int kolik = count(text.begin(), text.end(), 'k');
```

1147 Počet prvků vyhovujících dané podmínce



Chceme-li zjistit, kolik prvků v našem daném kontejneru vyhovuje dané podmínce, použijeme funkci `count_if<>()`, které předáme dvojici dopředných iterátorů určujících prohledávaný rozsah a predikát určující, zda daný prvek vyhovuje podmínce.

Vezmeme náš oblíbený řetězec

```
string text = "Žluťoučký kůň příšerně úpěl";
```

a zkusme zjistit, kolik obsahuje znaků s diakritickými znaménky. Za tím účelem napíšeme následující predikát:

```
class podm {
    static string diakritika;
public:
    bool operator()(char c)
    {
        string::iterator it = find(diakritika.begin(),
                                   diakritika.end(), c);
        return it != diakritika.end();
    }
};
string podm::diakritika = "áéěíóúůýžščřďňáěííóúůýžščřďň";
```

Počet znaků s danou vlastností zjistíme příkazem

```
int kolik = count_if(text.begin(), text.end(), podm());
```

Mimochodem, v tomto textu jich je 12.

1148 Kopírování do jiného kontejneru



Chceme zadaný rozsah prvků z jednoho kontejneru překopírovat na udané místo do jiného kontejneru. K tomu použijeme funkci `copy<>()`, které předáme dvojici vstupních iterátorů určujícího kopírovaný rozsah a výstupní iterátor určující místo, od kterého se mají kopie prvků z prvního kontejneru ukládat.

Máme pole celých čísel

```
int pole[] = {1,3,5,6,8,9, 99, 77, 22};
```

a chceme jeho obsah okopírovat do vektoru `vek` celých čísel. K tomu použijeme příkaz

```
copy(pole, pole+9, vek.begin());
```



Upozornění: Prvky cílového kontejneru již musí existovat, nestačí, byla-li jim např. vyhrazena paměť pomocí metody `reserve()` třídy `vector<>`.

1149 Odstranění prvků s danou hodnotou



Chceme-li z kontejneru odstranit prvky s jistou hodnotou, použijeme funkci `remove<>()`, které předáme dvojici vstupních iterátorů určujících rozsah, který chceme zpracovat, a hodnotu, kterou chceme odstranit. Máme-li např. oblíbený řetězec

```
string text = "Žluťoučký kůň příšerně úpěl";
```

a chceme z něj odstranit všechny mezery. Pak stačí napsat

```
remove(text.begin(), text.end(), ' ');
```

1150 Kopírování prvků, které splňují danou podmínku



Chceme ze zadaného rozsahu prvků v jednom kontejneru překopírovat na udané místo do jiného kontejneru prvky, které splňují jistou podmínku. K tomu použijeme funkci `remove_copy_if<>()`, které předáme dvojici vstupních iterátorů určující kopírovaný rozsah a výstupní iterátor určující místo, od kterého se mají kopie prvků z prvního kontejneru ukládat.

Vezmeme opět text o žlutém koni a chceme vytvořit jeho kopii, a to v poli znaků

```
char vysledek[100];
```

Tato kopie ovšem nebude obsahovat znaky s diakritickými znaménky; ty chceme odstranit. K tomu použijeme predikát `podm` z tipu 1147 a napíšeme

```
char* kde = remove_copy_if(text.begin(), text.end(),
                           vysledek, podm());
```

```
*kde = 0;
```

Tato funkce vrací iterátor ukazující za místo, kam byl překopírován poslední znak, a protože přesně tam potřebujeme doplnit nulu ukončující řetězec uložený v poli znaků, uložili jsme si ho do pomocné proměnné `pom`. Výsledkem bude řetězec "luouk k pern pl".

1151 Transformace všech prvků v kontejneru po jednom



Máme vektor celých čísel a chceme všechny hodnoty v něm nahradit jejich druhými mocninami. K tomu použijeme jednu z variant funkce `transform<>()`, jež očekává dvojici vstupních iterátorů určujících transformovaný rozsah, výstupní iterátor určující místo, kam se mají zapisovat výsledné hodnoty, a unární funktor, který se o transformaci postará.

Výsledky můžeme zapisovat do jiného nebo do téhož kontejneru.

V našem případě použijeme jako funktor např. ukazatel na funkci

```
int sqr(int n){return n*n;}
```

Vlastní transformaci obstará příkaz

```
transform(vek.begin(), vek.end(), vek.begin(), sqr);
```

1152 Fibonacciova čísla: transformace prvků po dvou



Chceme zjistit prvních 40 Fibonacciových čísel. Nulté číslo je 0, první je 1 a každé další je součtem předchozích dvou, $a_{n+1} = a_n + a_{n-1}$. K tomu lze také využít funkci `transform<>()`, tentokrát variantu s funktorem se dvěma parametry. Tato varianta očekává dvojici vstupních iterátorů určujících zpracovávaný rozsah, vstupní iterátor určující, odkud brát druhý argument transformačního funktoru, a nakonec samozřejmě transformační funktor.

Získání pole Fibonacciových čísel může vypadat takto: Nejprve definujeme potřebný funktor,

```
int f(int n1, int n2){return n1+n2;}
```

Pak definujeme pole, uložíme do něj první dvě hodnoty a transformaci dopočítáme ostatní.

```
int Fib[N] = {0,1}; // Uložíme pouze první dvě hodnoty
transform(Fib, &Fib[N-2], Fib+1, Fib+2, f);
```

Nyní bude pole `Fib` obsahovat posloupnost 0, 1, 1, 2, 3, 5, 8, 13...

1153 Procházíme postupně permutace prvků



Při řešení některých úloh nám nezbude než použít metodu hrubé síly – tedy probrat všechny možnosti. V určitých úlohách potřebujeme projít všechna možná přeuspořádání, tedy všechny permutace prvků kontejneru. K tomu slouží funkce `next_permutation<>()` a `prev_permutation<>()`. Obě funkce očekávají dvojici obousměrných iterátorů určujících rozsah permutovaných prvků (první z nich ukazuje jako obvykle na první prvek tohoto rozsahu, druhý ukazuje za poslední prvek rozsahu). Funkce `next_permutation<>()` přerovná prvky v daném rozsahu tak, že vznikne lexikograficky následující permutace, funkce `prev_permutation<>()` přerovná prvky v daném rozsahu tak, že vznikne lexikograficky předcházející permutace (pokud takové permutace existují). Obě funkce vracejí hodnotu typu `bool`, která říká, zda následující, resp. předcházející permutace existuje a byla vytvořena.

Máme-li např. pole

```
int A[] = {1, 6, 4, 7, 9, 0, -1};
```

vznikne z něj příkazem

```
next_permutation(A, A+7);
```

pole obsahující permutaci 1, 6, 4, 9, -1, 0, 7, zatímco po provedení příkazu

```
prev_permutation(A, A+7);
```

bude totéž výchozí pole obsahovat permutaci 1, 6, 4, 7, 9, -1, 0.

Poznamenejme, že třetím parametrem obou těchto funkcí může být binární predikát určující způsob porovnávání prvků permutované posloupnosti.

1154 Binární vyhledávání



Máme seřazenou posloupnost a chceme zjistit, zda se v ní vyskytuje prvek se zadanou hodnotou. (Máme například seřazený seznam struktur obsahujících data našich zaměstnanců a chceme zjistit, zda u nás pracuje Josef Novák.) Pak nám pomůže funkce `binary_search<>()`, jež očekává dvojici dopředných iterátorů určujících prohledávaný rozsah a hledanou hodnotu. Čtvrtým parametrem může být binární predikát pro porovnání prvků.

Tato funkce vrátí `true`, najde-li hledanou hodnotu, a `false` v opačném případě.

Podívejme se na příklad: V programu je definována struktura `zaměstnanec`,

```
struct zamestnanec {
    string jmeno, prijmeni;
    int plat;
};
```

a vektor zaměstnanců

```
vector<zamestnanec> oni;
```

Naším úkolem je zjistit, zda je mezi nimi Josef Novák. Definujeme tedy predikát pro porovnání pole příjmení a jména

```
struct porovnej {
    bool operator()(zamestnanec z1, zamestnanec z2)
    {
        return (z1.prijmeni < z2.prijmeni) ||
            ((z1.prijmeni == z2.prijmeni) && (z1.jmeno < z2.jmeno));
    }
};
```

Dále si připravíme instanci zaměstnance s patřičnými daty

```
zamestnanec jn = {"Josef", "Novák", 0};
```

a spustíme hledání

```
bool je = binary_search(oni.begin(), oni.end(), jn, porovnej());
```



Poznámka: Složitost tohoto algoritmu je podle standardu úměrná $\log(n)$, kde n je počet prvků v prohledávaném rozmezí. To však lze splnit pouze pro iterátory s náhodným přístupem; v případě dopředných iterátorů je složitost úměrná počtu n prvků.

1155 Kam vložit nový prvek?



Máme seřazenou posloupnost a chceme do ní vložit nový prvek tak, aby seřazení zůstalo zachováno. K tomu nám poslouží dvojice funkcí `lower_bound<>()` a `upper_bound<>()`, které očekávají stejné parametry jako funkce `binary_search<>()` popsaná v předchozím tipu, ale vracejí dopředný iterátor určující rozmezí, do něhož je třeba novou

hodnotu vložit. Přesněji, funkce `lower_bound<>()` vrátí iterátor určující první hodnotu, která je menší nebo rovna zadané, a funkce `upper_bound<>()` vrátí první hodnotu, která je větší než zadaná.

Vezměme vektor celých čísel, který obsahuje posloupnost 1, 3, 3, 3, 4, 4, 7, 8, 9, 9, 9:

```
vector<int> vek;
```

a zkusme určit, kam vložit nový prvek s hodnotou 4 uloženou v proměnné `co`, aby zůstalo zachováno řazení:

```
vector<int>::iterator i1 = lower_bound(vek.begin(), vek.end(), co);
```

```
vector<int>::iterator i2 = upper_bound(vek.begin(), vek.end(), co);
```

Z rozdílů `i1 - vek.begin()`, resp. `i2 - vek.begin()` zjistíme, že první místo, kam lze nový prvek vložit, má index 4, a poslední má index 6. Přitom „vložení“ znamená, že se následující prvky odsunou o jednu pozici doprava. Po provedení operace

```
vek.insert(i1, co);
```

bude vek obsahovat posloupnost 1, 3, 3, 3, 4, 4, 4, 7, 8, 9, 9, 9.

1156 Vyplňujeme kontejner zadanou hodnotou



začátečník

Máme kontejner a chceme v něm zadaný rozsah prvků vyplnit jistou hodnotou. K tomu můžeme použít funkci `fill<>()` nebo `fill_n<>()`. První z nich očekává dvojici dopředných iterátorů určujících vyplňovaný rozsah a hodnotu, která se má do prvků v tomto úseku vložit; druhá očekává iterátor určující počátek rozsahu, počet prvků, které má vyplnit, a hodnotu, která se má do prvků v tomto úseku vložit.

Máme vektor `vek` celých čísel, který již obsahuje 10 prvků, a my chceme prvních 5 nahradit nulami. K tomu nám poslouží příkaz

```
fill_n(vek.begin(), 5, 0);
```

Upozornění: Vyplňované prvky již musí v kontejneru existovat. Funkce `fill_n<>()` žádné nové prvky nevytvoří.

1157 Vyplňujeme kontejner generovanými hodnotami



znalec

Chceme vyplnit zadaný rozsah kontejneru hodnotami vytvořenými generátorem zadaným postupem.

K tomu použijeme funkci `generate<>()`, které předáme dvojici dopředných iterátorů určujících vyplňovaný rozsah a generátor – funkční objekt s přetíženým operátorem volání funkce (bez parametrů), který vrací hodnoty, jež budou vloženy do prvků v zadaném rozsahu. (Můžeme také použít ukazatel na funkci bez parametrů.)

Vezměme pole celých čísel, které chceme vyplnit náhodnými čísly v rozmezí od 0 do N . Generátor bude mít tvar

```
struct gener{
    int operator()()
    {
        return rand() % N;
    }
};
```

Ke generování náhodných čísel jsme využili standardní funkci `rand()` z knihovny jazyka C, jejíž deklaraci najdeme v hlavičkovém souboru `<stdlib.h>`. Vlastní příkaz pro vyplnění pole bude mít tvar

```
generate(pole, pole+sizeof(pole)/sizeof(int), gener());
```

1158 Sloučení dvou setříděných úseků



Máme dva seřazené rozsahy (v jednom nebo ve dvou kontejnerech) a potřebujeme je sloučit do dalšího rozsahu tak, aby vznikl opět seřazený úsek obsahující hodnoty z obou původních úseků. K tomu nám poslouží funkce `merge<>()`, které předáme dvojici vstupních iterátorů určujících první úsek, dvojici vstupních iterátorů určujících druhý úsek a výstupní iterátor určující počátek úseku, kam chceme zapsat výsledek. *Všechny tři úseky mohou ležet v tomtéž kontejneru, avšak úsek, kam zapisujeme výsledek, se nesmí překrývat se žádným z úseků obsahujících slučované hodnoty.*

Tato funkce vrátí iterátor ukazující na první prvek za koncem úseku obsahujícího výsledek.

Máme-li například pole

```
int pole1[] = {1,3,3,3,4,4,7,8,9,9,9};
int pole2[] = {2,2,3,7,7,9,11};
```

sloučíme je a výsledek zapíšeme do `pole3` příkazem

```
merge(pole1, pole1+sizeof(pole1)/sizeof(int), pole2,
      pole2+sizeof(pole2)/sizeof(int), pole3);
```

Přetížená verze této funkce umožňuje zadat jako poslední parametr binární predikát určující způsob porovnávání prvků.

1159 Sloučení dvou setříděných úseků na místě



Máme dva seřazené úseky, které v kontejneru následují bezprostředně za sebou, chceme je sloučit a nahradit je výsledkem slučování. K tomu použijeme funkci `inplace_merge<>()`, které předáme tři obousměrné iterátory. První a druhý určují první slučovaný úsek, druhý a třetí určují druhý slučovaný úsek.

Přetíženou verzi této funkce můžeme jako další parametr zadat binární predikát určující způsob porovnávání prvků ve slučovaných úsecích. Ani jedna z těchto funkcí nic nevrací.

Vezmeme např. pole

```
int pole[10] = {1,5,7,19,44,4,7,8,9,9};
```

které je tvořeno dvěma seřazenými úseky; první tvoří prvky s indexy 0 – 4, druhý prvky s indexy 5 – 9. Voláním

```
inplace_merge(pole, pole+5, pole+10);
```

z něj vytvoříme seřazené pole obsahující hodnoty 1, 4, 5, 7, 7, 8, 9, 9, 19, 44.



Poznámka: Čtenář obeznámený se základními třídícími algoritmy v této proceduře jistě poznal slučovací proceduru, na které je založen algoritmus třídění slučováním (mergesort).

1160 Množiny a operace s nimi



STL obsahuje třídu `set<>` určenou k reprezentaci množiny. Tato třída ovšem neimplementuje žádné běžné množinové operace; k tomu musíme použít funkce z hlavičkového souboru `<algorithm>`. Tyto funkce však ve skutečnosti umožňují používat jako množinu *libovolný seřazený úsek jakékoli posloupnosti* nebo množiny s opakováním (multiset).

Najdeme zde funkci `includes()`, jež umožňuje testovat, zda jedna množina je podmnožinou jiné množiny, funkce `set_intersection()`, `set_union()`, `set_difference()` a `set_symmetric_difference()` pro výpočet průniku, sjednocení, rozdílu a symetrické difference množin. Všechny tyto funkce očekávají množiny zadané prostřednictvím dvojice vstupních iterátorů; všechny kromě první také očekávají výstupní iterátor určující kontejner a místo v něm, do něhož bude zapsán výsledek.

Funkce testující, zda A je podmnožinou B, vrací hodnotu typu `bool`; zbylé funkce vrací výstupní iterátor ukazující na první prvek za úsekem, kde je zapsán výsledek.

1161 Je A podmnožinou B?



Máme množinu A reprezentovanou jako seřazený úsek posloupnosti a chceme zjistit, zda jde o podmnožinu jiné množiny B, opět reprezentované jako seřazený úsek posloupnosti obsahující hodnoty stejného typu jako A. Zjišťujeme tedy, zda platí relace $A \subset B$.

K tomu nám poslouží funkce `includes<>()`. Vezměme např. množinu A reprezentovanou polem

```
int mnozina_A[4] = {1,5,6,8};
```

a chceme zjistit, zda je A podmnožinou množiny B, kterou reprezentuje seznam

```
list<int> mnozina_B;
```

obsahující prvky 1, 2, 5, 6, 7, 9, 99, 991, 992, 993. Příkaz

```
bool je = includes(mnozina_B.begin(), mnozina_B.end(), mnozina_A,
                  mnozina_A+4);
```

nám zjistí, že A není podmnožinou B .

Všimněte si, že nejprve uvádíme množinu B (tedy tu, která by měla být nadmnožinou), a pak teprve uvádíme množinu A (tedy tu, která by měla být podmnožinou).

1162 Průnik množin A a B



Máme opět dvě množiny celých čísel, A a B , obě reprezentované jako seřazené úseky posloupností, a chceme najít jejich průnik, $A \cap B$. Přitom A je množina čísel typu `int` s opakováním

```
multiset<int> mnozina_A;
```

obsahující prvky 6, 7, 88, 7, 1. Poznamenejme, že instance této třídy se samy postarají o seřazení hodnot, které obsahují. Množina B je dána jako instance třídy `vector<int>`, jež obsahuje hodnoty 1, 2, 5, 6, 7, 9, 99, 991, 992, 993.

Hledáme průnik těchto dvou množin a chceme ho zapsat do pole `vysledek`.

```
int * p = set_intersection(mnozina_A.begin(), mnozina_A.end(),
                           mnozina_B.begin(), mnozina_B.end(),
                           vysledek);
```

V poli `vysledek` budou zapsány hodnoty 1, 6, 7 a ukazatel `p` bude ukazovat na prvek `vysledek[3]`.

1163 Větší nebo menší ze dvou hodnot



Mezi algoritmy patří také šablonová implementace funkce pro určení větší ze dvou hodnot. Jmenuje se podle očekávání `max<>()` a lze jí zadat dvě hodnoty typu, pro který je definováno porovnání pomocí operátoru `<`; tyto hodnoty také musí být možné předávat jako parametry hodnotou.

Přetížená verze této funkce očekává jako třetí parametr predikát určující způsob porovnání.

Pro určení menší ze dvou hodnot máme k dispozici dvojici šablonových funkcí `min<>()` se stejnými parametry jako má `max<>()`.

Jsou-li a a b dvě proměnné typu `double`, zjistíme menší z nich zápisem

```
double c = min(a, b);
```

1164 Nejmenší nebo největší prvek v daném úseku kontejneru



Chceme-li najít nejmenší nebo největší prvek v daném úseku kontejneru, použijeme funkci `min_element<>()`, resp. `max_element<>()`. Obě tyto funkce jsou k dispozici ve dvou verzích. První z nich očekává dvojici dopředných iterátorů, které určí úsek (rozsah prvků), v němž chceme maximum nebo minimum hledat. Přetížené verze očekávají navíc binární predikát určující způsob porovnávání.

Tyto funkce vracejí dopředný iterátor ukazující na nalezený nejmenší, resp. největší prvek.

Máme například celočíselné pole `pom` a chceme najít největší prvek mezi prvními deseti. K tomu použijeme příkaz

```
int* p = max_element(pom, pom+10);
```

Ukazatel `p` bude obsahovat adresu nalezeného největšího prvku.

1165 Prohození obsahu dvou proměnných



začátečník

S úlohou prohodit obsah dvou proměnných se setkáme při programování složitějších algoritmů poměrně často. I když jde o velmi jednoduchý algoritmus, standardní knihovna nabízí jeho šablonovou implementaci pod jménem `swap<>()`.

Jsou-li `a` a `b` dvě proměnné typu `double`, prohodíme jejich obsahy zápisem

```
swap(a, b);
```

Potřebujeme-li prohodit prvky kontejneru, oceníme funkci `iter_swap<>()`, jež očekává dvojici dopředných iterátorů. Tyto iterátory mohou ukazovat do téhož kontejneru nebo do dvou různých kontejnerů, ovšem na prvky stejného typu.

Máme například pole a vektor celých čísel

```
int pom[10];
vector<int> vl;
```

a chceme prohodit jejich první prvky. Napíšeme tedy

```
iter_swap(pom, vl.begin());
```

1166 Prohození úseků dvou kontejnerů



pokročilý

Někdy potřebujeme prohodit obsahy celých úseků kontejnerů. K tomu použijeme funkci `swap_ranges<>()`, jež očekává dvojici dopředných iterátorů určujících úsek v prvním kontejneru a dopředný iterátor určující začátek druhého úseku. Tyto úseky mohou ležet v témže kontejneru nebo ve dvou různých kontejnerech, jestliže ovšem obsahují hodnoty stejného typu.

Funkce `swap_ranges<>()` vrací dopředný iterátor ukazující za prohozený úsek ve druhém kontejneru.

Máme například celočíselné pole a vektor

```
int cisla[10];
vector<int> numera;
```

První tři prvky v poli prohodíme s prvními třemi prvky vektoru příkazem

```
vector<int>::iterator it = swap_ranges(cisla, cisla+3, numera.begin());
```

Iterátor `it` pak bude ukazovat na čtvrtý prvek vektoru `numera`, tedy za poslední vyměněný prvek.

1167 Prohození obsahu kontejnerů



Máme dva kontejnery stejného typu a chceme prohodit jejich obsah. K tomu slouží metoda `swap()`, kterou najdeme ve všech kontejnerech s výjimkou adaptérů (fronty, prioritní fronty a zásobníku).

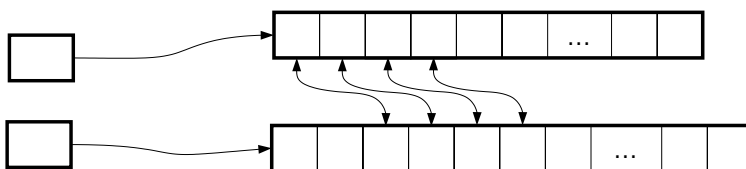
Nejjednodušeji lze rozdíl mezi metodou `swap()` a funkcí `swap(>>())` popsat takto:

- Funkce `swap(>>())` prohazuje jednotlivé prvky.
- Metoda `swap()` prohodí ukazatele na obsah kontejneru (na dynamicky alokovanou paměť, v níž je uložen obsah vektoru, množiny atd.).

To znamená, že při použití metody `swap()` se nestaráme o to, zda jde o stejně velké úseky. Metoda `swap()` by také nikdy neměla vyvolat výjimku. Musíme však mít dva kontejnery stejného typu.

Instance kontejneru

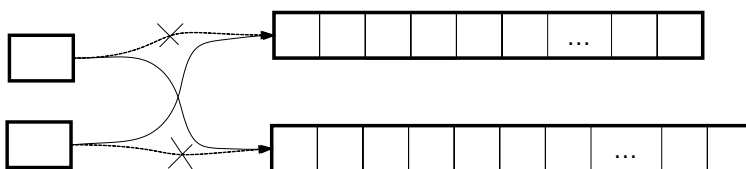
Jeho dynamicky alokovaná paměť



Obrázek 38 Funkce `swap(>>())` vymění obsahy odpovídajících úseků kontejnerů

Instance kontejneru

Jeho dynamicky alokovaná paměť



Obrázek 39 Metoda `swap()` prohodí ukazatele na dynamicky alokovanou paměť

1168 Standardní predikáty



Ve standardní knihovně najdeme několik předem připravených predikátů, které překladač použije, jestliže sami žádný nezadáme. Můžeme je však také použít ve svých programech explicitně. Jejich jména jsou dostatečně výstižná, takže se omezíme na výčet některých z nich.

- Pro aritmetické operace jsou k dispozici funkční objekty `plus<>`, `minus<>`, `multiplies<>`, `divides<>`, `modulus<>`, `negate<>`.

- Pro porovnávání jsou k dispozici `equal_to<>`, `not_equal_to<>`, `greater<>`, `less<>`, `greater_equal<>`, `less_equal<>`.
- Pro logické operace jsou k dispozici `logical_and<>`, `logical_or<>` a `logical_not<>`.

Dále jsou k dispozici objekty, které zapouzdří existující funkci s jedním nebo dvěma parametry do funkčního objektu, které udělají z funkce se dvěma parametry funkční objekt s jedním parametrem (tím, že dosadí pevnou hodnotu jednoho z parametrů) atd. Najdeme je v hlavičkovém souboru `<functional>`.

1169 Funkční objekt vracející větší ze dvou hodnot



Vytvoříme si šablonu funkčního objektu, jehož přetížený operátor volání funkce vrátí hodnotu většího se dvou parametrů. K porovnání použijeme předaný predikát; jako implicitní hodnotu použijeme standardní predikát `less<>`.

```
template<class T, class porovnavani = less<T> >
class Max
{
    porovnavani c;
public:
    T operator()(T a, T b)
    {
        return c(a, b) ? b : a;
    }
};
```

Jsou-li `a` a `b` dvě proměnné typu `long`, můžeme napsat

```
Max<int> mi;
int c = mi(a, b); // Vrátí větší z čísel a a b
```

Poznamenejme, že poslední zápis můžeme – za cenu přehlednosti – zkrátit do tvaru

```
int c = Max<int>()(a, b);
```

První závorky znamenají volání konstruktoru, které vytvoří dočasný objekt, další závorky pak použití přetíženého operátoru volání funkce pro tento dočasný objekt.

1170 Další iterátory



Většinou vystačíme s iterátory, které nám poskytují kontejnery jako vnořené typy. Někdy však budeme potřebovat ještě další iterátory.

Hlavičkový soubor `<iterator>` nabízí mj. třídu `iterator<>`, která je společným předkem všech tříd iterátorů. Mezi odvozenými typy najdeme iterátory pro práci s datovými proudy a jejich vyrovnávacími paměťmi, vkládací iterátory a mnohé další.

V tomto hlavičkovém souboru jsou také definovány operátory pro práci s iterátory.

1171 Kopírování přímo do proudu



Máme pole znakových řetězců

```
string pole[] = {"haha", "hehe", "hihi", "hoho"};
```

a toto pole chceme vypsát do proudu `cout`. K tomu lze využít algoritmus `copy<>()`, kterému jako cíl zadáme iterátor na tomto proudu.

```
#include <iterator>
ostream_iterator<string> oi(cout, " ");
```

Prvním parametrem konstruktoru je proud, na kterém má tento výstupní iterátor pracovat, a druhým je omezovač, znakový řetězec, který bude vložen mezi jednotlivé vystupující položky. Pak už můžeme kopírovat:

```
copy(pole, pole+4, oi);
```