

Kontejnery ve standardní knihovně

1096 Co jsou kontejnery a jak se dělí



začátečník

Kontejnery jsou třídy, jejichž instance slouží k uskladňování dat. Obvykle se dělí na dvě základní skupiny – *posloupnosti* (sequence) a *asociativní kontejnery* (slovníky, dictionary).

Posloupnosti jsou kontejnery, u nichž má smysl hovořit o pořadí dat. Data jsou v posloupnosti uložena zpravidla v pořadí, v němž byla do kontejneru uložena. Není to ale podmínkou: Mnohé kontejnery umožňují vložit nový údaj na místo, které si sami určíme.

V případě posloupností má také smysl operace seřazení dat podle velikosti („třídění“).

Implementace posloupností je zpravidla založena na dynamicky zvětšovaném poli nebo na lineárním spojovém seznamu.

V případě *asociativního kontejneru* neboli slovníku nemá pořadí dat význam. Kontejner si sám určuje, kam který údaj uloží. Údaje mají zpravidla tvar dvojice klíč – hodnota, a zadáme-li klíč, umožňuje asociativní kontejner k němu efektivně vyhledat odpovídající hodnotu.

Implementace asociativních kontejnerů bývá založena na hešové tabulce nebo na některé z variant vyváženého binárního vyhledávacího stromu. V současné verzi STL není hešová tabulka k dispozici, takže je to vždy strom.

1097 Jaké posloupnosti máme v STL k dispozici



začátečník

Asi nejpoužívanější posloupností je vektor, tedy šablonová třída `vector<>` deklarovaná v hlavičkovém souboru `<vector>`. Je to v podstatě zapouzdřené pole, jehož velikost se může v případě potřeby automaticky zvětšovat.

Šablona `list<>`, deklarovaná v hlavičkovém souboru `<list>`, představuje implementaci obousměrně zřetězeného spojového seznamu.

Jako podkladová třída pro implementace jiných posloupností se často používá dvoustanná fronta; šablona `deque<>` je deklarována v hlavičkovém souboru `<deque>`.

Další užitečné třídy jsou *zásobník* a *fronta* (`stack<>` a `queue<>`, deklarované ve stejnojmenných hlavičkových souborech). Máme k dispozici i prioritní frontu (frontu s předbíráním), a to jako třídu `priority_queue<>`, deklarovanou opět v `<queue>`.

1098 Jaké asociativní kontejnery máme k dispozici



Prvním z asociativních kontejnerů je *mapa* (šablona `map<>`, deklarovaná v hlavičkovém souboru `<map>`). Obsahuje dvojice klíč – hodnota a umožňuje rychle vyhledat hodnotu odpovídající zadanému klíči. Přitom klíč musí být jednoznačný. Data jsou do kontejneru umísťována podle hodnoty klíče a podle ní jsou také vyhledávána. Hodnotu sdruženou s jistým klíčem lze měnit přímo; klíč takto měnit nelze – z kontejneru musí být odstraněna celá dvojice klíč – hodnota.

Vedle toho je v témže hlavičkovém souboru definována třída `multimap<>` (*multimapa*); podobá se mapě až na to, že klíče se mohou opakovat, tj. nemusí být jednoznačné.

Množina sice nepatří mezi posloupnosti, ale není to ani asociativní kontejner, protože do něj neukládáme dvojice klíč – hodnota, ale pouze hodnoty (které ovšem slouží zároveň jako klíče). Žádná hodnota se nesmí opakovat. V STL ji představuje šablonová třída `set<>` definovaná v hlavičkovém souboru `<set>`.

Multimnožina (šablonová třída `multiset<>` definovaná v hlavičkovém souboru `<set>`) je množina, ve které se hodnoty mohou opakovat.

1099 Další kontejnery



Vedle kontejnerů, o nichž jsme se zmínili v tipech 1097 a 1098, nabízí STL ještě některé další.

Šablona `bitset<size_t N>` definovaná v hlavičkovém souboru `<bitset>` a některé související funkce, jsou určeny k manipulacím s posloupnostmi bitů o pevně zadané délce. (Parametrem této šablony je celé číslo, nikoli datový typ.)

Specializace šablony `vector<bool>` pro typ `bool` by měla optimalizovat práci s pamětí – tedy nejspíše ukládat logické hodnoty jako jednotlivé bity.

Obě tyto šablony „vybočují z řady“, nechovají se v mnoha ohledech jako ostatní kontejnery z STL.

1100 Současná verze STL není úplná



Na první pohled je zřejmé, že v STL chybí hešová tabulka. Vedle toho zde chybí objektové zapouzdření klasického pole z jazyka C, které by splňovalo požadavky kladené na kontejner z STL, třída pro vyjádření *n*-tice, jednosměrně zřetězený seznam, binární vyhledávací strom a případně další kontejnery.

Místo binárního stromu lze zpravidla použít množinu (třidu `set<>`), neboť ta je typicky implementována jako vyvážený binární vyhledávací strom.

Nová verze standardu by měla mimo jiné přinést hešovou tabulku, a to jako třídy `unordered_set<>`, `unordered_map<>`, `unordered_multiset<>` a `unordered_multimap<>`. Dále by měla obsahovat třídu `array<>` jako zapouzdření pole a třídu `tuple<>` představující *n*-tici.



Poznámka: Důvodem, proč tyto kontejnery zatím nejsou součástí STL, je, že standardizační komise nestihla připravit jejich návrh; i tak trvala příprava první verze standardu devět let. Očekává se však, že připravovaná nová verze standardu bude potřebné kontejnery (a mnohé další užitečné třídy a funkce) obsahovat.

Příčina, proč se třídy pro různé varianty hešové tabulky nebudou v nové verzi standardu jmenovat `hash_map<>` apod., spočívá nejspíše v tom, že třídy s těmito jmény nabízely jako nestandardní rozšíření starší implementace standardní knihovny.

1101 Jak používat kontejnery z STL: parametry šablony



začátečník

Šablony v STL jsou navrženy tak, aby je bylo možno používat jednotným způsobem – samozřejmě pokud to jde. To znamená, že typ a pořadí parametrů šablony je u většiny tříd buď stejné, nebo alespoň podobné, metody, které mají podobný účel, mají stejná jména atd.

Prvním parametrem šablony je vždy typ ukládaných hodnot; u šablon `map<>` a `multi-map<>` představují první dva parametry typy klíče a hodnoty.

Posledním parametrem je vždy alokátor – instance třídy, která se stará o alokaci paměti pro daný kontejner. Implicitní hodnotou tohoto parametru je standardní alokátor – třída `allocator` definovaná v hlavičkovém souboru `<memory>`. Tento parametr zadáváme, pouze když potřebujeme řídit práci s pamětí jinak než standardním způsobem.

Třídy, které fungují jako adaptéry, tj. pouze poskytují alternativní rozhraní k existujícím třídám, mají jako druhý parametr třídu podkladového kontejneru. (Například frontu nebo zásobník lze vytvořit jako adaptér dvoustranné fronty nebo seznamu.)

1102 Jak používat kontejnery z STL: konstruktory



začátečník

Všechny kontejnery mají konstruktor bez parametrů, který zkonstruuje prázdný kontejner. Všechny kontejnery kromě „obyčejné“ fronty (`queue<>`) a zásobníku (`stack<>`) mají konstruktor, kterému lze jako parametry zadat dvojici iterátorů určujících rozmezí prvků v jiném kontejneru, jejichž kopiemi se má nově vytvořený kontejner naplnit.

Třídy `queue<>` a `stack<>` místo toho nabízejí konstruktor, kterému lze zadat jiný kontejner; obsah nově vytvořeného kontejneru bude opět kopií obsahu předaného kontejneru.

Kontejnery mohou samozřejmě mít ještě další konstruktory – ty se ale již liší typ od typu.

1103 Jak používat kontejnery ze STL: metody



začátečník

Také jména metod se řídí u většiny kontejnerů podobnou logikou. Podívejme se alespoň na některé z nich.

Metody pro vkládání prvků se jmenují `push()`; má-li smysl vkládání na počátek, resp. na konec, jmenují se `push_front()`, resp. `push_back()`.

Je-li možný efektivní náhodný přístup k uloženým hodnotám, je pro daný kontejner přetížen operátor indexování a je definována metoda `at()`.

Má-li smysl odstranění prvního, resp. posledního prvku z kontejneru, slouží k tomu účelu metody `pop_front()`, resp. `pop_back()`. Ke zjištění hodnoty prvního, resp. posledního prvku slouží metody `front()`, resp. `back()`.

Funkce `empty()` testuje, zda je kontejner prázdný, funkce `size()` zjistí počet prvků (údajů) v kontejneru a funkce `capacity()` zjistí, kolik prvků do něj lze uložit, aniž by bylo třeba alokovat novou paměť.

Funkce `begin()`, resp. `end()` vracejí iterátor ukazující *na první*, resp. *za poslední* prvek kontejneru.

Metoda `clear()` odstraní všechny prvky z kontejneru (vyprázdní ho) a metoda `erase()` odstraní zadaný prvek nebo rozmezí prvků.

Metoda `swap()` prohodí obsah daného kontejneru s jiným kontejnerem téhož typu.

1104 Jak používat kontejnery z STL: zveřejňované datové typy



začátečník

Každý kontejner povinně zveřejňuje řadu datových typů, a to jako veřejně přístupné deklarace `typedef` uvedené v šabloně třídy kontejneru; jedná se především o typy odvozené od typu ukládaných hodnot.

Typ `value_type` představuje typ ukládaných hodnot. Ve třídách `map<>` a `multimap<>` najdeme ještě typ `key_type` představující typ klíče. Typy `pointer`, resp. `reference` představují typ ukazatele, resp. reference na typ uložených hodnot. To znamená, že máme-li např. kontejner typu `vecor<int>`, je `vecor<int>::pointer` typ `int*` a typ `vecor<int>::reference` je typ `int&`.

Typy `const_pointer`, resp. `const_reference` představují konstantní ukazatel, resp. konstantní na typ uložených hodnot.

Typ `iterator` představuje typ iterátoru použitelného k procházení daného kontejneru a ke změnám jeho hodnot. Typ `reverse_iterator` představuje reverzní iterátor, tedy iterátor určený k procházení kontejneru od konce a ke změnám hodnot. Typy `const_iterator` a `const_reverse_iterator` představují konstantní iterátory, tedy iterátory, které umožňují číst konstantní hodnoty uložené v kontejneru.

1105 Typy zveřejňované frontou a zásobníkem



pokročilý

Třídy `stack<>`, `queue<>` a `priority_queue<>` jsou definovány jako adaptéry, tedy třídy, které obalují jinou třídu, využívají jejích služeb a poskytují pouze potřebné rozhraní. Jako „podkladové“ třídy můžeme využít libovolnou posloupnost, zpravidla se ale užívá dvoustranná fronta (`deque`), která je také definována jako implicitní hodnota druhého parametru šablony v případě zásobníku a „obyčejné“ fronty.

Tyto adaptéry zveřejňují pouze typ ukládaných hodnot (typ `value_type`) a typ podkladového kontejneru (typ `container_type`). Ostatní typy musíme získat prostřednictvím podkladového typu.

1106 Co jsou to iterátory



Chceme-li procházet kontejner nebo jeho část, použijeme iterátor. O iterátorech jsme podrobněji hovořili v tipu 832; zde si jen připomeneme, že iterátor je objekt implementující stejnojmenný návrhový vzor, který umožňuje oddělit kontejner od algoritmu, který zpracovává jeho prvky.

Na iterátor se můžeme dívat jako na zobecnění ukazatele na prvek pole. Umožňuje přejít na následující prvek, získat hodnotu uloženou v prvku, změnit ji apod.

1107 Proč máme různé kategorie iterátorů



STL rozděluje iterátory do několika kategorií. Podívejme se, proč to dělá.

Je asi jasné, že nejširší možnosti jako iterátor nabízí ukazatel na prvek pole; umožňuje totiž náhodný přístup. (Ukazatel na prvek pole lze opravdu využít jako iterátor.)

Vezmeme-li místo pole obousměrně zřetězený seznam, zjistíme, že implementovat náhodný přístup je neefektivní; na obousměrném seznamu je snadné přejít na předchozí, resp. na následující prvek, ale přejít na náhodně zvolený prvek znamená dojít tak v jednotlivých krocích od počátku nebo od konce.

Vezmeme-li jednosměrně zřetězený seznam, zjistíme, že je neefektivní chtít se vrátit na předchozí prvek.

Vedle toho se setkáme s kontejnery, jejichž obsah se může měnit, protože s ním pracuje v pozadí i jiná část programu; to se týká např. vyrovnávacích pamětí datových proudů, jejichž obsah se může při opakovaném čtení lišit.

1108 Rozdělení iterátorů v STL



Různé algoritmy z STL pracující s kontejnery požadují různé druhy iterátorů. Přitom platí, že na místě, kde je požadován iterátor jisté kategorie, lze použít i iterátor kterékoli vyšší kategorie.

Nejnižší kategorii tvoří *vstupní* a *výstupní* iterátory (input, output iterator), které se používají např. pro práci s vyrovnávacími paměťmi datových proudů.

Vyšší kategorii tvoří dopředné iterátory (forward iterator). Takovýto iterátor by šlo používat např. u jednosměrně zřetězeného seznamu; STL však v současné době neobsahuje kontejner, který by poskytoval iterátor této kategorie.

Vyšší kategorii představují obousměrné iterátory (bidirectional iterator). Takovýto iterátor se hodí např. pro práci s obousměrným spojovým seznamem (třídou `list<>`).

Nejvyšší kategorii představují iterátory s náhodným přístupem (random access iterator); to jsou např. ukazatele na prvky polí nebo iterátory tříd `vector<>` nebo `deque<>`.

1109 Dereferencování iterátorů



Pro všechny druhy iterátorů je definována operace `*`, jež vrací hodnotu uloženou v kontejneru na místě, na které iterátor ukazuje. Tomu se říká *dereferencování iterátoru* (podobně jako mluvíme o dereferencování ukazatele).

Při práci s úseky polí je obvyklé udávat ukazatel *na první* prvek a ukazatel *za poslední* prvek pole. Také pro iterátory jsou definovány zvláštní hodnoty, které se chovají, jako kdyby ukazovaly na první prvek za kontejnerem. Tyto hodnoty nejsou dereferencovatelné.

1110 Vlastnosti vstupních iterátorů



Vstupní iterátory stejného typu lze porovnávat pomocí operátorů `==` a `!=`. Dva iterátory si jsou rovny, ukazují-li na týž prvek téhož kontejneru. Výstupní operátory lze přiřazovat; po přiřazení jsou si oba iterátory rovny.

Operace `++` znamená přechod na další prvek; je k dispozici jak prefixová, tak i postfixová verze. Pro vstupní iterátory neplyne ze vztahu `a == b` vztah `++a == ++b`.

Operátor `*` vrací hodnotu, na kterou iterátor ukazuje. Podobně lze použít i operátor `->`. Iterátor ovšem musí ukazovat na dereferencovatelnou hodnotu.

Algoritmus používající vstupní iterátor by neměl nikdy procházet totéž místo dvakrát.

1111 Vlastnosti výstupních iterátorů



Pro výstupní iterátory nemusí být definovány relace `==` a `!=`. Operace `++` znamená přechod na další prvek; je k dispozici jak prefixová, tak i postfixová verze.

Operátor `*` vrací hodnotu, na kterou iterátor ukazuje. V případě výstupního iterátoru lze tento operátor použít pouze na levé straně přiřazení. Iterátor ovšem musí ukazovat na dereferencovatelnou hodnotu.

Pro vstupní iterátory je definován operátor `++`, který způsobí přechod na následující prvek kontejneru. Pro vstupní iterátory neplyne ze vztahu `a == b` vztah `++a == ++b`.

1112 Vlastnosti dopředných iterátorů



Dopředné iterátory stejného typu lze porovnávat pomocí operátorů `==` a `!=`. Dva iterátory si jsou rovny, ukazují-li na týž prvek téhož kontejneru. Dopředné iterátory operátory lze přiřazovat; po přiřazení jsou si oba iterátory rovny.

Operace `++` znamená přechod na další prvek; je k dispozici jak prefixová, tak i postfixová verze. Pro dopředné iterátory plyne – na rozdíl od vstupních a výstupních iterátorů – ze vztahu `a == b` vztah `++a == ++b`.

Operátor `*` vrací hodnotu, na kterou iterátor ukazuje; iterátor ovšem musí ukazovat na dereferencovatelnou hodnotu. Podobně lze použít i operátor `->`.

Dopředné iterátory dovolují, aby algoritmus kontejner *opakovaně* procházel ve směru odzadu dopředu.

1113 Vlastnosti obousměrných iterátorů



Pro obousměrné iterátory jsou definovány tytéž operace jako pro dopředné iterátory (tip 1112) a navíc operace `--`, jež znamená přechod na předcházející prvek.

Obousměrné iterátory dovolují, aby algoritmus kontejner opakovaně procházel, a to ve směru dopředu i dozadu.

1114 Vlastnosti iterátorů pro náhodný přístup



Pro iterátor pro náhodný přístup jsou definovány stejné operace jako pro obousměrné iterátory a navíc operace přičtení a odečtení celého čísla, tedy $i + n$, $i - n$, $i += n$ a $i -= n$. Jejich význam je podobný jako v aritmetice ukazatelů: např. $i + n$ je iterátor, který ukazuje o n prvků dál, než kam ukazoval iterátor i .

Také rozdíl dvou iterátorů s náhodným přístupem má podobný význam jako v adresové aritmetice: Jsou-li i a j dva iterátory ukazující do téhož kontejneru, udává rozdíl $i - j$ vzdálenost mezi prvky, na které ukazují (ukazují-li i a j na po sobě následující prvky, je jejich rozdíl 1).

Dále jsou k dispozici relace $i > j$, $i < j$, $i \leq j$, $i \geq j$. Jsou-li i a j iterátory ukazující do téhož kontejneru, platí např. relace $i > j$, jestliže i ukazuje na prvek, který je v kontejneru za prvkem, na který ukazuje j .

Pro iterátory s náhodným přístupem je také k dispozici operace indexování. Výraz $i[n]$ znamená totéž co $*(i + n)$.

1115 Platnost iterátorů



Jestliže máme ukazatel na prvek seznamu a tento prvek ze seznamu odstraníme, ztratí tento ukazatel platnost. Máme-li ukazatel na poslední prvek pole a přidáme-li doprostřed pole nový prvek, nebude náš ukazatel již ukazovat na poslední prvek. Máme-li ukazatel na prvek dynamicky alokovaného pole a my toto pole zvětšíme voláním funkce `realloc()`, může se umístění tohoto pole v paměti změnit a ukazatel na jeho prvek ztratí platnost.

Na iterátory se můžeme dívat jako na objektové zapouzdření ukazatelů na prvky kontejnerů. To znamená, že určité operace, jako je změna kapacity, přidání nebo odebrání prvku atd., mohou změnit platnost iterátorů. Pokus o operaci s neplatným iterátorem má typicky za následek běhovou chybu programu.

1116 Co lze ukládat do kontejnerů z STL



Datový typ T objektů, které lze ukládat do kontejnerů z STL, musí mít kopírovací konstruktor, pro který platí:

Je-li x instance typu T , musí příkazem $T(x)$ vzniknout instance, která je ekvivalentní původní instanci x . Podobně je-li y instance typu `const T`, musí příkazem $T(y)$ vzniknout konstantní instance, která je ekvivalentní původní instanci y .

Třída T musí mít přístupný destruktork a výrazy $\&x$ a $\&y$, kde x je instance typu T a y je konstantní instance typu T , musí představovat adresu x , resp. y .

Vedle toho pro ně musí být definován operátor přiřazení, který vrací odkaz na levou stranu po přiřazení a jehož výsledkem je, že objekt na pravé straně je ekvivalentní objektu na levé straně.



Poznámka: I když tyto požadavky vypadají na první pohled komplikovaně, většina běžných datových typů je splňuje. Splňují je samozřejmě i základní typy, ukazatele, výčtové typy apod.

Na druhé straně těmto požadavkům nevyhovují např. automatické ukazatele (třída `auto_ptr<>` deklarovaná v hlavičkovém souboru `<memory>`) nebo třídy datových proudů.

1117 Jak se ukládají data do kontejneru



Do kontejneru se ukládají vždy kopie předaných dat vytvořené pomocí kopírovacího konstruktorku nebo kopírovacího přiřazovacího operátoru.

1118 Vkládáme data do vektoru



Máme instanci třídy `vector<string>` a potřebujeme do ní vložit jednotlivé řádky přečtené z proudu `kam`. K tomu použijeme příkaz (srovnej tip 1075)

```
vector<string> soubor;
string lajna;
while(getline(kam, lajna))
{
    soubor.push_back(lajna);
}
```

1119 Počáteční kapacita vektoru



Na počátku nemá vektor přidělenou žádnou kapacitu. Při vložení prvního prvku metodou `push_back()` se vyhradí paměť pro 1 prvek. Při vložení dalšího se paměť zvětší (typicky zdvojnásobí, ale to není podmínkou). Vždy když se zaplní celá dosavadní kapacita, vyhradí se nová, větší paměť, stávající prvky se do ní překopírují a původní paměť se uvolní.

To je poměrně neefektivní, a proto známe-li alespoň přibližně počet prvků, které v kontejneru budou, můžeme hned po vytvoření vyhradit potřebnou kapacitu voláním metody `reserve()`.

```
vector<string> lajna;  
lajna.reserve(100);
```

Kapacitu lze měnit i kdykoli později. Tato metoda ovšem neumožňuje zmenšit kapacitu vektoru; je-li již alokováno více paměti, než prikazuje `reserve()`, nestane se nic.

Změna kapacity může způsobit, že iterátory ztratí platnost.

1120 Výpis všech prvků vektoru



začátečník

Máme vektor řetězců `vek` a chceme všechny řetězce v něm uložené vypsát. To uděláme následujícím cyklem:

```
for(vector<string>::iterator it = vek.begin();  
    it != vek.end(); it++)  
    cout << *it << endl;
```

Iterátor zde slouží jako parametr cyklu. Protože vektor má iterátor s náhodným přístupem, mohli bychom podmínku opakování napsat ve tvaru `it < vek.end()`; v případě iterátorů však raději používáme operátor `!=`, neboť ten lze použít u všech druhů iterátorů, nejen u iterátoru pro náhodný přístup.

Nechceme-li použít iterátory, můžeme použít indexování, neboť ve třídě `vector<>` je k dispozici přetížený operátor indexování. Prvky vektoru jsou indexovány od 0 a jejich počet nám řekne metoda `size()`:

```
for(int i = 0; i < vek.size(); i++)  
    cout << vek[i] << endl;
```

1121 Přístup k prvkům vektoru



začátečník

Máme opět vektor řetězců `vek`. V předchozím tipu jsme viděli dva způsoby přístupu k prvkům vektoru: Pomocí dereferencovaného iterátoru a pomocí přetíženého operátoru indexování. Zde si ukážeme ještě třetí, pomocí metody `at()`.

Chceme změnit hodnotu prvku, na který ukazuje iterátor `it`, a uložit do něj řetězec "haha"; napíšeme tedy

```
*it = "haha";
```

Víme-li, že jde o 3. prvek, napíšeme

```
vek[2] = "haha";    // Indexujeme od 0!
```

Metoda `at()` funguje podobně jako operátor indexování, jestliže však použijeme index, který leží mimo kapacitu vektoru, vyvolá výjimku typu `out_of_range`.

```
vek.at(3) = "haha";
```

Pozor, ani metodu `at()`, ani operátor indexování nelze použít k připojení nových prvků na konec vektoru.

1122 Odstranění prvku z vektoru



začátečník

Chceme odstranit prvek, na který ukazuje iterátor `it`. K tomu nám poslouží metoda `erase()`:

```
vek.erase(it);
```

Chceme-li odstranit prvky ležící v rozmezí od `it1` do `it2`, použijeme metodu `erase()` se dvěma parametry,

```
vek.erase(it1, it2);
```

Přitom `it2` ukazuje na první prvek *za* odstraňovaným úsekem, takže prvek, na který `it2` ukazuje, už nebude odstraněn.



Upozornění: Odstraněním prvku z vektoru se nezmenší jeho kapacita!

1123 Chceme jen tolik paměti, kolik potřebujeme



znalec

Vektor si umí paměť brát, ale neumí ji vrátet. I když budeme do vektoru pouze přidávat prvky, může se stát, že bude zabírat mnohem více paměti, než kolik potřebuje. Jedna z obvyklých strategií totiž je, že se pokaždé, když se při vkládání údajů vyčerpá přidělená paměť, její množství zdvojnásobí. To znamená, že se může stát, že náš vektor obsahuje 1025 prvků, ale jeho kapacita je 2048 prvků – a my víme, že další prvky už nepřibudou.

Máme tedy např. vektor řetězců `vek` a chtěli bychom, aby zabíral jen tolik paměti, kolik je nezbytně nutné. V takovém případě si lze vypomoci následujícím trikem:

```
vek.swap(vector<string>(vek));
```

Pomocí kopírovacího konstrukturu vytvoříme nepojmenovanou pomocnou instanci. Kopírovací konstruktorem ovšem pro kopii vyhradí jen nezbytné množství paměti. Pak pomocí metody `swap()` prohodíme obsahy obou vektorů. Nyní pomocná proměnná obsahuje zbytečnou paměť, ale `vek` obsahuje jen nezbytnou.

Po vyhodnocení tohoto výrazu pomocná proměnná zanikne a její paměť se uvolní.

1124 Tabulka proměnných: mapa



začátečník

Píšeme program, v němž si uživatel definuje reálné proměnné, např. pomocí grafického uživatelského rozhraní, a náš program je zpracovává. Každá proměnná má jméno (znakový řetězec) a hodnotu (reálné číslo), takže pro ukládání proměnných použijeme

asociativní kontejner. A protože klíče jsou jednoznačné, neboť v našem programu nemohou mít dvě proměnné stejné jméno, zvolíme mapu.

```
map<string, double> tabulka_prom;
```



Poznámka: Anglické slovo *mapping* se používá také ve významu „zobrazení“ známém z matematiky. Mapa v STL je asociativní kontejner, který určuje jednoznačné přiřazení hodnot jménům – tedy implementuje opravdu zobrazení v matematickém smyslu.

1125 Vytvoření nové proměnné: přidání prvku do mapy



začátečník

V předchozím tipu 1124 jsme si vytvořili tabulku proměnných jako mapu dvojic klíč (znakový řetězec) a hodnoty typu `double`; tuto instanci jsme pojmenovali `tabulka_prom`.

Pro přístup k datům lze použít přetížený operátor indexování, přičemž jako index použijeme jméno proměnné. Napíšeme-li

```
tabulka_prom["x"] = 98;
```

stane se toto:

- Jestliže proměnná `x` (tedy prvek mapy s klíčem `"x"`) již existuje, změní se její hodnota na 98.
- Jestliže prvek mapy s klíčem `"x"` ještě neexistuje, vytvoří se a uloží se do něj zadaná hodnota.

1126 Přístup k prvkům mapy



začátečník

Máme tabulku proměnných jako mapu dvojic klíč (znakový řetězec) a hodnoty typu `double`; tuto proměnnou jsme pojmenovali `tabulka_prom`. Napíšeme-li

```
double pom = tabulka_prom["z"];
```

stane se toto:

- Jestliže proměnná `z` (tedy prvek mapy s klíčem `"z"`) již existuje, vrátí operátor indexování její hodnotu (a ta se uloží do proměnné `pom`).
- Jestliže prvek mapy s klíčem `"z"` ještě neexistuje, vytvoří se, uloží se do něj implicitní hodnota tohoto typu (v našem případě hodnota vytvořená zápisem `double()`, tedy 0) a tuto hodnotu operátor indexování také vrátí.

1127 Výpis obsahu mapy



začátečník

I když mapa nepatří mezi posloupnosti, lze ji procházet pomocí iterátoru. Ovšem dereferencujeme-li iterátor ukazující na nějaký prvek mapy, dostaneme dvojici klíč – hodnota, a to jako instanci struktury `pair<>`. Její složky se jmenují výstižně `first` (první)

a second (druhý). Máme-li instanci třídy `map<string, double>`, která se jmenuje `tabulka`, vypíšeme její obsah příkazem

```
for(map<string, double>::iterator it = tabulka.begin();
    it != tabulka.end(); it++)
    cout << it -> first << " " << it -> second << endl;
```

1128 Hodnota na vrcholu zásobníku



začátečník

Zásobník je datová struktura, která svým chováním připomíná zásobník automatické palné zbraně, nebo třeba sloupec knih na stole: je to posloupnost, v níž lze přidávat nové prvky jen na jeden konec (vrchol zásobníku) a jen z tohoto konce je lze také odebírat. Proto bývá také označován anglickou zkratkou LIFO – Last In, First Out, tedy poslední vložený prvek musíme vyjmout jako první.

Máme zásobník celých čísel

```
stack<int> zc;
```

Jako podkladový kontejner jsme použili implicitní dvoustrannou frontu. Chceme-li do něj vložit novou hodnotu, uloženou v proměnné `a`, napíšeme

```
zc.push(99);
```

Chceme-li odstranit hodnotu z vrcholu zásobníku, napíšeme

```
zc.pop();
```

Pozor: tato metoda nevrací odstraněnou hodnotu! Chceme-li zjistit hodnotu na vrcholu zásobníku, použijeme metodu `top()`:

```
int pom = zc.top();
```

Zjištění hodnoty na vrcholu zásobníku a odstranění hodnoty na vrcholu zásobníku obstarávají dvě různé metody.

1129 Prvek v čele fronty



začátečník

Fronta je datová struktura, která se chová podobně jako fronta při čekání na pokladnu v obchodě: Prvky se v ní řadí v pořadí, ve kterém do ní byly vloženy, a jako první musí být vyjmut prvek, který jsme do ní jako první vložili. Proto bývá označována anglickou zkratkou FIFO – First In, First Out neboli „kdo dřív přijde, ten dřív mele“.

Zacházení s frontou je velice podobné jako zacházení se zásobníkem. Vytvoříme-li si frontu celých čísel založenou na vektoru,

```
queue<int, vector<int> > fc;
```

můžeme do ní vložit novou hodnotu, uloženou v proměnné `a`, příkazem

```
fc.push(a);
```

Chceme-li odstranit hodnotu z čela fronty, napíšeme

```
zc.pop();
```

Pozor, tato metoda nevrací odstraněnou hodnotu! Chceme-li zjistit hodnotu na čele fronty, použijeme metodu `front()`:

```
int pom = zc.front();
```

Zjištění hodnoty na čele fronty a odstranění hodnoty z čela fronty obstarávají dvě různé metody.

1130 Proč dvě metody?



V klasických učebnicích programování, z nichž jsem se kdysi učil tohle nádherné umění, a kterých jsem později také několik napsal, se v souvislosti s frontou nebo zásobníkem téměř vždy objevovala metoda `pop()` (nebo třeba `vyjmi()`), která odstranila prvek z čela fronty nebo z vrcholu zásobníku a vrátila jeho hodnotu. Proč je v STL tato operace rozdělena na dvě?

Odpovědi jsou dvě a obě si zasluhují pozornost.

1. *Princip jediné zodpovědnosti* (viz též tip 821). Jde o dvě různé operace, a proto je rozumné, aby je prováděly dvě různé metody.
2. *Bezpečnost vzhledem k výjimkám*. Operace „odstraň prvek a vrať jeho hodnotu“ vyžaduje po řadě zjistit hodnotu uloženou v prvku, překopírovat ji do pomocné proměnné, odstranit tento prvek a vrátit hodnotu z pomocné proměnné. Kdyby přitom vznikla výjimka (např. v kopírovacím konstruktoru uložené hodnoty), znamenalo by to, že prvek byl odstraněn, ale jeho hodnota nebyla vrácena – je tedy ztracena. Rozdělení operací umožňuje předejít takovéto ztrátě dat: Jestliže vznikne výjimka při zjišťování hodnoty, zůstane její kopie v kontejneru, a operace `pop()` bude odstraňovat už jen hodnotu, kterou nepotřebujeme.