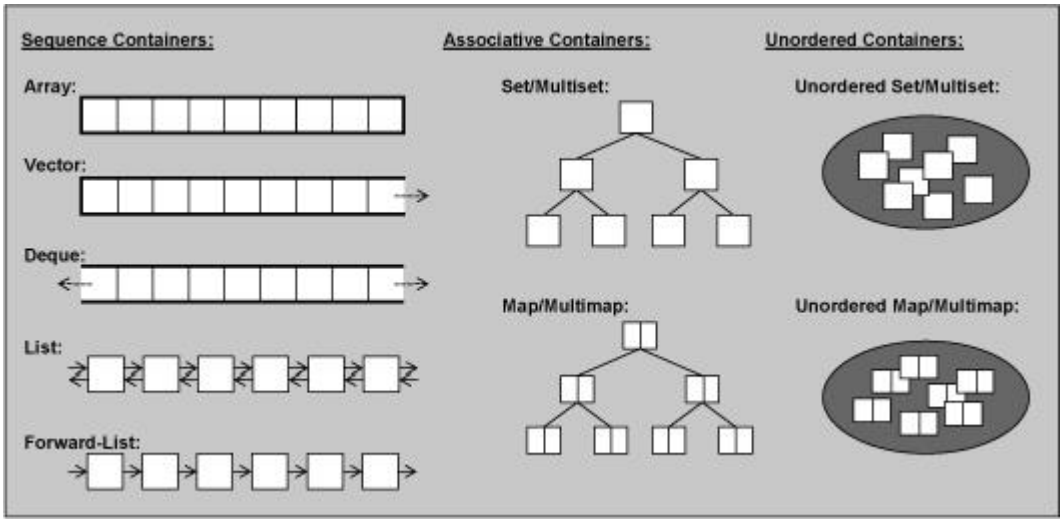
Knihovna standardních šablon (STL - Standard Template Library)

Je součást standardu jazyka C++ a obsahuje:

- Datové struktury pro práci s kolekcemi datových prvků - kontejnery datových prvků



- Iterátory – umožňují postupně projít všechny datové prvky uložené v kolekci
- Algoritmy pro práci s datovými prvky uloženými v kolekci – třídění, vyhledávání a další

Sekvenční kontejnery

vector

Je dynamické pole. Umožňuje přímý přístup přes index. Vložení prvku na konec nebo odebrání z konce je rychlé. Přidání prvku na jiné místo, než je konec, nebo odebrání prvku z jiného místa než konec vyžaduje posunutí prvků, které budou za přidávaným prvkem nebo které jsou za odebíraným prvkem.

Některé konstruktory

`vector<typ_prvku> v`	Vytvoří prázdný vektor ***v***.
`vector<typ_prvku> v1(v2)`	Vytvoří vektor ***v1***, který je kopií vektoru ***v2***. Prvky ve vektoru ***v2*** musí mít stejný datový typ.
`~vector()`	Destruktor zruší všechny prvky uložené ve vektoru a uvolní paměť, kterou vektor alokoval pro uložení prvků.

Příklad.

```
#include <vector>
vector<int> v;
```

Vektor na začátku nemá přidělenou žádnou paměť. Tu si přiděluje postupně, jak jsou do něho ukládány prvky. Přitom provádí realokaci, pokud je potřeba stávající paměť zvětšit. Realokaci lze předejít funkcí **`reserve`**, ve které uvedeme, kolik prvků bude do vektoru uloženo, čímž si vektor při jejím vykonání zajistí potřebné množství paměti.

Příklad. Vektor pro uložení 100 čísel typu **double**.

```
vector<double> d;
d.reserve(100);
```

Pro kolik prvků je momentálně ve vektoru rezervovaná paměť, lze zjistit funkcí **capacity**.

Příklad.

```
vector<int> v;
v.reserve(50);
cout << v.capacity();    // 50
v.reserve(100);
cout << v.capacity();    // 100
v.reserve(20);
cout << v.capacity();    // 100
```

Stávající kapacitu vektoru lze zvětšit, nelze ji zmenšit.

Operace neměnící obsah vektoru

size()	Počet uložených prvků.
empty()	Zda prázdný, tj. zda platí **size()==0** .
max_size()	Maximální počet prvků, který lze do vektoru uložit.
capacity()	Aktuální kapacita
reserve(n)	Zvýšení kapacity na ***n*** prvků
v1 == v2	Srovnání, zda dva vektory ***v1*** a ***v2*** vektory jsou stejné.
v1 != v2	Srovnání, zda vektory jsou různé.
v1 < v2	Vyžaduje, aby datový typ prvků měl definovánu operaci srovnání **<.**
v1 > v2	Vyžaduje, aby datový typ prvků měl definovánu operaci srovnání **>.**
v1 <= v2	Vyžaduje, aby datový typ prvků měl definovánu operaci srovnání **<=.**
v1 >= v2	Vyžaduje, aby datový typ prvků měl definovánu operaci srovnání **>=.**

U operací srovnání lze porovnávat jen vektory, které mají stejný datový typ prvků. Srovnání na rovnost vyžaduje, aby mezi prvky vektorů byla definována operace srovnání ==. Srovnání dalších operátorů vyžaduje, aby byla definována rovněž operace srovnání <. Zbývající operace jsou realizovány těmito dvěma operátory:

operátor	realizace
a!=b	**!(a==b)**
a>b	**b<a**
a<=b	**!(b<a)**
a>=b	**!(a<b)**

Operace přiřazení

`v1 = v2`	Do vektoru ***v1*** zkopíruje obsah vektoru ***v2*** **.**
`assign(n,prvek)`	Do vektoru uloží ***n*** × ***prvek*** .
`assign(zac,kon)`	Vektoru přiřadí prvky z jiného vektoru, které jsou v něm obsaženy počínaje počáteční pozicí ***zac***, dokud nedosáhne koncovou pozici ***kon***.
`v1.swap(v2)`	Zamění obsah vektorů ***v1*** a ***v2*** .
`swap(v1,v2)`	Zamění obsah vektorů.

U operace **`swap`** musí oba vektory mít stejný datový typ prvků. Obsahy vektorů se vymění včetně jejich kapacit.

Přímý přístup k prvkům

`v[idx]`	Vrací referenci na prvek s daným indexem. Nekontroluje, zda hodnota indexu je v rozsahu vektoru.
`at(idx)`	Vrací referenci na prvek s daným indexem. Je-li index mimo rozsah, generuje výjimku `out_of_range`.
`front()`	Vrací referenci na první prvek. Nekontroluje, zda takový prvek existuje.
`back()`	Vrací referenci na poslední prvek. Nekontroluje, zda takový prvek existuje.

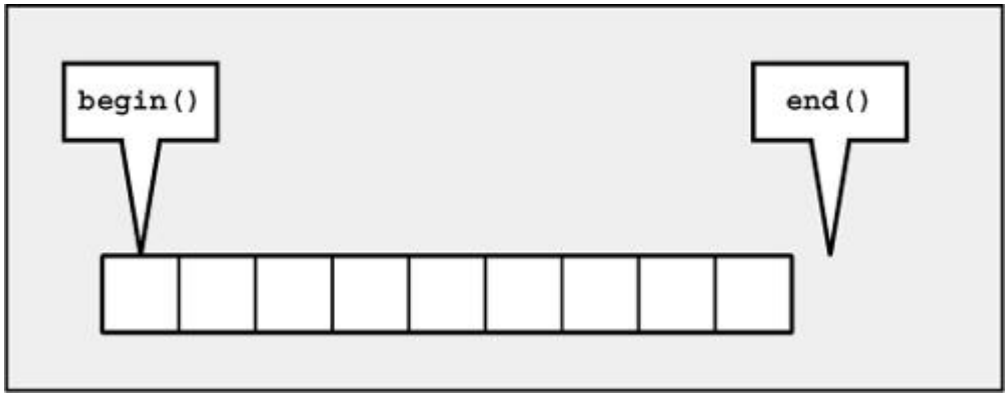
Příklad.

```
vector<int> v;
try { cout << v.at(1); }
catch (out_of_range) { cout << "Mimo rozsah"; }
if (v.size()>1) v[1]=5;
if (!v.empty()) cout << v.front();
```

Iterátory

`begin()`	Vrací ukazatel na první prvek (pro nastavení iterátoru na začátek).
`end()`	Vrací ukazatel ukazující za poslední prvek (pro zjištění, zda iterátor už není mimo vektor).
`rbegin()`	Vrací ukazatel na poslední prvek (pro nastavení reverzního iterátoru na začátek průchodu).
`rend()`	Vrací ukazatel ukazující před první prvek (pro zjištění, zda reverzní iterátor není už mimo vektor).
`cbegin()`	Vrací ukazatel na konstantní hodnotu ukazující na první prvek (pro nastavení `const` iterátoru na začátek).
`cend()`	Vrací ukazatel na konstantní hodnotu ukazující za poslední prvek (pro zjištění,

	zda `const` iterátor už není mimo vektor).
`crbegin()`	Vrací ukazatel na poslední prvek (pro nastavení reverzního `const` iterátoru na začátek průchodu).
`crend()`	Vrací ukazatel ukazující před první prvek (pro zjištění, zda reverzní `const` iterátor není už mimo vektor).



Vložení a odebrání prvku

`insert(poz,prvek)`	Vloží na pozici zadanou iterátorem ***poz*** kopii ***prvku*** a vrátí jeho pozici ve vektoru.
`insert(poz,n,prvek)`	Vloží na pozici zadanou iterátorem ***poz*** ***n***× kopii ***prvku***. Nevrací žádnou hodnotu.
`insert(poz,zac,kon)`	Vloží na pozici zadanou iterátorem ***poz*** kopii prvků od pozice určené iterátorem ***zac***, dokud se nedosáhne pozice určené iterátorem ***kon***. Nevrací žádnou hodnotu.
`emplace(poz,hodnoty)`	Sestaví prvek ze zadaných ***hodnot*** a umístí ho na pozici zadanou iterátorem ***poz***. Nevrací žádnou hodnotu.
`push_back(prvek)`	Přidá ***prvek*** na konec. Nevrací žádnou hodnotu.
`emplace_back(hodnoty)`	Sestaví prvek ze zadaných ***hodnot*** a umístí ho na konec. Nevrací žádnou hodnotu.
`erase(poz)`	Odstraní prvek, který je na pozici ***poz***. Vrací pozici následujícího prvku.
`erase(zac,kon)`	Odstraní prvky od pozice ***zac***, dokud nedosáhne pozice ***kon***. Vrací pozici následujícího prvku.
`pop_back()`	Odstraní poslední prvek. Nevrací žádnou hodnotu.
`resize(n)`	Změní počet prvků ve vektoru na hodnotu ***n***. Je-li ***n*** větší, než je současný počet prvků vektoru (hodnota **`size()`**), doplní se prvky s implicitní hodnotou (hodnota je dána

	implicitním konstruktorem datového typu prvků). Nevrací žádnou hodnotu.
`resize(n,prvek)`	Změní počet prvků ve vektoru na hodnotu ***n***. Je-li ***n*** větší, než je současný počet prvků vektoru, doplní se kopií zadaného ***prvku***. Nevrací žádnou hodnotu.
`clear()`	Odstraní všechny prvky ve vektoru, vektor bude nyní prázdný. Nevrací žádnou hodnotu.

Příklad.

```
vector<char> v;
v.assign(2,'+');                    // ++
v.push_back('C');                   // ++C
v.insert(v.begin(), 1, 'c');        // c++C
v[0]='C';                           // C++C
v.insert(v.begin(), v.begin(), v.begin()+2);        // C+C++C
v.insert(v.begin()+1, v.begin()+3, v.begin()+5);    // C+++C++C
v.pop_back();                                       // C+++C++
v.erase(v.begin()+1, v.begin()+3);  // C+C++
v.erase(v.begin()+1);               // CC++
v.erase(v.begin());                 // C++
```

```
struct Zlomek { unsigned cit,jmen;
  Zlomek(unsigned c,unsigned j) { cit=c,jmen=j; }
  void vypis() const { cout << cit << '/' << jmen << endl; }
};
vector<Zlomek> z;
z.emplace_back(2,3);    // místo z.push_back(Zlomek(2,3));
z.emplace_back(1,2);
z.emplace_back(2,5);
```

Sekvenční iterátory

Jsou ukazatelé na prvky a umožňují sekvenční průchod směrem dozadu nebo směrem dopředu.

Příklad.

```
for (vector<Zlomek>::iterator it=z.begin(); it!=z.end(); ++it)
                                                  it->vypis();
```

2/3
1/2
2/5

```
for (vector<Zlomek>::const_reverse_iterator it=z.crbegin();
                                            it!=z.crend(); ++it)
                                                  it->vypis();
```

2/5
1/2
2/3

Jednodušší zápis:

```
for (auto it=z.begin(); it!=z.end(); ++it) it->vypis();
for (auto it=z.crbegin(); it!=z.crend(); ++it) it->vypis();
```

Průchod všemi prvky

`for_each(zac,kon,f)`	Prochází prvky od pozice ***zac***, dokud nedosáhne pozici ***kon***. Pro každý pocházený prvek volá funkci ***f***, která má jeden parametr a tím je právě procházený prvek (předaný hodnotou nebo referencí). Na typu návratové hodnoty funkce nezáleží (návratová hodnota není využita).

Příklad.

```
#include <algorithm>
void vypis(const Zlomek &z)
{
  z.vypis();
}
for_each(z.begin(), z.end(), vypis);
```

Příklad. Využití nového formátu příkazu *for*.

```
for (const Zlomek &i: z) i.vypis();
```

Sekvenční kontejnery

Algoritmy hledání

`min_element(zac,kon)`	Najde nejmenší nebo největší prvek mezi prvky od pozice ***zac*** po pozici ***kon*** a vrátí pozici tohoto prvku. Lze zadat funkci ***mensi*** pro srovnání dvou prvků (funkce má tyto prvky jako parametry). Funkce vrací hodnotu *true*, je-li první z prvků menší než druhý. Není-li funkce zadána, pro srovnání je použit operátor srovnání **<** .
`min_element(zac,kon,mensi)`	
`max_element(zac,kon)`	
`max_element(zac,kon,mensi)`	
`find(zac,kon,hodnota)`	Hledá od pozice ***zac*** po pozici ***kon*** první prvek, který má zadanou ***hodnotu***. Je-li nalezen, vrátí jeho pozici. Není-li nalezen vrátí pozici ***kon***.

`find_if(zac,kon,f)`	Hledá od pozice ***zac*** po pozici ***kon*** první prvek, pro který funkce ***f*** vrátí hodnotu *true*. Funkce má jeden parametr – procházený prvek. Je-li prvek nalezen, vrátí jeho pozici. Není-li nalezen vrátí pozici ***kon***.

Příklad.

```
vector<string *> v;
v.push_back(new string("Petr"));
v.push_back(new string("Jana"));
v.push_back(new string("Eva"));
v.push_back(new string("Marek"));
bool mensi(const string *s1,const string *s2)
{
  return *s1<*s2;
}
vector<string *>::iterator poz;
poz = min_element(v.begin(), v.end(), mensi);
cout << **poz;          // Eva    **poz = *(*poz)
```

```
bool jeEvaNeboJana(const string *s)
{
  return *s == "Eva" || *s == "Jana";
}
poz = find_if(v.begin(), v.end(), jeEvaNeboJana);
cout << **poz;          // Jana
```

Jednodušší zápis. Místo

```
vector<string *>::iterator poz;
poz = min_element(v.begin(), v.end(), mensi);
```

lze použít

```
auto poz = min_element(v.begin(), v.end(), mensi);
```

Algoritmy srovnání

`equal(zac1,kon1,zac2)`	Zjistí, zda prvky od pozice ***zac1*** po pozici ***kon1*** jsou stejné jako prvky od pozice ***zac2***. Lze zadat funkci ***stejne*** pro srovnání dvou prvků (funkce má tyto prvky jako parametry). Funkce vrací hodnotu *true*, jsou-li prvky stejné. Není-li funkce zadána, pro srovnání je použit operátor srovnání == .
`equal(zac1,kon1,zac2,stejne)`	

Příklad.

```
vector<int> v1,v2;
```

```
v1.push_back(3);
v1.push_back(1);
v1.push_back(5);
v2.push_back(1);
v2.push_back(5);
equal(v1.begin()+1,v1.end(),v2.begin())  // true
equal(v1.begin(),v1.end(),v2.begin())    // false
```