



Výnimky, string, stringstream, STL

Pavol Marák | Programovacie techniky

OBSAH

- Výnimky
- Trieda `std::string`
- Trieda `std::stringstream`
- STL
 - Kontajnery
 - Iterátory



Výnimky

Výnimky

- C++ výnimky sú spôsob ako reagovať na výnimočné situácie počas behu programu.

Výnimky

- C++ výnimky sú spôsob ako reagovať na výnimočné situácie počas behu programu.
- Používanie výnimiek pomáha oddeliť hlavnú logiku programu od logiky, ktorá rieši vzniknuté chyby a problémy.

Výnimky

- C++ výnimky sú spôsob ako reagovať na výnimočné situácie počas behu programu.
- Používanie výnimiek pomáha oddeliť hlavnú logiku programu od logiky, ktorá rieši vzniknuté chyby a problémy.
- Pri vzniku výnimky sa riadenie toku programu odovzdáva do špeciálnej časti kódu nazývanej *exception handler*.

Try-catch konštrukcia

Try blok

- Je v ňom umiestnený sledovaný kód, v ktorom môže vzniknúť výnimka.

```
try{  
    // sledovany kod  
}
```

Try-catch konštrukcia

Try blok

- Je v ňom umiestnený sledovaný kód, v ktorom môže vzniknúť výnimka.

Catch blok (bloky)

- Predstavuje tzv. exception handler, ktorý obsluhuje konkrétny typ výnimky.

```
try{  
    // sledovany kod  
}  
catch(int e){  
    // handler pre typ int  
}  
catch(MyException& e){  
    // handler pre typ MyException  
}  
catch(...) {  
    // ostatne typy  
}
```


Throw

Výnimka vzniká (resp. je „vyhodená“) použitím klíčového slova **throw** v try bloku.


```
int main() {  
    try {  
        throw expression;  
    }  
    catch (MyException &e) {  
        // handler pre typ MyException  
    }  
    return 0;  
}
```

Throw

Výnimka vzniká (resp. je „vyhodená“) použitím klúčového slova **throw** v try bloku.

```
int main() {  
    try {  
        throw expression;  
    }  
    catch (MyException &e) {  
        // handler pre typ MyException  
    }  
    return 0;  
}
```

Copy inicializácia
výnimkového objektu
použitím *expression*



Exception handler

- Ak v try bloku vznikne výnimka, riadenie je presunuté do exception handlera (hľadá sa vhodný catch blok). Kód nasledujúci za throw v rámci try bloku nie je vykonaný.

Exception handler

- Ak v try bloku vznikne výnimka, riadenie je presunuté do exception handlera (hľadá sa vhodný catch blok). Kód nasledujúci za throw v rámci try bloku nie je vykonaný.
- Po obslúžení výnimky v catch bloku, program pokračuje po celom try-catch bloku.

Exception handler

- Ak v try bloku vznikne výnimka, riadenie je presunuté do exception handlera (hľadá sa vhodný catch blok). Kód nasledujúci za throw v rámci try bloku nie je vykonaný.
- Po obslúžení výnimky v catch bloku, program pokračuje po celom try-catch bloku.
- Ak v try bloku nevznikne výnimka, program pokračuje ďalej a všetky exception handlers (catch bloky) sú ignorované.

Stack unwinding


- Je proces odstraňovania funkcií zo zásobníka volaní funkcií za účelom hľadania funkcie, v ktorej existuje vhodný exception handler pre vzniknutú výnimku (vid' obrázok na ďalšej strane).

Stack unwinding

```
int main() {  
    try {  
        f1();  
        fnext();  
    }  
    catch (const char *e) {  
        // obsluha vyjimky  
    }  
    return 0;  
}
```

Stack unwinding

```
int main() {  
    try {  
        f1();  
        fnext();  
    }  
    catch (const char *e) {  
        // obsluha vyjimky  
    }  
    return 0;  
}  
  
void f1() {  
    f2();  
}
```



Stack unwinding

```
int main() {  
    try {  
        f1();  
        fnext();  
    }  
    catch (const char *e) {  
        // obsluha vyjimky  
    }  
    return 0;  
}  
  
void f1() {  
    f2();  
}  
  
void f2() {  
    f3();  
}
```

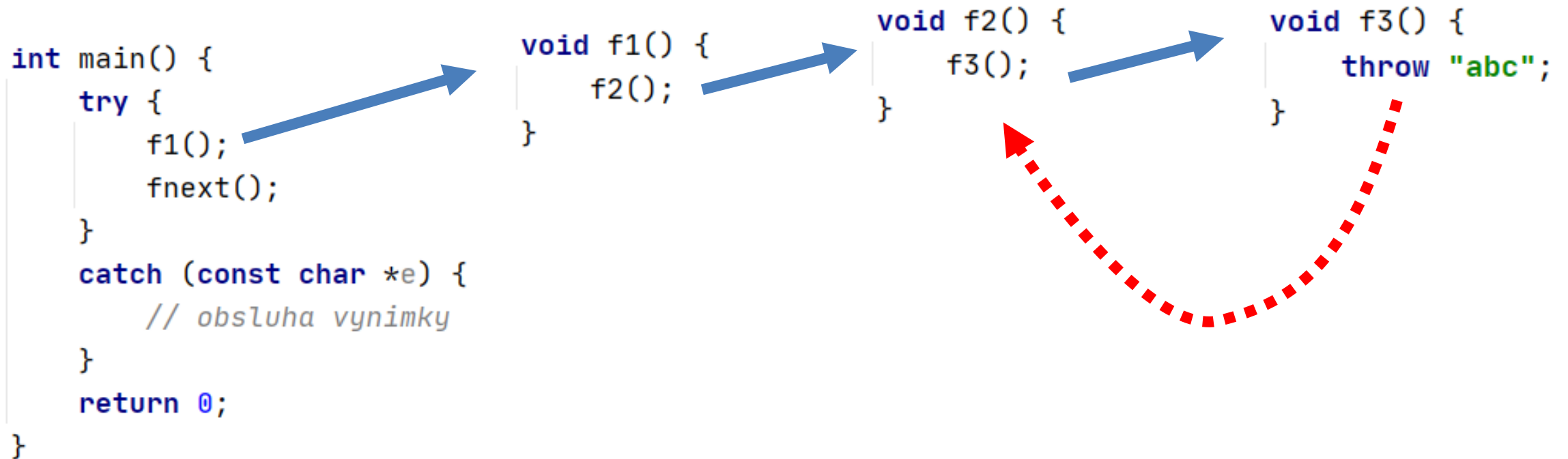
The diagram illustrates the process of stack unwinding. It shows three function definitions: `main()`, `f1()`, and `f2()`. A blue arrow points from the `f1();` line in `main()` to the `f1()` function definition. Another blue arrow points from the `f2();` line inside `f1()` to the `f2()` function definition. This visualizes the sequence of function calls and the subsequent unwinding of the stack when an exception is thrown.

Stack unwinding

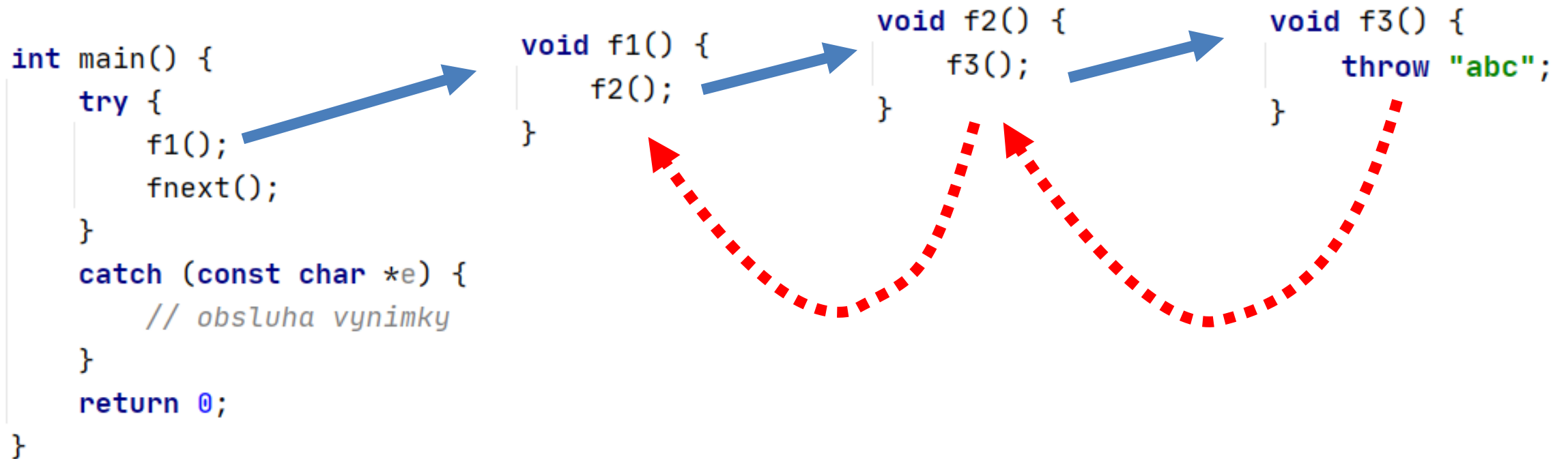
```
int main() {  
    try {  
        f1();  
        fnext();  
    }  
    catch (const char *e) {  
        // obsluha vyjimky  
    }  
    return 0;  
}  
  
void f1() {  
    f2();  
}  
  
void f2() {  
    f3();  
}  
  
void f3() {  
    throw "abc";  
}
```

The diagram illustrates the call stack during exception unwinding. It shows four function definitions: `main()`, `f1()`, `f2()`, and `f3()`. Blue arrows indicate the sequence of calls: from `main()` to `f1()`, from `f1()` to `f2()`, and from `f2()` to `f3()`. The `throw "abc";` statement in `f3()` is highlighted in green, indicating the point where an exception is thrown.

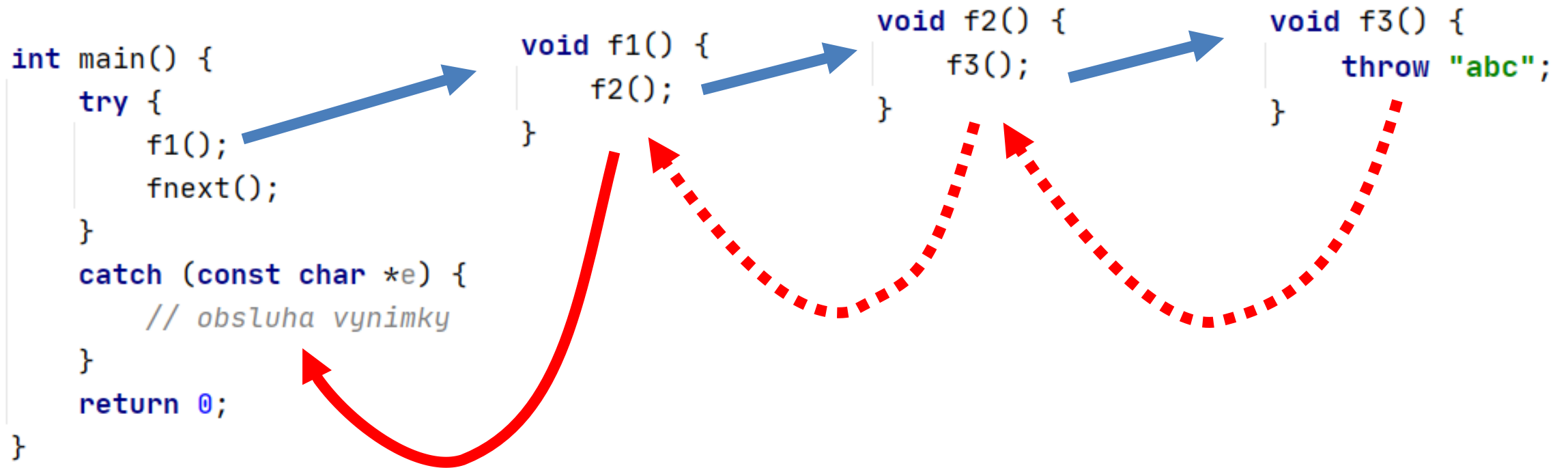
Stack unwinding



Stack unwinding



Stack unwinding



Štandardné výnimky C++

- Jazyk C++ má svoje preddefinované štandardné výnimky (vznikajú pri práci so štandardnu knižnicou).

Štandardné výnimky C++

- Jazyk C++ má svoje preddefinované štandardné výnimky (vznikajú pri práci so štandardnu knižnicou).
- Štandardné výnimky sú deklarované v hlavičkovom súbore `<stdexcept>`.

Štandardné výnimky C++

Standard library header `<stdexcept>`

This header is part of the [error handling](#) library.

Classes

<code>logic_error</code>	exception class to indicate violations of logical preconditions or class invariants (class)
<code>invalid_argument</code>	exception class to report invalid arguments (class)
<code>domain_error</code>	exception class to report domain errors (class)
<code>length_error</code>	exception class to report attempts to exceed maximum allowed size (class)
<code>out_of_range</code>	exception class to report arguments outside of expected range (class)
<code>runtime_error</code>	exception class to indicate conditions only detectable at run time (class)
<code>range_error</code>	exception class to report range errors in internal computations (class)
<code>overflow_error</code>	exception class to report arithmetic overflows (class)
<code>underflow_error</code>	exception class to report arithmetic underflows (class)

Štandardné výnimky C++

Ukážka štandardnej výnimky
std::out_of_range

```
#include <iostream>
#include <stdexcept>

using namespace std;

int main() {
    try {
        string s = "abc";
        s.at(n: 10);
        cout << "After throw";
    }
    catch (out_of_range &e) {
        cout << e.what() << endl;
    }
    cout << "After try-catch";
    return 0;
}
```



Špecifikátor noexcept

- Označuje funkciu, v ktorej nevznikajú výnimky.
- Ak vo funkcii označenej pomocou noexcept vznikne výnimka, program predčasne ukončí svoju činnosť zavolaním `std::terminate`.

```
void nonThrowing() noexcept{  
    // ...  
    // throw 0; ...v tomto prípade sa vola std::terminate  
}
```

Vlastná výnimková trieda

```
#include <iostream>
#include <cstdio>

using namespace std;

class InvalidRectangleException {
private:
    const int a, b;
public:
    InvalidRectangleException(const int a, const int b) :
        a(a), b(b) {}

    const char *what() const {
        char *message = new char[100];
        sprintf(message, format: "(EXCEPTION) Invalid rectangle: a=%i, b=%i", a, b);
        return message;
    }
};

int rectangleArea(const int a, const int b) {
    if (a < 0 || b < 0) {
        throw InvalidRectangleException(a, b);
    }
    return a * b;
}
```

```
int main() {
    try {
        cout << rectangleArea(a: 5, b: 3) << endl;
        cout << rectangleArea(a: -9, b: 10) << endl;
    }
    catch (InvalidRectangleException &e) {
        cout << e.what() << endl;
    }
    return 0;
}
```



std::string

std::string

- Trieda `std::string` je súčasťou štandardnej knižnice jazyka C++.
- Poskytuje metódy na jednoduchú prácu s reťazcami.
- Hlavičkový súbor: `#include <string>`
- Dokumentácia:

<http://www.cplusplus.com/reference/string/string/>

https://en.cppreference.com/w/cpp/string/basic_string

C-string vs. std::string

```
const char * str1 = "C-string";
```

```
std::string str2 = "C++ string";
```

Konštruktory std::string

- Default
- Z C-ret'azca
- Kopírovací
- Substring
- Fill

```
string a1; // default konstruktor
string a2("abcdefgh"); // z C-retazca
string a3(a2); // copy konstruktor
string a4(a3, pos: 1, n: 2); // substring konstruktor
string a5(n: 10, c: 'x'); // fill konstruktor
```

Kopírovanie reťazcov

Dosiahneme pomocou operátora „=“. Urobí sa hlboká kópia.

```
string b1("nieco");  
string b2;  
b2 = b1; // skopirovanie stringu b1 do b2  
  
// porovnanie adries stringov  
cout << (void*)b1.data() << endl;  
cout << (void*)b2.data() << endl;
```


Užitečné operátory

- Operátor +
- Operátor +=
- Operátor ==

```
string c1 = "Hello";  
c1 = c1 + " world ";  
cout << c1 << endl;
```

```
string c2 = "Ako sa";  
c2 += " mas?";  
cout << c2 << endl;
```

```
string c3 = "A";  
string c4 = "a";  
cout << (c3 == c4 ? "true" : "false") << endl;
```

Prístup k znaku

- Operátor []
- Metóda at()

```
string d1 = "dlhy retazec";  
cout << "Dĺzka retazca: " << d1.length() << endl;  
  
// metoda at() kontroluje, ci  
// pristupujeme k platnemu znaku  
cout << d1[3] << " = " << d1.at(n: 3) << endl;
```

Vloženie obsahu

- Metóda insert()

```
string f1 = "...[]...";  
f1.insert(pos: 4, s: "ABC");  
cout << f1 << endl;
```

Výstup: ...[ABC]...

Vymazanie obsahu

- Metóda erase()

```
string g1 = "Ahoj ako [vymazat] sa mas?";  
g1.erase(pos: 9, n: 10);  
cout << g1 << endl;
```

Výstup: Ahoj ako sa mas?

Vymazanie obsahu

- Metóda erase()

```
string g1 = "Ahoj ako [vymazat] sa mas?";  
g1.erase(pos: 9, n: 10);  
cout << g1 << endl;
```

Výstup: Ahoj ako sa mas?

Vymazanie obsahu

- Metóda erase()

9
↓

```
string g1 = "Ahoj ako [vymazat] sa mas?";  
g1.erase(pos: 9, n: 10);  
cout << g1 << endl;
```

10 znakov

Výstup: Ahoj ako sa mas?

Podret'azec

- Metóda substr()

```
string h1 = "Dobry den.";
cout << h1.substr(pos: 6, n: 3) << endl;
```

Výstup: den

Vyhľadávanie

- Metóda find()

```
string i1 = "A very long text";  
cout << (i1.find(s: "very") != string::npos ? "naslo sa" : "nenaslo sa");
```

Výstup: naslo sa

Prevod do C ret'azca

- Metóda c_str()

```
string j1 = "C retazec";  
const char* text = j1.c_str();  
cout << text << endl;
```

Prevod čísla na reťazec

- Globálna funkcia to_string()

```
string k1 = to_string(val: 500);  
string k2 = to_string(val: 7.123);  
cout << k1 << "    " << k2;
```

Výstup: 500 7.123000

Načítanie riadku

- Globálna funkcia `getline()`

```
string m1;  
// funkcia nacita text az po prvy vyskyt znaku '.'  
getline(&: cin, &: m1, delim: '.');  
cout << "Nacitany text: " << m1 << endl;
```

std::stringstream

std::stringstream

- Umožňuje vykonávať I/O operácie s textovým prúdom.
- Prístupný cez hlavičkový súbor `#include <sstream>`
- Formátované čítanie: operátor >>
- Formátovaný zápis: operator <<
- Dokumentácia:

<http://cplusplus.com/reference/sstream/stringstream>

std::stringstream

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main() {
    stringstream ss;

    // nastavenie obsahu streamu
    ss.str(s: "abc a 6 3.14");

    // formatovane citanie
    string p1;
    char p2;
    int p3;
    double p4;
    ss >> p1 >> p2 >> p3 >> p4;
```

```
// nastavenie good bitu
// po precitani obsahu zostal stream s
// nastavenym eofbit-om
ss.clear();

// vymazanie obsahu
ss.str(s: "");

// formatovane zapisovanie
ss << 1.89 << " text " << 'u' << endl;

string c;
while (ss >> c) {
    cout << c << endl;
}
return 0;
```

```
}
```



STL

STL knižnica

- Skratka pre Standard Template Library.
- Poskytuje implementáciu rôznych abstraktných dátových štruktúr a algoritmov.
- Prináša generické programovanie (rovnaký kód fungujúci pre viacero odlišných dátových typov).
- Je založená na template triedach a funkciách.

Komponenty STL

- Kontajnery
- Iterátory
- Algoritmy
- Funktory

Kontajnery

- Kontajnery sú objekty, ktoré uchovávajú kolekciu iných objektov a k nim asociované operácie.
- Kontajnery sú implementované ako class template.
- Kontajnery implementujú rôzne abstraktné dátové štruktúry líšiace sa efektivitou operácií.

<http://www.cplusplus.com/reference/stl>



Kontajnery

sekvenčné



Kontajnery

sekvenčné

kontajnerové adaptéry



Kontajnery

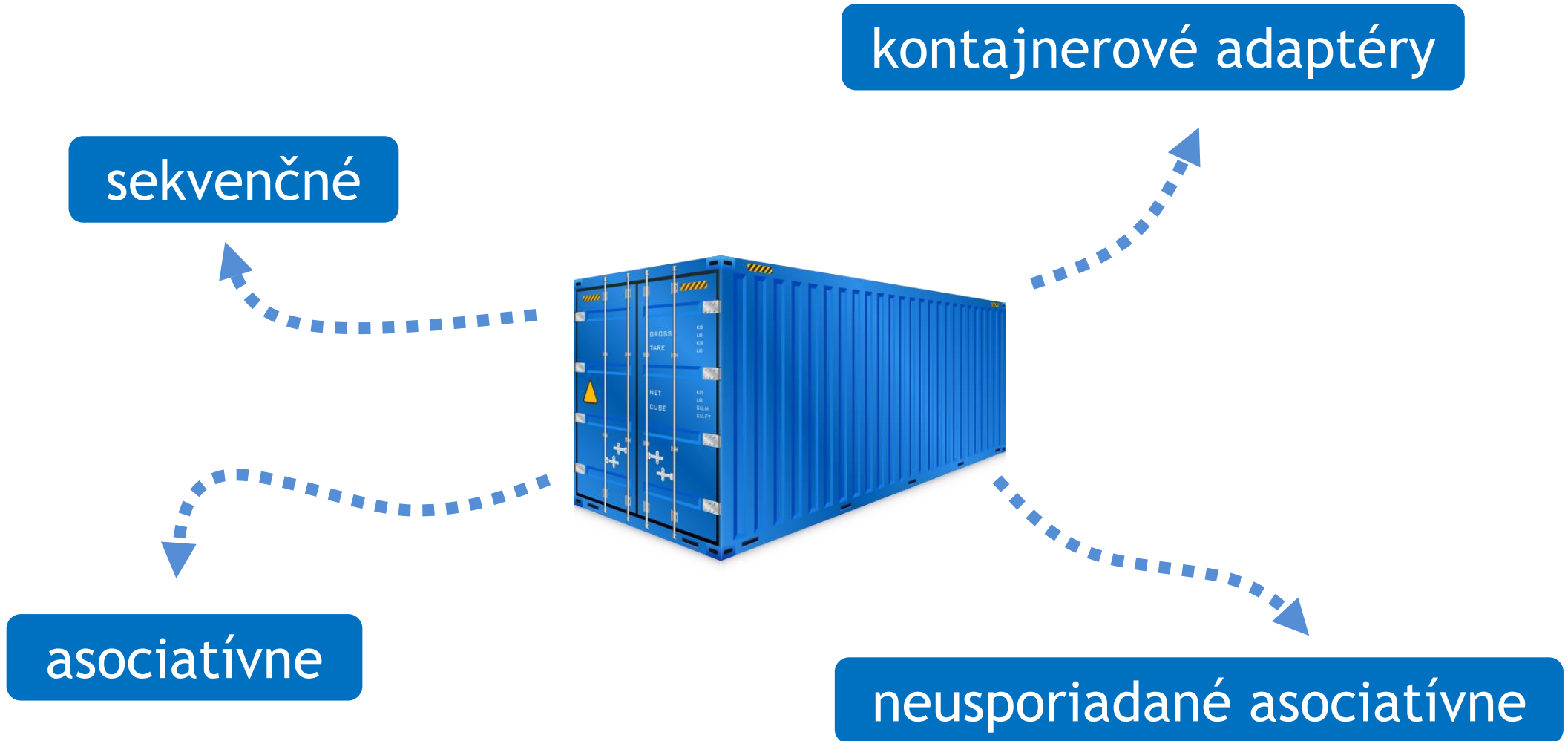
kontajnerové adaptéry

sekvenčné

asociatívne



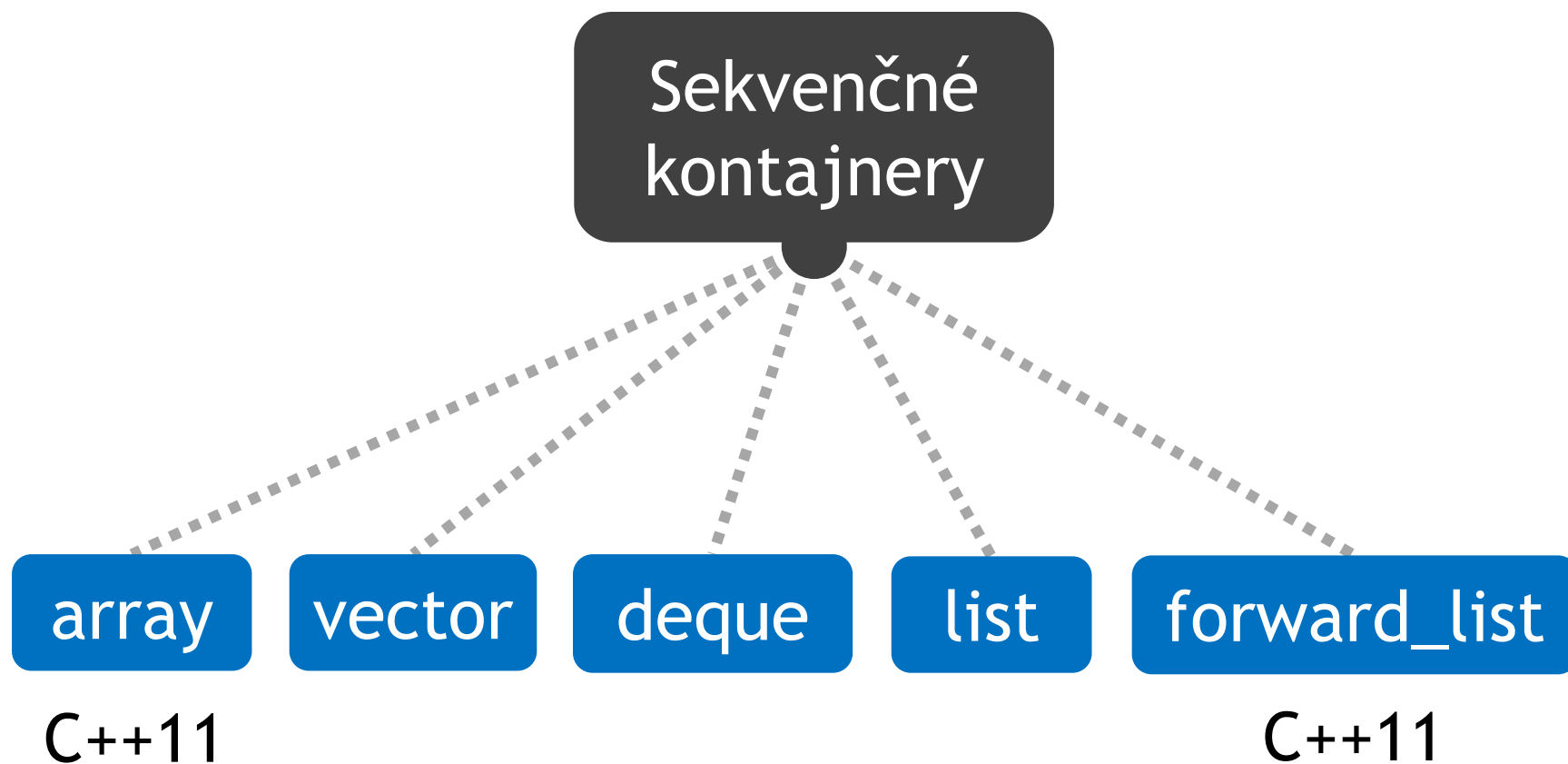
Kontajnery



Sekvenčné kontajnery

- Kontajnery, v ktorých poradie prvkov je určené programátorom.
- Lineárna sekvencia prvkov (v pamäti môžu byť fyzicky uložené spojitо aj nespojitо).

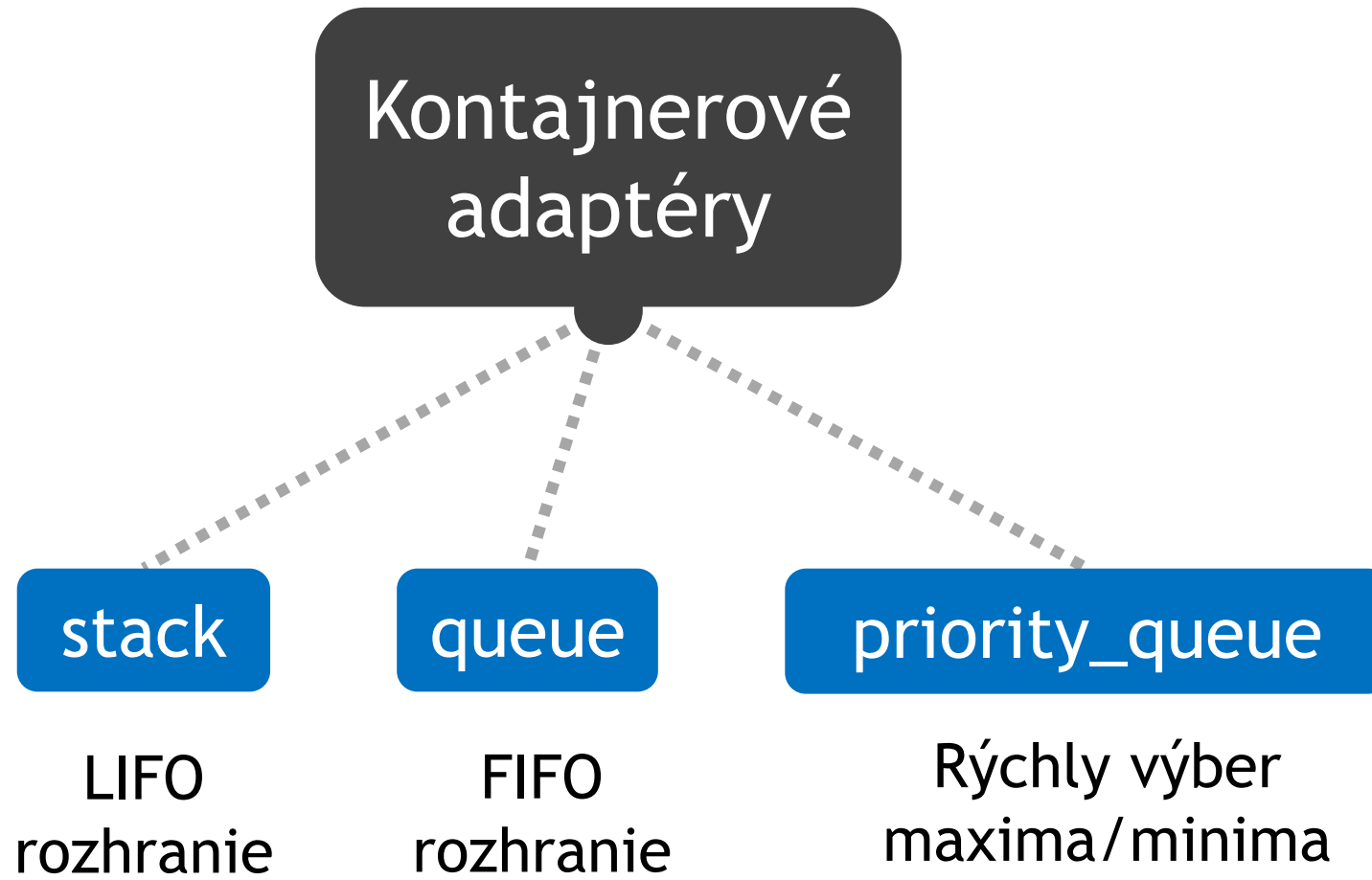
Sekvenčné kontajnery



Adaptéry

- Nie sú úplné kontajnery, ale enkapsulujú (zabaľujú) existujúce sekvenčné kontajnery.
- Pri vytváraní adaptérov môžeme špecifikovať ich tzv. „underlying container“ (kontajner, ktorý enkapsulujú).
- Adaptéry menia rozhranie „underlying“ kontajnera tak, aby dosiahli želané správanie.

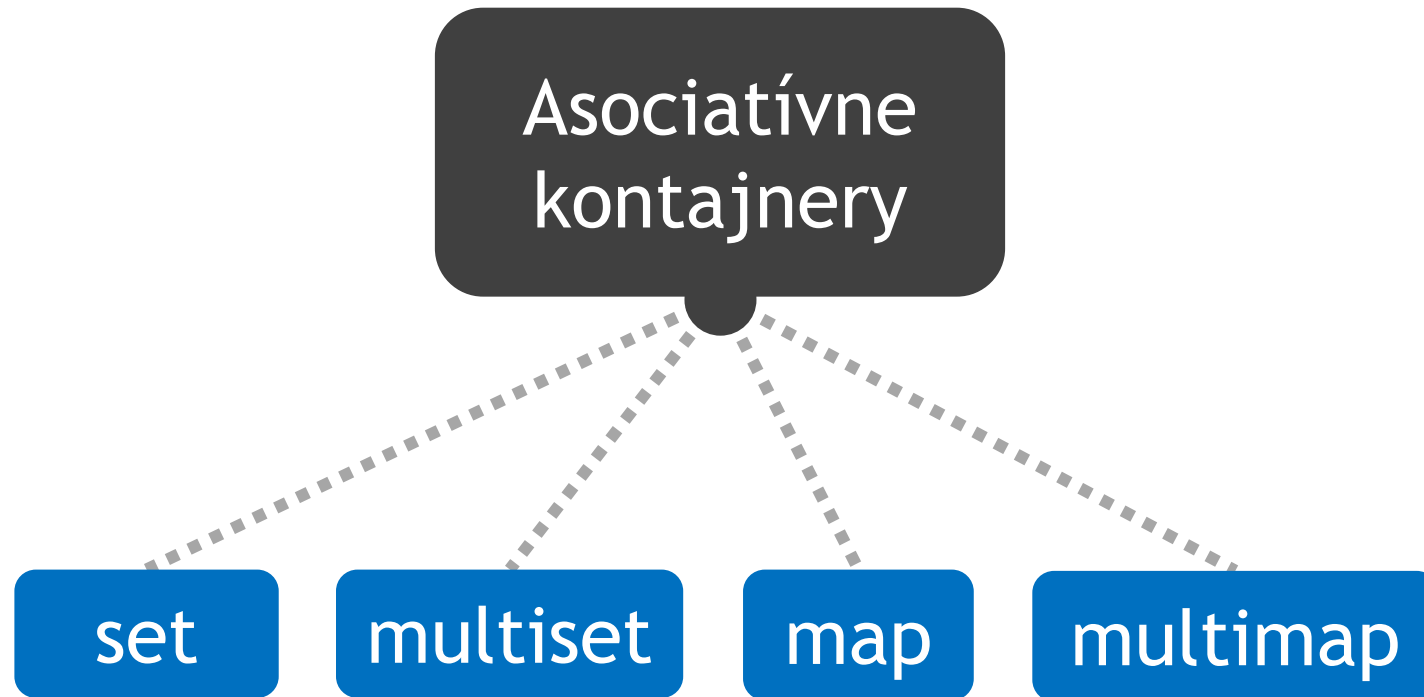
Adaptéry



Asociatívne kontajnery

- Sú to dátové štruktúry, v ktorých sa efektívne vyhľadáva.
- Kontajnery, v ktorých poradie/rozloženie prvkov je určené kontajnerom.
- Prvky kontajnera sú sprístupnené pomocou tzv. kľúča a nie pomocou ich pozície v kontajneri.

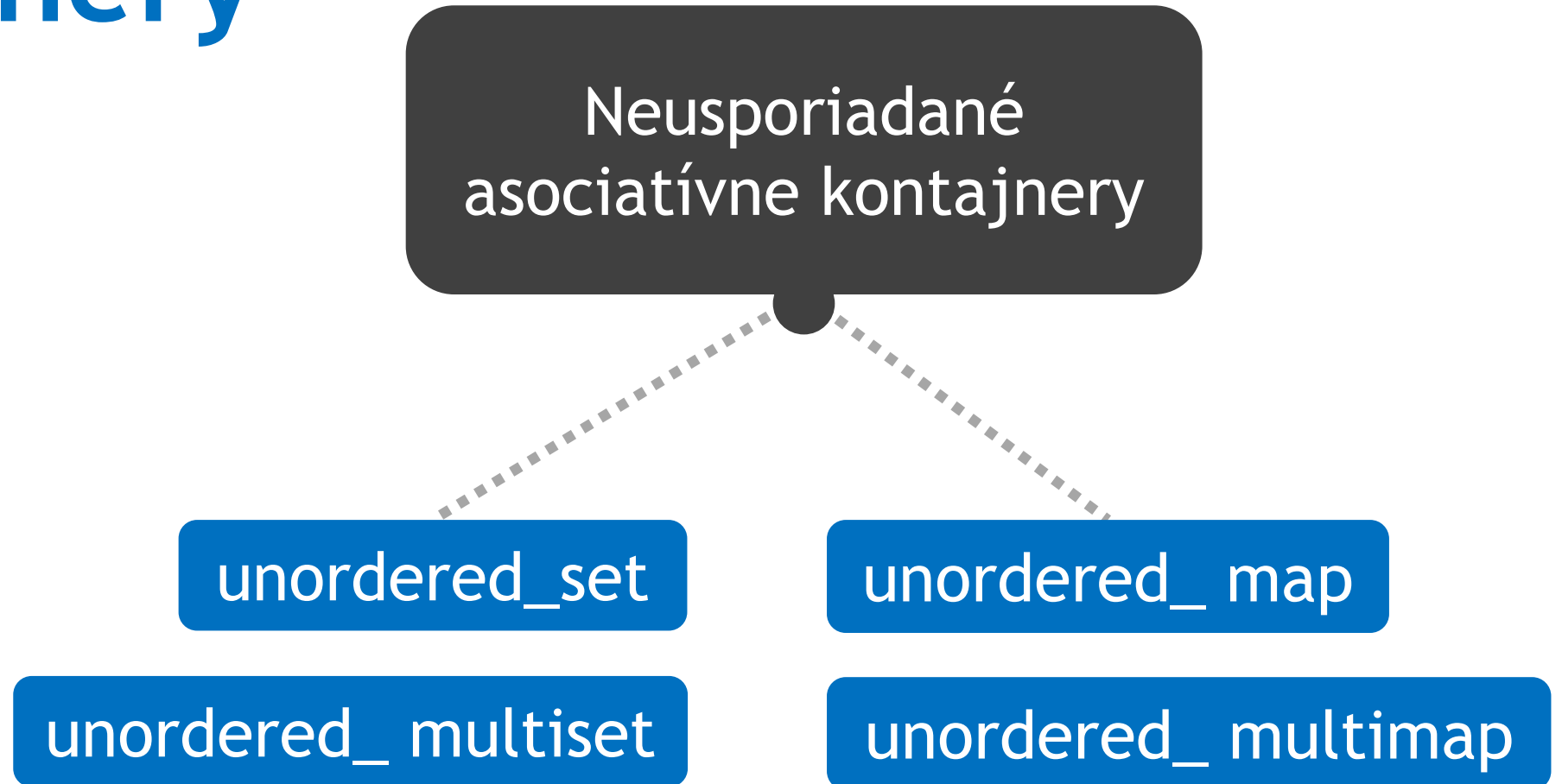
Asociatívne kontajnery



Neusporiadané asociatívne kontajnery

- Zavedené od C++11.
- Veľmi rýchle vyhľadávanie prvkov v kontajneri.
- Prvky nemajú žiadne usporiadanie.
- Prvky sa sprístupňujú pomocou hash funkcie a sú organizované v tzv. bucketoch.
- Potreba dobrej bezkolíznej hash funkcie.

Neusporiadané asociatívne kontajnery



Voľba optimálneho kontajnera

- **Ako zvoliť vhodný kontajner?** Rozhodnutie závisí od charakteru operácií, ktoré sa chystáme vykonávať.

Charakter operácie	Vhodný kontajner
Rýchly náhodný prístup	vector, array
Časté pridávanie/odstraňovanie zo začiatku/konca	deque
Časté pridávanie/odstraňovanie zo stredu	list
Rýchle vyhľadávanie	<ul style="list-style-type: none">• Asociatívny kontajner (ak záleží na poradí prvkov)• Neusporiadaný asociatívny kontajner (ak nezáleží na poradí)

Voľba optimálneho kontajnera

- <https://www.hackerearth.com/practice/notes/c-stls-when-to-use-which-stl/>
- <https://embeddedartistry.com/blog/2017/08/23/choosing-the-right-stl-container-general-rules-of-thumb/>

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast

Iterátory

- Iterátory ukazujú (podobne ako smerníky) na určitý prvok v kontajneri.
- Môžeme pomocou nich prechádzať obsah kontajneru (t.j. iterovať).
- STL obsahuje rôzne druhy iterátorov, ktoré sa odlišujú ich obmedzením pri sprístupňovaní prvkov (vid' nasledujúci obrázok).

<http://www.cplusplus.com/reference/iterator/>

Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo

Input iterátor

Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo

Input iterátor

$a==b$, $a!=b$

*a (ako rvalue)

Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo

Input iterátor

$a == b$, $a != b$

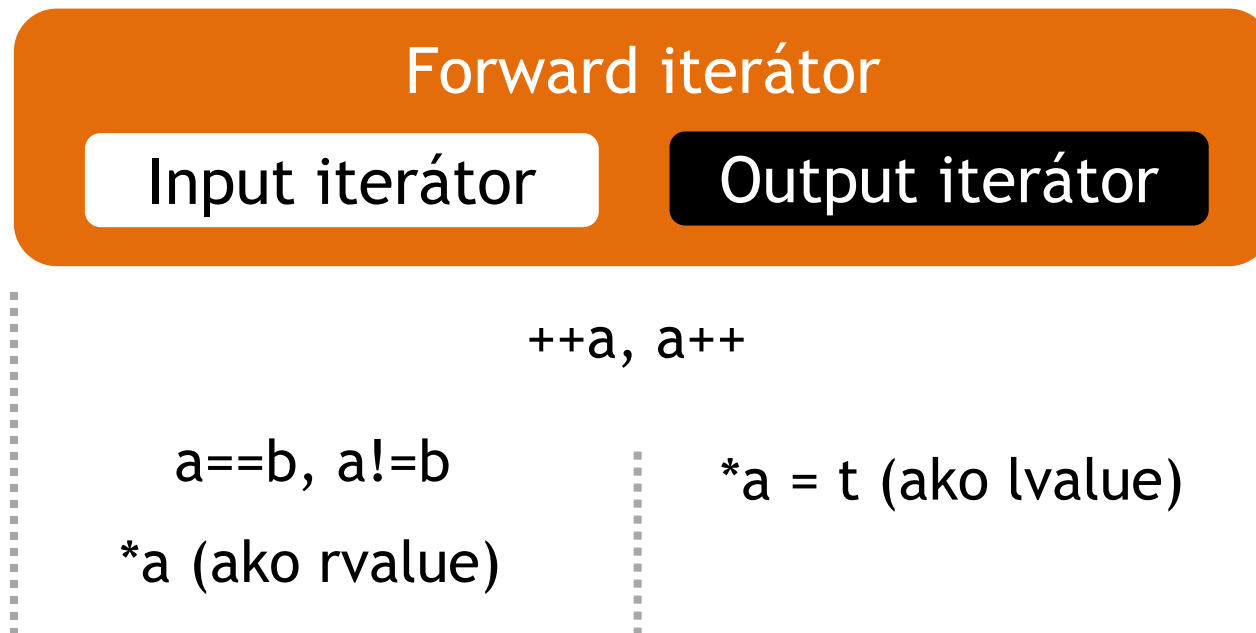
$*a$ (ako rvalue)

Output iterátor

$*a = t$ (ako lvalue)

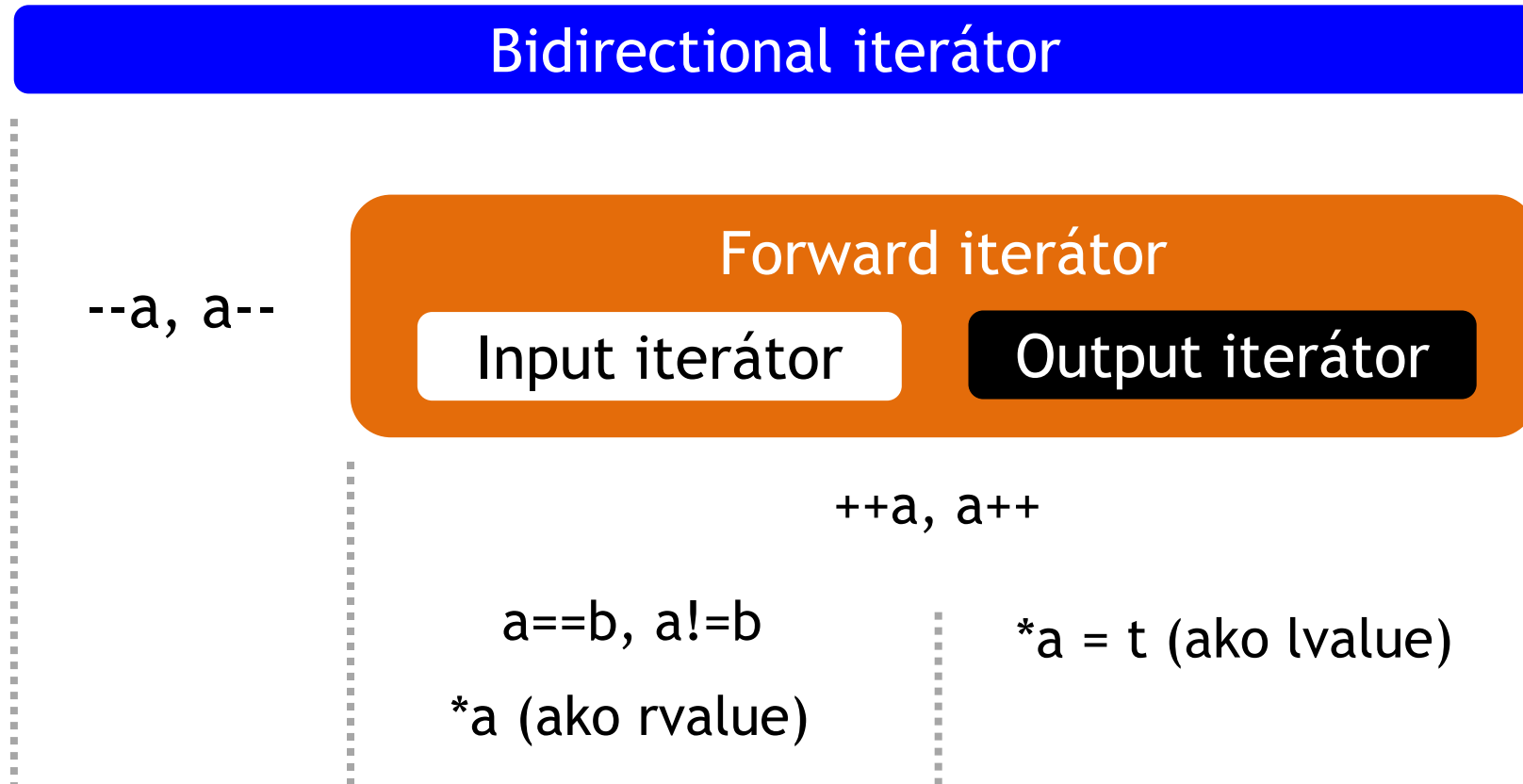
Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo



Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo



Taxonómia iterátorov

Pozn.: a,b sú iterátory, n je celé číslo

Random access iterátor

a+n, n+a

a<b

a>b

a<=b

a>=b

a+=n

a-=n

a[n]

Bidirectional iterátor

--a, a--

Forward iterátor

Input iterátor

Output iterátor

++a, a++

a==b, a!=b

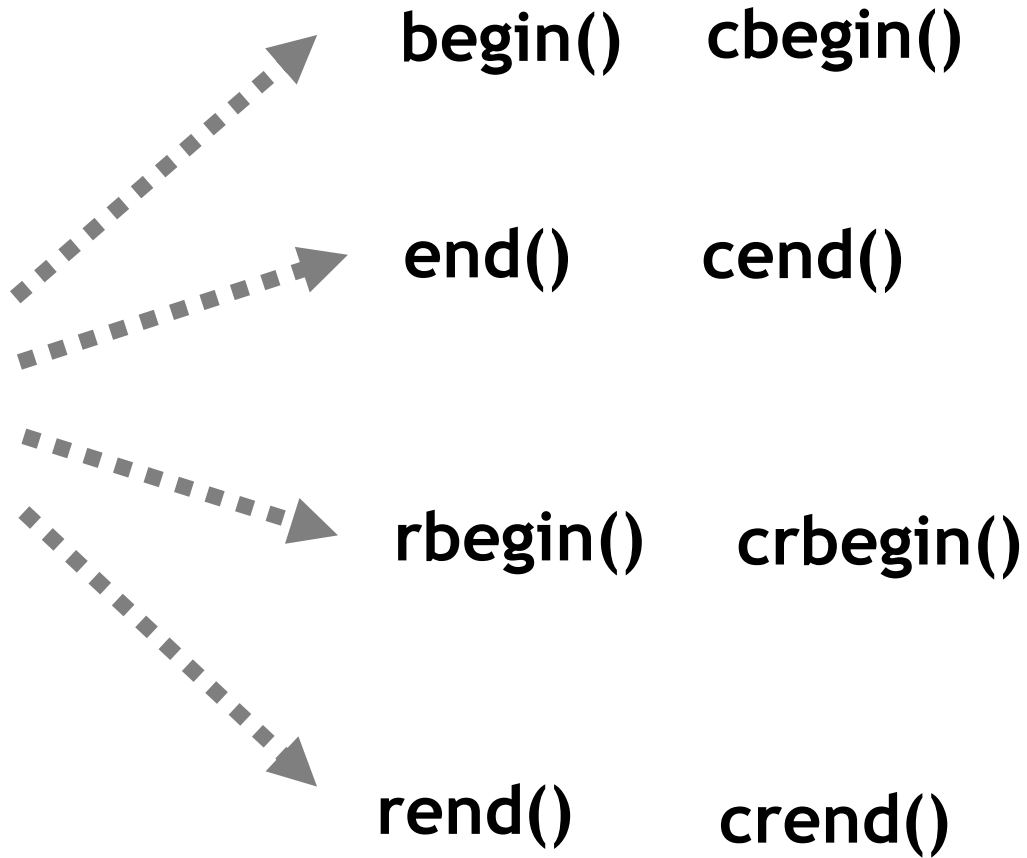
*a (ako rvalue)

*a = t (ako lvalue)

Iterátor



Kontajner

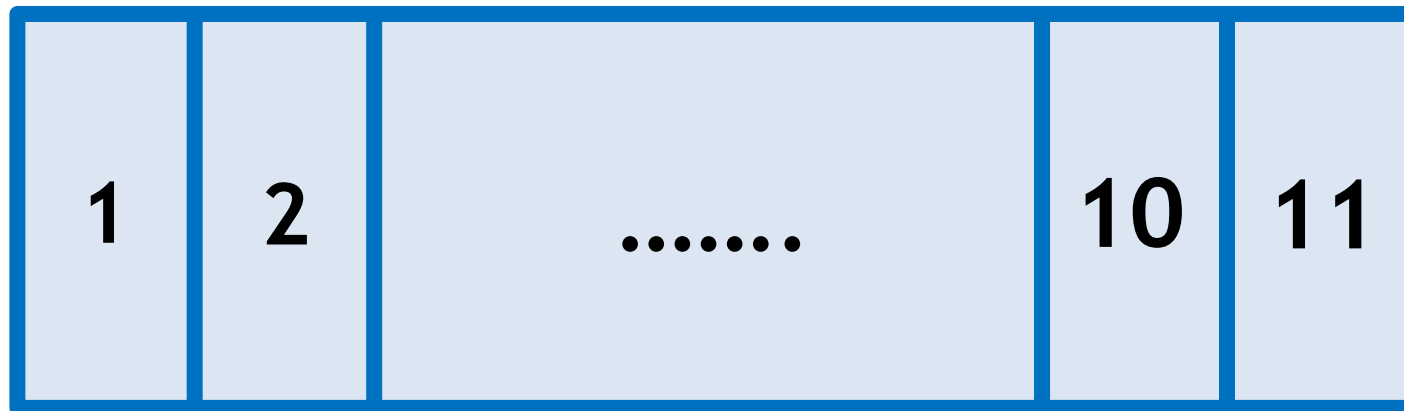


Iterátor

begin()



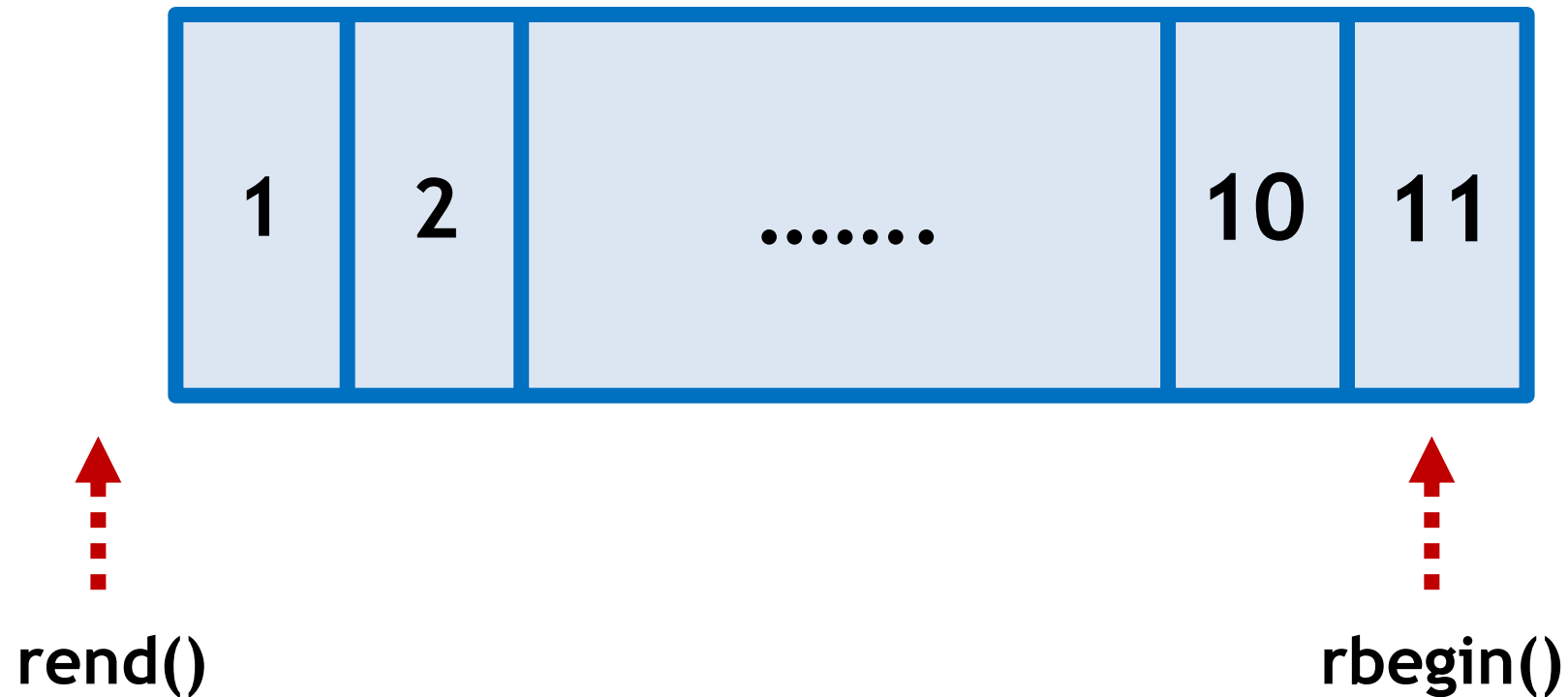
end()



Kontajner

Iterátory

Kontajner



std::vector

- Hlavičkový súbor: `#include <vector>`
- Predstavuje pole prvkov, ktoré môže dynamicky meniť svoju veľkosť.
- Zmena kapacity vektora je časovo náročná operácia.
- **Rýchle operácie:**
 - Náhodný prístup k prvku
 - Pridávanie/odstraňovanie z konca

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v{1, 2, 3, 4, 5};
    // pristup k prvkom
    cout << v[0] << " " << v.at(1) << endl;
    // pristup k prvemu/poslednemu prvku
    cout << v.front() << " " << v.back() << endl;
    // pocet prvkov v kontajneri
    cout << v.size() << endl;
    // pridavanie na koniec
    v.push_back(6);
    // pridanie pred zvolenu poziciu
    v.insert(v.begin() + 2, 1000);
    // odstranenie prvku
    v.erase(v.begin());
    // vymazanie vsetkych prvkov
    v.clear();
    return 0;
}
```

std::deque

- Hlavičkový súbor: `#include <deque>`
- Deque je akronym pre Double-Ended Queue.
- Predstavuje obojsmernú frontu s efektívnym pridávaním/odstraňovaním na koncoch.
- Nemusia uchovávať svoje prvky v spojitom pamäťovom bloku (to im umožňuje efektívnu zmenu veľkosti).

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
    deque<int> d{1, 2, 3, 4, 5};
    // pocet prvkov
    cout << d.size() << endl;
    // pristup k prvemu a poslednemu prvku
    cout << d.front() << " " << d.back() << endl;
    // pridanie prvku na zaciatok
    d.push_front(x: 0);
    // pridanie prvku na koniec
    d.push_back(x: 6);
    // vymazanie prveho prvku z
    d.pop_front();
    // vymazanie posledneho prvku
    d.pop_back();
    return 0;
}
```

std::list

- Hlavičkový súbor: `#include <list>`
- Vhodný kontajner pre situácie častého pridávania/vymazávania na ľubovoľnej pozícii (vykonané v konštantnom čase).
- Nemôžeme v ňom sprístupniť prvok podľa indexu (treba sa k nemu posúvať pomocou iterátora).

```
#include <iostream>
#include <list>

using namespace std;

int main() {
    list<int> a{1, 2, 3, 4, 5, 6, 7};

    // przechod
    list<int>::iterator i;
    for (i = a.begin(); i != a.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;

    // odwrotny przechod
    list<int>::reverse_iterator ri;
    for (ri = a.rbegin(); ri != a.rend(); ri++) {
        cout << *ri << " ";
    }
    return 0;
}
```



```
#include <iostream>
#include <list>

using namespace std;

int main() {
    list<int> a{3, 2, 2, 1, 4, 3, 4, 5, 4, 4, 3};
    // vymazanie vsetkych vyskytov hodnoty
    a.remove(value: 3);
    // vymazanie po sebe iducich duplikatov
    a.unique();
    // zotriedenie prvkov
    a.sort();

    return 0;
}
```

std::stack

- Hlavičkový súbor: `#include <stack>`
- Je to kontajnerový adaptér pracujúci FIFO štýlom (prvky sú pridávané/odstraňované len z jedného konca).
- **Základné operácie**
 - `empty()`, `size()`
 - `top()`
 - `push()`, `pop()`

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> s;
    // pridavanie na vrchol zasobnika
    for (int i = 0; i < 5; i++) {
        s.push(i);
    }
    // pocet prvkov v zasobniku
    cout << s.size() << endl;
    // sprístupnenie vrcholu zasobnika
    cout << s.top() << endl;
    // odstranovanie z vrcholu zasobnika
    while (!s.empty()) {
        s.pop();
    }
    // overenie, ci je zasobnik prazdny
    cout << (s.empty() ? "prazdny" : "neprazdny") << endl;
    return 0;
}
```

std::priority_queue

- Hlavičkový súbor: `#include <queue>`
- Je to kontajnerový adaptér, ktorý je navrhnutý tak, že prvý prvok kontajnera je maximum všetkých prvkov (rovnako ako max-heap).
- **Operácie**
 - `top()`
 - `push()`
 - `pop()`

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    priority_queue<int> pq;
    int data[5]{4, 6, 1, 8, 7};
    // pridavanie prvkov
    for (int i = 0; i < 5; i++) {
        pq.push(data[i]);
    }
    cout << pq.top() << endl; // ziskanie maxima
    pq.pop(); // vymazanie maxima
    cout << pq.top() << endl;
    return 0;
}
```

std::set

- Hlavičkový súbor: `#include <set>`
- Asociatívny kontajner uchovávajúci unikátne prvky v špecifickom poradí (určené vnútorným porovnávacím objektom).
- Prvky množiny sa nedajú meniť.
- Hodnota prvku v množine je jeho kľúčom.
- Rýchle vyhľadávanie.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
    set<int> s{9, 9, 4, 11, 3, 4}; // nepovoli duplikaty
    s.insert(x: 10); // pridavanie prvku do mnoziny
    pair<set<int>::iterator, bool> result;
    result = s.insert(x: 11); // duplikat
    cout << "Prvok 11 " << (result.second ? "bol" : "nebol") << " vlozeny" << endl;
    for (set<int>::iterator i = s.begin(); i != s.end(); i++) { // prechod mnozinou
        cout << *i << " ";
    }
    set<int>::iterator result_iter = s.find(x: 9); // vyhľadavanie v mnozine
    cout << endl << "Prvok 9 sa v mnozine ";
    if (result_iter != s.end()) { cout << "nachadza"; }
    else { cout << "nenachadza"; }
    return 0;
}
```

std::map

- Hlavičkový súbor: `#include <map>`
- Asociatívny kontajner uchovávajúci key-value páry v špecifickom poradí (určené vnútorným porovnávacím objektom).
- Kľúče sú unikátne a nedajú sa meniť.
- Hodnota prvku v mape sa dá sprístupniť kľúčom pomocou operátora `[]` a dá sa meniť.
- Rýchle vyhľadávanie (podľa kľúča).


```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<int, string> m{
        make_pair(x: 1, y: "first"), make_pair(x: 2, y: "second"),
        make_pair(x: 3, y: "third"), make_pair(x: 3, y: "other") // nepovoli duplikatne kluce
    };
    m[4] = "fourth"; // pridavanie pomocou []
    m.insert(x: make_pair(x: 5, y: "fifth")); // funkcia insert
    for(map<int, string>::iterator i = m.begin(); i!=m.end(); i++){ // prechod
        cout << i->first << " " << i->second << endl;
    }
    map<int, string>::iterator result_iter = m.find(x: 4); // vyhľadavanie
    cout << endl << "Prvok 4 sa v mape ";
    if(result_iter!=m.end()) { cout << "nachadza"; }
    else { cout << "nenachadza"; }
    return 0;
}
```

Vzorová implementácia v C/C++