

# **PB161 – PROGRAMOVÁNÍ V JAZYCE C++**

## **OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ**



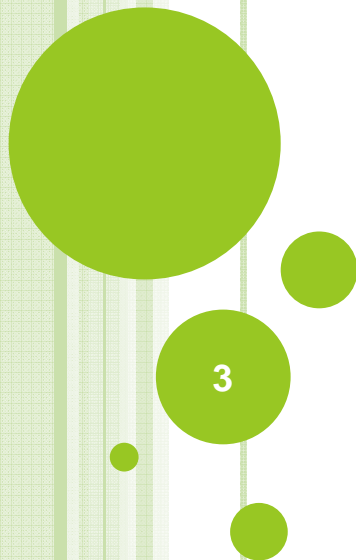
1

**Manipulátory proudů, Úvod do Standard Template Library, Kontejnery, Iterátory**

# ORGANIZAČNÍ

- Přednáška příští týden odpadá (28.10.)
  - státní svátek
  - cvičení (s výjimkou pondělí) normálně probíhají
  - pondělní skupiny si mají možnost nahradit cvičení v jiné skupině
- Průběžný test za 14 dnů (4.11.)
  - dva termíny, 14:00 a 15:00
  - nutno se přihlásit v ISu! (je již vypsáno)

# MANIPULÁTOR



# MANIPULÁTORY PROUDŮ

- `#include <iomanip>`
- Způsob jak ovlivnit chování proudů oproti defaultnímu
  - formátování vkládaných dat (typicky čísel)
  - zarovnání textu výstupu
  - chování interního vyrovnávacího pole
- Je jich velké množství, studujte dokumentaci
  - <http://www.cplusplus.com/reference/iostream/manipulators/>

## MANIPULÁTORY PROUDŮ (2)

nemali ste na mysli manipulátory?

- **Modifikátory** jsou funkce upravené pro použití s operátory `<< a >>`
  - `cout << endl;`
- Mohou být ale použity i jako běžné funkce
  - `endl(cout);`
- Modifikátory mají různou oblast platnosti
  - pouze následující znak
  - až do další explicitní změny nebo zániku proudu

# MANIPULÁTORY PROUDŮ – POČET CIFER

- Formátování reálných čísel
  - `setprecision`( ) – počet cifer za desetinou čárkou
  - `fixed` – zobrazení všech cifer do zobrazované délky čísla
  - `showpoint` – vždy zobrazovat cifry za desetinou čárkou
- Viz. ukázka

# POČET CIFER - UKÁZKA


```
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::ios;

int main() {
    const float PI = 3.141592653589793238462643383279502884197169399375105820974944592
    // Pi has 3.1415927410125732422 value
    cout << "Pi has" << std::setprecision(20) << PI << "value" << endl;
    // Pi has 3.141500 value
    cout << "Pi has" << std::setprecision(6) << std::fixed << 3.1415 << "value" << endl;

    // Problem - why? Pi has 3.141500000000000*181188397618825547397136688232421
    cout << "Pi has" << std::setprecision(50) << std::fixed << 3.1415 << "value" << endl;

    // Pi has 3.14159 value
    cout << "Pi has" << std::setprecision(4) << std::scientific << 3.1415 << "value" << endl;
    cout.unsetf(ios::scientific);
    return 0;
}
```

nieje to opačne? unitbuf použije cache  
a nunitbuf ju nepoužije?



## VYROVNÁVACÍ POLE PROUDU

- Defaultně je použito
- Nastavení chování vyrovnávacího pole
  - **unitbuf** – nepoužije se vyrovnávací paměť
  - **nunitbuf** – může se použít vyrovnávací paměť (default)
  - využití např. pro debug konzoli



# VYROVNÁVACÍ POLE PROUDU - UKÁZKA

```
#include <iostream>
#include <iomanip>
#include <sstream>
using std::cout;
using std::endl;
using std::ios;

int main() {
    cout << std::unitbuf;      // force cache flushing on insert
    cout << 255 << endl;      // output 255 and newline
    cout << std::setbase(16) << 255 << endl; // output 255 in hexadecimal and newline
    cout << 16 << endl;        // output 16 in hexadecimal and newline
    cout << std::setbase(8) << 255 << endl; // output 255 in oct and newline
    //cout << std::setbase(2) << 255 << endl; // wrong: only 8, 10 and 16 base is allowed

    cout << std::nounitbuf; // disable flushing of cache on insert
    cout << 255;             // will not display immediately
    cout << 255 << std::flush; // will force flush of this line
    cout << endl;
    return 0;
}
```

# MANIPULÁTORY PROUDŮ – ČÍSELNÁ SOUSTAVA

- Změna číselné soustavy pro výpis
  - defaultně dekadická
- Metoda `setbase( )`
  - argument 8, 10 nebo 16
- Modifikátory `dec`, `hex`, `oct`
  - analogie `setbase( )`

# ČÍSELNÁ SOUSTAVA - UKÁZKA

```
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::ios;

int main() {
    cout << std::unitbuf;    // force cache flushing on insert
    cout << 255 << endl;    // output 255 and newline
    cout << std::setbase(16) << 255 << endl; // output 255 in hexadecimal and newline
    cout << 16 << endl;    // output 16 in hexadecimal and newline
    cout << std::setbase(8) << 255 << endl; // output 255 in oct and newline
    //cout << std::setbase(2) << 255 << endl; // wrong: only 8, 10 and 16 base is allowed

    return 0;
}
```

# MANIPULÁTORY PROUDŮ – ZAROVNÁNÍ TEXTU

- Šířka výpisovaného pole `setw( )`
  - `setw(10)`
  - pokud není specifikovaná délka dostatečná, automaticky se zvětší na nejmenší potřebnou velikost
- Zarovnání vypisovaného pole v textu
  - `<< left, right`

# ŠÍŘKA A ZAROVNÁNÍ TEXTU - UKÁZKA

```
#include <iostream>
#include <iomanip>
#include <sstream>
using std::cout;
using std::endl;
using std::ios;

int main() {
    const float PI = 3.1415926535897932384626433832795028841971693993751058209749445
    cout << "Pi has" << PI << "value" << endl;
    // Pi has3.14159value

    cout << "Pi has" << std::setw(10) << PI << "value" << endl;
    // Pi has  3.14159value

    cout << "Pi has" << PI << "value" << endl;
    // Pi has3.14159value

    cout << "Pi has" << std::setw(10) << std::left << PI << "value" << endl;
    // Pi has3.14159  value

    return 0;
}
```

# DALŠÍ MANIPULÁTORY – UKÁZKA

```
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::ios;

int main() {
    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::showbase);
    cout << 255 << endl;           // 0xff
    cout.unsetf(ios::showbase);
    cout << 255 << endl;           // ff

    float value2 = 3;
    cout << std::showpoint << value2 << endl; // 3.000

    cout << std::boolalpha << (5 == 4) << endl; // false
    cout << std::noboolalpha << (5 == 4) << endl; // 0
    return 0;
}
```

přidání modifikátorů  
nastavením příznaku

přidá identifikaci  
číselné soustavy

odstranění příznaku  
modifikátorů

vynucení desetinné  
čárky

logické hodnoty slovně

# TVORBA VLASTNÍCH MANIPULÁTORŮ

- Pomocí funkce akceptující istream resp. ostream

```
#include <iostream>
using std::cout;
using std::endl;

std::ostream& insertVooDoo(std::ostream& out) {
    out << "VooDoo" << endl;
    out.flush();
    return out;
}

int main() {
    cout << "Hello " << insertVooDoo;
    return 0;
}
```

## KDY TYPICKY POUŽÍT MANIPULÁTORY?

- Defaultní hodnoty jsou většinou vhodné pro výstup čísel do “vět”
- Formátování čísel do “tabulkového” výpisu typicky vyžaduje použití `setw( )`
- Zbytek záleží na potřebách aplikace



# STANDARD TEMPLATE LIBRARY (STL)



17

# STANDARDNÍ KNIHOVNA C++

- Knihovní funkce z jazyka C
  - hlavičkové soubory s prefixem **cxxx**
  - většina standardních funkcí pro jazyk C
  - např. `cstdlib`, `cstdio`
  - C funkce obaleny ve jmenném prostoru `std::`
- C++ knihovna STL
  - překryv s funkčností z C a výrazné rozšíření
  - např. `iostream` namísto `cstdio`
  - obvykle snazší použití, snazší udržitelnost
  - může být lepší typová kontrola

# STANDARDNÍ KNIHOVNA C++ - KOMPONENTY

## 1. Bez využití šablon

- jen malá část mimo knihovních funkcí převzatých z C

## 2. Založené na šablonách, ale s fixovaným typem

- např. string (#include <string>)
- **typedef** basic\_string<**char**> string;
- (basic\_string je šablona)

## 3. Založené na šablonách, možnost volného typu

- např. dynamické pole s kontrolou mezí *vector*
- **template** < **class** T, **class** Allocator = allocator<T> > **class** vector;
- (bude probíráno později)

# CO JE GENERICKÉ PROGRAMOVÁNÍ?

- Typově nezávislý kód
- Okolní kód manipulující s objektem zůstává stejný nezávisle na typu objektu

```
float value;  
int value;  
CComplexNumber value;
```

```
cout << value;
```

- Využití dědičnosti pro částečnou typovou nezávislost

# STANDARD TEMPLATE LIBRARY (STL)

- Sada tříd, metod a funkcí do značné míry typově nezávislá
- Je to standardní součást jazyka C++ !
  - dostupné na všech platformách (s C++ překladačem)
- Několik příkladů už znáte
  - `std::string`
  - souborové a řetězcové proudy
  - `cin`, `cout`...

## NÁVRHOVÉ CÍLE STL

- Poskytnout konzistentní způsobem širokou sadu knihovných funkcí
- Doplnit do standardních knihoven běžně požadovanou funkčnost nedostupnou v C
- Zvýšení uživatelské přívětivosti
  - snadnost použití základních paměťových struktur
  - třízení, dynamická úprava velikosti, foreach...
- Využít co nejvíce typové nezávislosti
  - jednotný kód pro široké spektrum datových typů

## NÁVRHOVÉ CÍLE STL (2)

- Zvýšení robustnosti a bezpečnosti kódu
  - např. automatická správa paměti
  - např. volitelná kontrola mezí polí
  - automatické ukazatele – uvolní dynamicky alokovanou paměť
- Zachovat vysokou rychlost
  - nemusí vše kontrolovat (např. meze polí)
  - většinou existují i pomalejší, ale kontrolující metody

# STL ZAVÁDÍ NOVÉ KONCEPTY

## 1. Kontejnery

- objekty, které uchovávají jiné objekty bez ohledu na typ
- kontejnery různě optimalizovány pro různé typy úloh
- např. `std::string` (uchovává pole znaků)
- např. `std::list` (zřetězený seznam)

## 2. Iterátory

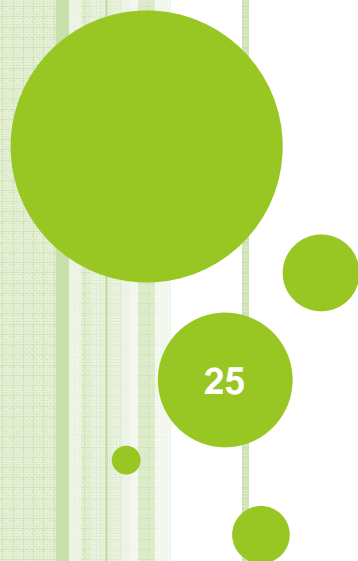
- způsob (omezeného) přístupu k prvkům kontejneru
- např. `std::string.begin()`
- přetížené operátory `++` pro přesun na další prvek atd.

## 3. Algoritmy

- běžné operace vykonané nad celými kontejnery
- např. `sort(str.begin(), str.end())`
- A mnohé další 😊



# KONTEJNERY



# SYNTAXE STL

- STL obsahuje velké množství tříd a metod
- Důležité je umět hledat

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

- A rozumět syntaxi

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

- **template** <**class** T, **class** Allocator = allocator<T>> **class** list;

- např. list<int> seznam;

- např. list<string> seznam;

klíčové slovo  
pro šablony

způsob alokace v paměti  
– většinou použita  
defaultní možnost (tj.  
nevyplňuje se)

místo doplnění  
konkrétního typu pro  
šablonu

jméno kontejneru

## KONTEJNERY - MOTIVACE

- Většinu dat je nutné uchovávat ve složitějších strukturách
  - z C známe pole a struct, z C++ třídu
- Větší množství dat vyžaduje speciální organizaci v paměti
  - dosažení rozumného výkonu vůči častým operacím
  - vkládání/odstranění/hledání prvku
  - paměťová úspornost může také hrát roli
- Není obecně nejlepší typ organizace pro všechny potřeby
  - viz. Návrh algoritmů

## KONTEJNERY – MOTIVACE (2)

- V C bylo nutné implementovat vlastní struktury
  - zřetězený seznam, strom...
  - nebo použít (nestandardizované) dodatečné knihovny
- Základní operace jsou ale typicky pořád shodné
  - např. vložení a odstranění prvku
  - např. procházení struktury po prvcích
- C++ obsahuje nejběžnější způsoby reprezentace dat přímo v základní knihovně (kontejnery)
- Jsou definovány společné funkce pro nejčastější operace
  - je díky tomu snadné „vyměnit“ druh kontejneru

```
#include <vector>
#include <list>
int main() {
    std::vector<int> kontejner(10);
    // std::list<int> kontejner(10);
    kontejner.push_back(123);
    return 0; }
```

# TYPY STL KONTEJNERŮ

- Typy kontejnerů
  - sekvenční: `vector`, `deque`, `list`
  - asociativní: `set`, `multiset`, `map`, `multimap`, `bitset`
  - s upraveným chováním (adaporty): `stack` (LIFO), `queue` (FIFO), `priority_queue`
- Interní organizace kontejnerů se liší
  - vyberte vhodný pro daný typ operace
  - dynamické pole, řetězený seznam, strom...
- Detailní srovnání parametrů možností
  - <http://www.cplusplus.com/reference/stl/>

# ZÁKLADNÍ METODY VĚTŠINY KONTEJNERŮ

- Konstruktory a destruktory
  - počáteční inicializace, kopírovací konstruktor...
- Iterátory
  - `begin()`, `end()`, `rbegin()`...
- Metody pro zjištění a manipulaci velikosti
  - `size()`, `empty()`, `resize()`...
- Metody přístupu (jen čtení)
  - `operátor []`, `front()`...
- Metody pro změnu obsahu kontejneru
  - např. `push_back()`, `clear()`...

# KONTEJNER VECTOR



- Sekvenční kontejner pro rychlý přístup indexem
  - rychlost srovnatelná s běžným polem
  - „inteligentní“ pole: možnost zvětšení, kontroly mezí...
- Podobné sekvenčnímu poli známému z C
  - např. `int pole[10];`
  - (C++11 navíc zavádí `std::array`)
- Syntaxe použití

```
#include <vector>
int main() {
    std::vector<int> vect(10);
    return 0;
}
```

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

# KONTEJNER VECTOR – DOKUMENTACE

In their implementation in the C++ Standard Template Library vectors take two template parameters:

```
template < class T, class Allocator = allocator<T> > class vector;
```

Where the template parameters have the following meanings:

- **T:** Type of the elements.
- **Allocator:** Type of the allocator object used to define the storage allocation model. By default, the allocator defines the simplest memory allocation model and is value-independent.

In the reference for the vector member functions, these same names are assumed for the template parameters.

## Member functions

<b>(constructor)</b>	Construct vector (public member function)
<b>(destructor)</b>	Vector destructor (public member function)
<b>operator=</b>	Copy vector content (public member function)

### Iterators:

<b>begin</b>	Return iterator to beginning (public member function)
<b>end</b>	Return iterator to end (public member function)
<b>rbegin</b>	Return reverse iterator to reverse beginning (public member function)
<b>rend</b>	Return reverse iterator to reverse end (public member function)

### Capacity:

<b>size</b>	Return size (public member function)
<b>max_size</b>	Return maximum size (public member function)
<b>resize</b>	Change size (public member function)



# KONTEJNER VECTOR - DOKUMENTACE

In their implementation in the C++ Standard Template Library vectors take two template parameters:

```
template < class T, class Allocator = allocator<T> > class vector;
```

Potřebné pro syntaxi vytvoření instance šablony.  
Typicky jeden parametr, zde např. `vector<int> myVect;`

## Seznam přetížených konstruktorů

```
vector<int> myVect;
```

```
vector<int> myVect(20);
```

(constructor)	Construct vector (public member function)	;
(destructor)	Vector destructor (public member function)	
operator=	Copy vector content (public member function)	

## Iterators:

begin	Return iterator to beginning (public member type)
end	
rbegin	
rend	

Dostupné iterátory (bude probráno později)

## Capacity:

size
max_size
resize

Další metody kontejneru (typicky dostupné pro většinu kontejnerů)...

# VECTOR – JAK POUŽÍVAT DOKUMENTACI

[http://www.cplusplus.com/reference/stl/vector/operator\[\]/](http://www.cplusplus.com/reference/stl/vector/operator[]/)

## **vector::operator[]**

```
reference operator[] ( size_type n );  
const reference operator[] ( size_type n ) const;
```

- Dvě přetížené verze operátoru [].
- První vrací referenci na n+1 prvek vektoru.
- Reference je nutná, aby se výraz dal použít jako l-hodnota.
  - např. `vect[10] = 4;`
- Druhá verze je určena pro konstantní objekty

### **Parameters**

n

Position of an element in the vector.  
Notice that the first element has a position of 0, not 1.  
Member type `size_type` is an unsigned integral type.

- Popis očekávaných argumentů

### **Return value**

The element at the spe

- Popis návratové hodnoty

# VECTOR – UKÁZKY

```
#include <iostream>
#include <vector>
using std::vector;
using std::cout;
using std::endl;

int main() {
    const int ARRAY_LEN = 10;
    vector<int> vect(ARRAY_LEN);

    // Fill with some data
    for (int i = 0; i < ARRAY_LEN; i++) {
        vect[i] = i + 10;
    }
    cout << "Vector size: " << vect.size();
    cout << "Value of 10. element: " << vect[9];

    // problem: reading outside allocated array
    cout << "Value of 31. element: " << vect[30]; // no bounds checking
    //cout << "Value of 31. element: " << vect.at(30); // with bounds checking

    // Add something to the end of vector
    vect.push_back(97);
    vect.push_back(98);
    vect.push_back(99);
    cout << "Vector size: " << vect.size() << endl;
    cout << "Value of last element: " << vect[vect.size() - 1] << endl;

    // Play with vector capacity (reserved space)
    cout << "Vector capacity: " << vect.capacity() << endl;
    vect.reserve(100);
    cout << "Vector size: " << vect.size() << endl;
    cout << "Vector capacity: " << vect.capacity() << endl;

    return 0;
}
```

hlavičkový soubor pro  
kontejner vector

instance šablony s typem  
int, počáteční velikost 10

přístup pomocí  
operátoru []

počet prvků ve vektoru

přidání dodatečných prvků

zvětšení předalokované  
maximální kapacity  
vektoru

## KOPÍROVÁNÍ KONTEJNERŮ

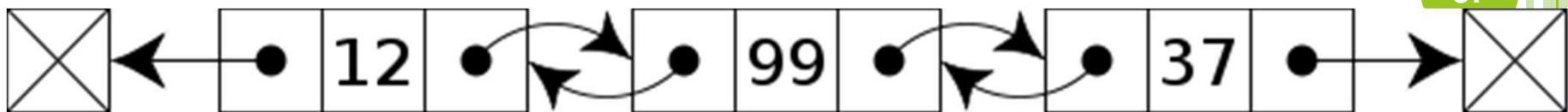
- Kontejnery lze kopírovat (operátor =)
- Dochází ke kopii všech položek
- Jednotlivé položky jsou kopírovány jako hodnoty
  - tj. plytká kopie
  - není problém u primitivních datových typů
  - není problém u typů s vhodným kopírovacím konstruktorem
- Je problém při kopírování položek typu ukazatel

## KONTEJNER `LIST`

- Spojovaný seznam pro rychlé vkládání/odebírání
  - používán velice často
  - spojovaný seznam schránek
  - jednotlivé schránky obsahují zvolený datový typ
- Nepřístupuje se přes index, ale pomocí iterátorů
- Syntaxe použití

```
#include <list>
int main() {
    std::list<int> myList;
    return 0;
}
```

<http://en.cppreference.com/stl/list/>



## KONTEJNER LIST – UKÁZKA DOKUMENTACE

**Kam vložit.** Za zadaný iterátor.

**Co vložit.** Stejný datový typ s jakým je instance šablony vytvořena.

### **list::insert**

```
iterator insert ( iterator position, const T& x );  
    void insert ( iterator position, size_type n, const T& x );  
template <class InputIterator>  
    void insert ( iterator position, InputIterator first, InputIterator last );
```

**Lze vložit i sekvence prvků zároveň.** Od iterátoru `first` po dosažení iterátoru `last`.

## KONTEJNER LIST - UKÁZKA

```
std::list<int> myList;

myList.push_back(1);
myList.push_back(2);
myList.push_back(3);
myList.push_front(4);
myList.push_front(5);

cout << "List size: " << myList.size() << endl;
// Use iterator to output all value in list
std::list<int>::iterator iter;
for (iter = myList.begin(); iter != myList.end(); iter++) {
    cout << *iter << endl;
}

iter = myList.begin(); // get iterator to begin
iter++;                // jump to next item
myList.insert(iter, 10); // insert after this item

myList.clear();
cout << "List size: " << myList.size() << endl;
```

## DYNAMICKÁ ALOKACE U KONTEJNERŮ

```
//std::list<A&> myList;    // wrong: Why?
std::list<A*> myList;      // list of pointers to A
std::list<A*>::iterator iter;
for (int i = 0; i < 10; i++) myList.push_back(new A(i));

// Memory leak, if not all A's deleted before
// myList.clear()

for (iter = mylist.begin(); iter != myList.end(); iter++) {
    // release dynamically allocated object A
    delete *iter;
}
myList.clear();
```

- Musíme uvolnit jednotlivé položky před clear()



# KONTEJNER MAP

- Motivace:
  - chceme rychlý přístup k prvku (jako u pole), ale nemáme index
  - např. UČO studenta dle jeho jména: jméno → UČO
  - idea: jméno → `hash("jméno")` → index → `pole[index]` → UČO
- Kontejner pro rychlý přístup k hodnotě prvku dle jeho klíčové hodnoty (znáte z databází)
  - realizováno prostřednictvím asociativních polí (hašovací tabulky)
  - vždy dvojice klíč a hodnota
  - instance šablony specifikujeme dvěma datovými typy
    - *typ* klíče a *typ* hodnoty
- Přístup zadáním klíče (jméno), obdržíme asociovanou hodnotu (UČO)

```
1 template < class Key, class T, class Compare = less<Key>,  
2         class Allocator = allocator<pair<const Key,T> > > class map;
```

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

## KONTEJNER MAP(2)

- Obsahuje dvojici hodnot
  - pro sekvenční procházení je nutný speciální iterátor

```
#include <map>
int main() {
    std::map<string, int> myMap;
    return 0;
}
```

- Klíče musí být unikátní
  - nelze vložit dvě různé hodnoty se stejným klíčem
  - pokud v aplikaci nastává, použijte multimap

## KONTEJNER `MAP` - UKÁZKA

```
// Key is string, value is integer
std::map<std::string, int> myMap;

// Insert key with associated value
myMap.insert(std::make_pair<std::string, int>("Jan Prumerny", 3));
myMap.insert(std::make_pair<std::string, int>("Michal Vyborny", 1));
myMap.insert(std::make_pair<std::string, int>("Tonda Flakac", 5));
// C++11 adds additional possibility: myMap.insert( {"Karel", 12} );

// Another way of insert
myMap["Lenka Slicna"] = 2; // if key does not exist yet, it is created
myMap["Lenka Slicna"] = 1; // if key already exists, just set associated value

cout << "Map size: " << myMap.size();

// Access value by key "Tonda Flakac"
cout << "Tonda Flakac: " << myMap["Tonda Flakac"];
```

# POZNÁMKY K VYUŽITÍ KONTEJNERŮ (1)

## ○ `vector`

- pro rychlý přístup indexem za konstantní čas
- časté vkládání na konec

## ○ `list`

- pro časté vkládání „někam doprostřed“

## ○ `map`, `multimap`

- pokud potřebujete asociativní pole (klíč→hodnota)

## ○ `bitset`

- pro pole bitů s ohledem na paměť
- vhodnější než `vector<bool>`

## POZNÁMKY K VYUŽITÍ KONTEJNERŮ (2)

- Pro počet položek v kontejneru je funkce `size()`
- Pro test na prázdnost
  - použijte `empty()` ne `size() == 0`
  - výpočet velikosti u konkrétního kontejneru může trvat
- Kontejnery řeší automaticky svoje zvětšování
  - pokud se už vkládaný prvek nevejde, zvětší se velikost (některých) kontejnerů o více než 1 prvek (např. o 10)
    - to ale samozřejmě chvíli trvá
  - pokud budeme vkládat 1000 prvků, je zbytečné provádět 100x zvětšování
  - využijte `reserve(1000);`
  - nebo přímo v konstruktoru `std::list<int> myList(1000);`

# ADAPTORY

- Změní defaultní chování kontejneru
- STL adaptor
  - stack – zásobník LIFO `#include <stack>`
  - queue – fronta FIFO `#include <queue>`
  - priority\_queue – prioritní fronta FIFO `#include <queue>`
- Každý adaptor má přiřazen svůj defaultní kontejner
  - použije se, pokud nespecifikujeme jiný
  - můžeme změnit na jiný, pokud poskytuje požadované metody
  - (viz. dokumentace)
- Každý adaptor má sadu podporovaných metod
  - např. stack metody `push()`, `top()` a `pop()`
  - (viz. dokumentace)

# ADAPTORY – UKÁZKA

změna defaultního kontejneru  
**POZOR:** koncové zobáky musí být '> >',  
ne '>>' (operátor vstupu)  
Pozn. Od C++11 už mohou být zobáky za  
sebou (překladač rozliší dle kontextu)

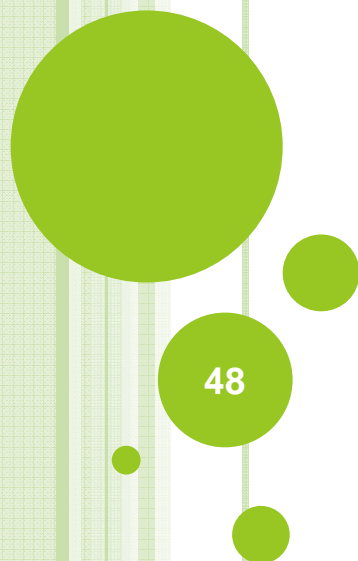
```
#include <iostream>
#include <stack>
using std::cout;
using std::endl;

int main() {
    std::stack<int> myStack;
    std::stack<int, std::list<float>> > myStack2;

    myStack.push(10);
    myStack.push(2);
    myStack.push(35);
    myStack.push(14);

    while (!myStack.empty()) {
        cout << myStack.top() << endl;
        myStack.pop();
    }
    return 0;
}
```

# ITERÁTOR





## ITERÁTORY - MOTIVACE

- Nástroj pro přístup k položkám daného kontejneru
- Lze si představit jako inteligentní ukazatel na položku kontejneru
- Ne vždy je dostupný operátor []
  - u některých kontejnerů nelze přímo určit, která položka je i-tá (resp. takové řazení nemá pro uživatele význam)
  - např. kontejner map
- Možnost sekvenčního procházení
- Možnost aplikace operace na oblast od do
  - např. vypiš od aktuální položky do konce
- Může být mírně rychlejší než indexování
  - <http://stackoverflow.com/questions/776624/whats-faster-iterating-an-stl-vector-with-vectoriterator-or-with-at>

## ITERÁTORY - SYNTAXE

- `std::jméno_třídy<parametry_šablony>::iterator`
- `std::list<int>::iterator iter;`
- `std::map<string, int>::iterator iter;`
- Kontejnery mají alespoň metody `begin()` a `end()`
  - `begin()` vrátí iterátor na první položku
  - `end()` vrátí iterátor těsně “za” poslední položkou
  - přetížené i pro `const` varianty (překladač rozliší dle kontextu)
  - C++11 přidává `cbegin()` a `cend()` pro explicitní specifikaci

```
std::list<int>::iterator iter;  
for (iter = myList.begin(); iter != myList.end(); iter++) {  
    cout << *iter << endl;  
}
```

# ITERÁTORY – ZÍSKÁNÍ, TEST KONCE

## ○ Získání iterátoru

- kontejner má metody `begin()` a `end()` (a další)
- jako výsledek dalších metod
  - např. STL algoritmu `find(hodnota)`

## ○ Testování na konec procházení kontejneru

- metoda `end()` vrací iterátor na prvek **za posledním**
- `for(iter=vect.begin(); iter!=vect.end(); iter++) { }`
- typicky jako `if((iter = find())!=vect.end()) { }`

## ○ Testování na prázdnotu kontejneru z hlediska iterátorů

- splněno když `begin() == end()`
- použijte ale metodu `empty()`
- nepoužívejte `size()` – může trvat

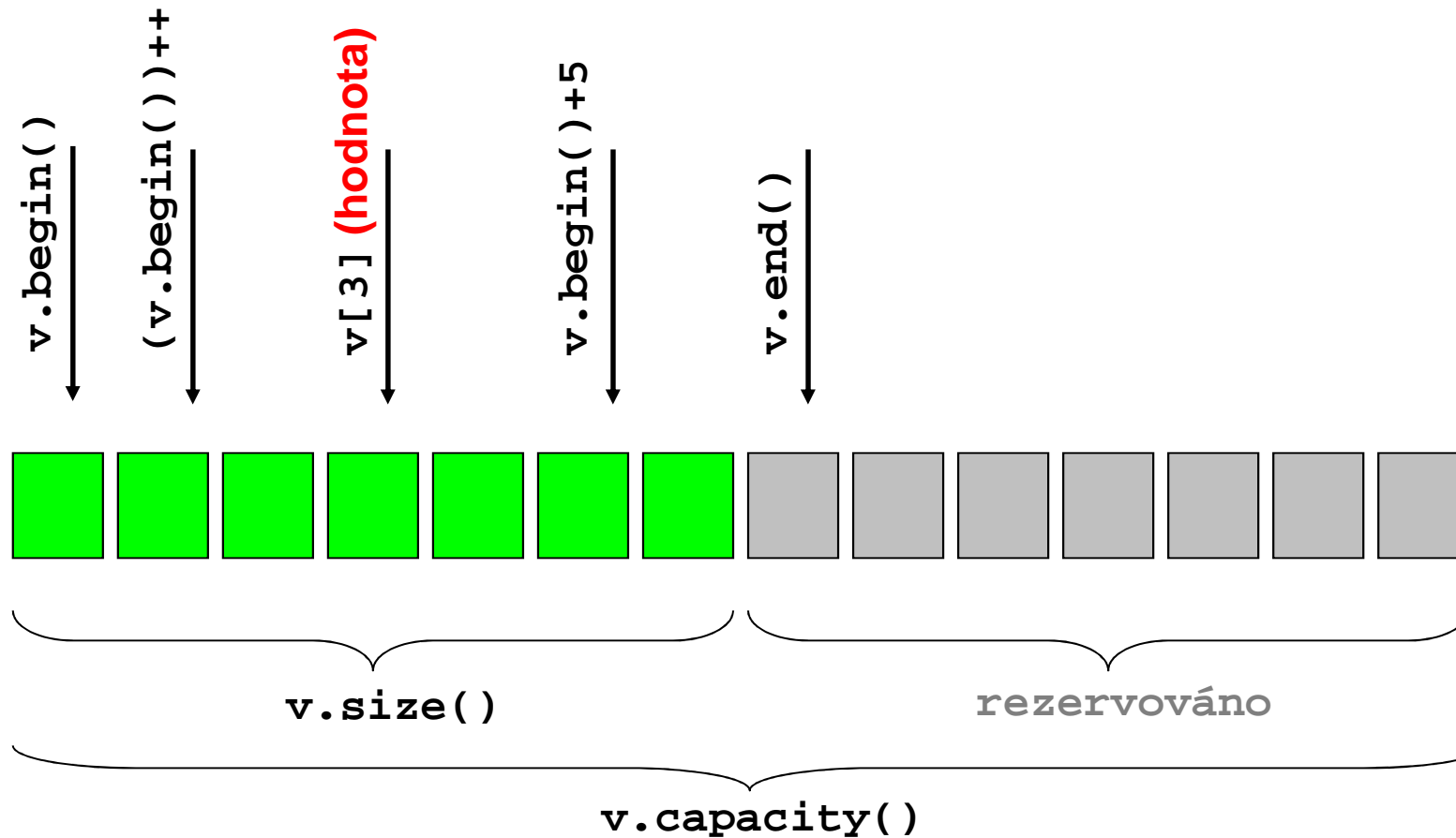
## ITERÁTORY – POSUN, PŘÍSTUP

- Iterátor přetěžuje operátory pro posun a přístup
- Operátor `++` a `--`
  - posun na další resp. předchozí prvek
  - prefixové i postfixové verze
- Operátor `*`
  - iterátor je forma ukazatele => dereference
  - **Jaký je rozdíl mezi `iter = a * iter = ?`**
- Operátor `->`
  - např. `map<typ_klíč, typ_hodnota>::iterator iter;`
  - `klíč = iter->first; hodnota = iter->second;`

# ITERÁTOR – ILUSTRACE NA KONTEJNERU

## VECTOR

o `std::vector<int> v(7);`



## ITERÁTORY - PŘÍMÝ PŘÍSTUP

- Některé kontejnery umožňují přístup dle indexu
  - operátor `[]`
- Umožňují ukazatelovou aritmetiku
  - `&vect[x] == &vect[0]+x`
- Nemusí kontrolovat konec kontejneru!
  - stejná situace jako `[]` u pole hodnot
  - hrozí čtení a zápis za koncem pole
- Kontejner umožňující přímý přístup typicky nabízí i metodu s kontrolou
  - např. `vector::at()` vyvolá výjimku při přístupu mimo

## ITERÁTORY – DALŠÍ VYUŽITÍ

- Použití pro označení pozice pro vybranou operaci
  - např. specifikace pozice, kam se má vkládat prvek
    - `insert(iterator, hodnota);`
  - např. specifikace prvku, který se má odebrat
    - `erase(iterator);`
- Použití pro STL algoritmy (později)
  - specifikace rozsahu pro vykonání algoritmu
  - např. tříděný nebo kopírovaný úsek

## ITERÁTORY – UKÁZKA VÝPIS KONTEJNERU VECTOR

- přes index (operátor [])
- přes iterátor (procházení od začátku do konce)

```
// Create small vector and fill ascendingly
std::vector<int> myVect(10);
for (unsigned int i = 0; i < myVect.size(); i++) myVect[i] = i;

// Print via direct access
for (unsigned int i = 0; i < myVect.size(); i++) {
    cout << myVect[i] << endl;
}

// Print via iterator
std::vector<int>::iterator iter2;
for (iter2 = myVect.begin(); iter2 != myVect.end(); iter2++) {
    cout << *iter << endl;
}
```



# TYPY ITERÁTORŮ

- vstupní (istream\_iterator)
  - jen čtení (nemodifikuje kontejner)
- výstupní (ostream\_iterator)
  - jen zápis (neumožňuje čtení, jen zápis)
- dopředný (operace ++)
  - čtení i zápis, posun možný pouze “dopředu”
- zpětný (operace --)
  - čtení i zápis, posun možný pouze “dozadu”

## TYPY ITERÁTORŮ (2)

- obousměrný (list, map...)
  - nejpoužívanější, obousměrný posun, čtení i změna
- přímý přístup (vector, deque)
  - umožňuje přístup přes index
- const\_iterator
  - pro použití s konstantními objekty (nelze měnit odkazovaný prvek)

## REVERZNÍ ITERÁTOR

- Obrací význam operátorů ++ a --
- `std::kontejner::reverse_iterator`
- Využití metod `rbegin()` a `rend()`

```
// Wrong: naive print from end will not print first item
for (iter = myList.end(); iter != myList.begin(); --iter) {
    cout << *iter << endl;
}

// Use reverse iterator
std::list<int>::reverse_iterator riter;
for (riter = myList.rbegin(); riter != myList.rend(); riter++) {
    cout << *riter << endl;
}
```

čo? asi nedokočená myšlienka,

## ITERÁTORY - POZNÁMKY

- Iterátor abstrahuje koncept procházení a manipulace pro různé kontejnery
- Množina podporovaných iterátorů ale není stejná pro všechny kontejnery
  - nelze vždy pouze vyměnit typ kontejneru
- Iterátor nekontroluje meze kontejneru
  - může číst/zapisovat za koncem vyhrazené paměti
  - některé implementace (např. MSVS2012 debug) ale provádějí
- Iterátor může být zneplatněn, pokud se výrazně změní obsah souvisejícího kontejneru (**erase( )**)

# SHRNUTÍ

- Modifikátoru proudů
  - umožňující změnit dílčí vlastnosti proudu
  - jednorázově vs. do další změny
- Standard Template Library (STL)
  - pohodlná a rozsáhlá – nutné umět hledat v dokumentaci
  - Zvolte vhodný kontejner
- Kontejnery + iterátory (+ algoritmy)
  - ušetří hodně práce