

# Programování v C++

---

Filip Zavoral

[zavoral@ksi.mff.cuni.cz](mailto:zavoral@ksi.mff.cuni.cz)

[www.ksi.mff.cuni.cz/~zavoral](http://www.ksi.mff.cuni.cz/~zavoral)

NPRG041 - cvičení

ZS 2020/21

# Organizace cvičení

- účast na cvičeních
  - aktivní  $\rightsquigarrow$  praktická znalost předchozí látky
  - dokončené programy  $\rightsquigarrow$  GitLab
  - 3 nepřítomnosti OK, delší domluvit předem
- 2 domácí úlohy - Recodex
  - během semestru jedna menší (15b) a jedna větší (25b) DÚ
  - hodnocení se započítává **do zkoušky!**
- zápočtový program
  - do konce října - vlastní návrh, kreativita
  - do 20.11. schválené zadání
  - vývoj v GitLabu
    - každý den commit!
  - do 30.4. první pokus o odevzdání hotové verze
    - stránka ke cvičení  $\Leftarrow$  obsahuje požadavky na program a jeho odevzdání
  - do konce výuky v LS komplet hotovo vč. doc
- zkouškový test
  - během zimního zkouškového období v labu (60b)
  - poslední termín během LS

# Zaměření cvičení

- očekávané znalosti
  - Počítačové systémy
    - C a trochu C++, dekompozice, syntaxe jazyka, třídy
  - Programování 2
    - algoritmizace, ovládání Visual Studia
- zaměření cvičení
  - důkladná znalost jazyka
    - pokročilé konstrukce, praxe v používání
    - efektivita!
  - knihovny
  - best practices ~> profesionální úroveň
    - kultura a kvalita zdrojového kódu
      - čitelnost, udržitelnost
    - ladění
- vývojové prostředí
  - norma C++ (17, 20, ...)
    - korektní program na **všech** platformách
  - Visual Studio 2019
    - language standard C++17 / latest

Create New Project [Language: C++]  
Console App  
Name, Location - Create

Solution Explorer  
Solution / Project / Source Files  
Add New/Existing Item  
Visual C++ / C++ File (.cpp)  
(... Header File)

ctrl-shift-B  
F5, ctrl-F5  
F10, F11, F9  
  
Debug / Window  
Watch, Auto, Locals, Call Stack

*ctrl F5*: Solution Explorer  
Project / Properties / Linker / System /  
SubSystem: Console

# Hello world

---

a další základní obraty

# Hello world, parametry

```
#include <iostream>
```

```
int main()  
{
```

```
    std::cout << "Hello world" << std::endl;
```

```
    return 0;
```

```
}
```

přetížený  
operátor <<

nepoužívejte staré C  
knihovny (stdio, ...)

[www.cppreference.com](http://www.cppreference.com)

C++ knihovny:  
namespace std

*C++20: format*

rozbalení std  
nikdy v headeru !

předávání parametrů odkazem  
konstantní reference

přístup k prvkům vektoru

Solution Explorer / Project Properties /  
Debugging / Command Arguments

vektor pro komfortnější zpracování

ošetření parametrů příkazové řádky

vlastní funkcionalita mimo main!

```
#include <string>  
#include <vector>  
using namespace std;
```

```
int doit( const string& s ) { ... }
```

```
void zpracuj( const vector<string>& a ) {  
    ... a[i] ...  
}
```

```
int main( int argc, char ** argv )  
{
```

```
    vector<string> arg( argv, argv+argc);
```

```
    if ( arg.size() > 1 && arg[1] == "--help" ) {  
        cout << "Usage: myprg .... " << endl;  
        return 8;  
    }
```

```
    zpracuj( arg);
```

předávání parametrů  
vždy hodnotou → kopie!


efektivita!

# Násobilka



- vypište násobilku všech čísel z parametrů příkazové řádky
  - postupný meziúkol: vypište parametry příkazové řádky
- rozšíření: parametry
  - rozsah hodnot násobilky
    - f ≈ from (default 1), -t ≈ to (default 10)
  - nepovinné, lze zadat i jen jeden parametr
  - nasobilka **-f 3 -t 12** 5 32

následující parametr ≡ od kolika

```
1 * 7 = 7
2 * 7 = 14
...
10 * 7 = 70
```

-  dekompozice
  - smysluplné jednotky s jasnou funkčností
  - pojmenované
  - co nejužší rozhraní
  - efektivní** předávání parametrů

-  varování

- zapomeňte na  ~~globální proměnné~~ 
- lokalita přístupu
- minimalizace rozhraní funkcí / metod
  - prevence chyb, zjednodušení diagnostiky

konverze string → číslo

```
int stoi( const string& s);
```

```
.... f( const vector<string>& a)
{
    for( int i = 1; i < a.size(); ++i) {
        .... a[i]
```

velikost kontejneru

range-base for

```
for( auto&& s : a) {
    .... s
```

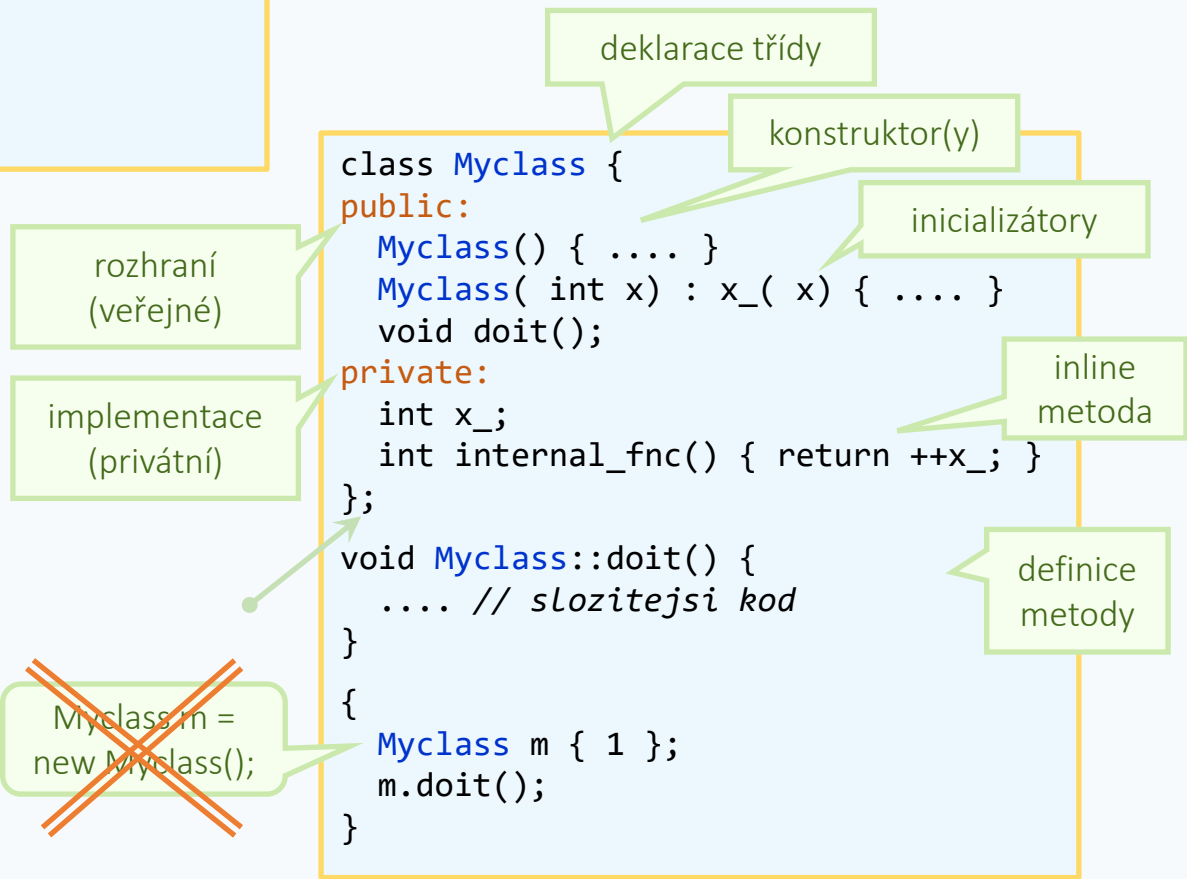
typová dedukce

# Rozhraní

```
void nasobilka( int cislo, int od, int do) {  
    ....  
}  
  
int main( ....) {  
    .... // zpracovani parametru  
    int od = ....  
    int do = ....  
    for( ....) {  
        cislo = atoi( ....);  
        nasobilka( cislo, od, do);  
    }  
}
```

- dekompozice
  - funkční?
  - efektivní?

- objekt
  - běhová instance třídy
  - sdílená data
    - různými metodami třídy
    - různými běhy jedné metody
- násobilka
  - třída
    - sdílená data
  - ➡ GitLab **c01-nasoblika**



# Řetězce

---



# Řetězce, stringy, C-stringy, string\_view

pouze podporované operace

nepoužívejte, pokud vás k tomu nikdo nenutí

```
char a[] {"ahoj"};  
char* b {"ahoj"};  
char* c {a};
```

pole znaků ≡ C-string

ukazatel na C-string

```
string d {"ahoj"};  
string e {a};  
string f {d + c + a};
```

std:: knihovna

přetížené operátory

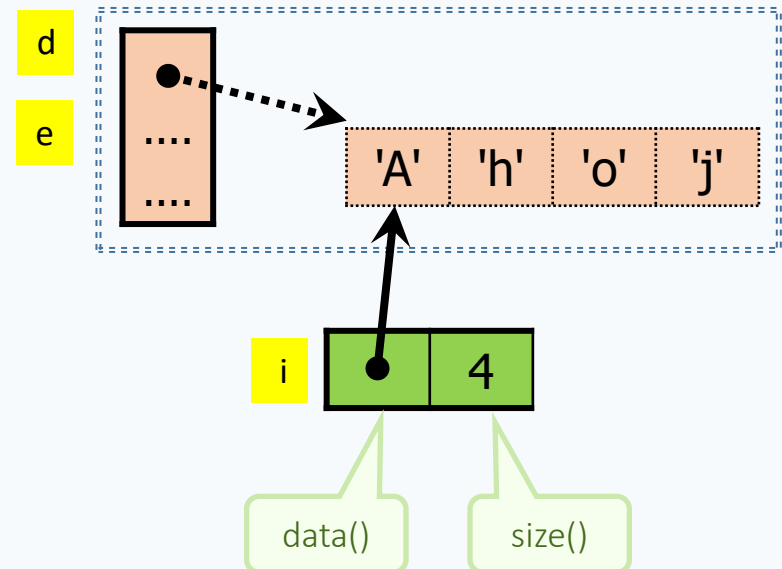
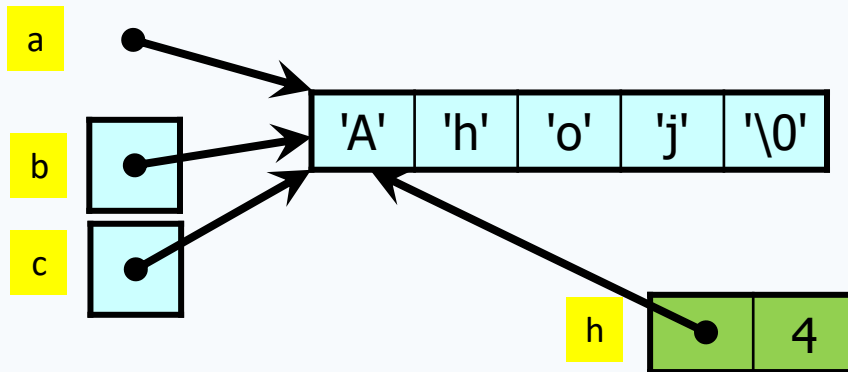
```
string_view g {"ahoj"};  
string_view h {c};  
string_view i {e};
```

pohled na existující objekt

```
size_t f( string_view s )  
{ .... }
```

```
string x { "ahoj"};  
f( x);  
f( "ahoj");
```

typické  
použití



# Nepoužívejte char\*

- "Karel", "Gott"  $\rightsquigarrow$  "Karel Gott"

```
string cele_jmeno( const string& jm, const string& prij)
{
    return jm + " " + prij;
}
```

Jméno, Příjmení  
↓  
Jméno Příjmení  
☹

low-level  
chybové  
pracné  
ne-bezpečné

```
int cele_jmeno( char * buf, size_t bufsize,
               const char * jm,
               const char * prij)
{
    size_t lj = strlen( jm);
    size_t lp = strlen( prij);
    if ( lj + lp + 2 > bufsize )
    { /* error */ return -1; }
    memcpy( buf, jm, lj);
    buf[ lj ] = ' ';
    memcpy( buf + lj + 1, prij, lp);
    buf[ lj + lp + 1 ] = 0;
    return lj + lp + 1;
}
```

# Řetězce a čísla

```
f( string_view sv) {  
    string x { sv};  
    stoi( x);  
    stoi( string{ sv});  
}
```

string\_view  $\rightsquigarrow$  string

pozor kopie!  
efektivita!

~~string  $\rightsquigarrow$  string\_view  $\rightsquigarrow$  string ....~~

```
#include <string>
```

```
int stoi( const string& s);  
int stoi( s, size_t& idxRet = nullptr, int base = 10);  
stol, stoul, stoll, stof, stod, ...  
string to_string( val);
```

nepovinné parametry:

- první nezkonvertovaný znak  
(reference - návratový parametr)
- soustava

[www.cppreference.com](http://www.cppreference.com)

```
#include <cctype>  
isalpha( c)  
isalnum( c)  
isdigit( c)
```

isdigit je lepší

```
if( c >= '0' && c <= '9')  
if( isdigit( c))  
int n = c - '0';
```

OK - číslice jsou  
uspořádané

~~... = c - 48;~~

'0'  $\neq$  48

písmena nejsou  
uspořádaná !!

~~if( c >= 'a' && c <= 'z')~~

# Počítání oveček

---

# Počítání oveček, streamy

- spočtete
  - počet znaků, řádek, slov, vět
  - počet a součet čísel
- upřesnění zadání
  - zdroj dat: cin, obecný istream
  - co to je slovo, věta
  - rozhraní
    - návratové hodnoty
    - různá funkčnost vs. neopakovatelný vstup
- postup
  - 1. funkční návrh
  - 2. objektový návrh, rozhraní !
    - dekompozice, encapsulace
    - výpočet vs. vstup/výstup
  - 3. implementace

nikdy  
dohromady!

jakýkoliv vstupní stream  
(cin, soubor, řetězec, ...)

(pokus o) načtení znaku  
(nemusí se povést!)

načtení řetězce

detekce jakékoliv chyby  
(např. EOF)

platná načtená hodnota

```
#include <iostream>

fce( istream& s) {
    char c;
    string word;
    for(;;) {
        c = s.get();
        s >> word;
        if( s.fail())
            return;
        process( c);
    }
}
```

zpracování std vstupu

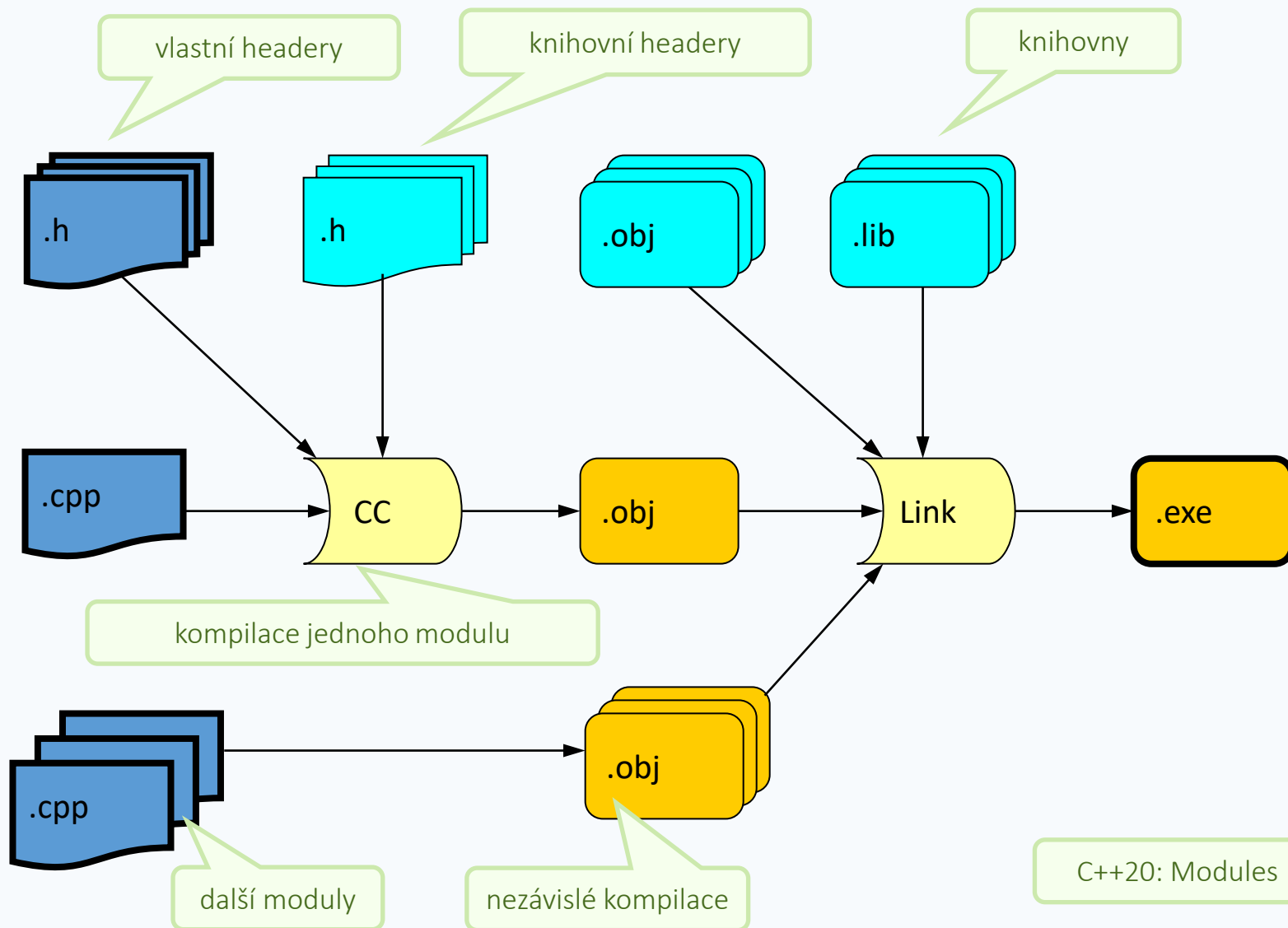
zpracování souboru

```
#include <fstream>

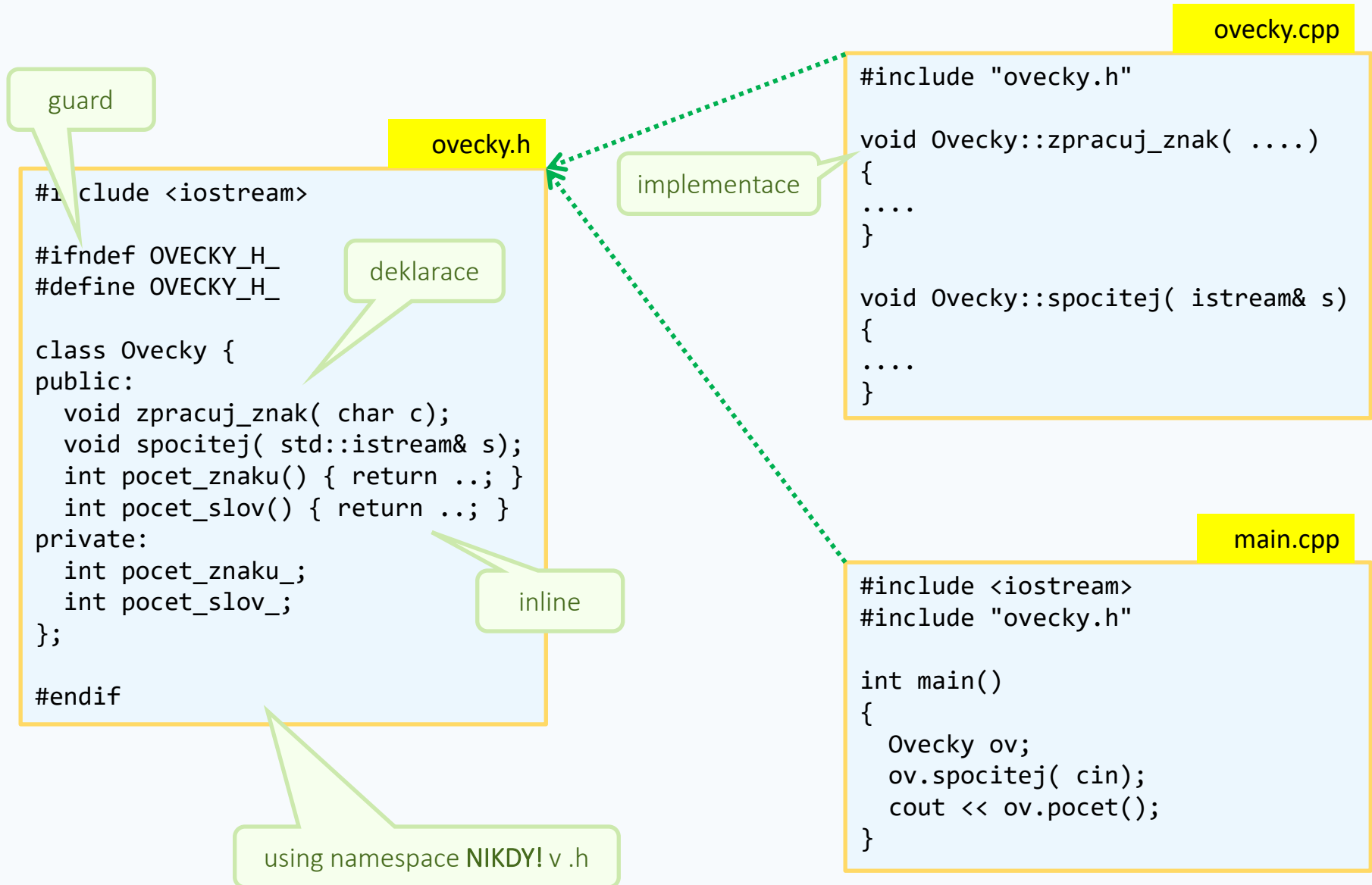
fce( cin);

ifstream f;
f.open( "file.txt");
if( ! f.good()) ....
fce( f);
```

# Překlad a linkování



# Modularita a zdrojové soubory



# Počítání oveček – upřesnění

- spočtete
  - počet **znaků**, **řádek**, **slov**, **vět**, **počet** a **součet** čísel
  - **znaky**: vše včetně mezer, konců řádek apod.
    - neuvažujte diakritiku, resp. všechny speciální znaky považujte za nepísmena
  - **slovo**: nejdelší posloupnost alfanumerických znaků nezačínající číslicí
  - **číslo**: posloupnost číslic následující za nealfanumerickým znakem
    - `' .12ab. '` je jedno číslo a žádné slovo
  - **řádky**: započítat jen ty, kde je alespoň jedno slovo nebo číslo
    - poslední řádka nemusí být ukončená `' \n '`
  - **věta**: neprázdná posloupnost **slov** ukončená oddělovačem
    - oddělovače vět jsou `' . '`, `' ! '`, `' ? '`
    - `' ... '` ani `' 31.12.2021 '` nejsou tři věty
  - spočítat z `cin` nebo ze **všech** souborů uvedených na příkazové řádce
    - žádné číslo/slovo/věta/řádek nejde přes hranici souboru
  - dekompozice, objektovost, modularita, efektivita
    - elegantní a efektivní rozhraní třídy pro vstup (data) a výstup (výsledky)
    - nemíchat výpočet vs. I/O
  - (samozřejmě) GitLab

znaku: 999  
slov: 999  
vet: 999  
radku: 999  
cisel: 999  
soucet: 999



# Inline a ne-inline metody

```
class Trida
{
    string slozitaFce( int x);
    string getResult () { return r; }
    int inlineFce( int x)
    { int y = -1;
      for( i = 0; i < 10; ++i)
      ....
    }
};
```

trida.h

inline metoda  
rozvinutí místo volání  
s = q.r

```
#include "trida.h"

{
    Trida q;
    string s;
    s = q.getResult();
    s = q.slozitaFce( 1);
    int z = q.inlineFce( 2);
}
```

předání parametrů a volání  
push 1  
s = call q.slozitaFce  
add esp, 8

```
#include "trida.h"

string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i < ...; ++i) {
        ....
    }
}
```

trida.cpp

~~push 2~~  
~~y = -1~~  
~~i = 0~~  
~~loop:~~  
~~if( i >= 10) goto ...~~  
~~...~~  
~~++i~~  
~~goto loop~~

ale: šablony!  
nutná definice při kompilaci  
⇒ vše v headeru

# Inicializace třídy

```
class Trida {  
public:  
    Trida() { x_ = 0; }  
private:  
    int x_;  
};
```

kód konstrukturu  
přiřazení !

kopírování referencí  
přiřazení nelze!

```
class Trida {  
public:  
    Trida( Y& y ) { x_=0; y_=y; ... }  
private:  
    int x_;  
    Y& y_;  
};
```

totéž pro const

```
class Trida {  
public:  
    Trida() : x_(0) { }  
private:  
    int x_;  
};
```

seznam  
inicializátorů

inicializace - OK

```
class Trida {  
public:  
    Trida( Y& y ) : x_(0), y_(y) { }  
private:  
    int x_;  
    Y& y_;  
};
```

```
class Trida {  
public:  
    Trida() { .... }  
private:  
    int x_ { 0 };  
};
```

direct initialization

☹ inicializace na  
různých místech

```
class Trida {  
public:  
    Trida( Y& y ) : y_(y) { .... }  
private:  
    int x_ { 0 };  
    Y& y_;  
};
```

# Kontejnery

# Kontejnery a iterátory

- sekvenční kontejnery
  - **vector** - pole prvků s přidáváním zprava
  - **deque** [dek] - double-ended queue
  - **list, forward\_list** - obousměrně / jednosměrně vázaný seznam
  - **array** - pole pevné velikosti

```
vector<int> x;  
list<string> y;  
array<MyClass, 8> z;  
map<string,int> m;
```

- asociativní kontejnery
  - **setříděné** - dle operátoru <
    - **set/multiset**<V> - množina / s opakováním
    - **map/multimap**<K,V> - asociativní pole / relace
  - **nesetříděné** - hash table, vyhledávání pouze ==
    - **unordered\_set/multiset/map/multimap**

kontejnery obsahují  
vždy hodnoty  
vlození = kopie

- iterátor
  - odkaz na prvky kontejneru + operátory
  - *kontejner*<T>::iterator, **const\_iterator**
  - **k.begin(), cbegin, end, cend** - iterátor na začátek / **za(!)** konec kontejneru
  - **\*it, it->x** - přístup k prvku/položce přes iterátor
  - **++it** - posun na následující prvek

# Základní práce s kontejnery

```
#include <vector> .. list, map, ..  
vector<int> pole { 0, 10, 20 };  
pole.push_back( 30);  
x = pole[3];  
x = pole[99]  
for( auto&& x : pole)  
    x *= 2;
```

initializers

přidání za konec

kontrola mezí !

cyklus

```
string s;  
cin >> s;
```

jednoduché načtení  
jednoho 'slova'  
zkontrolovat!

přidání do mapy  
vytvoření pairu

iterator - typová dedukce  
*map<string,int>::const\_iterator*

vyhledání a změna  
nebo vložení hodnoty

```
#include <map>  
map<string,int> m;  
m.insert( pair{ "jedna", 1});  
auto it = m.find( "jedna");  
if( it != m.end())  
    *it = ....;  
m["dva"] = 2;  
cout << it->first << it->second;
```

kontrola!!!

přístup k prvku  
přes iterátor

prvkem mapy  
je pair

spočítejte frekvence slov  
(data: cpp / cin / argv)  
funkčnost ~> třída ~> rozhraní ~> I/O

# Sekvenční kontejnery

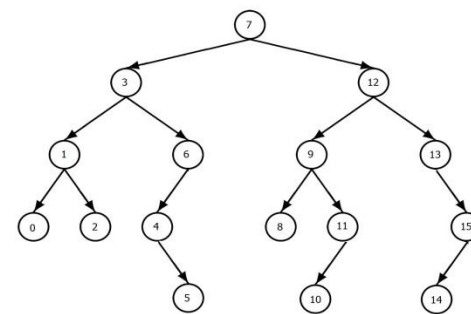
- **vector** - pole prvků s přidáváním zprava
  - celočíselně indexováno, vždy od 0
  - všechny prvky umístěny v paměti **souvisle** za sebou
  - při přidání možná změna lokace  $\rightsquigarrow$  **neplatnost iterátorů a referencí!**
  - odvozené: queue, stack, priority\_queue
- **deque** [dek] - fronta s přidáváním a odebíráním z obou stran
  - double-ended queue
    - lze přidávat i zepředu
  - libovolný rozsah indexů
  - prvky nemusejí být umístěny v paměti souvisle
    - přidávání neinvaliduje reference 😊
- **forward\_list** - jednosměrně vázaný seznam
- **list** - obousměrně vázaný seznam
  - vždy zachovává umístění prvků
  - nepodporuje přímou indexaci []
  - vkládání doprostřed
- **array** - pole pevné velikosti
- **basic\_string** - string, wstring

[dekjú]  $\approx$  dequeue  
*odebrat z fronty*

# Asociativní kontejnery

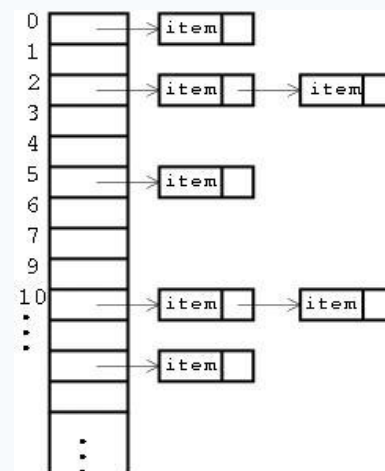
- **setříděné**

- setříděné podle operátoru <
  - pro neprimitivní typy (třídy) nadefinovat operator<
- **set**<V> - množina
- **multiset**<V> - množina s opakováním
- **map**<K,V> - asociativní pole
  - it = find(K)
- **multimap**<K,V> - relace s rychlým vyhledáváním podle klíče K
  - it1 = upper\_bound(K), it2 = lower\_bound(K)
- **pair**<A,B> - pomocná šablona - uspořádané dvojice
  - položky first, second
  - šablona funkce make\_pair( f,s)



- **nesetříděné**

- **unordered\_set/multiset/map/multimap**
- hash table - neseříděné, vyhledávání pouze na ==
- pro neprimitivní typy (třídy) nadefinovat
  - porovnání: bool **operator==** ( const X&)
  - hashovací funkci: size\_t **hash<X>**( const X &)



# Základní metody kontejnerů

- jednotné rozhraní nezávislé na typu kontejneru
  - **!!** ne všechny kontejnery podporují vše
- vkládání
  - `push_back(T)`, `push_front(T)` přidání prvku na konec / začátek - copy/move
  - `emplace_back(par)`, `emplace` konstrukce prvku na místě (v kontejneru)
  - `insert (T)`, `(it, T)` vložení prvku, před prvek
  - `insert (it, it b, it e)` vložení interval z jiného kontejneru
  - `insert( make_pair(K,T))` vložení do mapy - klíč, hodnota
- přístup k prvkům
  - `front()`, `back()` prvek na začátku / konci
  - `operator[]`, `at()` přímý přístup k prvku
    - bez kontroly / s kontrolou (výjimka)
  - `find(T)` vyhledání prvku
  - `upper_bound`, `lower_bound` hledání v multisetu/multimapě
- další
  - `size()`, `empty()` velikost / neprázdnost
  - `pop_front()`, `pop_back()` odebrání ze začátku / konce
    - nevrací hodnotu, jen odebírá!
  - `erase(it)`, `erase(it b, it e)` smazání prvku, intervalu
  - `clear()` smazání kontejneru

*... and many **many** others*



# Iterátory

- iterátor

- objekt reprezentující odkazy na prvky kontejneru
- operátory pro přístup k prvkům
- operátory pro procházení kontejneru

- deklarace

- `kontejner<T>::const_iterator`
- `kontejner<T>::iterator`
- `auto it = ....`

iterátor vždy typovaný

konstantní iterátor - **používejte!**

(mutabilní) iterátor příslušného kontejneru

typová dedukce - používejte všude, kde lze

- vytvoření

- `k.begin()`, `cbegin`, `end`, `cend`

iterátor na začátek / **za(!)** konec kontejneru

- operátory

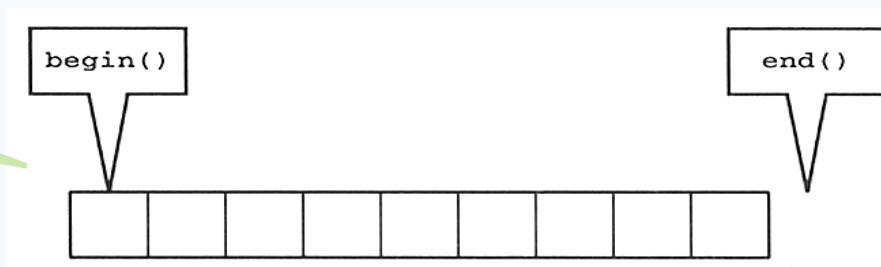
- `*it`, `it->x`
- `++it`
- `+(int)` `-(int)`

přístup k prvku/položce přes iterátor

posun na následující prvek

posun iterátoru

polootevřený  
interval



# Překladačový slovník (basic)

*k jednomu slovu může být více překladů*

## ▶ operace

- přidat slovo a jeho překlad
  - akceptovat **více** překladů jednoho slova
- nalézt všechny překlady slova
- odebrat jeden překlad slova
- odebrat všechny překlady slova

```
add( slovo, cizi)
?? find( slovo) // -> cizi cizi cizi
del( slovo, cizi)
del( slovo)
```

## ▶ k rozmyšlení

- kontejner(y) pro ukládání dat
- jak 'vracet' více slov
  - kontejner hodnot ✖ rozmezí
  - perzistentní ✖ tranzientní

## ▶ rozhraní

- API - public metody
- volání s přímými daty z mainu

it m.lower_bound( val);	val >= *it
it m.upper_bound( val);	val < *it

hledání  
v multi- kontejnerech

# Pojmenování typů

```
map<string,int> frekvence;  
map<string,int>::const_iterator it;  
fce( map<string,int>& fr);
```

neopisujte stále  
deklarace !

```
using Frekvence = map<string,int>;  
typedef map<string,int> Frekvence;
```

```
Frekvence::const_iterator it;  
fce( const Frekvence& fr);
```

scope - oblast platnosti:  
globální, třída, funkce/metoda

Proč:

- neupíšu se
- změna druhu nebo typu
- čitelnost
- rozlišení logicky různých typů

```
class Trida {  
    using Frekvence = map<string,int>;  
    fce( const Frekvence& fr);  
};
```

```
Trida::Frekvence f;
```

# Procházení kontejnerů

```
vector<int> pole;  
vector<int>::const_iterator i;  
for( i = pole.cbegin(); i != pole.cend(); ++i)  
    cout << *i;
```

cyklus s iterátory

```
vector<int> pole;  
for( size_t i=0; i<pole.size(); ++i)  
    cout << pole[i];
```

```
vector<int> pole;  
for( auto i = pole.cbegin(); i != pole.cend(); ++i)  
    cout << *i;
```

auto  
typová dedukce

range-based for  
pouze pro celý kontejner

```
vector<int> pole;  
for( auto&& x : pole)  
    cout << x;
```

kopie !

```
for( auto x : pole)
```

```
map<string,int> mapa;  
for( auto&& x : mapa)  
    cout << x.first << x.second;
```

prvkem mapy je pair  
vždy first, second

structural bindings

Project / Properties / C++ / Language /  
Standard / ISO C++17 (nebo latest)

```
map<string,int> mapa;  
for( auto&& [key, value] : mapa)  
    cout << key << value;
```

# Složitost operací

složitost	přidání / odebrání na začátku	přidání / odebrání na i-té pozici	přidání / odebrání m prvků	přidání / odebrání na konci	nalezení i-tého prvku
funkce	push_front pop_front	insert erase	insert erase	push_back pop_back	begin() + i [i]
list	konst	konst	m, konst přesuny mezi sezn. (splice)	konst	neex
deque	konst	min( i, n - i)	m + min( i, n - i)	konst	konst
vector	neex	n - i	m + n - i	konst (*)	konst
asocia tivní	ln	(s klicem k) ln	(s klicem k) ln + m	ln	nalezení podle hodnoty ln
unsorted	konst	konst	m	konst	konst

fyzická velikost: capacity() ↔ reserve()  
logická obsazenost: size() ↔ resize()

při překročení kapacity rozšíření  
a kopie stávajících prvků

# Konstruktory a vkládání do kontejneru

```
class MyClass {  
    MyClass( X x, Y y);  
    MyClass(const  MyClass& mc);  
    MyClass(MyClass&& mc) noexcept;  
    MyClass& operator=(const MyClass& mc);  
    MyClass& operator=(MyClass&& mc) noexcept;  
    ~MyClass();  
};
```

typická sada konstruktorů

```
vector<MyClass> v;  
MyClass m{ x, y };
```

push	emplace		
v.push_back( m)	v.emplace_back( m)	(ctor) <b>copy_ctor</b>	☹️
v.push_back( MyClass{x,y})	v.emplace_back( MyClass{x,y})	ctor, move_ctor, dtor	😐
	v.emplace_back( x,y)	ctor	😊

efektivita!

# Funktory

```
class ftor {  
public:  
    ftor( int step) : step_(step), qq_(0) {}  
    int operator() (int& x) { return x += (qq_ += step_); }  
private:  
    int step_;  
    int qq_;  
};  
for_each( v.cbegin(), v.cend(), ftor{2});
```

Přičíst ke všem prvkům  
+n, +2n, +3n, ...

Funktor – třída s  
přetíženým operátorem ()

oddělení inicializace  
a běhového parametru

anonymní instance

najít v kontejneru  
prvek větší než n

```
it = find_if( bi, ei, fnc);
```

```
class cmp {  
public:  
    cmp( int n) : n_(n) {}  
    bool operator() (int& x) { return x > n_; }  
private:  
    int n_;  
};
```

```
auto fnd = find_if( v.cbegin(), v.cend(), cmp{9});  
cout << ((fnd == v.end()) ? -1 : *fnd);
```

# Lambdy

- najděte všechny prvky větší než 9
  - binder, funktor, lambda

spousta kódu  
i pro triviality

```
class greater_than {  
public:  
    greater_than( int val) : val_(val)) {}  
    bool operator() (int& x) { return x > val_; }  
private:  
    int val_;  
};
```

```
find_if( v.begin(), v.end(), bind2nd( greater<int>(), 9));  
bind( &greater<int>::operator(), placeholders::_1, 9));
```

```
find_if( v.begin(), v.end(), greater_than{ 9});
```

```
find_if( v.cbegin(), v.cend(), [](int& x) { return x > 9; });  
find_if( v.cbegin(), v.cend(), [](auto x) { return x > 9; });
```

binder  
obsolete

vlastní  
funktör

lambda

lambda type inference C++17

- 😊 mnohem jednodušší zápis a syntaxe
- 😞 složitější logika → plnohodnotný funktor



# Lambdy - syntax

- Syntax

```
[ captures ] ( params )opt mutableopt -> rettypeopt { statements; }
```

- **[ captures ]**

- access to external local variables - **initialization**
- explicit/implicit, by-value/by-reference, generalized

- **( params )**

- call parameters
- optional but usual

- **mutable**

- local copy of external variable can be modified

- **-> rettype**

- return type, "new" syntax
- optional - compiler deduces type from expression
- template type deduction

- **{ statements; }**


- lambda body

?

```
[ ] ( ) { }  
[ ] { } ( )
```

# Lambda = funktor

```
[ captures ] ( params ) -> rettype { statements; }
```



```
class ftor {  
private:  
    CaptTypes captures_;  
public:  
    ftor( CaptTypes captures ) : captures_( captures ) {}  
    rettype operator() ( params ) { statements; }  
};
```

```
std::vector<int> v { .... };  
  
size_t compCounter = 0;  
sort( v.begin(), v.end(), [&compCounter](int a, int b) {  
    ++compCounter; return a<b; });  
  
cout << "number of comparisons: " << compCounter << '\n';
```

# Kontejnery a třídění - vector, list, set

```
#include <vector>
#include <algorithm>

string s;
vector<string> v;
for(;;) {
    cin >> s;
    if( cin.fail())
        break;
    v.push_back(s);
}
sort( v.begin(),v.end());
```

```
list<string> v;
for(;;) {
    cin >> s;
    if( cin.fail())
        break;
    for( auto i = v.begin(); i != v.end() && *i <= s; ++i)
        ;
    v.insert( i, s);
}
```

```
string s;
set<string> v;
for(;;) {
    cin >> s;
    if( cin.fail())
        break;
    v.insert(s);
}
```

jak to setřídít? 😊

## ► dva problémy

- chci jiné setřídění než standardní
  - např. řetězce primárně dle délky
- kontejner složených typů
  - není na něm definováno standardní porovnání - operator <
  - struktury, objekty, ...

## ► řešení - vlastní komparátor

- operator<
- externí komparátor - funkce / funktor / lambda

# Třídění - vlastní kritéria

## ▶ vlastní komparátor

### ◦ operator<

- 😊 lze u třídící funkce i šablony kontejneru
- 😞 jen jeden, nelze měnit pro primitivní typy

přetížení operátoru  
 $a \text{ ⌘ } b \equiv a.operator\text{⌘}(b)$

```
class T {  
    string s; int i;  
    bool operator<( const T& y) const  
        { return i<y.i || (i == y.i && s<y.s); }  
};  
  
set<T> v;  
v.insert( T {"jedna", 1});
```

### ◦ externí komparátor - funkce

- 😊 může jich být několik
- 😞 nelze jako parametr šablony kontejneru
  - parameter šablony musí být typ
- funkce není typ

```
bool mysort( const string& s1, const string& s2) {  
    return s1.size() < s2.size() ? true :  
        (s2.size() < s1.size() ? false : s1<s2);  
}  
  
vector<string> v;  
sort( v.begin(),v.end(), mysort);
```

# Třídění - vlastní kritéria

## ▶ vlastní komparátor

### ◦ externí komparátor - funktor

- 😊 nejobecnější, může jich být několik
- 😐 malinko složitější

cmp x;  
x(t1,t2);

```
class T { string s; int i; };  
  
struct cmp {  
    bool operator()( const T& x, const T& y) const  
    { return x.i < y.i || .....; }  
};  
  
set<T, cmp> v;  
v.insert( T{"jedna", 1});
```

funktor

### ◦ externí komparátor - lambda

- 😊 kompaktnější než funktor
- 😐 trochu pokročilejší syntaxe

```
auto cmp = [](const string& s1, const string& s2) { return .. };  
set< T, decltype(cmp)> v;
```

typ lambdy

# Filmová databáze, překladový slovník full

- ▶ filmová databáze
  - struct/class název filmu, režisér, rok, ...
  - seříděné dle **roku** a **názvu** filmu
  - neřešte vstup - jen API
  - vyzkoušejte všechny komparátory

- ▶ slovník

- přidat slovo a jeho překlad
  - akceptovat **více** překladů jednoho slova
- nalézt všechny překlady slova
- odebrat jeden překlad slova
- odebrat všechny překlady slova
- **nalézt všechny překlady slov začínajících prefixem**
- **nalézt slovo když znáte překlad**
- **vypsát překlady seříděné podle délky**

```
add slovo cizi  
find slovo -> cizi cizi cizi  
del slovo cizi  
del slovo
```

```
pfind slovo ->  
  slovxxx cizi cizi  
  slovyyy cizi  
  slovzzz cizi cizi  
  
rfind cizi -> slovo slovo
```

- ▶ rozhraní

- API - public metody

- ▶ ... až po odladění "apka"

- UI - console

# Makroprocesor

- jednoduchý makroprocesor



- vstup: text #novemakro obsah makra # dalsi novemakro konec
- výstup: text dalsi obsah makra konec
- vstup: zz #m1 xx # #m2 yy m1 # m2
- výstup: zz yy xx

pozor - v definici makra může být  
obsažena hodnota jiného makra

- definice jednoho makra na příkazové řádce
  - makroproc mojemakro obsah meho makra az do konce cmdln

# Makroprocesor

- všechny ostatní znaky (kromě definice a vyvolání makra) → cout
- výstup white spaces v okolí definice makra:
  - `ws1definice_makraws2` → `ws1ws2`
  - (definice makra se jednoduše vypustí)
- identifikátor
  - posloupnost `isalnum` začínající `isalpha`
  - např. `2A` není identifikátor, `X$B1` je identifikátor `X` a identifikátor `B1`
- oddělovače
  - oddělovače `#nazevmakra` a `#` jsou `isspace`
  - `#` na vstupu za jiným znakem než `isspace` se chová jako běžný znak
- rozhraní
  - API → `cin(+argv) / cout`
- ošření nekorektního vstupu
  - např. definice makra uvnitř jiné definice, dva konce definice makra za sebou apod.
  - výstup až do posledního korektního znaku vstupu, dále na výstup **Error**↵
  - podrobnější chybovou diagnostiku lze vypsát na `cerr` (není kontrolováno)
- stabilita
  - na žádný (ani nekorektní) vstup nesmí program 'odletět', ani se chovat nedefinovaně



# Algoritmy

---

funktory, lambda

# Nejpoužívanější algoritmy

#include <algorithm>

- **it find**( it first, it last, T&)
  - asociativní kontejnery: k.find(T&)
- **int count**( it first, it last, T&)
- **for\_each**( it first, it last, fnc( T&))
- **sort**( it first, it last, sort\_fnc(x&, y&))
- **copy**( it first, it last, output\_it out)
- **transform**( it first, it last, it out, fnc( T&))
- **transform**( it first, it last, it **first2**, it out, fnc( T&x, T&y))
  - vkládání za konec kontejneru: back\_inserter( kont)
- **find\_if**, **count\_if**, **remove\_if**( it first, it last, **pred&** p)
- **remove**, **remove\_if** - přesun (move-assignment) na konec, **nic nemaže!**
- **unique** - zjednoznačnění - přesun **následných** duplicit na konec
- **kontejner.erase** - skutečné smazání

specializovaná metoda rychlejší

funkce modifikuje argument

vrátí kopii funktoru  
získání výsledku

vrací modifikovaný argument  
možnost jiného kontejneru

spojování

predikát: bool fnc( const T&)

stejná hodnota nebo predikát

parametry, přesná sémantika, další algoritmy:  
<https://en.cppreference.com/w/cpp/algorithm>

podívejte se!

# Algoritmy - použití

```
#include <algorithm>
vector<int> v { 1, 3, 5, 7, 9 };
// vector<int>::const_iterator result;
auto result = find( v.cbegin(), v.cend(), 5);
```

```
bool greater10( int value ) {
    return value>10;
}

result = find_if( v.cbegin(), v.cend(), &greater10);
if( result == v.cend())
    cout << "Nothing";
else
    cout << "Found:" << *result;
```

predikát

jak parametricky?

vždy otestovat!

```
void mul2( int& x)
{
    x *= 2;
}
```

jak zrestartovat?  
jak krok parametricky?

```
for_each( begin, end, fnc( T&))
// vynásobit všechny prvky 2

// přičíst ke všem prvkům +1, +2, +3, ...
```

```
int fce( int& x) {
    static int qq = 0;
    return x += (qq +=1);
}

for_each( v.cbegin(), v.cend(), fce);
for_each( v.rbegin(), v.rend(), fce);
```

# Algoritmy a funktory

Přičíst ke všem prvkům  
+n, +2n, +3n, ...

najít v kontejneru  
prvek větší než n

součet všech čísel  
větších než parametr

jak získat výsledek?

po skončení  
hodnota použitého funktoru

pozor!  
nejde o identický objekt

```
it = find_if( bi, ei, fnc);

class cmp {
public:
    cmp( int n) : n_(n) {}
    bool operator() (int& x) { return x > n_; }
private:
    int n_;
};

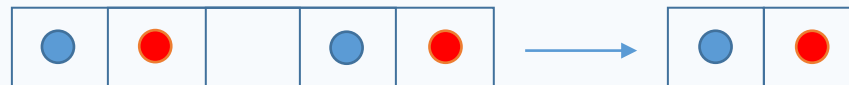
auto fnd = find_if( v.cbegin(), v.cend(), cmp{9});
cout << ((fnd == v.end()) ? -1 : *fnd);
```

```
class scitacka {
public:
    scitacka( int limit) : limit_(limit), vysledek_(0) {}
    int operator() (int& x) { if( x > limit_) vysledek += x; }
    int vysledek_;
private:
    int limit_;
};

auto s = for_each( v.cbegin(), v.cend(), scitacka{10});
cout << s.vysledek_;
```

# Příklady na algoritmy, funktory a lambdy

- 1  
vektor čísel  $\Rightarrow$  multiset čísel větších než **X** inkrementovaných o **Y**
- 2  
najít (první) prvek odlišný od předchozího alespoň o  $n$
- 3  
inkrementovat čísla v zadaném **rozsahu hodnot**  
(první  $+n$ , druhé  $+2n$ , ...)
- 4  
najít číslo za největší dírou (rozdíl sousedních hodnot)
- 5  
součet druhých mocnin první a druhé poloviny vektoru do jiného kontejneru

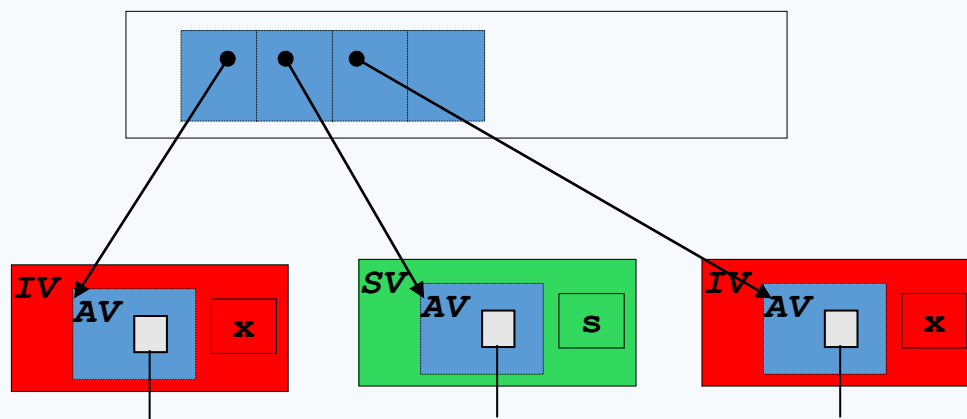


# Polymorfní struktury

# Polymorfní datové struktury

- problém
  - kontejner obsahující hodnoty libovolného typu
  - int, double, string, complex, zlomky, ...
- technické upřesnění
  - třída Seznam, operace add, print
  - společný předek prvků **AbstractVal**
  - konkrétní prvky **IntVal**, **StringVal**, ...
  - stačí jednoduchá implementace vektorem
  - pole objektů vs. pole 'odkazů'

jiné možnosti:  
structural typing  
type erasure  
variant  
☞ Advanced C++



# Polymorfní struktury - základní idea

```
class AbstractVal {  
public:  
    virtual void print() = 0;  
};  
  
using Valptr =  $\Psi$ <AbstractVal>;
```

typ odkazu

abstraktní předek  
umí existovat a vytisknout se

```
class Seznam {  
public:  
    void add( Valptr p);  
    void print();  
private:  
    vector<Valptr> pole_  
};
```

vektor odkazů

```
int main() {  
    Seznam s;  
    s.add(  $\Psi$ <IntVal>(123) );  
    s.add(  $\Psi$ <StringVal>("abc") );  
    s.print();  
}
```

použití

## • $\Psi$ ?

- AbstractVal \*
- AbstractVal &
- unique\_ptr<AbstractVal>
- shared\_ptr<AbstractVal>
- iterator
- ... ?



# Polymorfní struktury - implementace

```
#include <memory>
class AbstractVal;
using Valptr = unique_ptr<AbstractVal>;
```

unique\_ptr ≈  
vlastnictví objektu

...  
templates  
variadic templates

```
class Seznam {
public:
    void add( Valptr p) { pole.push_back( move( p)); }
    void print() { for(auto&& x : pole_) x->print(); }
private:
    vector<Valptr> pole_;
};
```

proč '->' ?

```
int main() {
    Seznam s;
    s.add( make_unique<IntVal>(123));
    s.add( make_unique<StringVal>("456"));
    s.print();
}
```

konstruktory?  
destruktory?

# Polymorfní struktury - konkrétní datové typy

```
class IntVal : public AbstractVal {  
public:  
    IntVal( int x) : x_( x) {}  
    virtual void print() { cout << x_; }  
private:  
    int x_;  
};
```

```
class StringVal : public AbstractVal {  
public:  
    StringVal( string x) : x_( x) {}  
    virtual void print() { cout << x_; }  
private:  
    string x_;  
};
```



what's the difference?

```
class DoubleVal : public AbstractVal;  
class ComplexVal : public AbstractVal;  
class LongintVal : public AbstractVal;  
class FractionVal : public AbstractVal;
```

# Polymorfní struktury - přiřazení

```
int main() {  
    Seznam s1, s2;  
    s1.add( make_unique<IntVal>(123));  
    s1.add( make_unique<StringVal>("456"));  
    s2 = s1;  
    s2.print();  
}
```

čím je to zajímavé?

compiler error:  
XXXX unique\_ptr XXX attempting to reference a deleted function

```
class Seznam {  
    ....  
    Seznam( const Seznam& s) = delete;  
    Seznam& operator=(const Seznam& s) = delete;  
};
```

možné řešení: zakázat !!!

copy constructor  
a operator=  
by se měly chovat stejně

```
Seznam& Seznam::operator=(const Seznam& s)  
{  
    for( auto&& x : s.pole_)  
        pole_.push_back( x);  
    return *this;  
}
```

ale co když  
přiřazení potřebuju?

compiler error:  
XXXX unique\_ptr XXX attempting to reference a deleted function

# Polymorfní struktury - make\_unique

```
Seznam& Seznam::operator=(const Seznam& s)
{
    for( auto&& x : s.pole_)
        pole_.push_back( make_unique<..>( *x));
    return *this;
}
```

nechci kopírovat ukazatel  
chci vytvořit nový objekt

jaký typ použít?

motivace

```
int main() {
    Seznam s;
    s.add( make_unique<IntVal>(123));
    s.add( make_unique<StringVal>("abc"));
    s.print();
}
```

```
Seznam& Seznam::operator=(const Seznam& s)
{
    for( auto&& x : s.pole_)
        pole_.push_back( make_unique<AbstractVal>( *x));
    return *this;
}
```

compiler error:  
cannot instantiate abstract class

.... 4:30 v noci, ráno je deadline

tak tu abstraktnost zrušíme!

... a ono se to konečně zkompiluje!

```
class AbstractVal {
public:
    virtual void print() = 0;
};
```

```
class AbstractVal {
public:
    virtual void print() {}
};
```



# Polymorfní struktury - slicing

- je to správně?

- **není !!**

- **slicing**

- pouze **část** objektu

- společný předek

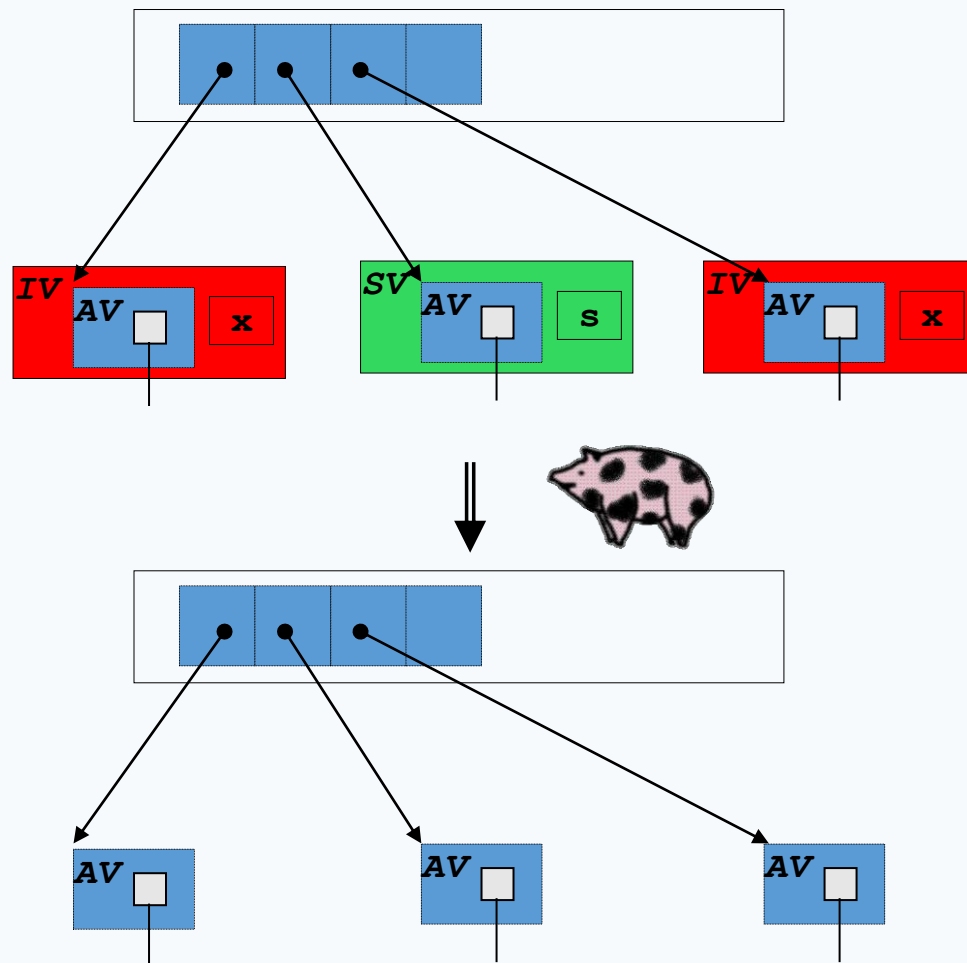
- horší chyba než předchozí případ!

- ☹ projde kompilátorem

- ☹ nespadne

- ☹☹ dělá **nesmysly!**

```
for( auto&& x : s.pole_ )  
    pole_.push_back( make_unique<AbstractVal>( *x));
```



# Polymorfní struktury - kopie podle typu

- co s tím?
  - skutečná hodnota **IntVal**  
⇒ vytvořit **IntVal**
  - skutečná hodnota **StringVal**  
⇒ vytvořit **StringVal**

```
class AbstractVal {  
public:  
    enum T { T_INT, T_STRING, ...};  
    virtual T get_t() const;  
};
```

```
Seznam& Seznam::operator=(const Seznam& s)  
{  
    for( auto&& x : s.pole_) {  
        switch( x->get_t()) {  
            case AbstractVal::T_INT:  
                pole_.push_back( make_unique<IntVal>( *x));  
                break;  
            case AbstractVal::T_STRING:  
                pole_.push_back( make_unique<StringVal>( *x));  
                break;  
        }  
    }  
    return *this;  
}
```

**FUJ !!!**

- ▶ ošklivé
- ▶ těžko rozšiřitelné
- ▶ zásah do předka
- ▶ výhybky plné kufrů
- ▶ **toto není polymorfismus!**

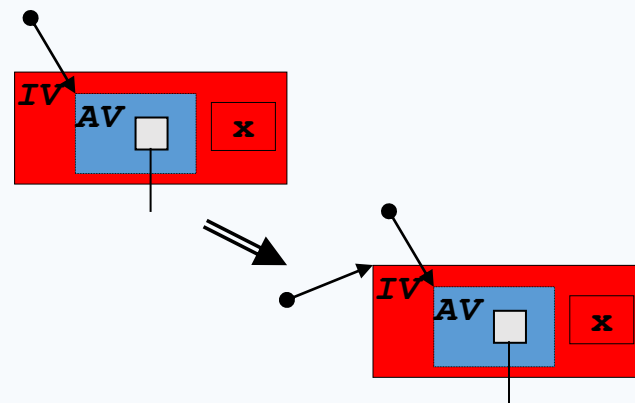
# Polymorfní struktury - klonování

- jak to udělat lépe?
  - využít mechanismus pozdní vazby
  - každý prvek bude umět naklonovat sám sebe
  - rozhraní v `AbstractVal`, implementace v `IntVal`, ...
  - virtuální **klonovací** metoda

```
class AbstractVal {  
public:  
    virtual void print() = 0;  
    virtual valptr clone() = 0;  
};  
  
class IntVal : public AbstractVal {  
    ....  
    virtual valptr clone() override  
    { return make_unique<IntVal>(*this); }  
};
```

kovariantní návratový typ

```
... operator=(const Seznam& s)  
{  
    for( auto&& x : s.pole_)  
        pole_.push_back( x->clone());  
    return *this;  
}
```



# Polymorfní struktury - copy constructor

- copy-constructor a operator=
  - společné chování
  - operator= navíc úklid starého stavu, vrací referenci
  - společné tělo

```
class Seznam
{
public:
    ....
    Seznam() {}
    Seznam( const Seznam& s) { clone( s); }
    Seznam& operator=(const Seznam& s) { pole_.clear(); clone( s); return *this; }
private:
    void clone( const Seznam& s)
        { for( auto&& x : s.pole_) pole_.push_back( x->clone()); }
    vector< valptr> pole_;
};
```

naklonování  
celého seznamu

naklonování  
jednoho prvku



# Polymorfní struktury - self-assignment

čím je to zajímavé?

```
class Seznam
{
public:
    ....
    Seznam& operator=(const Seznam& s)
    { pole_.clear(); clone( s); return *this; }
};
```

nejdřív si sám celé pole smažu  
... a potom nakopíruju ... .. NIC!

rovnost ukazatelů ⇒ stejný objekt

```
Seznam& operator=(const Seznam& s)
{
    if( this == &s) return *this;
    pole_.clear();
    clone( s);
    return *this;
};
```

```
int main() {
    Seznam s;
    s.add( make_unique<IntVal>(123));
    s.add( make_unique<StringVal>("abc"));
    s = s;
}
```

takhle blbě by to asi  
nikdo nenapsal, ale....

```
int main() {
    vector<Seznam> s;
    ....
    s[i] = s[j];
}
```

- je třeba *trocha* opatrnosti  
... a rozumět tomu, co se v programu děje
- naimplementujte sami  
= jen dát dohromady předchozí moudra
- k rozmyšlení  
sémantika (chování) při použití `shared_ptr`

# Šablony

---

templates

# Šablony

```
class Scitacka
{
public:
    Scitacka() : val_( 0) {}
    void add( int x);
    int result() { return val_; }
private:
    int val_;
};
```

x.h

```
void Scitacka::add( int x)
{ val_ += x; }
```

x.cpp

```
int main()
{
    Scitacka s;
    s.add( 1);
    s.add( 2);
    auto x = s.result();
}
```

main

hlavička  
šablony

hlavička i u  
definice těla

tělo v headeru

použití  
instanciace

šablona těla  
musí být při  
kompilaci  
viditelná

scitacka.h

```
template<typename T> class Scitacka
{
public:
    Scitacka() : val_( 0) {}
    void add( T x);
    T result() { return val_; }
private:
    T val_;
};
```

```
template <typename T>
void Scitacka<T>::add( T x)
{ val_ += x; }
```

```
#include "Scitacka.h"

int main()
{
    Scitacka<unsigned long long> s;
    s.add( 1);
    s.add( 2);
    auto x = s.result();
}
```

# Šablony a operatory

```
class S
{
public:
    friend S operator+
        ( const S& x, const S& y);
private:
    int val_;
};

S operator+( const S& x, const S& y)
{ return S { x.val_ + y.val_; }

int main()
{
    S a, b, c;
    c = a + b;
}
```

globální funkce  
není metoda třídy

```
template<typename T> class S
{
public:
    S<T>( T val) : ....
    template<typename X>
    friend S<X> operator+
        ( const S<X>& x, const S<X>& y);
private:
    T val_;
};

template<typename X>
S<X> operator+( const S<X>& x, const S<X>& y)
{ return S<X> { x.val_ + y.val_; }
}
```

scitacka.h

friend template  
jiná šablona

- naprogramujte
  - zlomek<T> se sčítáním a přiřazením
  - sčítačka <zlomek>, sčítačka <string>
  - polymorfní ConcreteVal<T> pomocí šablon

```
#include
"Scitacka.h"

int main()
{
    S<long> a, b, c;
    c = a + b;
}
```

# Gumové pole

► problém

- `std::vector` nezachovává umístění prvků
- přidání → invalidace referencí, iterátorů, ...
- vynuceno požadavkem na spojitě uložené prvky

```
x.push_back(n)
x[i]
```

► chci

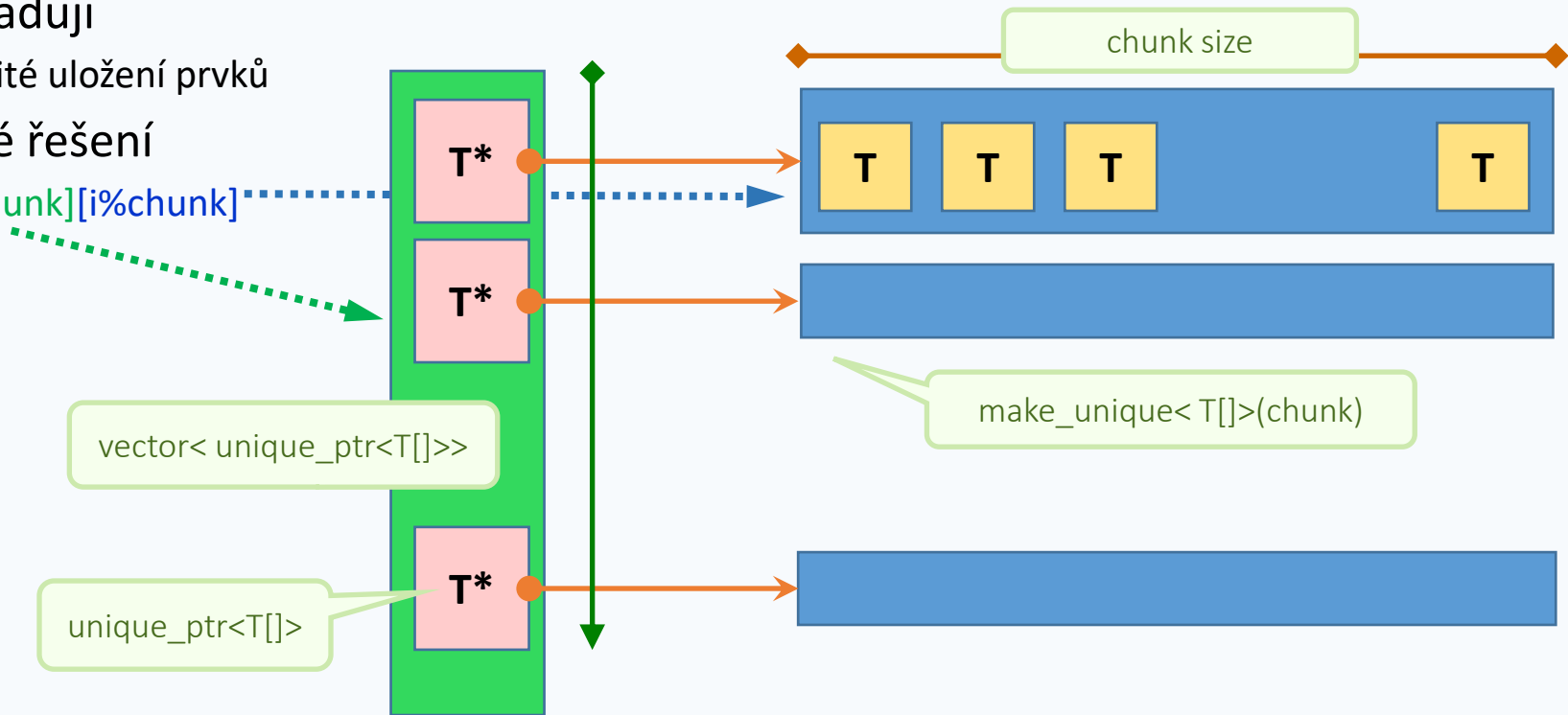
- datová struktura zachovávající umístění
- žádné invalidace
- konstantní časová složitost přístupu k prvkům

- ▶ nevyžadují

- spojité uložení prvků

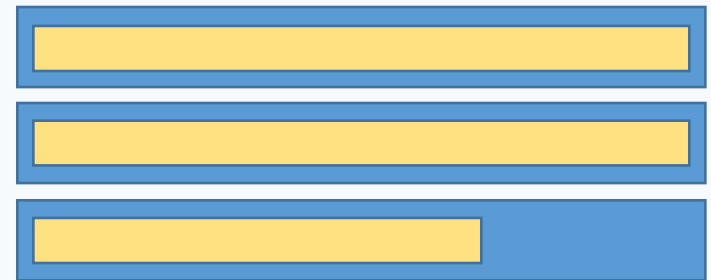
► možné řešení

- `[i/chunk][i%chunk]`



# Gumové pole - deklarace

```
template<typename T> class Pole {  
public:  
    Pole( size_t chunk = 100) : .... {}  
    void push_back( const T& x);  
    T& operator[] ( size_t i) { return ..; }  
    T& at( size_t i) { check(i); return ..; }  
  
private:  
    void check( size_t i);  
    void resize( size_t i);  
    ....  
    vector< unique_ptr<T[]>> hrabe_;  
};
```



logická obsazenost

fyzická velikost

# Gumové pole - iterator

```
template<typename T> class Pole {  
public:  
    void push_back( const T& x);  
    T& operator[] ( size_t i);  
  
    iterator begin() { return iterator{..}; }  
    .... end() { return ....; }  
private:  
    ....  
};
```

stejně jako  
std:: iterátory

```
Pole<xyz>::iterator it = ....  
auto it = p.begin();
```

```
for( auto it = p.begin();  
    it != p.end(); ++it) ....
```

nemusí být  
stejné typy

Pole::iterator

```
class iterator {  
    iterator() : .... {}  
    iterator( const iterator& it) : .... {}  
    iterator( ....) : .... {}  
    T& operator* () { return .... }  
    bool operator != ( ....) { return ....; }  
    iterator operator++ () { ....; return *this; }  
private:  
    ....  
};
```

???

## ▶ iterator:

- \* - dereference prvku
- ++ - inkrementace

## ▶ end()

- různý od  $\forall$  platných iterátorů
  - ... i v budoucnosti!
- nemusí být iterator
  - přetížený operator !=
- na posledním prvku musí platit  $++it == end()$

default konstruktor  
Pole<T>::iterator it;  
it = p.begin();

# Gumové pole - kopie, implementace

## ▶ základ implementace

- `x.push_back(n)`
- `x[i]`

## ▶ iterator

- operator `* != ++`
  - `++` na posledním prvku
- `begin()`, `end()`
  - `end()` -> iterator
  - operator `!=`
- `for( auto&& i : v ) { }`

## ▶ operace s kontejnerem

- liší se od default?
  - copy-constructible
  - move-constructible
  - assignable (`=`)

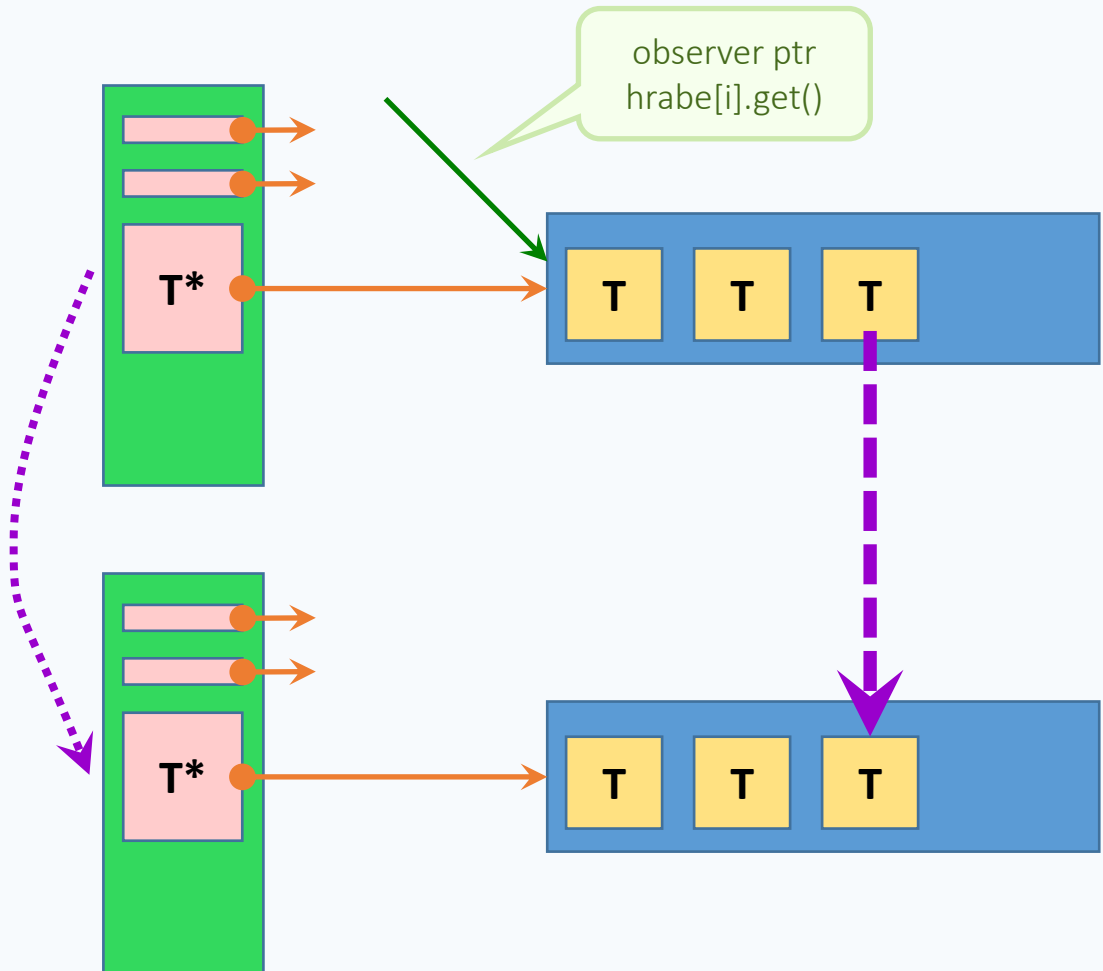
## ▶ řádně otestujte

- všechny funkce / kombinace
- okrajové případy

## ▶ k rozmyšlení

- `const_iterator`
- `Pole<unique_ptr<T>>`

```
Pole<T> a;  
....  
Pole<T> b = a;
```





# Výjimky

---

exceptions

# Výjimky

- vyvolání výjimky
  - try blok
  - nejbližší **vyhovující** catch blok
    - dědičnost
  - **stack unwinding**
    - destrukce *všech* objektů
- dvojitá výjimka
  - výjimka při zpracování výjimky
  - terminate ☠

stack  
unwinding

anonymní  
instance

vyvolá se  
nejspecifičtější

zachytí vše

potomci  
std::exception

```
g() {  
    yy  
    if( error) throw exctype{aa};  
}  
  
f() {  
    xxx  
    g();  
}  
  
try {  
    xxx f() xxx  
} catch( exctype& e) {  
    e.yy();  
} catch( ...) {  
    yy;  
}
```

```
#include <stdexcept>  
  
class exception {  
public:  
    exception();  
    virtual const char *what() const;  
};  
  
bad_alloc, bad_cast, domain_error,  
invalid_argument, length_error, out_of_range,  
overflow_error, range_error, underflow_error  
  
} catch( exception& e) {  
    cout << e.what() << endl;  
}
```

# Výjimky při inicializaci a destrukci

- výjimky v destruktore
  - ☠ **nikdy!**
    - destruktory se volají při obsluze výjimek
- výjimky v konstruktore
  - ☠ **ne globální!**
    - není kde chytit
  - základní třída
    - konstruktor může vyvolat výjimku
  - odvozená třída
    - výjimku inicializace je vhodné zachytit
    - objekt není vytvořen
    - tělo konstrukturu odvozené třídy se neprovede

tělo try bloku je tělem  
konstrukturu

```
class A {  
public:  
    A( X& x) { ... throw ... }  
};  
  
class B : public A {  
public:  
    B( X& x) try : A(x) {  
        ...  
    } catch( ...) {  
    }  
};
```

# Vlastní typ výjimky

```
#include <stdexcept>, <cstdio>
```

```
class myexc : public std::out_of_range {
```

```
public:
```

```
    myexc( int ix) : ix_(ix), s_( "Chyba na indexu:") { s_ += to_string( ix); }
```

```
    virtual const char *what() const override { return s_.c_str(); }
```

```
    int getIndex() const { return ix_; }
```

```
private:
```

```
    int ix_;
```

```
    string s_;
```

```
};
```

vše  
v konstruktoru

žádné výjimky !!!

vlastní diagnostika

kompatibilita

anonymní instance

```
myclass::myfnc() {  
    whatever();  
    if( error_occured)  
        throw myexc{ 17};  
    whatever_else();  
}
```

```
class myexc : public std::out_of_range {  
public:  
    myexc( int ix) : out_of_range( "XXX"), ix_(ix), ....
```

```
try {  
    nejakymujkod  
} catch( myexc& me) {  
    cout << "Chyba indexu: " << me.getIndex();  
} catch( exception& e) {  
    cout << e.what();  
} catch( ...) {  
    cout << "unexpected exception";  
}
```

strcpy, atoi, ... errors  
Properties ▶ C/C++ ▶ General  
▶ SDL checks ▶ No (/sdl-)

# noexcept

- specifikace, že funkce nikdy nevyvolá výjimku
  - kontrola kompilátorem
- lze testovat
  - `std::is_nothrow_move_constructible`
- lze vygenerovat efektivnější kód
  - normou zaručený noexcept
  - vector - realokace
    - noexcept move konstruktor
      - move
    - bez noexcept
      - kopie prvků a jejich smazání
- praktické důsledky
  - nepište vlastní konstruktory/destruktory
    - v ideálním případě stačí automaticky generované
    - kompilátor si noexcept odvodí sám
  - v případě vlastních konstruktorů pište noexcept
    - u move konstruktorů
    - u destruktů

```
class Klasa {  
    Klasa() { .... }  
    Klasa( const Klasa& k) { .... }  
    Klasa( Klasa&& k) noexcept {..}  
    ~Klasa() noexcept { .... }  
};
```

# random

- C: `stdlib rand`
- generators
  - uniformně rozložené hodnoty
  - *linear\_congruential, mersenne\_twister, subtract\_with\_carry, ...*
  - `default_random_engine`
- distributors
  - transform a generated sequence to a particular distribution
  - *uniform, normal, poisson, student, exponential, ...*
- použití: `distributor( generator)`

seed - náhodná inicializace

náhodné číslo 1..6

```
#include <random>
#include <ctime>

default_random_engine generator;
default_random_engine generator{ time(0)};
uniform_int_distribution<int> distrib{1,6};
int dice_roll = distrib( generator);
```

generator je volán distributorem

```
auto dice = bind( distrib, mt19937_64);
auto dice = [](){ return distrib(mt19937_64); };
int wisdom = dice() + dice() + dice();
```

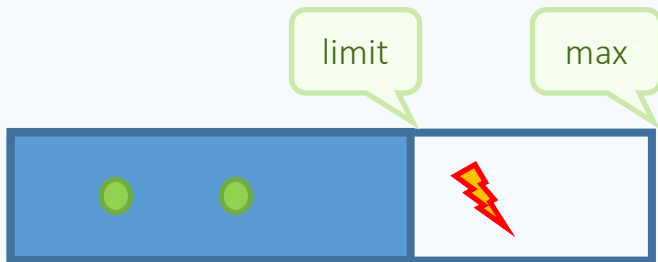
spojení generatoru  
and distributoru

# Výjimky - použití

```
const size_t max = 100;

size_t limit = myrandom( max);
init_up_to( limit);
try {
    for(;;) {
        i = myrandom( max);
        xxx.at(i)
    }
} catch( myexc &e) {
    cout << e.getIndex();
} catch( .....
```

- opravit Gumu
  - ++ vs. end
  - neinicializovaná data
- doplnit
  - at()
    - výjimka: přístup za poslední prvek
  - chráněný iterator
    - výjimka: ++end(), \*end()
- podrobnější diagnostika
  - vlastní typ výjimky
  - špatný index a velikost pole
- vyzkoušet střelbu
  - nějak velké pole - max
  - zaplnit až do limit
  - náhodně střílet až do max
  - chytat výjimky



# Filesystem

---



# filesystem

- platformově nezávislá práce FS
  - paths, files, directories, symlinks/hardlinks, attributes, rights, ...
  - nativní / generalizovaná vnitřní reprezentace
- iterátory nad FS
  - lze rekurzivní
  - directory entry
- operace se soubory
  - copy, remove, rename, file\_size, ... ..

<https://en.cppreference.com/w/cpp/filesystem>

copy\_options

import std.filesystem;

```
#include <filesystem>
namespace fs = std::filesystem;

void dir( const string& root)
{
    fs::path treep{ root};
    for( auto&& de : fs::recursive_directory_iterator{ treep}) {
        if( fs::is_directory( de.status())) ....
        else ....
    }
}
```

rekurzivní průchod

directory entry

# filesystem

- iterátory pro path/filename
  - části složeného jména

```
for( auto&& de : fs::recursive_directory_iterator{ treep}) {  
    fs::path p{ de};  
    cout << p.root_path() << ":" << p.parent_path() << ":" << p.filename()  
        << ":" << p.stem() << ":" << p.extension() << endl;  
  
    for( auto&& pi : p) cout << pi;    cout << pi.string()  
}
```

části path

oduzozovování

- normalizované skládání
  - append / /=
  - concat + +=

```
fs::path p{ "temp"};  
p /= "user" / "data";
```

část path  
temp\user\data

```
fs::path p{ "temp/"};  
p += "user" + "data";
```

zřetězení  
temp\userdata

# raw string

```
f = open( "C:\\temp\\new.txt");
```

co je tu špatně?

```
( '(?:[^\\"']|\\.)*' | "(?:[^\\""]|\\.)*" )|
```

\\ ⇒ \\\\

" ⇒ \"

- raw string literal
  - neplatí escape chars
  - user-defined multi-char delimiter
- syntaxe
  - **R" delim ( chars ) delim "**
  - delim: libovolná (i prázdná) posloupnost znaků
  - chars: platí **všechny** znaky (*newline, tab, \, ", ...*)

R" delim ( chars ) delim "

```
R"('(?:[^\\"']|\\.)*' | "(?:[^\\""]|\\.)*" )|"
```

```
"(\\'(?:[^\\"\\'']|\\.)*\\' | \\"(?:[^\\"\\"]|\\.)*\\")|"
```

```
R"""(A      \b
C)""" "\0" R"raw(Mooh)raw";
```

```
"A\t\\b\\nC\\0Mooh";
```

# Operace nad stromy

- treeproc op path [path]
  - treeproc **print** path
    - elegantně vytiskne obsah stromu adresářů
    - nakonec celkovou velikost souborů a čas strávený výpisem
  - treeproc **delete** path
    - smaže celý strom
  - treeproc **copy** source\_path dest\_path
    - okopíruje celý strom
  - treeproc **copydir** source\_path dest\_path
    - okopíruje pouze adresářovou strukturu bez souborů
  - treeproc **flat** source\_path dest\_path
    - okopíruje všechny soubory bez adresářové struktury
- API vs. commandline
- kulturně implementace
  - už umíte třídy, lambdy, šablony, algoritmy, ..., ...
- netestujte na vlastních zdrojících ☠

```
[C:\moje\pokus]  
  file1.txt  
  file2.bin  
[C:\moje\pokus\sub]  
  whatever.xxx
```

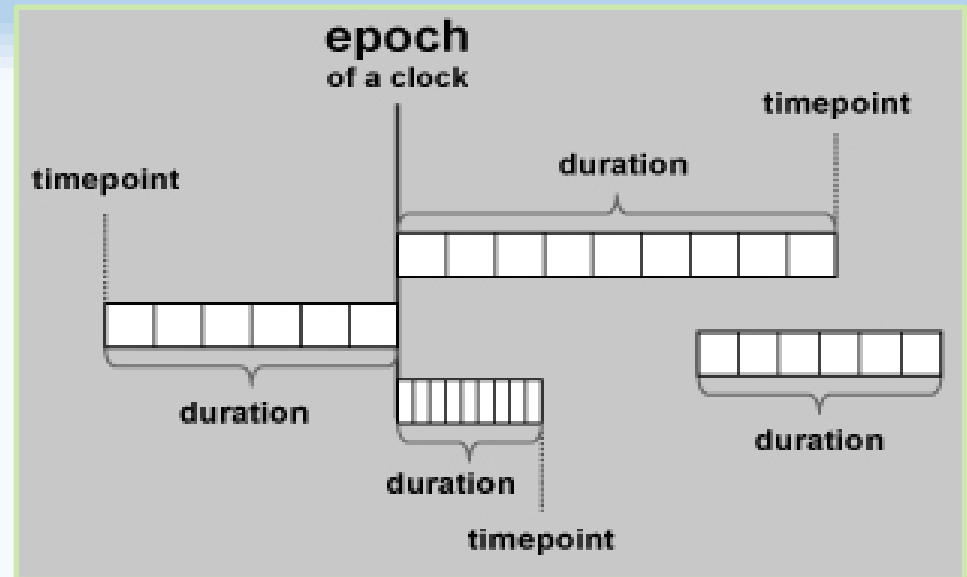


# Chrono

---

# chrono

- epoch
- duration
  - various time units
  - time arithmetic
  - strong type checking
- clock
  - system\_clock
    - system **real-time** clock
    - to\_time\_t(), from\_time\_t() - conversion from/to time\_t
  - steady\_clock
    - monotonic clock, never decreasing
    - not related to wall clock time, suitable for measuring
  - high\_resolution\_clock
    - the clock with the shortest tick period available
- timepoint - time interval from the start of the clock's epoch



# chrono

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace chrono;

void sleep_ms( int ms)
{
    auto t0 = high_resolution_clock::now();
    this_thread::sleep_for( milliseconds( ms));
    auto t1 = high_resolution_clock::now();
    milliseconds total_ms = duration_cast<milliseconds>( t1 - t0);
    cout << total_ms.count();
}
```

various time units  
strong type checking

h min s ms us ns  
y month d

```
chrono::seconds twentySeconds{20};
chrono::hours aDay{24};
chrono::milliseconds ms;
using namespace chrono_literals;

auto tm = 1h + 23min + 45s;
ms = tm + twentySeconds + aDay;
--ms;
ms *= 2;
```

# chrono calendars & timezones

Howard Hinnant: Design Rationale for Chrono  
[www.youtube.com/watch?v=adSAN282Ylw](http://www.youtube.com/watch?v=adSAN282Ylw)

- C++20 - significant chrono extension

- **calendar support**

- time\_of\_day, day, month, year, weekday, month\_day, year\_month\_day, ..., ...

```
year_month_day ymd = 14d/11/2019;  
sys_days d{ ymd};  
d += weeks{ 1};  
cout << ymd << d << format( "{:%d.%m.%Y}", ymd);  
auto d2 = Thursday[2]/November/2019;
```

- **time zone**

- tzdb, locate\_zone, current\_zone, time\_zone, sys\_info, zone\_time, leap, ...

- various real-world / IT clocks

- utc\_clock, tai\_clock, gps\_clock, file\_clock, local\_t

- conversions

- clock\_time\_conversion, clock\_cast

- input/output

- format, parse

```
auto zt = chrono::zoned_time{..  
cout << format( locale{ "cs_CZ"}, "Local time: {:%c}", zt)  
      << format( "{:%d.%m.%Y %T}", zt);
```



**variant**

---

# variant & visit

```
#include <variant>
using myvar = variant<int, double, string>;
myvar v, w;
v = 12;
auto x = get<int>(v);
v = "abcd";
auto y = get<2>(v);
w = v;

cout << v.index();    // 2
if( holds_alternative<string>(v))
    ....

if( auto pv = get_if<int>(&v))
    cout << *pv;
else
    cout << "not an integer";
```

změna typu

přístup pro  
konkrétní typ

přístup  
přes index

```
vector< myvar> vec{ 1, 2.1, "tri" };
for (auto&& v : vec) {
    visit( [](auto&& arg) { cout << arg; }, v);
    myvar w = visit( [](auto&& arg) -> myvar { return arg + arg; }, v);
}
```

polymorfní kód

návratový typ:  
opět variant

# variant & visit

```
myvar v { 3.14 };
struct myVisitor {
    void operator()( const int& i)    const { get<int>(v).... }
    void operator()( const double& f) const { .... }
    void operator()( const string& s) const { .... }
};
visit( myVisitor(), v);
```

typově specifický kód

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

```
for (auto&& v : vec) {
    visit( overload {
        [](auto arg) { cout << arg; },
        [](double arg) { cout << fixed << arg; },
        [](const string& arg) { cout << quoted(arg); },
    }, v);
}
```

variadic templates  
custom argument deduction guides  
pack expansions in using declarations  
aggregate initialization  
implicit constructors

typově specifický kód

# Moduly

---

# include vs. import

- separátní kompilace
  - 50 let stará
  - `#include`
    - mechanické vložení zdrojového textu
    - mnohonásobná kompilace
    - rozdělení header/source
    - one definition rule
    - závislosti, cykly
- moduly
  - C++20
    - implementace VS 19.8 (20.11.2020)
    - gcc zatím ne
  - export/import
  - integrace s build systémem

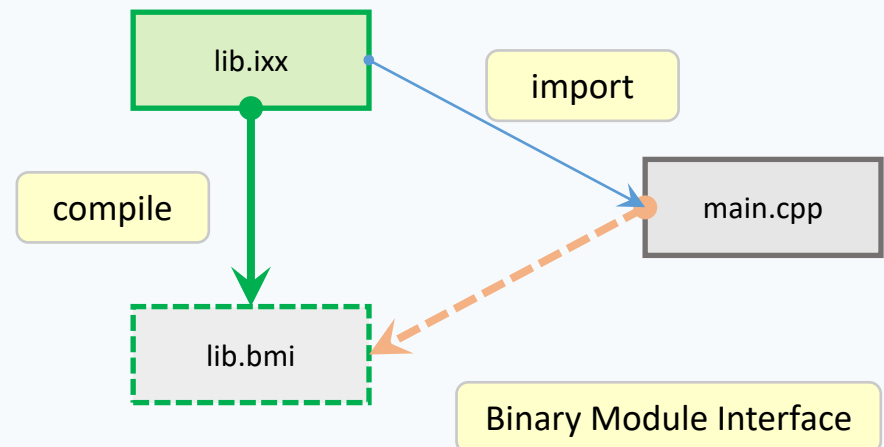
Install: C++ Modules for v142 build tools  
Project properties: Language  
C++ Language Standard: /std:c++latest  
Enable C++ Modules

```
#include <iostream>

int main()
{
    std::cout << "Includes";
}
```

```
import std.core;

int main()
{
    std::cout << "Modules";
}
```



# export / import, template

```
export module mylib;

import std.core;

export std::string get_text() {
    return "I am here";
}
```

mylib.ixx

```
import mylib;

s = get_text();
```

whatever.cpp

```
export module mylib;

export {
    std::string get_text() {...}
    class S { .... };
    ....
}
```

```
export module temple;

export template <typename T>
struct foo
{
    T value;
    foo(T const v):value(v) {}
};

export template <typename T>
foo<T> make_foo(T const value)
{
    return foo<T>(value);
}
```

BMI - přeložený mezikód

není nutný  
mnohonásobný  
překlad

```
import temple;

int main()
{
    auto fi = make_foo( 42);
    cout << fi.value;

    auto fs = make_foo( "modules"s);
    cout << fs.value;
}
```

# partitions, private

```
export module mylib:classes; mlc.ixx
```

```
export {  
    class S { .... };  
    ....  
}
```

```
export module mylib;
```

**mylib.ixx**

```
export import :fnc;  
export import :classes;
```

```
export module mylib:fnc; mlf.ixx
```

```
export {  
    std::string get_text() {..}  
    ....  
}
```

```
import mylib;
```

**whatever.cpp**

```
s = get_text();
```

```
export module imp;
```

```
struct Impl;
```

```
export class S {  
public:  
    void doit();  
    Impl* get() { return i_.get(); }  
private:  
    std::unique_ptr<Impl> i_;  
};
```

```
module :private;  
struct Impl { .... };
```

```
import imp;
```

```
int main() {  
    S s;  
    s.doit();  
    s.get();
```

OK: metoda / pointer

Error: undefined type

```
auto impl = *s.get();  
}
```

modul gumové pole

# Ranges

---



# ranges

- stl
- iterator based algorithms - verbosity

```
set_difference( v2.begin(), v2.end(), v3.begin(), v3.end(), back_inserter(v4));
```

- no orthogonal composition

```
transform( input.begin(), input.end(), back_inserter(output), f);  
copy_if( input.begin(), input.end(), back_inserter(output), p);  
transform_if
```

- ranges

- (it,it), (it,count), (it,predicate)
- all std:: containers
- composability
- lazy evaluation

transform\_view<  
filter\_view<  
ref\_view< >>>

vyhodnocení zde

```
#include <ranges>  
vector<int> numbers = { .... };  
  
auto results = numbers  
| ranges::views::filter( [](int n) { return n%2 == 0; })  
| ranges::views::transform( [](int n) { return n*2; });  
  
for( auto v : results)  
    cout << v << " ";
```

zde se nic  
nevyhodnocuje

# C++17 vs. C++20

- tisk lichých prvků obráceně

```
for_each( v.crbegin(), v.crend(),  
    [](auto const x) {  
        if(x % 2 == 0) print(x);  
    }  
);
```

```
for( auto&& x : v | view::reverse  
    | view::filter(is_even))  
    print(x);
```

- počet slov

```
istringstream iss(text);  
vector<string> words(  
    istream_iterator<string>{iss},  
    istream_iterator<string>{});  
auto count = words.size();
```

```
auto count = distance(  
    view::c_str(text) | view::split(' '));
```

- setřídít kopii bez dvou nejmenších a největších prvků

```
vector<int> v2 = v;  
sort( v2.begin(), v2.end());  
auto first = v2.begin;  
advance( first, 2);  
auto last = first;  
advance( last, v2.size() - 4);  
v2.erase( last, v2.end());  
v2.erase( v2.begin(), first);
```

```
auto v2 = v | copy | action::sort  
    | action::slice( 2, end - 2);
```

není v C++20

# ranges - materializace

- `ranges::to`

```
auto v = ranges::to<vector>(r);
```

není v C++20

- `ranges::copy`

kompilační if

```
auto r = ....  
vector<ranges::range_value_t<decltype(r)>> v;  
if constexpr( ranges::sized_range<decltype(r)> ) {  
    v.reserve( ranges::size(r) );  
}  
ranges::copy( r, back_inserter(v) );
```

GCC 10.2 OK  
VS 18.6 aktivní algoritmy zatím neumí

VS 18.6 ani GCC 10.2 neumí

- `for`

```
for( auto&& x : r ) ....
```

# Různé

---

# xvalues

## ▶ xvalue

- **e**xpiring value, movable
- temporary, `std::move`

```
x = std::move( victim);
```

## ▶ glvalue

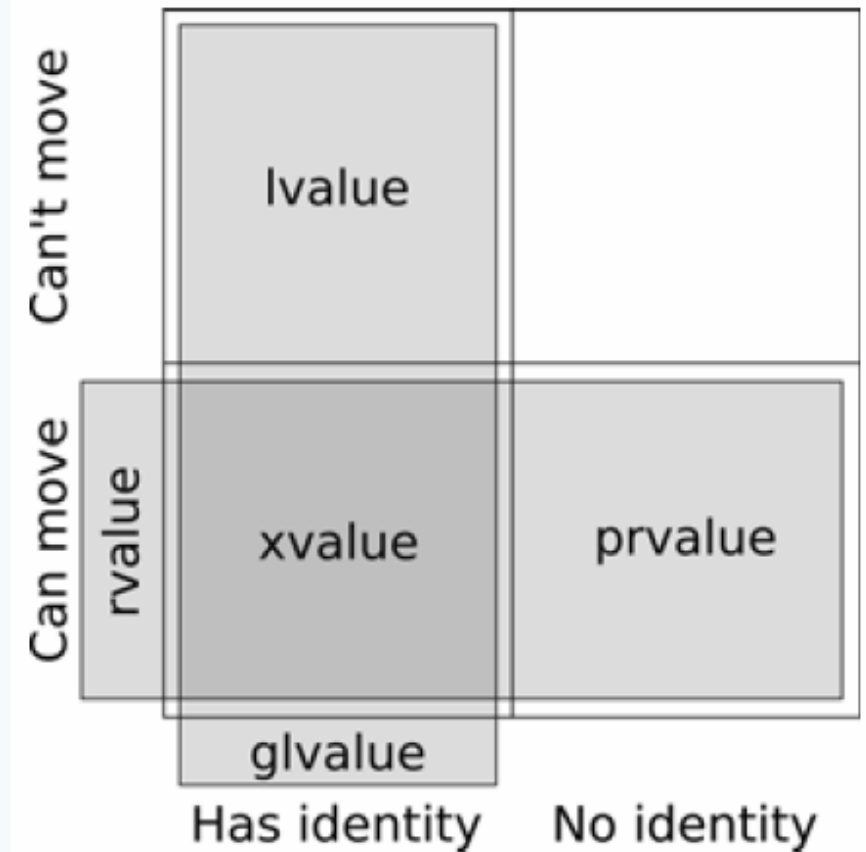
- **g**eneralized **l**value
- lvalue + xvalue

## ▶ prvalue

- **p**ure **r**value
- pre-C++11 rvalue

## ▶ rvalue (C++11)

- prvalue + xvalue



# Special members

compiler implicitly declares

user declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

# Audio player

- interpret
  - album
    - rok, název
    - žánr: rock, pop, folk, ...
    - track
- favorit
  - interpret, album, track
- přehrávat
  - konkrétní album
  - žánr
  - alba v letech
  - náhodně interpreta
  - favority
  - ...



- ▶ dekompozice
- ▶ datové struktury
- ▶ funkce a jejich implementace
- ▶ optimalizace
  - malá mobilní zařízení
  - pomalý procesor
  - málo paměti

# Zápočtový program

- termín
  - odevzdání **kompletně** hotového programu **do 30.4.** (velmi doporučuji dříve)
  - poslední opravy do konce výuky v LS (21.5.)
- mff GitLab
  - komplet zdrojáky, knihovny, projekt
    - ne .obj, .dbg, ... !
  - dokumentace, data
  - používat během **celého** vývoje
  - před odevzdáním vyzkoušet na čistém počítači!
- funkčnost
  - stabilita, efektivita
- kvalita kódu
  - dekompozice, encapsulace, API, GUI vs. app logika, komentáře, udržitelnost, čitelnost, elegance a estetika, dobré mravy, deployment
- mnohem podrobnější informace
  - <https://www.ksi.mff.cuni.cz/teaching/nprg041-zavoral-web/cviceni.html>

zjevení se zdrojáků najednou  
den před odevzdáním  
nebude akceptováno

čtěte!



# The End.

... to be continued

zkouškový test

zápočtový program

Pokročilé programování v C++

Ročníkový projekt / Bc. práce / ...

# starší slajdy

---

... mohou být někomu užitečné

# Streamy

- čtení ze souboru i std vstupu
- záměnnost
  - std vstup i soubor jsou streamy
  - lze přiřadit za běhu

```
#include <iostream>
#include <fstream>
```

```
ifstream x;
x.open( "file.txt");
if( ! x.good()) { "chyba" }
for (;;) {
    x >> a;
    if( x.fail())
        break;
    f( a);
}
x.close();
```

stav streamu

výsledek  
předchozí  
operace

lepší než  
test na eof

nepsat zvláštní kód pro  
čtení souboru

```
process( istream& f) {
    f >> ....
}

if( ....) {
    ifstream f( ....);
    process( f);
} else {
    process( cin);
}
```

# operátor <<

- přetížení operátoru <<
- není to metoda třídy ale friend globální funkce
  - nemáme přístup do implementace ostream

```
class Complex {  
public:  
    Complex() : re_(0), im_(0) {}  
    friend ostream& operator<< ( ostream& out, const Complex& x);  
private:  
    double re_, im_;  
};  
  
ostream& operator<< ( ostream& out, const Complex& x) {  
    out << "[" << x.re_ << "," << x.im_ << "]" << endl;  
    return out;  
}
```

toto není  
metoda

# Stream manipulátory

endl	vloží nový řádek
setw(val)	nastaví šířku výstupu
setfill(c)	nastaví výplňový znak
dec, hex, oct	čte a vypisuje v dané soustavě
left, right	zarovnávání
fixed, scientific	formát výpisu čísla
precision(val)	nastaví přesnost
ws	přeskočí bílé znaky
(no)skipws	nastavení/zrušení přeskakování bílých znaků při čtení
(no)showpoint	nastaví/zruší výpis desetinné čárky
...	

# Bezparametrický manipulátor

- speciální funkce
  - předávané ukazatelem
  - vrátí referenci na modifikovaný stream

```
cout << 1 << mriz << 2 << mriz << 3 << endl;  
  
ostream& mriz( ostream& io)  
{  
    io << " ### ";  
    return io;  
}
```

ukazatel na funkci

funkce  
zavolá ji op<<

- jak to funguje

přetížená metoda na  
ukazatel na funkci

- přesněji: šablona

```
ostream& operator<< (ostream& (* pf)(ostream&));
```

# Parametrický manipulátor

```
cout << 1 << mriz(5) << 2 << mriz(3) << 3 << endl;
```

- nelze předdefinovaná funkce
  - libovolné možné parametry
- ošklivé řešení
  - vlastní funkce s extra parametrem

```
cout << mriz(cout,5) << ...
```

- hezčí řešení
  - zvláštní třída, zvláštní přetížení <<

# Parametrický manipulátor

- vlastní třída
  - anonymní instance
  - parametr konstruktoru
  - přetížení << na tuto třídu

příklad:  
cout << oddel( '-', 8);

-----

```
class tecka {  
private: int n_;  
public: explicit tecka( int n ) : n_( n ) {  
        int get_n() const { return n_; }  
};  
  
ostream& operator<<( ostream& io, const tecka & p)  
{  
    int n = p.get_n();  
    while( n-- ) io << ".";  
    return io;  
}  
  
cout << 1 << tecka(5) << 2 << tecka(3) << 3 << endl;
```

známý trik:  
separace inicializace a  
volání

zřetězení <<

jiná instance

příklad:  
cout << zlomek( 3, 4);

3 / IV



# streams - ios\_base::iostate

- ios\_base::iostate

- bits - badbit, failbit, eofbit
- methods - good(), bad(), fail(), eof()
- operators - bool, !

[https://en.cppreference.com/w/cpp/io/ios\\_base/iostate](https://en.cppreference.com/w/cpp/io/ios_base/iostate)

číst dál?




správně načteno?

**good() ≠ ! fail()**

eofbit   failbit   badbit   good()   fail()   bad()   eof()   bool   oper !

false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

# ios\_base::iostate::failbit

- recoverable errors
  -  file cannot be opened
  - if eofbit or badbit or eof while consuming ws
  -  op>>, op<< if no characters are extracted/inserted
  -  op>> if the input cannot be parsed as a valid value or if the value does not fit in the destination type
  - getline if the function extracts no characters or if it manages to extract basic\_string::max\_size characters, or if it fills in the provided buffer without encountering the delimiter
  - read if the eof occurs on the input stream before all requested characters could be extracted
  - seekg/tellp on failure

# badbit, exceptions

- `ios::badbit`
- non-recoverable errors
  - `put`, `write` if it fails
  - `op<<`, `putback`, `unget` if `eof`
  - exception is thrown by any member function
- exceptions

```
try {  
    ifstream f;  
    f.exceptions( ios::badbit | ios::failbit);  
    f.open(fname);  
    while( ! f.eof()) {  
        f >> a >> b >> c;  
    }  
    f.close();  
} catch ( ios_base::failure& fail) {  
    cerr << e.what() << e.code();  
}
```

# Čtení vstupu - slova oddělená ws

```
string s1, s2;  
int i1, i2;  
f >> s1 >> i1 >> s2 >> i2;  
if( f.fail()) ...;  
...
```

ws (mezery, ...)  
se automaticky přeskočí

stream zůstává  
za posledním čtením

# Čtení vstupu - celé řádky

```
const int MaxBuf = 4095;
char buffer[ MaxBuf+1];

for( ;;) {
    f.getline( buffer, MaxBuf);
    if( f.fail()) break;
    cout << "[" << buffer << "]" << endl;
}
```

pozor na zásobník!

vždy limit

```
string s;
for( ;;) {
    getline( f, s);
    if( f.fail()) break;
    cout << "[" << s << "]" << endl;
}
```

parsování řádku

```
string r, s1, s2;
for( ;;) {
    getline( f, r);
    if( f.fail()) break;
    stringstream radek(r);
    radek >> s1 >> s2;
    cout << "[" << s1 << s2 << "]" << endl;
}
```

# Čtení vstupu - oddělovače

```
string s;  
string::iterator b, e;  
char delim = ',';  
  
while( getline( f, s) ) {  
    b = e = s.begin();  
    while( e != s.end() ) {  
        e = find( b, s.end(), delim);  
        string val{ b, e};  
        cout << "[" << val << "];"  
        b = e;  
        if( e != s.end())  
            b++;  
    }  
    cout << endl;  
}
```

dokud přečtené slovo není na konci

iterator na oddělovač

hodnota mezi oddělovači

přeskočí oddělovač

# Čtení vstupu - výhled

```
f >> ws;  
if( isdigit( f.peek())) {  
    int i;  
    f >> i;  
    cout << "[" << i << "]" << endl;  
} else {  
    string s;  
    f >> s;  
    cout << "{" << s << "}" << endl;  
}
```

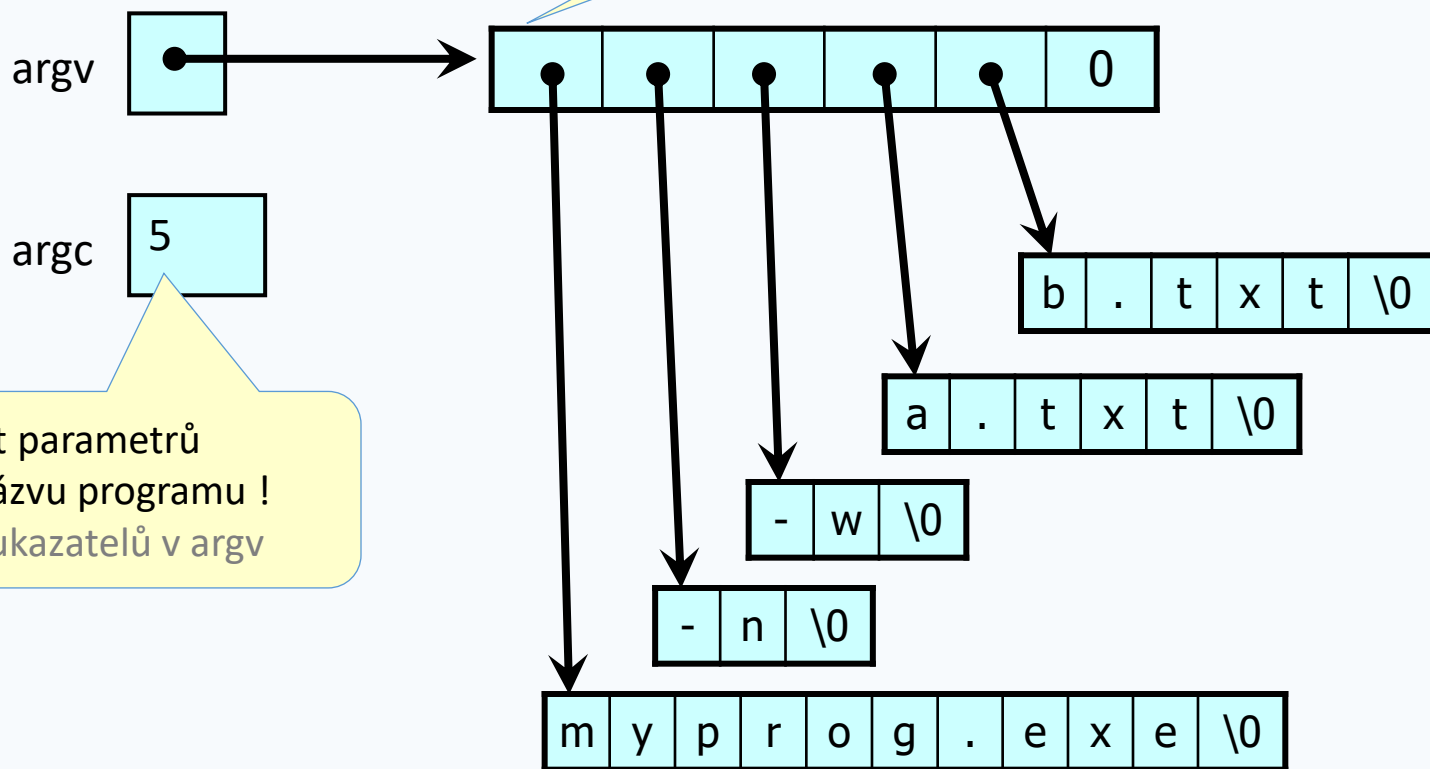
přečte nejbližší znak,  
ale nechá ve streamu

# Parametry příkazové řádky

```
C:\> myprog.exe -n -w a.txt b.txt
```

```
int main( int argc, char** argv)
```

pole řetězců  
(ukazatelů na char)



```
vector<string> arg( argv, argv+argc);
```



# Zpracování příkazové řádky

usage: myprog [-n] [-w] fileA fileB

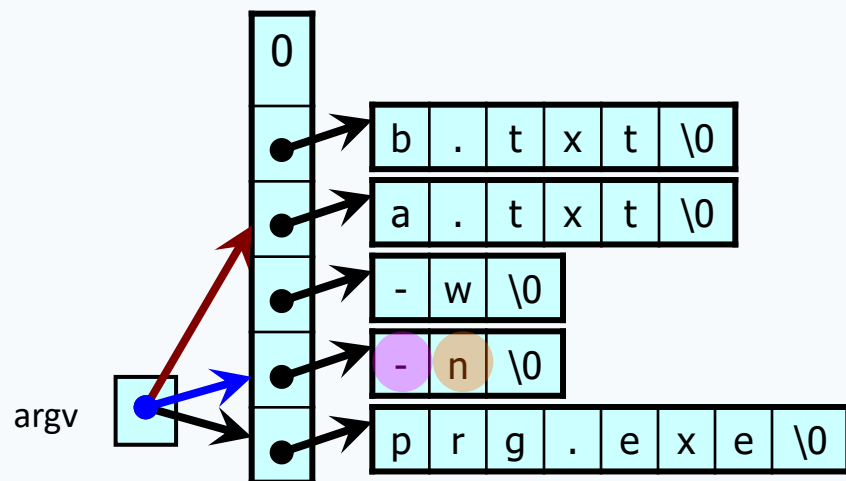
```
int main( int argc, char** argv)
{ int n=0, w=0;
  while( *++argv && **argv=='-' )
  { switch( argv[0][1]) {
      case 'n': n = 1; break;
      case 'w': w = 1; break;
      default: error();
    }
  }
  if( !argv[0] || !argv[1])
    error();
  doit( argv[0], argv[1], n, w);
  return 0;
}
```

options

nastavení  
přepínače

zbývající  
parametry

výkonná funkce



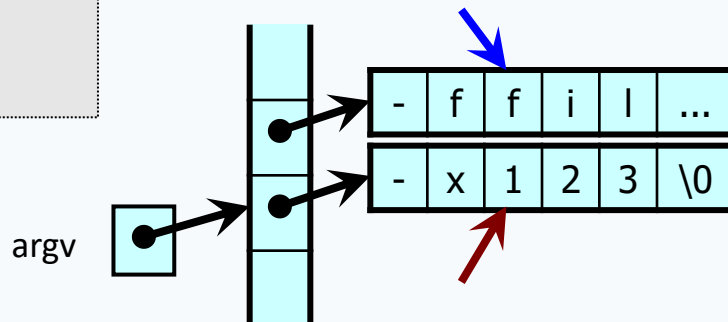
# Zpracování příkazové řádky

```
int main( int argc, char** argv)
{ int n=0, w=0;
  int x = 0;
  char* f = 0;
  while( *++argv && **argv=='-')
  { switch( argv[0][1]) {
    case 'n': n = 1; break;
    case 'w': w = 1; break;
    case 'x': x = atoi( *argv+2; break;
    case 'f': f = *argv+2; break;
    default: error();
  }
}
if( !argv[0] || !argv[1]) error();
doit( argv[0], argv[1], n, w, x, f);
return 0;
}
```

usage: myprog [-n] [-w] [-  
x123] [-filename]  
fileA fileB

číselný  
parametr

řetězcový  
parametr



# Zpracování příkazové řádky

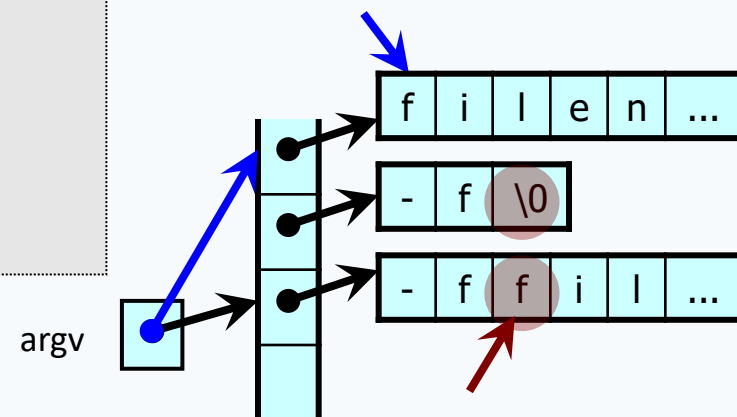
```
int main( int argc, char** argv)
{ int n=0, w=0;
  int x = 0;
  char* f = 0;
  while( *++argv && **argv=='-')
  { switch( argv[0][1]) {
      case 'n': n = 1; break;
      case 'w': w = 1; break;
      case 'x': x = atoi( argv[0]+2); break;
      case 'f': if( argv[0][2]) f = *argv+2;
                else f = *++argv;
                break;
      default: error();
    }
  }
  if( !argv[0] || !argv[1]) error();
  doit( argv[0], argv[1], n, w, x, f);
  return 0;
}
```

usage: myprog [-n] [-w] [-x123] [-f filename]  
fileA fileB

≡ &(argv[0][2])

-ffile

-f file



# vidle

multiset čísel větších než X inkrementovaných o Y

```
class fnc_vidle {  
public:  
    fnc_vidle(int x, int y) : x_(x), y_(y) {}  
    int operator()(int val) { if (val > x_) s_.insert(val + y_); return val; }  
    multiset<int> get_s() { return s_; }  
private:  
    multiset<int> s_;  
    int x_, y_;  
};  
  
auto fe = for_each( v.begin(), v.end(), fnc_vidle(x, y));  
fe.get_s() ....
```

# slovník

```
class Dict {
public:
    using Data = multimap< string, string>;
    void add(const string& src, const string& dest) { data_.insert(make_pair(src, dest)); }
    void del(const string& src, const string& dest);
    void del_all(const string& src) { data_.erase(src); }
    tuple< Data::const_iterator, Data::const_iterator> find(const string& src);
    tuple< Data::const_iterator, Data::const_iterator> find_prefix(const string& src);

private:
    Data data_;
};

void Dict::del(const string& src, const string& dest)
{
    auto b = data_.lower_bound(src);
    if (b == data_.end())
        return;
    for (auto e = data_.upper_bound(src); b != e; ++b) {
        if (b->second == dest) {
            data_.erase(b);
            return;
        }
    }
}
```

# slovník

```
auto Dict::find(const string& src) // -> tuple< Data::const_iterator, Data::const_iterator>
{
    auto b = data_.lower_bound(src);
    if (b == data_.end())
        return { data_.end(), data_.end() };
    auto e = data_.upper_bound(src);
    return { b, e };
}

auto Dict::find_prefix(const string& src) // -> tuple< Data::const_iterator, Data::const_iterator>
{
    auto b = data_.lower_bound(src);
    if (b == data_.end() || src.empty())
        return { data_.end(), data_.end() };
    string src_end = src;
    ++src_end[src_end.size() - 1];
    auto e = data_.lower_bound(src_end);
    return { b, e };
}
```

# range-based for

```
auto && range = range_expr;  
auto begin = begin_expr;  
auto end = end_expr;  
for (; begin != end; ++begin) { .. }
```

```
class sentence {  
public:  
    struct const_iterator {  
        const_iterator(const sentence& sentence, size_t index = 0) : sentence_(sentence), index_(index) {}  
        char operator*() const { return sentence_[index_]; }  
        void operator++() { ++index_; }  
    private:  
        const sentence& sentence_;  
        size_t index_;  
    };  
    struct end_iterator {  
        end_iterator(char separator) : separator_(separator) {}  
        char separator_;  
    };  
  
    sentence(const string& s, char separator) : s_(s), separator_(separator) {}  
    char operator[](size_t i) const { return s_[i]; }  
    const_iterator begin() { return *this; }  
    end_iterator end() { return end_iterator(separator_); }  
    private:  
        string s_;  
        char separator_;  
};  
  
bool operator!=(const sentence::const_iterator& lhs, const sentence::end_iterator& rhs)  
{ return *lhs != rhs.separator_; }  
  
sentence x( "Ahoj babi. Dnes jsme prijeli. Mame hlad", '.');  
for (auto&& y : x)  
    cout << y;
```

není de-facto iterator  
pouze k porovnání na separátor

# Gumové pole

```
template<typename T> class Pole {  
public:  
    Pole( size_t chunk = 100) : chunk_(chunk), size_(0) {}  
    void push_back( const T& x) { resize( ++size_); (*this)[size_-1] = x; }  
    T& operator[] ( size_t i) { return rake_[i/chunk_][i%chunk_]; }  
    T& at( size_t i) { check(i); return (*this)[i]; }  
private:  
    void check( size_t i) { if (i >= size_) throw ....; }  
    void resize( size_t i) { while( rake_.size() < (i-1)/chunk_)  
        rake_.push_back( make_unique<T[]>(chunk_)); }  
    size_t chunk_;  
    size_t size_;  
    vector< unique_ptr<T[]>> rake_;  
};
```

počet alokovaných /  
potřebných chunků

≈ new T[chunk\_]

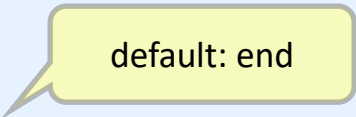
auto p = make\_unique< T[]>(chunk\_);  
rake\_.push\_back( move( p));



# Gumové pole - iterator

```
template<typename T> class Pole {
private:
    ....
    class iterator {
public:
    iterator() : k_(nullptr), i_(0) {}
    iterator( Pole<T>* k, size_t i = end_index) : k_(k), i_(i) {}
    iterator( const iterator& it) : k_(it.k_), i_(it.i_) {}
    T& operator* () { return (*k_)[i_]; }
    bool operator != ( const iterator& it2 ) { return this->k_ != it2.k_ || this->i_ != it2.i_; }
    iterator operator ++() { if( ++i_ >= k_->size_) i_ = end_index; return *this; }
private:
    static const size_t end_index = -1;
    Pole<T>* k_;
    size_t i_;
};

iterator begin() { return iterator( this, 0); }
iterator end() { return iterator( this); }
};
```



default: end

# Gitlab



- Development lifecycle tool
  - <https://gitlab.mff.cuni.cz>
  - integrace ve Visual Studiu
  - povinné odevzdávání zápočtů
  - **velmi** doporučené pro vývoj
    - source versioning
    - návrat k předchozím verzím
    - *"večer před deadlinem mi odešel disk"*
- Integrace se SIS
  - přihlašte se pomocí MFF username / password
  - potom bude pro každého studenta vytvořena repository  
/teaching/nprg041/2019-20/zavoral/**novakova**

# Gitlab ve VS

<https://gitlab.mff.cuni.cz/teaching/nprg041/2019-20/zavoral/novakova>

- Spojení VS s repository
  - Clone or check out code
    - vytvoří lokální repository - do ní zdrojáky
- File / New Project / Console C++
  - do stejného adresáře
    - ... *add source code* ...
- Team Explorer
  - sync = vztah mezi lokální a vzdálenou repo
  - changes = vztah mezi loc repo (tajna schovana) a soubory
  - Synchronization / Sync, Changes
    - Changes
      - .vs - rmb: **ignore** these local items
    - Commit Staged (.gitignore), Commit All (.cpp)
      - je to v loc repo -> sync do remote: Sync / Push
- Build
  - vytvoří spoustu tmp
  - Team Explorer / Changes
    - project\_name/Debug, Debug -> **ignore** these local items
    - comment / Commit Staged / Sync / Push

VS 16.8 - zcela  
přepřacováno