Objektovo-orientované programovanie v C++

Jazyk C++

- vytvoril Dán Bjarne Stroustrup v roku 1985 ako rozšírenie jazyka C (Pôvodne ho nazval „C s triedami“)
- je univerzálny jazyk, nie je určený pre nejaký konkrétny typ aplikácií
- dáva programátorovi na výber – podporuje procedurálne, objektové aj tzv. generické programovanie
- nie je jazyk závislý od konkrétnej platformy - multiplatformový
- ponúka programátorom širokú škálu tried a funkcií, z ktorých každá ale má svoje praktické opodstatnenie
- je použiteľné aj bez komplikovaného vývojového prostredia
- je jazyk so statickým typovaním – typy premenných sú známe už v čase kompilácie
- je takmer úplne kompatibilný s jazykom C
- posledná štandardizovaná (ISO) verzia je C++20 (predchádzajúce verzie sú C++17, C++14, C++11, C++03 a C++98). Aktuálne sa pracuje na novom štandarde C++23
- jeho najväčšia výhoda oproti iným vysokoúrovňovým jazykom (napr. Java a C#) je jeho rýchlosť

Hlavné rozdiely medzi C a C++

- C nepodporuje objektové a generické programovanie
- C++ má vybudovanú **obsluhu výnimiek**
- V C++ je dovolená **deklarácia lokálnych premenných** takmer na ľubovoľnom mieste, v C nie
- C++ má nový dátový typ **bool**, ktorý uchováva pravdivostnú hodnotu true alebo false
- V C++ je iný mechanizmus **vstupu a výstupu**, nemusí sa udávať formát
- C a C++ sa líšia príkazmi pre **dynamickú alokáciu a dealokáciu**
- V C++ musí byť každá funkcia definovaná ešte pred svojím použitím
- Ak sa v C nešpecifikuje návratový typ funkcie, automaticky sa predpokladá int. V C++ to tak nie je
- V C++ je o niečo jednoduchšia práca so štruktúrami

Namespace – menný priestor

- Predstavte si, že pracujete dvaja programátori na jednom programe a obaja sa rozhodnete, že vaša funkcia sa bude volať *abc()*, pričom každá bude robiť niečo iné. Čo sa stane? Ako bude kompilátor vedieť, ktorú funkciu ma použiť?
- Menné priestory (**namespaces**) sú v jazyku C++ navrhnuté tak aby „obalili“ funkcie a objekty do spoločného priestoru
- Definícia menného priestoru:

```
namespace namespace_name {
        // deklarácie premenných, funkcií, objektov
}
```

- Volanie z menného priestoru:

```
namespace_name::code; //code môže byť premenná alebo funkcia
```

- Čo ak sa mi nechce stále písať pred všetko namespace, lebo používam v celom programe iba jeden? Použijem **direktívu `using`:**

```
using namespace namespace_name;
```

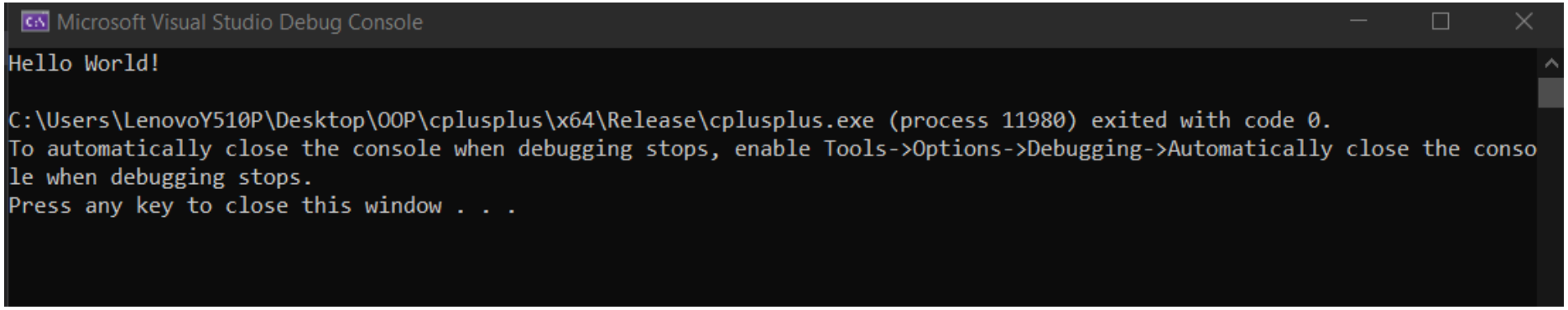
- Vo väčšom projekte kde je definovaných viacero menných priestorov je lepšie nepoužívať direktívu `using`

Štruktúra programu v C++

```
#include <iostream> //vlozenie hlavickového súboru
using namespace std; //miesto pre direktívy

int main(){ //funkcia main, musí mat aj návratový typ

        cout << "Hello World!\n"; // kód funkcie main
        return 0; //návratová hodnota funkcie main
}
```



Štandardné knižnice jazyka C

- Keďže C++ je spätne kompatibilne s jazykom C, môžeme používať knižnice napísane v jazyku C:

<cassert> (assert.h)	C Diagnostics Library
<cctype> (ctype.h)	Character handling functions
<cerrno> (errno.h)	C Errors
<cfloat> (float.h)	**Characteristics of floating-point types**
<ciso646> (iso646.h)	ISO 646 Alternative operator spellings
<climits> (limits.h)	**Sizes of integral types**
<clocale> (locale.h)	C localization library
<cmath> (math.h)	**C numerics library**
<csetjmp> (setjmp.h)	Non local jumps
<csignal> (signal.h)	C library to handle signals
<cstdarg> (stdarg.h)	Variable arguments handling
<cstddef> (stddef.h)	C Standard definitions
<cstdio> (stdio.h)	**C library to perform Input/Output operations**
<cstdlib> (stdlib.h)	**C Standard General Utilities Library**
<cstring> (string.h)	**C Strings**
<ctime> (time.h)	**C Time Library**

Štandardné knižnice jazyka C++ (C++ Standard Library)

- zbierka funkcií a objektov napísaných v jazyku C++
- všetky knižnice sú v mennom priestore `std`

- zbernice na dáta (Containers) - `<vector>, <array>, <list>, <map>, <set>,…`
- všeobecné funkcie/algoritmy - `<algorithm>, <memory>, <utility>,...`
- vstupno/výstupne operácie (I/O stream) - `<iostream>, <fstream>,...`
- knižnice na prácu so stringami –`<string>, <regex>,...`
- knižnice na numerické operácie - `<random>, <numeric>,...`
- a ďalšie iné..

Deklarácia lokálnych premenných

- V C++ je napr. možný nasledujúci zápis:

```
int main()
{
        ...
        int sum = 0;
        for (int i = 0; i <= 10; i++) {
                for (int j = 0; j < 10; j++) {
                        sum += 1;
                }
        }
        ...
        if (int flag = 200 > sum) {
                ...
        }
}
```

Dátový typ bool

- Dátový typ môže mať iba dva stavy:
  - true
  - false
- Výsledok operátorov == a != je dátového typu bool

```
int a = 10;
bool state = a == 10;
if (state) { //to iste ako keby tu je napisane state == true
        ...
}
```

Vstupy a výstupy v C++

- Základným pojmom pre vstupno-výstupné operácie je **stream**
- **Stream** (dátový prúd) je abstrakciou vstupno-výstupného zariadenia, všetky operácie vykonané na streame sa fyzicky prejavia na danom zariadení
- Na riadenie vstupov a výstupov máme v C++ dva operátory:

  operátor **<<** : posielanie dát do streamu

  operátor **>>** : vyberanie dát zo streamu
- Knižnica `<iostream>`:
  - Táto knižnica obsahuje funkcie potrebné na vstupy z klávesnice a výstupy na obrazovku
  - Na výstupné a vstupné operácie sa používajú objekty

    `cout` (console output) – objekt triedy ostream (output stream)

    `cin` (console input) – objekt triedy istream (input stream)

Výstup na obrazovku

- Príklad: Chceme vypísať hodnotu čísla a znakový reťazec

```
float cislo = 2.5;
char s[] = "Toto bol moj prvy vypis v C++";
cout << "Hodnota cisla je : " << cislo;
```

- Ak chceme po vypísaní ukončiť riadok:

```
cout << "Hodnota cisla je : " << cislo << endl;
```

- Vypísanie spolu s reťazcom:

```
cout << "Hodnota cisla je : " << cislo << endl << s << endl;
```

- Ak by sme nepoužili direktívu `using namespace std;`:

```
std::cout << "Hodnota cisla je : "<< cislo << std::endl << s << std::endl;
```

Výstup na obrazovku

- Formátovanie výstupu pomocou knižnice <iomanip>:

  `setprecision(int n)` - nastaví počet desatinných miest, ktoré sa vypíšu, alebo maximálnu dĺžku vypísaného argumentu. Musí sa ale použiť `fixed` formát čísla.

  `setw(int n)` - nastaví minimálnu dĺžku znakov vypísaného argumentu

- Príklad: Výpis desatinného čísla na rôzny počet miest

```
#include<iostream>
#include<iomanip>
...
double cislo = 12.3456;
cout << fixed << setprecision(2) << cislo << endl;
cout << scientific << setprecision(2) << cislo << endl;
...
Výstup:
12.35
1.23e+01
```

Vstup z klávesnice

- Príklad: Chceme načítať vek používateľa z klávesnice

```
float age;
cout << "Zadaje Vas vek : " << endl;
cin >> age;
cout << "Vas vek je " << age << endl;
```

- Môžeme načítať zo streamu aj do viacerých premenných za sebou:

```
float age, height;
cout << "Zadaje Vas vek a vysku : " << endl;
cin >> age >> height;
cout << "Vas vek je " << age << "a Vasa vyska je "<< height<< endl;
```

- Čo sa stane keď do premennej typu float budeme chcieť načítať text, alebo do premennej typu int budeme chcieť načítať desatinne číslo?

Knižnica <string>

- Na prácu s reťazcami znakov sa používa knižnica <string>
- Načítajme do premennej typu string reťazec zo vstupu z klávesnice:

```
#include <string>
...
string name;
cout << "Zadaje Vase meno : " << endl;
cin >> name;
cout << "Vase meno je " << name << endl;
```

- Problém nastane ak chceme do jedného stringu načítať viac slovný retazec -> cin vždy berie prázdny znak (medzeru, tab, nový riadok) ako znak, **po ktorý** má čítať znaky do premennej string.

Knižnica <string>

- Riešením je funkcia `getline` (z knižnice `<string>`):

```
#include <string>
...
string name;
cout << "Zadaje Vase meno a priezvisko: " << endl;
getline(cin, name);
cout << "Vase meno a priezvisko je " << name << endl;
```

Knižnica <string>

String operations:

c_str	Get C string equivalent (public member function)
data	Get string data (public member function)
get_allocator	Get allocator (public member function)
copy	Copy sequence of characters from string (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)
find_first_of	Find character in string (public member function)
find_last_of	Find character in string from the end (public member function)
find_first_not_of	Find absence of character in string (public member function)
find_last_not_of	Find non-matching character in string from the end (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

Knižnice <ifstream>,<ofstream>,<fstream>

- Tieto knižnice umožňujú vstupno-výstupné operácie z/do súboru
- `ifstream` – práca zo vstupnými súbormi (input file stream)
- `ofstream` – práca s výstupnými súbormi (output file stream)
- `fstream` – práca so vstupno-výstupnými súbormi
- Rovnako ako knižnice sa volajú aj triedy, ktoré sú abstrakciou skutočných dátových súborov

Otvorenie a zatvorenie súboru

- Uvedené funkcie sú metódy tried fstream, ifstream, ofstream
- Otvorenie súboru:

```
void open(const char *filename, openmode mode)
```

- Otvorí súbor, ktorý má na disku meno filename. mode udáva, akým spôsobom sa má súbor otvoriť a môže mať napr. tieto hodnoty:

`ios::in` – otvorenie na čítanie

`ios::out` – otvorenie na zapisovanie

`ios::app` – nové dáta sa pripoja na koniec súboru

`ios::binary` – znamená, že súbor je binárny

- Zatvorenie súboru:

```
void close()
```

Otvorenie a zatvorenie súboru

- Príklad: otvorte subor file.dat na zápis na koniec súboru:

```
#include <fstream>
...
fstream afile;
afile.open("file.dat", ios::out | ios::app);
...
afile.close();
```

- Keďže súbor sa z rôznych dôvodov nemusí podariť otvoriť (neexistuje, je poškodený, je problém s hardwarom a pod.), je korektné vždy testovať, či je súbor správne otvorený. Slúži na to funkcia:

```
bool is_open()
```

Otvorenie a zatvorenie súboru

- Príklad: Otvorenie súboru s testovaním

```
fstream textfile("text.txt", ios::out | ios::app); //použil som konštruktor
if (!textfile.is_open()) {
        cout << "Subor sa nepodarilo otvorit!" << endl;
}
…
textfile.close();
```

Výstup do súboru a vstup zo súboru

- Príklad: Zapísanie čísel do textového súboru

```
#include <fstream>
...
fstream textfile;
textfile.open("cisla.txt", ios::out);
if (textfile.is_open()) {
	for (int i = 0; i < 10; i++) {
		textfile << i << '\n';
	}
	textfile.close();
}
else {
	cout << "Subor sa nepodarilo otvorit!" << endl;
}
...
```

Výstup do súboru a vstup zo súboru

- Príklad: Načítanie čísel z textového súboru

```
#include <fstream>
...
int numbers[10];
fstream textfileInput;
textfileInput.open("cisla.txt", ios::in);
if (textfileInput.is_open()) {
        for (int i = 0; i < 10; i++) {
                textfileInput >> numbers[i];
        }
        textfileInput.close();
}
else {
        cout << "Subor sa nepodarilo otvorit!" << endl;
}
...
```

Výstup do súboru a vstup zo súboru

- Na zapísanie/čítanie jedného znaku (=1 byte) do súboru máme príkazy

```
fstream& put(char c)
fstream& put(char &c) / int get()
```

- Príklad: Vytvorenie obrázku vo formáte .raw (binárneho súboru)

```
fstream image;
image.open("image.raw", fstream::out | fstream::binary);
for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
                if ((i - 50) * (i - 50) + (j - 50) * (j - 50) <= 400) {
                        image.put(255);
                } else {
                        image.put(0);
                }
        }
}
image.close();
```

Výstup do súboru a vstup zo súboru

- Príklad: Načítanie dát z .raw súboru do poľa

```
...
fstream image;
unsigned char intensity[100][100];
image.open("image.raw", ios::in | ios::binary);
for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
                intensity[i][j] = (unsigned char)image.get();
        }
}
image.close();
```

Výstup do súboru a vstup zo súboru

- Príklad: Zápis reťazcov znakov do súbora

```
//nemusím definovať ios::out mode ak pouzijem ofstream

ofstream file("example.txt");
if (file.is_open()) {
        file << "Lorem ipsum dolor sit amet, consectetur adipiscing elit" << endl;
        file << "sed do eiusmod tempor incididunt ut labore" << endl;
        file << "Suspendisse interdum consectetur libero id faucibus" << endl;
        file.close();
}
else {
        cout << "Subor sa nepodarilo otvorit!" << endl;
}
```

Výstup do súboru a vstup zo súboru

- Príklad: Načítanie reťazcov znakov zo súbora

```
...
string line;
ifstream file("example.txt"); //nemusím definovať mode
if (file.is_open()) {
        while (getline(file, line)) {
                cout << line << endl;
        }
        file.close();
} else {
        cout << "Subor sa nepodarilo otvorit!" << endl;
}
```

Výstup do súboru a vstup zo súboru

- Na zistenie konca súboru môžeme použiť aj funkciu `bool eof()`

- Príklad: Načítanie reťazcov znakov zo súbora

```
...
string line;
ifstream file("example.txt");
if (file.is_open()) {
        while (!file.eof()){
                getline(file, line);
                cout << line << endl;
        }
        file.close();
}
```

Dynamická alokácia pamäte

- Alokácia sa robí sa pomocou príkazu

```
new typ
```

- Ak chceme premennú alokovať s priradením hodnoty:

```
new typ(value)
```

- Dealokácia premennej:

```
delete pointer;
```

- Príklad: Alokácia celočíselnej premennej

```
int* cislo = new int;
*cislo = 10;
cout << "Na adrese " << cislo << " je ulozena hodnota " << *cislo << endl;
delete cislo;
```

Dynamická alokácia pamäte

- Príklad: Alokácia reálneho čísla:

```
float a = 2.5;
float* b = new float(3.2);
float* c = new float;
cout << "a = " << a << " b = " << *b << " c = " << *c << endl;
*c = a + *b;
cout << "a = " << a << " b = " << *b << " c = " << *c << endl;
delete b;
delete c;
```

Dynamická alokácia pamäte

- Príklad: Alokácia štruktúry

```
struct bod {
        int x, y;
};

bod A = {100,150};
cout << "Bod A ma suradnice: " << A.x << " " << A.y << endl;
bod* B = new bod(A);
B->x += 20;  // to iste ako (*B).x += 20;
B->y -= 40; // to iste ako (*B).y -= 40;
cout << "Bod B ma suradnice: " << B->x << " " << B->y << endl;
delete B;
```

Dynamická alokácia pamäte

- Pole s dĺžkou n môžeme alokovať takto: `new typ[n]`
- A dealokujeme ho: `delete[] array`
- Príklad: Výpočet hodnôt funkcie f(x)=x^2 na intervale <0,1>

```
int n;
cout << "V kolkych bodoch sa ma funkcia vypocitat?" << endl;
cin >> n;
float* x = new float[n];
float* fx = new float[n];

for (int i = 0; i < n; i++) {
        x[i] = i * (1. / (n - 1));
        fx[i] = x[i] * x[i];
        cout << "f(" << x[i] << ")=" << fx[i] << endl;
}
delete[] x; delete[] fx;
```

Dynamická alokácia pamäte

- Pri dvojrozmerných poliach alokujeme jednorozmerne pole smerníkov na daný typ.
- Pre každý smerník tohto poľa alokujeme jednorozmerne pole prvkov daného typu.
- Stratégia po riadkoch – alokácia poľa s rozmermi $m \times n$:
- Alokácia:

```
int m = 3, n = 3;
int** pole = new int* [m];
for (int i = 0; i < m; i++) {
        pole[i] = new int[n];
}
```

- Prístup k prvkom:

```
for (int i = 0; i < m; i++){
        for (int j = 0; j < n; j++){
                pole[i][j] = i*n + j;
                cout << pole[i][j] << " ";
        }
        cout << endl;
}
```

Dynamická alokácia pamäte

- Dealokácia:

```
for (int i = 0; i < m; i++) {
        delete[] pole[i];
}
delete[] pole;
```

- Ako zapísať 2D pole ako 1D?

```
int m = 3, n = 3;
int* pole = new int [m*n];
for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
                pole[i * n + j] = i * n + j;
                cout << pole[i * n + j] << " ";
        }
        cout << endl;
}
```

Dynamická alokácia pamäte

- Ak sú všetky rozmery poľa okrem jedného známe v čase kompilácie, môžeme alokovať viacrozmerné pole nasledujúcim spôsobom:

```
typ (*pole)[const1][const2][const3]...=new typ[d][const1][const2][const3]...
```

- Dealokujeme klasickým spôsobom

```
delete[] pole;
```

- Príklad: Alokácia poľa celých čísel s rozmermi 5x10

```
int n = 5;
int(*pole)[10] = new int[n][10];
```

- Nasledujúci zápis nie je dovolený! - chyba: **expression did not evaluate to a constant**

```
int n = 5;
int m = 10;
int(*pole)[m] = new int[n][m];
```

Memory leak

- Ku každej dynamickej alokácii musí byť urobená príslušná dealokácia!!
- Predstavme si situáciu, keď máme nejakú funkciu f

```
void f(int n){
        char *pole=new int[n];
        ...
}
```

- Čo sa stane, ak vo funkcii f chýba dealokácia?
- Lokálna premenná pole zanikne, ale v pamäti ostane alokovaných n pamäťových miest, ku ktorým už nie je možný prístup.
- Takéto miesto v pamäti sa nazýva **memory leak** a je častou a niekedy ťažko odhaliteľnou programátorskou chybou.

Rozdiely medzi alokáciou v C a C++

- V **C** na alokáciu používame funkcie malloc(..) a free( ), zatiaľ čo v **C++** máme operátory new a delete
- Operátor **new** umožňuje alokovať premennú aj s nastavením jej hodnoty (pri objektoch zavolá koštruktor), malloc to neumožňuje
- Operátor **delete** pri objektoch zavolá ich deštruktor, free nie
- V C je možnosť **realokovať**, teda zmeniť veľkosť vyhradeného pamäťového miesta, v C++ to možné **nie je**
- malloc pri neúspešnej alokácii vráti hodnotu NULL, new oznámi výnimku
- Operátory new[] a delete[] umožňujú priamu prácu s poľami, čo malloc a free neumožňujú
- new je rýchlejšie ako malloc, ale free je rýchlejšie ako delete. Ale celková kombinácia new/delete je rýchlejšia ako malloc/free.

Argumenty funkcií

- Ako poslať premennú ako argument funkcie?
  - Ako hodnotu premennej:

```
int increase(int y) { //vytvorí sa lokálna kópia premennej
        return y += 1;
}
```

volanie: `int a = 1; a = increase(a); cout << a;`

  - Ako adresu premennej:

```
void increasePtr(int* y) { //nevytvára sa kópia, pristupuje sa priamo k premennej
        *y += 1;
}
```

volanie: `int a = 1; increase(&a); cout << a;`

  - Ako adresu na konštantnú (nemennú) premenu:

```
int increaseConstPtr(const int* y) { //nevytvára sa kópia, ale mám zabránené meniť premennú na adrese
        return *y + 1;
}
```

volanie: `int a = 1; int b; b = increase(&a); cout << b;`

Argumenty funkcií

- Ako poslať premennú ako argument funkcie?

  - **Ako referenciu na premennú**:

```
void increaseRef(int& y) { //nevytvorí sa kópia, upravuje sa priamo premenná
        y += 1;
}
```

volanie: `int a = 1; increase(a); cout << a;`

  - **Ako referenciu na konštatnú premmenu**:

```
//nevytvára sa kópia, ale mám zabránené meniť premmenú
int increaseConstRef(const int& y) {
        return y + 1;
}
```

volanie: `int a = 1; int b; b = increase(a); cout << b;`

Triedy v C++

- Definícia triedy:

```
class meno_triedy {
private:
        typD1 data1;
        typD2 data2;
        ...
        typM1 Metoda1(...);
        ...
public:
        typD11 data11;
        ...
        typM11 Metoda11(...);
        ...
};
```

← súkromné dáta a metódy

← verejné dáta a metódy

Triedy v C++

- Definovanie deklarovaných metód mimo definície triedy:

```
typM1 meno_triedy::Metoda1(...) {
        ...
}

typM11 meno_triedy::Metoda11(...) {
        ...
}
...
```

Zapúzdrenie – (Encapsulation)

- Mechanizmus, ktorý zväzuje dohromady dáta a kód
- V úplne objektovo orientovanom programe patria všetky dáta a funkcie nejakej triede (okrem hlavného programu)
- Zapúzdrenie (encapsulation) umožňuje lepšiu prehľadnosť programu a najmä môže chrániť dáta pred nežiadúcimi zásahmi zvonku
- Vo vnútri triedy môžu byť všetky dáta alebo metódy definované ako
  - **súkromné (private)** - prístupné len pre triedu samotnú
  - **verejné (public)** – prístupné aj pre ostatné triedy
- Technika **data hiding**:
  - Je to najbezpečnejší a najbežnejší spôsob návrhu tried
  - Základný princíp:
    - všetky dáta sú **súkromné (private)**
    - trieda má vytvorený **interface**, teda **verejné (public)** metódy, ktoré umožňujú zmenu a sprostredkovanie dát, ak je to potrebné
  - Takto sa zabezpečí, že pri použití triedy sú prístupné a meniteľné len tie dáta, ktorým to dovolí interface, čo je dôležité napr. pri programovaní knižníc, ktoré budú používať iní programátori a pod.

Vytvorenie objektu a prístup k dátam a metódam

- Keďže trieda je dátový typ, vytvorenie objektu je len vytvorením premennej daného typu

```
meno_triedy meno_objektu;
```

- Rovnako môžeme vytvoriť napr. aj statické pole objektov:

```
meno_triedy meno_objektu[rozmer];
```

- Ak chceme pristupovať k dátam objektu alebo použiť nejakú jeho metódu, robíme to rovnako ako pri štruktúrach:

```
meno_objektu.data1
meno_objektu.Metoda1(...)
...
```

Vytvorenie objektu a prístup k dátam a metódam

- Príklad: Vytvoríme **triedu** Human

  Táto trieda by mala obsahovať nasledujúce dáta:

  meno

  vek

  zamestnanie

  Ďalej v nej budú nasledujúce metódy:

  metóda na nastavenie hodnôt údajov

  metóda na výpis informácií o človeku

  metóda na získanie veku človeka

  metóda na zmenu zamestnania

  Trieda by mala byť vytvorená technikou **data hiding**.

Vytvorenie objektu a prístup k dátam a metódam

- Keďže máme použiť techniku data hiding, budú všetky dáta súkromné a prístup k nim bude umožnený len pomocou metód triedy

```
class Human {
private:
        std::string name, profession;
        int age;
public:
        void setData(std::string n, std::string p, int a);
        void printData();
        int getAge();
        void setProfession(std::string newP);
};
```

- Teraz definujeme metódy triedy:

```
void Human::setData(std::string n, std::string p, int a) {
        name = n;
        profession = p;
        age = a;
}
void Human::printData() {
        std::cout << "Ahoj, volam sa " << name << std::endl;
        std::cout << "Mam " << age << " rokov" << std::endl;
        std::cout << "Moje zamestnanie je " << profession << std::endl;
}
int Human::getAge() {
        return age;
}
void Human::setProfession(std::string newP) {
        profession = newP;
}
```

Vytvorenie objektu a prístup k dátam a metódam

- Volanie:

```
Human Jozef;
Jozef.setData("Jozef", "murar", 35);
Jozef.printData();


std::cout << Jozef.getAge() << std::endl;
Jozef.setProfession("elektrikar");
Jozef.printData();
```

- Výstup:

```
Ahoj, volam sa Jozef
Mam 35 rokov
Moje zamestnanie je murar
35
Ahoj, volam sa Jozef
Mam 35 rokov
Moje zamestnanie je elektrikar
```

Konštruktor a deštruktor

- Konštruktor je funkcia, ktorá sa zavolá vždy pri vytvorení objektu. Používa sa na počiatočné inicializácie (nastavenie hodnôt dát, alokáciu pamäte..). Ak nie je explicitne deklarovaný, je prednastavený ako **prázdna funkcia**.
- V C++ má konštruktor rovnaké meno ako samotná trieda. Je to metóda triedy, ale nemá návratový typ. Deklarujeme ho takto:

```
class meno_triedy {
        ...
        meno_triedy(...);
        ...
};
```

- Jeho obsah potom definujeme podobne ako pri iných metódach:

```
meno_triedy::meno_triedy() {
        ...
}
```

Konštruktor a deštruktor

- Deštruktor je funkcia, ktorá sa zavolá vždy pri zániku objektu (pri dealokovaní pamäte, zrušení lokálnych premenných pri ukončení funkcie...). Používa sa na **záverečné operácie** (dealokácia pamäte...)

- Meno deštruktora je zložené zo znaku “~” a mena triedy. Takisto nemá návratový typ a nemá ani parametre. Deklarácia:

```
class meno_triedy {
        ...
        ~meno_triedy();
        ...
};
```

- Definícia:

```
meno_triedy::~meno_triedy() {
        ...
}
```

Konštruktor a deštruktor

- Príklad: Trieda Human s konštruktorom bez parametrov. Triedu Human obohatíme o konštruktor, v ktorom sa hodnoty parametrov nastavia na počiatočné “nulové” hodnoty.

```
class Human {
private:
        std::string name, profession;
        int age;
public:
        Human(); //konštruktor musí byť public
        void setData(std::string n, std::string p, int a);
        void printData();
        int getAge();
        void setProfession(std::string newP);
};
```

Konštruktor a deštruktor

- Obsah konštruktora definujeme takto:

```
Human::Human() {
        name = "";
        profession = "";
        age = 0;
}
```

- Konštruktor sa zavolá po vytvorení objektu, t.j. po deklarácii

```
Human Martin;
Martin.printData();
```

- dostaneme:

```
Ahoj, volam sa
Mam 0 rokov
Moje zamestnanie je
```

Konštruktor a deštruktor

- Príklad: Trieda **Human** s konštruktorom s parametrami.
- Počiatočné nastavenie parametrov, ktoré sa vykonáva vo funkcii **setData()**, môžeme urobiť priamo pri vytvorení objektu v konštruktore:

```
class Human {
private:
        std::string name, profession;
        int age;
public:
        Human();
        Human(std::string n, std::string p, int a); //konštruktor s parametrami.
        void setData(std::string n, std::string p, int a);
        void printData();
        int getAge();
        void setProfession(std::string newP);
};
```

Konštruktor a deštruktor

- Konštruktor potom bude mať nasledujúci obsah:

```
Human::Human(std::string n, std::string p, int a) {
        name = n;
        profession = p;
        age = a;
}
```

- Objekt potom vytvárame spolu s uvedením hodnôt parametrov:

```
Human zamestnanec("Katarina", "uctovnicka", 26);
zamestnanec.printData();
```

- Výstup:

```
Ahoj, volam sa Katarina
Mam 26 rokov
Moje zamestnanie je uctovnicka
```

- Konštruktor s parametrami z predchádzajúceho príkladu môžeme zapísať aj takto:

```
class Human {  ...
        Human(std::string n, std::string p, int a) :name(n), profession(p), age(a) {};
...};
```

- Môžeme definovať aj konštruktor (alebo ľubovoľnú metódu triedy) s preddefinovanými parametrami:

```
class Human {  ...
        Human(std::string n, std::string p = "stazista", int a = 18);
...};
```

- Volanie:

```
Human zamestnanec("Katarina");
zamestnanec.printData();
```

- Výstup:

```
Ahoj, volam sa Katarina
Mam 18 rokov
Moje zamestnanie je stazista
```

Konštruktor a deštruktor

- Príklad: Trieda na uchovávanie znakových reťazcov (imitácia typu *string*), s definovaným deštruktorom

```
class myString {
private:
        char* s;
        int lenght;
public:
        myString(int d,const char* sInput);
        ~myString();
        char* getString();
        int getLength();
};
```

Konštruktor a deštruktor

- Konštruktor:

```
myString::myString(int d, const char* sInput) :lenght(d) {
        s = new char[lenght]; //dynamicka alokacia
        for (int i = 0; i < lenght; i++) {
                s[i] = sInput[i];
        }
}
```

- Definujeme deštruktor, ktorý použijeme na dealokáciu pamäte:

```
myString::~myString() {
        delete[] s; //dealokacia pamate alokovanej v konstruktore
        std::cout << "zavolany destruktor\n" ;
}
```

Konštruktor a deštruktor

- Ďalej definujeme ďalšie metódy triedy:

```
char* myString::getString() {
        return s;
}

int myString::getLength() {
        return lenght;
}
```

Konštruktor a deštruktor

- Použitie triedy:

```
int main(){
        myString s1(36, "Toto je test novej triedy myString\n");
        myString s2(25, "Asi sa nam to podarilo!\n");
        std::cout.write(s1.getString(), s1.getLength());
        std::cout.write(s2.getString(), s2.getLength());
        // aj keď to nie je explicitne povedané, tu sa zavolajú deštruktory
        return 0;
}
```

- Výstup:

```
Toto je test novej triedy myString
Asi sa nam to podarilo!
zavolany destruktor
zavolany destruktor
```

Konštruktor a deštruktor

- Ručné volanie deštruktora:

```
int main(){
        myString s1(36, "Toto je test novej triedy myString\n");
        myString s2(25, "Asi sa nam to podarilo!\n");
        std::cout.write(s1.getString(), s1.getLength());
        std::cout.write(s2.getString(), s2.getLength());
        s1.~myString();
        return 0;
}
```

- Takýto kód spôsobí pád programu. Prečo?
- Riešenie:

```
myString::~myString() {
        delete[] s; //dealokacia pamate alokovanej v konstruktore
        s = nullptr; //nastavenie pointera ako prazdny pointer
        std::cout << "zavolany destruktor\n" ;
}
```

Inline funkcie

- **Inline funkcia** – funkcia, ktorej obsah vloží prekladač (kompilátor) priamo na miesto jej použitia (podobne ako makro v C)
- Pri použití inline funkcie sa odbúra čas, ktorý program potrebuje na zavolanie štandardnej funkcie, inline funkcie teda urýchľujú priebeh programu
- Ako inline sa väčšinou definujú krátke a často používané funkcie, pri inlinovaní dlhých funkcií by sa program stal zbytočne dlhý a neprehľadný
- **Definícia**:

```
inline typ f(zoznam parametrov) {
        ...
}
```

- Aj keď teoreticky môže byť ľubovoľná funkcia inline, v skutočnosti **kompilátor nie vždy vyhovie požiadavkám** programátora. Niektoré kompilátory môžu odmietnuť inlinovať funkciu, ktorá obsahuje cykly, príkaz *switch*, rekurzívne funkcie a pod. V takom prípade nenastane chyba, ale funkcia sa zmení na štandardnú.
- Každá inline funkcia musí byť definovaná pred jej prvým volaním.
- Inline funkcie sú výhodnejšie ako makrá, pretože umožňujú typovú kontrolu, môžu byť štruktúrovanejšie a lepšie optimalizované kompilátorom

Inline funkcie a triedy

- Funkcie (najmä krátke) sa môžu zahrnúť priamo do definície triedy. V takom prípade sa automaticky stávajú inline funkciami. Nevyžaduje a ani nepovoľuje sa potom žiadna definícia mimo triedy.

- Príklad: Trieda `Human2` s inline funkciou `setProfession` a `getAge`

```
class Human2 {
private:
        std::string name, profession;
        int age;
public:
        Human2(std::string n, std::string p, int a);
        void printData();
        inline void setProfession(std::string newP) { profession = newP; };
        int getAge() { return age; }; //numusim ani napisat inline a bude inline
};
```

Polia objektov

- Keďže trieda je dátový typ ako každý iný, môžeme vytvárať aj polia objektov danej triedy. Postup je štandardný:

```
meno_triedy meno_pola[n]
```

- Podľa toho, aký konštruktor má daná trieda, rozlišujeme niekoľko prípadov:

1. Ak má trieda konštruktor bez parametrov, deklarujeme pole štandardne:

```
meno_triedy meno_pola[n];
```

2. Ak má trieda konštruktor s viacerými parametrami, musíme pole inicializovať

a) priamym volaním konštruktorov pre všetky prvky:

```
meno_triedy meno_pola[n]={meno_triedy(...), meno_triedy(...)...};
```

b) vymenujeme hodnoty prvkov poľa (objekty danej triedy):

```
meno_triedy meno_pola[n]={objekt1,objekt2,objekt3..};
```

Polia objektov

1. Ak má trieda konštruktor bez parametrov, deklarujeme pole štandardne:

```
class Human2 {...
public:
        Human2():name(""), profession(""), age(0){};
...}

int main(){
        Human2 zamestanci[10];
        for (size_t i = 0; i < 10; i++){
                zamestanci[i].setData("abc", "test", i);
                zamestanci[i].printData();

        }
        return 0;
}
```

Polia objektov

2. Ak má trieda konštruktor s viacerými parametrami, musíme pole inicializovať :

a) priamym volaním konštruktorov pre všetky prvky:

```
class Human2 {...
public:
    Human2(std::string n, std::string p, int a) :name(n), profession(p), age(a) {};
...}

int main(){
        Human2 zamestanci[4] = { Human2("Michal", "uradnik", 18),Human2("Maria", "uradnik", 18),
                Human2("Jozef", "uradnik", 18), Human2("Julia", "uradnik", 18)};
        for (size_t i = 0; i < 4; i++){
                zamestanci[i].printData();
        }
        return 0;
}
```

Polia objektov

2. Ak má trieda konštruktor s viacerými parametrami, musíme pole inicializovať :

   b) vymenujeme hodnoty prvkov poľa (objekty danej triedy):

```
int main(){
        Human2 a = Human2("Michal", "uradnik", 18);
        Human2 b = Human2("Jozef", "uradnik", 18);
        Human2 c = Human2("Maria", "uradnik", 18);
        Human2 d = Human2("Julia", "uradnik", 18);

        Human2 zamestanci[4] = { a,b,c,d};
        for (size_t i = 0; i < 4; i++){
                zamestanci[i].printData();
        }
        return 0;
}
```

Priraďovanie objektov

- Ak máme dva objekty rovnakého typu, môžeme jeden priradiť druhému. Vytvorí sa bitová kópia, teda hodnoty všetkých dát priraďovaného objektu sa skopírujú do príslušných dát objektu, ktorému sa priraďuje (vrátane obsahu polí).
- Priraďované objekty musia byť naozaj z tej istej triedy, nestačí, ak budú mať triedy identický obsah!

- Príklad: Priraďovanie trojuholníkov

  Najskôr si definujeme triedy, ktoré budeme potrebovať. Keďže trojuholník je definovaný tromi bodmi, najprv vytvoríme triedu Point:

```
class Point {
private:
        int x, y;
public:
        Point(int x1, int y1) :x(x1), y(y1) {}
        int X() { return x; }
        int Y() { return y; }
};
```

Priraďovanie objektov

- Teraz vytvoríme triedu Triangle:

```
class Triangle {
private:
	Point A, B, C;
public:
	Triangle(Point b1, Point b2, Point b3) :A(b1), B(b2), C(b3) {}
	void printPoints(std::string pointName);
	void changePoints(Point b1, Point b2, Point b3) { A = b1; B = b2; C = b3; }
};
void Triangle::printPoints(std::string pointName) {
	std::cout << "Triangle: " << pointName << std::endl;
	std::cout << "A[" << A.X() << ", " << A.Y() << "]" << std::endl;
	std::cout << "B[" << B.X() << ", " << B.Y() << "]" << std::endl;
	std::cout << "C[" << C.X() << ", " << C.Y() << "]" << std::endl;
}
```

Priraďovanie objektov

- Nakoniec definované triedy použijeme:

```
int main(){
	Point A(10, 10), B(20, 10), C(15, 15);
	Point D(0, 0), E(10, 0), F(3, 6);
	Triangle T1(A, B, C);
	Triangle T2(D, E, F);
	T1.printPoints("T1");
	T2.printPoints("T2");
	T2 = T1;
	T2.printPoints("T2");
	T1.changePoints(D, E, F); //obsah T2 sa už nezmení, aj keď bol kópiou T1
	T1.printPoints("T1");
	T2.printPoints("T2");
}
```

Priraďovanie objektov

- Výstup:

```
Triangle T1
A[10, 10]
B[20, 10]
C[15, 15]
Triangle T2
A[0, 0]
B[10, 0]
C[3, 6]
Triangle T2
A[10, 10]
B[20, 10]
C[15, 15]
```

```
Triangle T1
A[0, 0]
B[10, 0]
C[3, 6]
Triangle T2
A[10, 10]
B[20, 10]
C[15, 15]
```

Priraďovanie objektov

- Príklad: Priraďovanie objektov tried obsahujúcich pole

  Príklad s trojuholníkmi pozmeníme tak, že namiesto troch bodov A, B, C bude trieda *`Triangle`* obsahovať trojprvkové pole bodov.

  Keďže chceme použiť pole typu `Point`, a vnútri triedy nemáme možnosť pole inicializovať hodnotami, musíme predefinovať triedu `Point` tak, aby mala prázdny konštruktor:

```
class Point2 {
private:
        int x, y;
public:
        Point2():x(0),y(0) {}
        int X() { return x; }
        int Y() { return y; }
        void setPoint(int x1, int y1) { x = x1; y = y1; }
};
```

Priraďovanie objektov

- Teraz upravíme triedu *Triangle*. Keďže body trojholníka sú teraz uložené v poli, nemôžeme konštruktor zapísať v kompaktnej forme ako predtým.

```
class Triangle2 {
private:
        Point2 B[3];
public:
        Triangle2(Point2 b1, Point2 b2, Point2 b3) { B[0] = b1; B[1] = b2; B[2] = b3; }
        void printPoints(std::string name);
        void changePoints(Point2 b1, Point2 b2, Point2 b3) {B[0] = b1; B[1] = b2; B[2] = b3;}
};
void Triangle::printPoints(std::string printName) {
        std::cout << "Triangle " << printName << std::endl;
        std::cout << "A[" << B[0].X() << ", " << B[0].Y() << "]" << std::endl;
        std::cout << "B[" << B[1].X() << ", " << B[1].Y() << "]" << std::endl;
        std::cout << "C[" << B[2].X() << ", " << B[2].Y() << "]" << std::endl;
}
```

Priraďovanie objektov

```
int main(int argc, char* argv[]){
        Point2 A, B, C;
        A.setPoint(10, 10);
        B.setPoint(20, 10);
        C.setPoint(15, 15);
        Triangle2 T1(A, B, C);
        Triangle2 T2(C, B, A);
        T1.printPoints("T1");
        T2.printPoints("T2");
        T2 = T1;
        T1.printPoints("T1");
        T2.printPoints("T2");
        ...
}
```

Výstup:

```
Triangle T1
A[10, 10]
B[20, 10]
C[15, 15]
Triangle T2
A[15, 15]
B[20, 10]
C[10, 10]
Triangle T1
A[10, 10]
B[20, 10]
C[15, 15]
```

```
Triangle T2
A[10, 10]
B[20, 10]
C[15, 15]
```

Priraďovanie objektov

- Pri nesprávnom priraďovaní objektov môžu niekedy vzniknúť chyby, napr. memory leaky, nesprávne zásahy do pamäte..
- Príklad: Priraďovanie objektov typu `myString`

  Najskôr rozšírime triedu `myString` o metódu `changeString`:

```
class myString {
private:
        char* s;
        int lenght;
public:
        myString(int d, char* sInput);
        ~myString();
        char* getString();
        int getLength();
        void changeString(int d, const char* sInput);
};
```

Priraďovanie objektov

```
void myString::changeString(int d, const char* sInput) {
        delete[] s;
        lenght = d;
        s = new char[lenght];
        for (int i = 0; i < lenght; i++) {
                s[i] = sInput[i];
        }
}
```

Priraďovanie objektov

- V hlavnom programe máme tieto príkazy:

```
myString s1(35, "Toto je test novej triedy myString");
myString s2 = s1;
std::cout.write(s1.getString(), s1.getLength());std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());std::cout << std::endl;

s1.changeString(20, "Podarilo sa nam to?");
std::cout.write(s1.getString(), s1.getLength());std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());
```

- Nedostali sme očakávaný výstup a program spadol.
- **Čo sme urobili zle?**

Priraďovanie objektov

- Pozrime sa ešte raz na náš program:

```
myString s1(35, "Toto je test novej triedy myString");
myString s2 = s1;
```

- Reťazec *s2* ešte nebol inicializovaný, nestratíme teda žiadne informácie, a všetky dáta z *s1* sa doňho skopírujú. No práve to spôsobí problém - okrem toho, že reťazce *s1* a *s2* budú mať rovnaký obsah a rovnakú dĺžku, ich smerníky *s* **budú ukazovať na rovnaké miesto v pamäti.**

  Keď potom program vykoná tento príkaz:

```
s1.changeString(20, "Podarilo sa nam to?");
```

  zmení sa ním dĺžka a obsah *s1*. Ale nielen to – keďže sa zmení obsah na mieste v pamäti, kde ukazuje smerník *s1.s*, a *s1.s=s2.s*, tak sa **zmení aj obsah *s2***. Ale dĺžka *s2* sa nezmení. Teda okrem toho, že v *s2* budeme mať nesprávny obsah, tento obsah sa aj nesprávne vypíše.

  Nakoniec program spadne preto, pretože dva deštruktory sa budú pokúšať uvoľniť to isté pamäťové miesto a to sa im nepodarí.

Priraďovanie objektov

- Čo by sa stalo, ak by sme program prepísali nasledujúcim spôsobom?

```
myString s1(35, "Toto je test novej triedy myString");
myString s2(25, "Napiseme zakerny program");
s2 = s1;
```

- V tomto prípade by sa po priradení do *s2* nakopírovali hodnoty z *s1*, aj keď *s2* už bolo predtým inicializované. **To znamená, že smerník *s2.s* teraz ukazuje na to isté miesto, na ktoré ukazuje *s1.s*.**
- Pamäťové miesto, na ktoré *s2.s* ukazoval predtým, nebolo dealokované a už naň neukazuje nijaký smerník. Teda vznikol **memory leak**. Navyše na konci programu dôjde opäť k dvojnásobnej dealokácii a program spadne.

- **V podobných prípadoch sa teda musíme postarať o to, aby nám nezmizli nejaké dôležité dáta a aby nevznikali duplicity a nejednoznačnosti.**

Smerník na objekt

- Dynamické alokovanie objektov sa podobá alokovaniu premenných. Operátor *new* okrem vytvorenia potrebného miesta v pamäti zavolá aj konštruktor objektu. *delete* zase zavolá príslušný deštruktor.
- **Alokácia**:

  Ak má trieda konštruktor bez parametrov:

```
meno_triedy *meno_objektu=new meno_triedy;
```

  Ak má trieda konštruktor s parametrami:

```
meno_triedy *meno_objektu=new meno_triedy(zoznam parametrov);
```

- **Dealokácia:**

```
delete meno_objektu;
```

- Môžeme alokovať aj **polia objektov**, v tomto prípade je však nutné, aby trieda mala konštruktor bez parametrov:

```
meno_triedy *meno_pola = new meno_triedy[n];
delete[] meno_pola;
```

Smerník na objekt

- **Prístup k dátam a metódam objektu**

  Ak máme dynamicky alokovaný objekt:

  ```
  meno_triedy *meno_objektu=new meno_triedy;
  ```

  potom k jeho dátam a metódam pristupujeme podobne ako pri štruktúrach:

  ```
  (*meno_objektu).data
  (*meno_objektu).Metoda(...)
  ```

  alebo

  ```
  meno_objektu->data
  meno_objektu->Metoda(...)
  ```

- **Získanie adresy objektu**

  Ak máme už vytvorený objekt nejakej triedy, môžeme získať jeho adresu rovnako ako pri premenných ľubovoľného iného typu:

  ```
  adresa_objektu = &meno_objektu
  ```

Smerník na objekt

- Príklad: Smerník na objekt triedy Human

1. Najskôr uvažujeme konštruktor bez parametrov

```
Human* Jozef = new Human();
Jozef->setData("Jozo", "murar", 35);
Jozef->printData();
delete Jozef;
```

2. Teraz uvažujeme konštruktor s parametrami

```
Human* Maria = new Human("Maria", "učtovníčka", 30);
Maria->printData();
delete Maria;
```

Alokácia poľa objektov

- Príklad: Alokácia poľa objektov triedy Human

  Vytvoríme pole študentov a každý študent sa predstaví. Uvažujeme verziu triedy *Človek* s prázdnym konštruktorom.

```
Human* student = new Human[2];
student[0].setData("Peter", "Student", 20);
student[1].setData("Filip", "Student", 22);
for (int i = 0; i < 2; i++) {
        student[i].printData();
}
delete[] student;
```

Alokácia poľa objektov

- Príklad: Alokácia poľa smerníkov na objekty

  Budeme opäť pracovať so študentami, ale teraz vytvoríme pole tak, aby jeho prvkami boli smerníky na objekt triedy Human.

```
Human** student = new Human* [3];
student[0] = new Human("Peter", "Student", 20);
student[1] = new Human("Filip", "Student", 22);
student[2] = new Human("Maria", "Student", 21);
for (int i = 0; i < 3; i++) {
        student[i]->printData();
}
for (int i = 0; i < 3; i++) {
        delete student[i];
}

delete[] student;
```

Alokácia poľa objektov

- Treba si uvedomiť rozdiel pri **vytváraní poľa objektov a poľa smerníkov na objekty**
- Príklad: Pole objektov a pole smerníkov na objekt s konštruktorom bez parametrov

  Nech má trieda *Človek* teraz takýto konštruktor:

```
Human::Human() {
        name = "Mr. Nobody";
        profession = "unknown";
        age = 0;
}
```

- Alokovaním poľa objektov sa priamo spustí **konštruktor a objekty budú mať inicializované hodnoty.**

```
Human* student = new Human[2];
for (int i = 0; i < 2; i++) {
        student[i].printData();
}
```

Alokácia poľa objektov

- Dostaneme výstup:

```
Ahoj, volam sa Mr. Nobody
Mam 0 rokov
Moje zamestnanie je unknown
```

- Ak teraz vytvoríme pole smerníkov na objekty triedy Human, len vytvorením smerníkov **sa konštruktory nespustia**! Preto musíme ešte aj vytvoriť objekty, na ktoré budú smerníky ukazovať, tak, ako sme to videli v predchádzajúcom príklade. Ak použijeme podobný kód ako v prípade poľa objektov:

```
Human** studentPtr = new Human*[2];
for (int i = 0; i < 2; i++) {
        studentPtr[i]->printData();
}
```

- aký výstup dostaneme? ... Program spadne.

Alokácia poľa objektov

- Správne musí byť:

```
Human** studentPtr = new Human*[2];
for (int i = 0; i < 2; i++) {
        studentPtr[i] = new Human();
        studentPtr[i]->printData();
}
for (int i = 0; i < 2; i++) {
        delete studentPtr[i];
}
delete[] studentPtr;
```

Objekt ako parameter funkcie

- Objekt môže vstupovať do funkcie ako parameter, rovnako ako ľubovoľný iný dátový typ
- Príklad: Program na počítanie s n-uholníkmi

  Podobne ako pri príklade s trojuholníkmi si najprv definujeme triedu `Point`:

```
class Point {
private:
        int x, y;
public:
        Point() :x(0), y(0) {}
        int X() { return x; }
        int Y() { return y; }
        void setPoint(int x1, int y1) { x = x1; y = y1; }
};
```

Objekt ako parameter funkcie

```
class Polygon {
private:
        Point B[10];//vrcholy n-uholníka
        int n;//počet vrcholov
public:
        Polygon(int n, Point P[]) :n(n) { setPoints(P); }
        void setPoints(Point P[]) { for (int i = 0; i < n; i++) { B[i] = P[i]; } }
        void printPoints(std::string meno);
        int N() { return n; }
        int BX(int index) { return B[index].X(); } //vráti x-súradnicu bodu
        int BY(int index) { return B[index].Y(); } //vráti y-súradnicu bodu
};
void Polygon::printPoints(std::string name) {
        std::cout << "n-uholnik " << name << std::endl;
        for (int i = 0; i < n; i++) {
                std::cout << i << ". vrchol: " << B[i].X() << "  " << B[i].Y() << std::endl;
        }
}
```

Objekt ako parameter funkcie

- Teraz definujeme ďalšiu triedu – `Calculator` – ktorá bude obsahovať funkcie na počítanie s mnohouholníkmi. Pre jednoduchosť teraz uvedieme triedu `Calculator` len s jednou funkciou – `perimeter`. Parametrom metódy `perimeter` bude objekt triedy `Polygon`.

```
class Calculator {
public:
        double perimeter(Polygon U);
};


double Calculator::perimeter(Polygon U) {
        double o = 0.;
        for (int i = 0; i < U.N() - 1; i++) {
                o += sqrt(pow(U.BX(i + 1) - U.BX(i), 2) + pow(U.BY(i + 1) - U.BY(i), 2));
        }
        o += sqrt(pow(U.BX(U.N() - 1) - U.BX(0), 2) + pow(U.BY(U.N() - 1) - U.BY(0), 2));
        return o;
}
```

Objekt ako parameter funkcie

- Nakoniec vytvorené triedy použijeme:

```
int main() {
	Point B[4];
	B[0].setPoint(0, 0);
	B[1].setPoint(1, 0);
	B[2].setPoint(1, 1);
	B[3].setPoint(0, 1);

	Polygon square(4, B);
	Calculator K;

	square.printPoints("Stvorec");
	std::cout << "Obvod stvorca je: " << K.perimeter(square) << std::endl;
	return 0;
}
```

Objekt ako parameter funkcie

- Výstup:

```
n-uholnik Stvorec
0. vrchol: 0  0
1. vrchol: 1  0
2. vrchol: 1  1
3. vrchol: 0  1
Obvod stvorca je: 4
```

- Ak vstupuje objekt do funkcie ako parameter, vytvorí sa v pamäti z neho kópia a tá sa potom používa vnútri funkcie. Preto nie je možné vo funkcii meniť hodnoty dát, keďže zmeny by sa prejavili len na kópii. Po skončení funkcie kópia **objektu zanikne**.
- Pri vytváraní kópie sa **nevolá** konštruktor objektu, pretože stačí hodnoty skopírovať, nemusia sa inicializovať nové. Pri zániku kópie sa volá **deštruktor**.

Objekt ako parameter funkcie

- Ak posielame do funkcie ako parameter priamo objekt, má to niekoľko nevýhod. Okrem toho, že sa vytvára kópia celého objektu, teda spotrebováva sa ďalšia pamäť, skrýva tento spôsob aj isté nebezpečenstvo, ak objekt **dynamicky alokuje pamäť**.
- Príklad: Trieda Polygon s dynamicky alokovaným poľom

```
class Polygon {
private:
        Point* B;
        int n;
public:
        Polygon(int n, Point* P) :n(n) { setPoints(P); };
        ~Polygon();
        void printPoints(std::string name);
        void setPoints(Point* P);
        int N() { return n; }
        int BX(int index) { return B[index].X(); } //vráti x-súradnicu bodu
        int BY(int index) { return B[index].Y(); } //vráti y-súradnicu bodu
};
```

Objekt ako parameter funkcie

- Definujeme nový konštruktor a deštruktor:

```
void Polygon::setPoints(Point* P){
        B = new Point[n];
        for (int i = 0; i < n; i++) {
                B[i] = P[i];
        }
}
Polygon::~Polygon() {
        std::cout << "Dealokujem pole..." << std::endl;
        delete[] B;
}
```

- Obsah hlavného programu ostane nezmenený.
- Aký dostaneme výstup? ...Program spadne. Čo sa stalo?
- Pri vytvorení kópie sa do smerníka B kópie nakopírovala adresa zo smerníka B originálu. Pri zániku kópie sa pamäť na tejto adrese dealokovala, teda ani originál už nemá alokovanú pamäť. Pri zániku originálu na konci programu, t.j. pri pokuse znovu dealokovať tú istú pamäť, program spadne.

Objekt ako parameter funkcie

- Jeden spôsob, ako sa vyhnúť uvedeným problémom, je posielať do funkcie ako parameter **smerník na objekt**. V tom prípade odovzdávame funkcii len adresu, nevytvára sa teda žiadna kópia a nespúšťa sa deštruktor. Ušetrí sa tiež pamäť.

- Príklad: Počítanie s n-uholníkmi, so smerníkmi na objekty ako parametrami funkcií

  V triede `Calculator` prepíšeme metódu *perimeter*:

```
double Calculator::perimeterPtr(Polygon* U) {
        double o = 0.;
        for (int i = 0; i < U->N() - 1; i++) {
                o += sqrt(pow(U->BX(i + 1) - U->BX(i), 2) + pow(U->BY(i + 1) - U->BY(i), 2));
        }
        o += sqrt(pow(U->BX(U->N() - 1) - U->BX(0), 2) + pow(U->BY(U->N() - 1) - U->BY(0), 2));
        return o;
}
```

- V hlavnom programe ju potom zavoláme takto:

```
std::cout << "Obvod stvorca je: " << K.perimeterPtr(&square) << std::endl;
```

Objekt ako parameter funkcie

- Príklad: Počítanie obvodu n-uholníka pomocou odkazu (referencie) ako parametra funkcie

  V triede `Calculator` upravíme metódu *perimeter* tak, aby jej parametrom bola referencia na objekt triedy `Polygon`:

```
double Calculator::perimeterRef(Polygon& U) {
        double o = 0.;
        for (int i = 0; i < U.N() - 1; i++) {
                o += sqrt(pow(U.BX(i + 1) - U.BX(i), 2) + pow(U.BY(i + 1) - U.BY(i), 2));
        }
        o += sqrt(pow(U.BX(U.N() - 1) - U.BX(0), 2) + pow(U.BY(U.N() - 1) - U.BY(0), 2));
        return o;
}
```

- V hlavnom programe ju potom zavoláme bez operátora &:

```
std::cout << "Obvod stvorca je: " << K.perimeterRef(square) << std::endl;
```

Objekt ako parameter funkcie

- Príklad: Vytvorte funkciu v triede `Calculator`, ktorá dynamicky alokuje vstupný objekt triedy `Polygon` ktorému nastaví n bodov s náhodnými súradnicami.

  V triede `Calculator` vytvoríme metódu `generateRandomPoly` tak, aby jej parametrom bol smerník na objekt triedy `Polygon` a hodnota n.

```
void Calculator::generateRandomPoly(Polygon* U, int n){
        Point* P = new Point[n];
        for (int i = 0; i < n; i++){
                std::random_device r;
                std::default_random_engine generator(r());
                std::uniform_int_distribution<int> distribution(1, 6);
                int x = distribution(generator);
                int y = distribution(generator);
                P[i].setPoint(x, y);
        }
        U = new Polygon(n, P);
        delete[] P;
}
```

Objekt ako parameter funkcie

- V hlavnom kóde zavoláme:

```
Calculator K;
Polygon* randomPoly = nullptr;
K.generateRandomPoly(randomPoly, 5);
randomPoly->printPoints("Nahodny n-uholnik");
```

- Program spadne vo funkcii `printPoints()`. Prečo?

Objekt ako parameter funkcie

- Riešenie: **referencia na smerník**

  V triede `Calculator` vytvoríme metódu `generateRandomPolyRef` tak, aby jej parametrom **bola referencia na smerník** na objekt triedy `Polygon` a hodnota n.

```
void Calculator:: generateRandomPolyRef(Polygon*& U, int n){
        Point* P = new Point[n];
        for (int i = 0; i < n; i++){
                std::random_device r;
                std::default_random_engine generator(r());
                std::uniform_int_distribution<int> distribution(1, 6);
                int x = distribution(generator);
                int y = distribution(generator);
                P[i].setPoint(x, y);
        }
        U = new Polygon(n, P);
}
```

Objekt ako parameter funkcie

- V hlavnom kóde zavoláme:

```
Calculator K;
Polygon* randomPoly = nullptr;
K.generateRandomPolyRef(randomPoly, 5);
randomPoly->printPoints("Nahodny n-uholnik");
```

- Výstup (napr.):

```
n-uholnik Nahodny n-uholnik
0. vrchol: 2  2
1. vrchol: 5  5
2. vrchol: 2  5
3. vrchol: 2  1
4. vrchol: 6  4
```

Vrátenie objektu funkciou

- Príklad: Funkcia na posunutie n-uholníka

  Do triedy `Calculator` doplníme metódu, ktorá vráti n-uholník posunutý oproti pôvodnému o zadaný vektor (dx,dy). Uvažujeme verziu triedy `Polygon` **so staticky alokovaným poľom *B***.

```
class Calculator {
public:
        double perimeter(Polygon U);
        Polygon displacement(Polygon& U, int dx, int dy);
};
Polygon Calculator::displacement(Polygon& U, int dx, int dy) {
        Point* B = new Point[U.N()];
        for (int i = 0; i < U.N(); i++) {
                B[i].setPoint(U.BX(i) + dx, U.BY(i) + dy);
        }
        Polygon U1(U.N(), B);
        delete[] B;
        return U1;
}
```

Vrátenie objektu funkciou

- Do hlavného programu potom doplníme volanie funkcie:

```
Point B[4];
B[0].setPoint(0, 0);
B[1].setPoint(1, 0);
B[2].setPoint(1, 1);
B[3].setPoint(0, 1);
Polygon square(4, B);
square.printPoints("Originalny stvorec");
Calculator K;
Polygon newSquare = K.displacement(square, 1, 1);
newSquare.printPoints("Posunuty stvorec");
```

Vrátenie objektu funkciou

- Dostaneme výstup:

```
n-uholnik Originalny stvorec
0. vrchol: 0  0
1. vrchol: 1  0
2. vrchol: 1  1
3. vrchol: 0  1
n-uholnik Posunuty stvorec
0. vrchol: 1  1
1. vrchol: 2  1
2. vrchol: 2  2
3. vrchol: 1  2
```

Vrátenie objektu funkciou

- Tak ako v prípade, keď bol objekt parametrom funkcie, aj pri vracaní objektu funkciou sa vytvorí v pamäti kópia objektu, ktorý sa vracia. Po jeho priradení do iného objektu v programe kópia zanikne a zavolá sa deštruktor. To môže spôsobiť podobné problémy, ako sme videli v prípade objektu v úlohe parametra.
- Príklad: Trieda `Polygon` s dynamicky alokovaným poľom

  Ak máme v triede `Polygon` dynamicky alokované pole, a chceme použiť metódu `displacement` triedy `Calculator`, **program spadne**.
- Čo sa stalo?

  Vo funkcii `displacement` sa vytvorí lokálna premena *U1*– objekt triedy `Polygon`, pričom sa dynamicky alokuje pole objektov triedy `Point`. Keďže tento objekt figuruje ako parameter pre *return*, vytvorí sa z neho kópia, ktorá je vrátená funkciou. Po odchode z funkcie premena *U1* zanikne, spustí sa jej deštruktor a pole bodov sa dealokuje.

  Na konci funckie main sa spusti deštruktor pre newSqaure, ktorý sa pokúša **dealokovať to isté pole**. Vtedy program spadne.

- Ako teda správne vrátiť objekt z funkcie?

Vrátenie objektu funkciou

- Príklad: Funkcia **displacementRef** s vrátením odkazu

```
class Calculator {
public:
        double perimeter(Polygon U);
        Polygon displacement(Polygon& U, int dx, int dy);
        Polygon& displacementRef(Polygon& U, int dx, int dy);
};
Polygon& Calculator::displacementRef(Polygon& U, int dx, int dy) {
        Point* B = new Point[U.N()];
        for (int i = 0; i < U.N(); i++) {
                B[i].setPoint(U.BX(i) + dx, U.BY(i) + dy);
        }
        Polygon U1(U.N(), B);
        delete[] B;
        return U1;
}
```

Vrátenie objektu funkciou

- Volanie:

```
Polygon& newSquare = K.displacementRef(square, 1, 1);
newSquare.printPoints("Posunuty stvorec");
```

- V tomto prípade sme sa zbavili problému s vytváraním kópie, výstup však nie je o nič lepší. Program síce nespadne, ale dáta sú zlé.
- Vo funkcii sme vrátili referenciu na lokálnu premennú U1, ktorá pri odchode z funkcie **zanikne, teda zavolá sa jej deštruktor**.
- V hlavnom programe teda už táto referencia ukazuje na objekt, ktorý už neexituje.

Vrátenie objektu funkciou

- Príklad: Funkcia `displacementPtr` s vrátením smerníka

```
class Calculator {
public:
        double perimeter(Polygon U);
        Polygon* displacementPtr(Polygon& U, int dx, int dy);
};

Polygon* Calculator::displacementPtr(Polygon& U, int dx, int dy) {
        Point* B = new Point[U.N()];
        for (int i = 0; i < U.N(); i++) {
                B[i].setPoint(U.BX(i) + dx, U.BY(i) + dy);
        }
        Polygon* U1 = new Polygon(U.N(), B);
        delete[] B;
        return U1;
};
```

Vrátenie objektu funkciou

- Volanie:

```
Polygon newSquare = *(K.displacementPtr(square, 1, 1));
newSquare.printPoints("Posunuty stvorec");
```

- Teraz dostaneme výstup, ktorý vyzerá byť poriadku.

- V tomto prípade však nastane iný problém. Vo funkcii `displacementPtr` sme alokovali pomocou new objekt, ten však nikde nebol dealokovaný. Vznikol nám tak **memory leak**.
- Keďže vo funkcii main už nemáme kde objekt dealokovať, tento postup je takisto **nesprávny**.

Vrátenie objektu funkciou

- Ako odstrániť memory leak?

```
Polygon* newSquare = K.displacementPtr(square, 1, 1);
newSquare->printPoints("Posunuty stvorec");
delete newSquare;
```

- Semerník na dynamicky alokovaný `Polygon` vo funkcii `displacementPtr` sa skopíruje do `Polygon* newSquare` a na konci sa vymaze obsah na ktorý ukazuje.

Vrátenie objektu funkciou

- Príklad: Funkcia **displacement** ktorá na vstup dostane referenciu na posunutý n-uholník

```
class Calculator {
public:
        double perimeter(Polygon U);
        Polygon displacement(Polygon& U, int dx, int dy);
        void displacement(Polygon& U, Polygon& Unew, int dx, int dy);
};
void Calculator::displacementRef(Polygon& U, Polygon& Unew, int dx, int dy){
        Point* B = new Point[U.N()];
        for (int i = 0; i < U.N(); i++) {
                B[i].setPoint(U.BX(i) + dx, U.BY(i) + dy);
        }
        Unew.setPoints(B);
        delete[] B;
}
```

Vrátenie objektu funkciou

- Volanie:

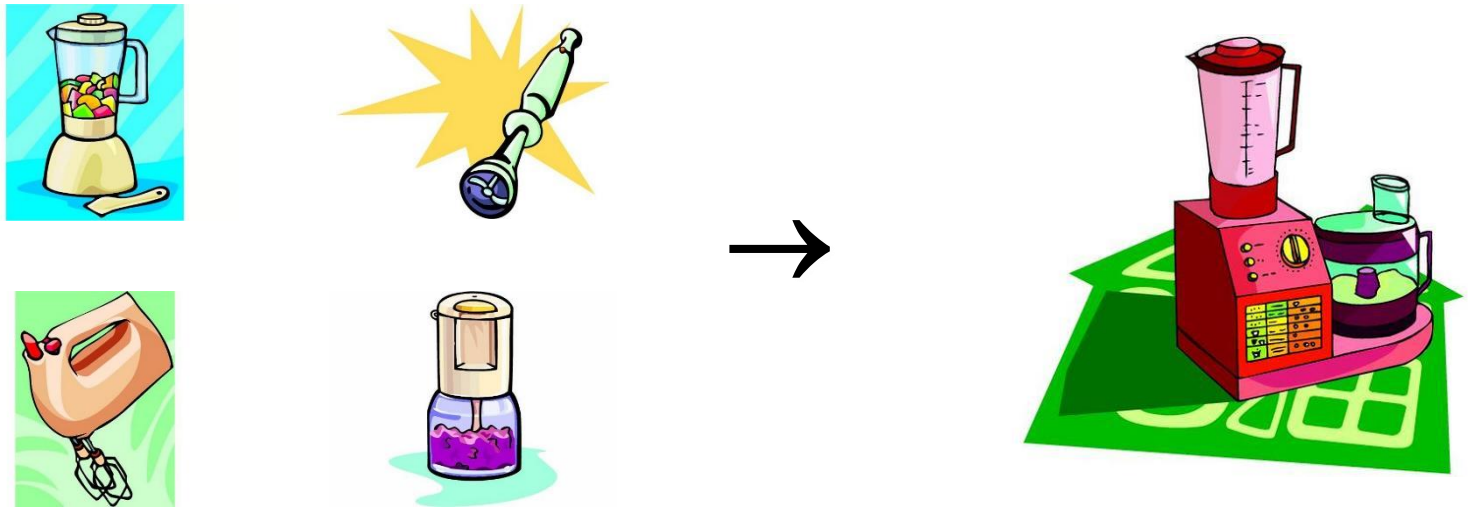
```
Polygon newSquare(4, B);
K.displacementRef(square, newSquare, 1, 1);
newSquare.printPoints("Posunuty stvorec");
```

- Do funkcie sme odovzdali odkaz na pôvodný aj nový štvorec – nevytvára sa žiadna kopia. Doležíte je aby bola pamäť pre body **nového štvorce alokovaná ešte pred zavolaním funkcie na posun** (to sa deje v tomto prípade v konštruktore)
- Výstup:

```
n-uholnik Stvorec
0. vrchol: 0  0
1. vrchol: 1  0
2. vrchol: 1  1
3. vrchol: 0  1
n-uholnik Posunuty stvorec
0. vrchol: 1  1
1. vrchol: 2  1
2. vrchol: 2  2
3. vrchol: 1  2
Dealokujem pole...
Dealokujem pole...
```

Polymorfizmus - mnohotvárnosť

- Ide o viacúčelové využitie funkcií alebo operátorov
- Metóda s jedným názvom môže byť použitá pre rôzne typy dát, rôzny počet vstupov alebo rôzne návratové hodnoty, čo uľahčuje orientáciu v programe



Preťažovanie funkcií

- **Ide o situáciu, keď v programe definujeme viacero funkcií s tým istým názvom, ale s rôznymi vstupmi alebo výstupmi**.

Mlynček(na mäso)

Mlynček(na mak)

Mlynček(na orechy)

- V programe to teda vyzerá takto:

  typ1 Funkcia(parametre1);

  typ2 Funkcia(parametre2);

  typ3 Funkcia(parametre3);

  ...
- Podľa situácie sa prekladač sám vie rozhodnúť, ktorý tvar funkcie mal programátor na mysli – jednotlivé verzie sa musia líšiť počtom alebo typom parametrov.
- **Preťažovanie môžeme využiť nielen v objektovom, ale aj v procedurálnom programovaní**

Preťažovanie funkcií

- Príklad: Počítanie absolútnych hodnôt čísel pomocou procedurálneho programovania

  Vytvoríme funkciu *absolut*, ktorá bude počítať absolútnu hodnotu celého alebo reálneho čísla.

```
int absolut(int x){
        if (x > 0) return x;
        else return -x;
}

float absolut(float x){
        if (x > 0) return x;
        else return -x;
}
```

Preťažovanie funkcií

- V hlavnom programe potom funkcie použijeme:

```
int main() {
        int a = -1;
        float b = 2.3f;
        std::cout << "Absolutna hodnota cisla " << a << " je " << absolut(a) << std::endl;
        std::cout << "Absolutna hodnota cisla " << b << " je " << absolut(b) << std::endl;
        return 0;
}
```

- Dostaneme výstup:

Absolutna hodnota cisla -1 je 1

Absolutna hodnota cisla 2.3 je 2.3

Preťažovanie funkcií

- Príklad: Počítanie absolútnych hodnôt čísel pomocou OOP

  Vytvoríme triedu `Calculator`, ktorá bude počítať absolútne hodnoty z celých, reálnych a komplexných čísel.

- Keďže budeme počítať aj s komplexnými číslami, vytvoríme si najprv triedu `ComplexNumber`:

```
class ComplexNumber {
private:
        float a, b;
public:
        ComplexNumber(float real, float imag) { a = real; b = imag; }
        float A() { return a; }
        float B() { return b; }
};
```

Preťažovanie funkcií

- Teraz definujeme triedu `Calculator`:

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        float abs(ComplexNumber& x);
};

int Calculator::abs(int x) {
        if (x > 0) return x;
        else return -x;
}
```

Preťažovanie funkcií

- Varianty funkcie pre reálne a komplexné čísla:

```
float Calculator::abs(float x) {
        if (x > 0) return x;
        else return -x;
}

float Calculator::abs(ComplexNumber& x) {
        return sqrt(x.A() * x.A() + x.B() * x.B());
}
```

Preťažovanie funkcií

- V hlavnom programe vytvoríme objekty a vypočítame absolútne hodnoty:

```
int n = -3;
float f = 2.5f;
ComplexNumber c(3, 4);
Calculator K;
std::cout << "Absolutna hodnota cisla " << n << " je " << K.abs(n) << std::endl;
std::cout << "Absolutna hodnota cisla " << f << " je " << K.abs(f) << std::endl;
std::cout << "Absolutna hodnota cisla " << c.A() << "+" << c.B() << "i je " << K.abs(c) << std::endl;
```

- Dostaneme výstup:

  Absolutna hodnota cisla -3 je 3

  Absolutna hodnota cisla 2.5 je 2.5

  Absolutna hodnota cisla 3+4i je 5

Použitie štandardných argumentov

- Príklad: Kalkulačka, ktorá počíta obsah obdĺžnika

  Do triedy `Calculator` doplníme ďalšiu funkciu – výpočet obsahu obdĺžnika pre zadané strany *a,b*. Keďže obdĺžnik môže byť v skutočnosti aj štvorec, definujeme aj verziu funkcie, ktorá bude mať len jeden parameter – dĺžku strany *a*.

- Nová trieda `Calculator` bude vyzerať takto:

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        float abs(ComplexNumber x);
        double area(double a, double b) { return a * b; }
        double area(double a) { return a * a; }
};
```

Použitie štandardných argumentov

- V hlavnom programe použijeme nové funkcie:

```
double a = 5.2, b = 3.4;
std::cout << "Obsah obdlznika so stranami " << a << ", " << b << " je: " << K.area(a, b) << std::endl;
std::cout << "Obsah stvorca so stranou " << a << " je: " << K.area(a) << std::endl;
```

- Výstup:

  Obsah obdlznika so stranami 5.2, 3.4 je: 17.68
  Obsah stvorca so stranou 5.2 je: 27.04

Použitie štandardných argumentov

- Existuje aj iný spôsob, ako vyriešiť uvedený problém. Namiesto dvoch funkcií definujeme len jednu, so **štandardnou hodnotou** jedného argumentu. Bude ním argument *b*, ak jeho hodnota bude nulová, znamená to, že ide o štvorec.

- Trieda `Calculator` bude definovaná takto:

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        float abs(ComplexNumber x);
        double area(double a, double b = 0) {double c = (b == 0) ? a * a : a * b; return c; }
};
```

- Hlavný program aj výstup bude rovnaký ako v predchádzajúcom prípade. Ak zavoláme funkciu len s jedným argumentom, *b* bude mať automaticky hodnotu 0 a funkcia spozná, že má vypočítať obsah štvorca.

Použitie štandardných argumentov

- Vo funkcii:

```
double area(double a, double b = 0) {
        double c = (b == 0) ? a * a : a * b;
        return c;
}
```

- sa podčiarknutá časť volá podmienený operátor (conditional/ternary operator)
- Tvar:

  podmienka ? ak je splnená : ak nie je splnená

- Ak je b == 0 do premennej double c sa priradí výsledok výpočtu a*a ak b != 0 do premennej double c sa priradí výsledok výpočtu a*b
- Funguje to aj opačne, napr. `(a > b ? a : b) = c;` priradí buď do premennej a alebo b, podľa toho, ktorá je väčšia hodnotu z premennej c

Použitie štandardných argumentov

- Štandardnú hodnotu môžu mať všetky argumenty. Upravme teraz funkciu tak, aby vrátila obsah jednotkového štvorca, ak je volaná bez argumentov.

- Nová definícia triedy `Calculator` teda bude:

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        float abs(ComplexNumber x);
        double area(double a = 0, double b = 0) {
                double c = (a == 0 && b == 0) ? 1 : ( (b == 0) ? a * a : a * b );
                return c;
        }
};
```

Použitie štandardných argumentov

- V hlavnom programe potom použijeme funkciu *obsah* všetkými tromi spôsobmi:

```
float a = 5.2, b = 3.4;
std::cout << "Obsah obdlznika so stranami " << a << ", " << b << " je: " << K.area(a, b) << std::endl;
std::cout << "Obsah stvorca so stranou " << a << " je: " << K.area(a) << std::endl;
std::cout << "Obsah stvorca so stranou 1 je: " << K.area() << std::endl;
```

- Výstup:

Obsah obdlznika so stranami 5.2, 3.4 je: 17.68

Obsah stvorca so stranou 5.2 je: 27.04

Obsah stvorca so stranou 1 je: 1

Použitie štandardných argumentov

- Príklad: Trieda `Calculator`, v ktorej funkcia *obsah* nie je inline

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        float abs(ComplexNumber x);
        float area(float a = 0, float b = 0);
};

float Calculator::area(float a = 0, float b = 0) {
        float c = (a == 0 && b == 0) ? 1 : ((b == 0) ? a * a : a * b);
        return c;
}
```

- Pri kompilácii takéhoto kódu kompilátor **vyhlási chybu**.
- Problém je v tom, že štandardné hodnoty argumentov nemôžu byť dané v deklarácii aj definícii funkcie. Musia byť dané len raz, nezáleží na tom, či v deklarácii alebo definícii.

Preťažovanie a nejednoznačnosti

- Príklad: Absolútna hodnota čísla typu *double*

  Do triedy `Calculator` doplníme ďalší variant funkcie *abs*, ktorý bude počítať absolútnu hodnotu čísel typu *double*.

```
class Calculator {
public:
        int abs(int x);
        float abs(float x);
        double abs(double x);
};
```

- Trochu pozmeníme aj definície metód:

```
int Calculator::abs(int x) {
        std::cout << "Pocitam absolutnu hodnotu cisla typu int" << std::endl;
        if (x > 0) return x;
        else return -x;
}
```

Preťažovanie a nejednoznačnosti

```
float Calculator::abs(float x) {
        std::cout << "Pocitam absolutnu hodnotu cisla typu float" << std::endl;
        if (x > 0) return x;
        else return -x;
}

double Calculator::abs(double x){
        std::cout << "Pocitam absolutnu hodnotu cisla typu double" << std::endl;
        if (x > 0) return x;
        else return -x;
}
```

- V hlavnom programe potom zavoláme funkcie takto:

```
std::cout << "Absolutna hodnota cisla -10 je " << K.abs(-10) << std::endl;
std::cout << "Absolutna hodnota cisla -10.0 je " << K.abs(-10.0) << std::endl;
```

Preťažovanie a nejednoznačnosti

- Ktorá verzia funkcie *abs* bude použitá pre čísla -10 a -10.0?

```
Pocitam absolutnu hodnotu cisla typu int
Absolutna hodnota cisla -10 je 10
Pocitam absolutnu hodnotu cisla typu double
Absolutna hodnota cisla -10.0 je 10
```

- Vidíme, že v prípade čísla 10 sa program rozhodol pre verziu pre typ int, v prípade čísla 10.0 použil verziu pre typ double. Máme však v programe **nejednoznačnosť**, a v závislosti od okolností a od kompilátora sa program ani nemusí podariť skompilovať. Mali by sme sa preto snažiť nejednoznačnosti z programu odstrániť.

Preťažovanie a nejednoznačnosti

- Príklad: Podávanie parametra hodnotou a odkazom

  Majme v triede `Calculator` ďalšiu verziu funkcie *abs*, ktorá nielen vráti absolútnu hodnotu čísla, ale aj ňou prepíše pôvodnú hodnotu.

```
class Calculator {
public:
        int abs(int x);
        int abs(int& x);
        ...
}
int Calculator::abs(int& x){
        std::cout << "Pocitam absolutnu hodnotu cisla typu int" << std::endl;
        if (x > 0) return x;
        else return -x;
}
```

Preťažovanie a nejednoznačnosti

- V hlavnom programe skúsime potom zavolať funkciu *abs*:

```
int m = -10;
std::cout << "Absolutna hodnota cisla je " << K.abs(m) << std::endl;
```

- V tomto prípade program nemá nijakú možnosť rozhodnúť, ktorú verziu funkcie *abs* mal programátor na mysli. Výsledkom teda bude závažná **nejednoznačnosť** a program sa **nepodarí skompilovať**.
- Kompilátor (VS2022) vypíše chybu:

```
error C2668: 'Calculator::abs': ambiguous call to overloaded function
message : could be 'int Calculator::abs(int &)'
message : or       'int Calculator::abs(int)'
message : while trying to match the argument list '(int)'
```

Preťažovanie a nejednoznačnosti

- Príklad: Metódy so štandardnými argumentami

  V triede `Calculator` majme tieto funkcie:

```
class Calculator {
public:
        double area(double a) { return a * a; }
        double area(double a = 0, double b = 0)       ;
};
```

- V hlavnom programe sa pokúsime funkciu *area* zavolať:

```
double k = 5.2;
std::cout << "Obsah stvorca so stranou " << k << " je: " << K.area(k) << std::endl;
```

- Ani v tomto prípade kompilátor nedokáže rozlíšiť, o ktorú verziu funkcie ide, a preto vyhlási chybu:

```
'Calculator::area': ambiguous call to overloaded function
message : could be 'double Calculator::area(double,double)‘
message : or       'double Calculator::area(double)‘
message : while trying to match the argument list '(double)'
```

Preťažovanie konštruktora

- Príklad: Sčítanie komplexných čísel

  Do triedy `ComplexNumber`, ktorú sme si už vytvorili, doplníme funkciu, ktorá k danému číslu pripočíta iné komplexné číslo. Budeme potrebovať aj metódy na zmenu hodnoty komplexného čísla a jeho výpis.

```
class ComplexNumber {
private:
        float a, b;
public:
        ComplexNumber(float real, float imag) { a = real; b = imag; }
        ComplexNumber Plus(ComplexNumber& x);
        float A() { return a; }
        float B() { return b; }
        void setA(float newA) { a = newA; }
        void setB(float newB) { b = newB; }
        void print() { std::cout << a << "+" << b << "i"; }
};
```

Preťažovanie konštruktora

- Definujme funkciu *Plus*:

```
ComplexNumber ComplexNumber::Plus(ComplexNumber& x) {
        ComplexNumber z(0, 0);
        z.setA(x.a + a);
        z.setB(x.b + b);
        return z;
}
```

- V hlavnom programe ju potom použijeme:

```
        ComplexNumber x(1, 2), y(3, -1), z(0, 0);
        z = x.Plus(y);
        std::cout << "Sucet cisel ";
        x.print();
        std::cout << " a ";
        y.print();
        std::cout << " je ";
        z.print();
```

- `Vystup : Sucet cisel 1+2i a 3+-1i je 4+1i`

Preťažovanie konštruktora

- Príklad: Súčet komplexných čísel s preťažením konštruktora

  V predchádzajúcom príklade sme videli, že sme na viacerých miestach potrebovali pomocnú premennú typu `ComplexNumber`. Do nej sa uložili výsledky sčítania. V našom prípade sme reálnu aj imaginárnu časť čísla nastavili na 0, ale v takomto prípade nie je potrebné, aby tieto premenné mali inicializovanú hodnotu. Je výhodné mať k dispozícii aj prázdny konštruktor:

```
class ComplexNumber {
private:
        float a, b;
public:
        ComplexNumber(float real, float imag) { a = real; b = imag; }
        ComplexNumber() {}
        ...
```

- Potom môžeme objekt vytvoriť aj bez inicializácie hodnôt:

```
ComplexNumber z;
```

Preťažovanie konštruktora

- Ďalším prípadom, keď je výhodné mať aj konštruktor bez parametrov, je vytváranie poľa objektov. Nemusíme tak inicializovať prvky poľa.
- Príklad: Počítanie násobkov komplexného čísla

```
int main(){
	ComplexNumber x(1, 2), z[4];
	for (int i = 0; i < 4; i++) {
		z[i].setA(x.A() * i);
		z[i].setB(x.B() * i);
		z[i].print();
		std::cout << std::endl;
	}
	return 0;
}
```

- Vystup:

```
0+0i
1+2i
2+4i
3+6i
```

Preťažovanie konštruktora

- Ak chceme dynamicky alokovať pole objektov, je nutné mať k dispozícii aj konštruktor bez parametrov.
- Príklad: Počítanie vopred neznámeho počtu násobkov komplexného čísla

```
ComplexNumber x(1, 2), *z;
int n;
std::cout << "Kolko nasobkov sa ma vypocitat? ";
std::cin >> n;
z = new ComplexNumber[n];
for (int i = 0; i < n; i++){
        z[i].setA(x.A() * i);
        z[i].setB(x.B() * i);
        z[i].print();
        std::cout << std::endl;
}
```

- Kolko nasobkov sa ma vypocitat? 3

  0+0i

  1+2i

  2+4i

Preťažovanie konštruktora

- Samozrejme, trieda môže mať neobmedzený počet konštruktorov s rôznym počtom a typom parametrov. Kvôli prehľadnosti nie je však vhodné veľké množstvo rôznych konštruktorov.
- Príklad: Program, ktorý vypočíta, koľko dní ostáva do najbližšej výplaty.

  Vytvoríme triedu Date, pričom dátum sa bude dať inicializovať číselnými hodnotami *day*, *month*, *year* alebo znakovým reťazcom.

```
class Date {
private:
        int day, month, year;
public:
        Date(std::string s, std::string delimiter);
        Date(int d, int m, int y) { day = d; month = m; year = y; }
        void print() { std::cout << day << ". " << month << ". " << year << std::endl; }
        int getDay() { return day; }
        int getMonth() { return month; }
        int getYear() { return year; }
};
```

Preťažovanie konštruktora

- Definujeme si konštruktor, ktorý rozdelí string na den, mesiac a rok. Vstup je aj delimiter, teda oddeľovač (znak medzi hodnotami dátumu):

```
Date::Date(std::string s, std::string delimiter) {
        size_t pos = 0;
        std::string token;
        int date[3] = {0,0,0};
        int counter = 0;
        while ((pos = s.find(delimiter)) != std::string::npos) {
                token = s.substr(0, pos);
                date[counter] = std::stoi(token);
                s.erase(0, pos + delimiter.length());
                counter++;
        }
        day = date[0];
        month = date[1];
        year = std::stoi(s);
}
```

Preťažovanie konštruktora

- Hlavný program:

```
int daysInMonth[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
int d, m, y;
std::string sdat;
std::cout << "Aky je dnesny datum? ";
std::getline(std::cin, sdat);

Date dat(sdat, ".");
std::cout << "Ktory den v mesiaci chodi vyplata? ";
std::cin >> d;

y = dat.getYear();
if (dat.getDay() <= d){
            m = dat.getMonth(); //vyplata tento mesiac
}else{
            if (dat.getMonth() == 12){
                        m = 1;
                        y++; //vyplata az v dalsom roku
            }else{
                        m = dat.getMonth() + 1; //vyplata buduci mesiac
            }
}
...
```

Preťažovanie konštruktora

- Hlavný program:

```
...
Date payout(d, m, y);
std::cout << "Najblizsia vyplata bude dna ";
payout.print();
if (payout.getMonth() == 1 && dat.getMonth() == 12) {
          d = d + 31;
}else{
          d = d + daysInMonth[dat.getMonth() - 1] * (payout.getMonth() - dat.getMonth());
}
std::cout << "Do vyplaty zostava " << d - dat.getDay() << " dni" << std::endl;
```

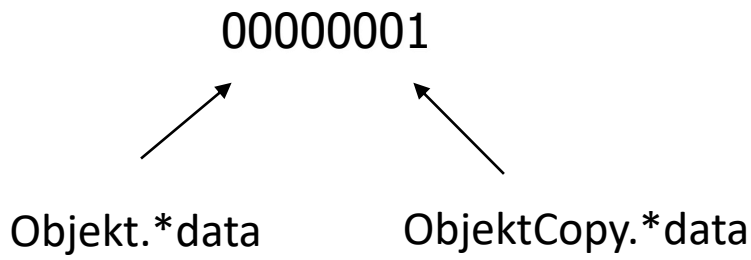
- Výstup:

```
          Aky je dnesny datum? 26.10.2022
          Ktory den v mesiaci chodi vyplata? 10
          Najblizsia vyplata bude dna 10. 11. 2022
          Do vyplaty zostava 15 dni
```

Kopírovací konštruktor

- V predchádzajúcich prednáškach sme sa stretli so situáciami, keď vytvorenie presnej kópie objektu spôsobilo problémy v programe, alebo dokonca jeho zrútenie. **Ukázalo sa, že nežiadúce je najmä vytváranie presnej kópie v prípade, ak objekt dynamicky alokuje pamäť.** Ak sa totiž skopíruje aj smerník, dva (alebo viac) objektov obsahuje smerníky ukazujúce na to isté miesto. To vedie k viacnásobným dealokáciám, memory leakom a podobne. Problémy nastali najmä v troch prípadoch:

  1. Pri priraďovaní objektov
  2. Ak objekt vystupoval ako parameter funkcie
  3. Ak bol objekt návratovou hodnotou funkcie



Kopírovací konštruktor

- V C++ existuje špeciálny typ konštruktora, ktorý sa zavolá práve v prípadoch, keď sa má vytvoriť bitová kópia objektu. Tento konštruktor zabezpečí správne kopírovanie objektov. Nazýva sa **kopírovací konštruktor** a volá sa v nasledujúcich prípadoch:

  1. Inicializácia priradením: `meno_triedy objekt1=objekt2`
  2. Objekt ako parameter funkcie: `f(objekt)`
  3. Objekt ako návratová hodnota funkcie: `objekt=f()`

- **Kopírovací konštruktor sa nevolá pri priradení objekt1=objekt2 (V tomto prípade nejde o vytvorenie nového objektu)**

Kopírovací konštruktor

- Príklad: Priraďovanie znakových reťazcov pomocou kopírovacieho konštruktora

  Na jednej z predchádzajúcich prednášok sme vytvorili triedu `myString`, ktorej súčasťou bolo dynamické alokovanie pamäte. Pri priraďovaní objektov tejto triedy vznikali problémy. Upravme teraz túto triedu tak, aby obsahovala kopírovací konštruktor.

```
class myString {
...
public:
        myString(const myString& str); //kopirovaci konstrutor
        ...
};
```

Kopírovací konštruktor

- Definujeme kopírovací konštruktor:

```
myString::myString(const myString& str) {
        lenght = str.lenght; //mozem pristupit k private premennej str.length
        s = new char[lenght];
        for (int i = 0; i < lenght; i++) {
                s[i] = str.s[i];
        }
}
```

- Hlavný program:

```
myString s1(35, "Toto je test novej triedy myString");
myString s2 = s1;  //Tu sa zavolá kopírovací konštruktor
std::cout.write(s1.getString(), s1.getLength());
std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());
std::cout << std::endl;
s1.changeString(20, "Podarilo sa nam to?");
std::cout.write(s1.getString(), s1.getLength());
std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());
```

Kopírovací konštruktor

- Kým bez kopírovacieho konštruktora program spadol, s kopírovacím konštruktorom dostaneme správny výstup:

```
Toto je test novej triedy myString
Toto je test novej triedy myString
Podarilo sa nam to?
Toto je test novej triedy myString
```

Kopírovací konštruktor

- Pomocou kopírovacieho konštruktora môžeme poslať objekt do funkcie ako parameter. Aj keď vytváranie kópie je samo o sebe nevýhodné vzhľadom na pamäť, dá sa urobiť technicky korektne.
- Príklad: Počítanie obvodu n-uholníka

  Už máme vytvorenú triedu `Polygon`, obohatíme ju o kopírovací konštruktor:

```
class Polygon {
        ...
public:
        ...
        Polygon(const Polygon& U); //Kopirovaci konstruktor
        ...
};
```

Kopírovací konštruktor

- Kopírovací konštruktor bude definovaný takto:

```
Polygon::Polygon(const Polygon& U) {
        std::cout << "kopirovaci konstruktor" << std::endl;
        n = U.n;
        setPoints(U.B);
}
void Polygon::setPoints(Point* P){
        B = new Point[n];
        for (int i = 0; i < n; i++) {
                B[i] = P[i];
        }
}
```

- V triede `Calculator` bude len jediná funkcia `perimeter`, ktorej parametrom bude objekt triedy `Polygon`:

```
class Calculator {
public:
        double perimeter(Polygon U);
};
```

Kopírovací konštruktor

- Hlavný program:

```
Point B[4];
B[0].setPoint(0, 0);
B[1].setPoint(1, 0);
B[2].setPoint(1, 1);
B[3].setPoint(0, 1);

Polygon square(4, B);
Calculator K;

square.printPoints("Stvorec");
std::cout << "Obvod stvorca je: " << K.perimeter(square) << std::endl;
```

Kopírovací konštruktor

- Keď sme nemali k dispozícii kopírovací konštruktor, program spadol. Pomocou kopírovacieho konštruktora môžeme pádu programu predísť:

```
n-uholnik Stvorec
0. vrchol: 0  0
1. vrchol: 1  0
2. vrchol: 1  1
3. vrchol: 0  1
kopirovaci konstruktor
Dealokujem pole...
Obvod stvorca je: 4
Dealokujem pole...
```

Kopírovací konštruktor

- Nakoniec si ukážeme aj to, ako sa dá kopírovací konštruktor využiť, ak je objekt návratovou hodnotou funkcie:
- Príklad: Posúvanie n-uholníka

  Do triedy `Calculator` doplníme funkciu, ktorá bude posúvať daný n-uholník o danú dĺžku v smere x a y. Návratovou hodnotou tejto funkcie bude posunutý n-uholník.

```
class Calculator {
public:
        double perimeter(Polygon U);
        Polygon displacement(Polygon& U, int dx, int dy);
};
Polygon Calculator::displacement(Polygon& U, int dx, int dy) {
        Point* B = new Point[U.N()];
        for (int i = 0; i < U.N(); i++) {
                B[i].setPoint(U.BX(i) + dx, U.BY(i) + dy);
        }
        Polygon U1(U.N(), B); //dynamicky alokované pole pre body
        delete[] B;
        return U1;
}
```

Kopírovací konštruktor

- Hlavný program:

```
Point B[4];
B[0].setPoint(0, 0);
B[1].setPoint(1, 0);
B[2].setPoint(1, 1);
B[3].setPoint(0, 1);

Polygon square(4, B);
Calculator K;

square.printPoints("Stvorec");
std::cout << "Obvod stvorca je: " << K.perimeter(square) << std::endl;

Polygon newSquare = K.displacement(square, 1, 1);
newSquare.printPoints("Posunuty stvorec");
```

Kopírovací konštruktor

- Opäť platí, že bez kopírovacieho konštruktora by program zlyhal. S kopírovacím konštruktorom máme problém vyriešený:

```
n-uholnik Posunuty stvorec
0. vrchol: 1  1
1. vrchol: 2  1
2. vrchol: 2  2
3. vrchol: 1  2
Dealokujem pole...
Dealokujem pole...
```

Preťažovanie operátorov

- Tak ako môžeme preťažovať funkcie, môžeme preťažiť aj operátory (+,-,=,++,[]...), ktoré sú vlastne istým typom funkcií. Takto môžeme napr. definovať operáciu + pre zlomky, komplexné čísla atď.
- Syntax preťaženia operátora:

  deklarácia:

  ```
  typ operator#(parametre);
  ```

  definícia:

  ```
  typ meno_triedy::operator#(parametre)
  {
        ...
  }
  ```

- Pri preťažovaní operátorov platia základné obmedzenia:

  1. Nemôžeme zmeniť počet operandov operátora

  2. Nedá sa zmeniť priorita operátora

Preťažovanie binárnych operátorov

- Sem patria operátory +, -, *, /, =
- Príklad: Počítanie s komplexnými číslami

  V triede `ComplexNumber` vytvoríme operátory + a * tak, aby sme mohli sčítať a vynásobiť dve komplexné čísla a tiež vynásobiť komplexné číslo reálnym.

```
class ComplexNumber {
        ...
public:
        ComplexNumber operator+(const ComplexNumber& x);
        ComplexNumber operator*(const ComplexNumber& x);
        ComplexNumber operator*(float t);
        ...
};
```

Preťažovanie binárnych operátorov

- Definujeme jednotlivé operátory:

```
ComplexNumber ComplexNumber::operator+(const ComplexNumber& x){
        ComplexNumber z;
        z.setA(a + x.a);
        z.setB(b + x.b);
        return z;
}

ComplexNumber ComplexNumber::operator*(const ComplexNumber& x){
        ComplexNumber z;
        z.setA(a * x.a - b * x.b);
        z.setB(a * x.b + b * x.a);
        return z;
}

ComplexNumber ComplexNumber::operator*(float t){
        ComplexNumber z;
        z.setA(a * t);
        z.setB(b * t);
        return z;
}
```

Preťažovanie binárnych operátorov

```
ComplexNumber x(1, 2), y(3, -1), z1, z2, z3;
z1 = x + y;
z2 = x * y;
z3 = x * 3;
std::cout << "Komplexne cisla: ";
x.print();
std::cout << ", ";
y.print();
std::cout << std::endl << "x+y: ";
z1.print();
std::cout << std::endl << "x*y: ";
z2.print();
std::cout << std::endl << "x*3: ";
z3.print();
```

- Výstup:

```
Komplexne cisla: 1+2i, 3+-1i
x+y: 4+1i
x*y: 5+5i
x*3: 3+6i
```

Preťažovanie relačných operátorov

- Ide o operátory ==, <, >, <=, >= ...
- Príklad: Porovnávanie veku dvoch osôb

  Budeme pracovať s triedou `Date`, ktorú sme už vytvorili. Definujeme pre ňu operátory <, >, ==.

```
class Date {
        ...
public:
        ...
        bool operator >(const Date& dat);
        bool operator <(const Date& dat);
        bool operator ==(const Date& dat);
        ...
};
```

Preťažovanie relačných operátorov

- Definujeme jednotlivé operátory:

```
bool Date::operator<(const Date& dat){
        if (year < dat.year) {
                return true;
        } else if (year > dat.year) {
                return false;
        } else {
                if (month < dat.month) {
                        return true;
                } else if (month > dat.month) {
                        return false;
                } else {
                        if (day < dat.day) { return true; }
                        else { return false; }
                }
        }
}
```

Preťažovanie relačných operátorov

```
bool Date::operator>(const Date& dat){
        if (year > dat.year) {
                return true;
        } else if (year < dat.year) {
                return false;
        } else {
                if (month > dat.month) {
                        return true;
                } else if (month < dat.month) {
                        return false;
                } else {
                        if (day > dat.day) { return true; }
                        else { return false; }
                }
        }
}
```

Preťažovanie relačných operátorov

```
bool Date::operator==(const Date& dat) {
        if (year == dat.year && month == dat.month && day == dat.day) {
                return true;
        } else {
                return false;
        }
}
```

Preťažovanie relačných operátorov

```
std::string meno1, meno2, sdat1, sdat2;
std::cout << "Meno a datum narodenia prvej osoby: " << std::endl;
std::getline(std::cin, meno1);
std::getline(std::cin, sdat1);
std::cout << "Meno a datum narodenia druhej osoby: " << std::endl;
std::getline(std::cin, meno2);
std::getline(std::cin, sdat2);
Date dat1(sdat1, "."), dat2(sdat2, ".");
if (dat1 < dat2) {
        std::cout << meno1 << " je starsi ako " << meno2 << std::endl;
} else if (dat1 > dat2) {
        std::cout << meno2 << " je starsi ako " << meno1 << std::endl;
} else {
        std::cout << "Tito dvaja su rovnako stari" << std::endl;
}
```

Preťažovanie relačných operátorov

- Výstup:

```
Meno a datum narodenia prvej osoby:
Maria
13.9.1988
Meno a datum narodenia druhej osoby:
Julia
15.5.1991
Maria je starsi ako Julia
```

Preťažovanie unárnych operátorov

- Sem patria napr. operátory ++, --
- Príklad: Určenie nasledujúceho dňa

  Pre triedu `Date` definujeme operátor ++ (nasledujúci deň)

```
class Date {
public:
        ...
        Date& operator ++();
        Date operator ++(int);
        ...
};
```

Preťažovanie unárnych operátorov

- Definujeme operátor ++Date:

```
Date& Date::operator++() {
        int daysInMonth[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
        if (day == 31 && month == 12){
                year++;
                month = 1;
                day = 1;
        } else if (day == daysInMonth[month - 1]) {
                month++;
                day = 1;
        } else {
                day++;
        }
        return *this;
}
```

Preťažovanie unárnych operátorov

- Definujeme operátor `Date++`:

```
Date Date::operator++(int) {
        int d=day, m=month, y=year;
        int daysInMonth[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
        if (day == 31 && month == 12) {
                year++;
                month = 1;
                day = 1;
        } else if (day == daysInMonth[month - 1]) {
                month++;
                day = 1;
        } else {
                day++;
        }
        return Date(d,m,y);
}
```

Preťažovanie unárnych operátorov

- Hlavný program:

```
std::string sdat;
std::cout << "Aky je dnesny datum? ";
std::getline(std::cin, sdat);
Date dat(sdat, ".");

Date tommorow = ++dat; //++dat zvysi datum o jeden den a vrati uz zvyseny datum.
std::cout << "Zajtrajsi datum je: ";
dat.print();
Date stillTommorow = dat++; //dat++ zvysi datum o jeden den, ale vrati originalny datum pred zvysenim.
std::cout << "Pozajtrajsi datum je: ";
dat.print();
```

- Výstup:

```
Aky je dnesny datum? 21.10.2020
Zajtrajsi datum je: 22. 10. 2020
Pozajtrajsi datum je: 23. 10. 2020
```

Preťažovanie operátora priradenia

- Už sme si hovorili, aké problémy môžu nastať pri priraďovaní objektov, ak súčasťou objektu je dynamicky alokovaná pamäť. Teraz si ukážeme, ako môžeme priraďovací operátor vhodne preťažiť, aby sme sa problémom vyhli.
- Príklad: Priraďovanie objektov triedy `myString:`

```
class myString {
        ...
public:
        myString& operator=(const myString& str);
        ...
};
```

Preťažovanie operátora priradenia

- Teraz definujeme operátor priradenia. Tento operátor má ako parameter odkaz na objekt triedy `myString` a návratová hodnota je takisto odkaz.

```
myString& myString::operator=(const myString& str) {
        if (lenght < str.lenght) {  //ak je nový retazec dlhší, realokujeme pamäť
                delete[] s;
                s = new char[str.lenght];
        }
        lenght = str.lenght;
        for (int i = 0; i < lenght; i++) {
                s[i] = str.s[i];
        }
        return *this;
}
```

Preťažovanie operátora priradenia

- Hlavný program:

```
myString s1(35, "Toto je test novej triedy myString");
myString s2(0, "");
std::cout.write(s1.getString(), s1.getLength());
std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());
std::cout << std::endl;
s2 = s1; //na tomto mieste sa nevola kopirovaci konstruktor! treba pretazit operator =
std::cout.write(s1.getString(), s1.getLength());
std::cout << std::endl;
std::cout.write(s2.getString(), s2.getLength());
std::cout << std::endl;
```

- Program s klasickým operátorom priradenia by padol. Naopak program s preťaženým operátorom funguje správne

Preťažovanie indexového operátora

- Príklad: Indexový operátor [ ] pre triedu *nUholník*

  Do triedy `Polygon` doplníme indexový operátor ktorý bude vracať daný vrchol n-uholníka (objekt typu `Point`)

```
class Polygon {
        ...
public:
        ...
        Point& operator [](int index) { return B[index]; } ..
};
```

Preťažovanie indexového operátora

- Pomocou nového operátora potom môžeme prepísať napr. funkciu `perimeter` triedy `Calculator` :

```
/*double Calculator::perimeter(Polygon U) { //Povodna funkcia
        double o = 0.;
        for (int i = 0; i < U.N() - 1; i++) {
                o += sqrt(pow(U.BX(i + 1) - U.BX(i), 2) + pow(U.BY(i + 1) - U.BY(i), 2));
        }
        o += sqrt(pow(U.BX(U.N() - 1) - U.BX(0), 2) + pow(U.BY(U.N() - 1) - U.BY(0), 2));
        return o;
}*/

double Calculator::perimeter(Polygon U) {
        double o = 0.;
        for (int i = 0; i < U.N() - 1; i++) {
                o += sqrt(pow(U[i+1].X() - U[i].X(), 2) + pow(U[i + 1].Y() - U[i].Y(), 2));
        }

        o += sqrt(pow(U[U.N() -1].X() - U[0].X(), 2) + pow(U[U.N() - 1].Y() - U[0].Y(), 2));
        return o;
}
```

Preťažovanie vstupno-výstupných operátorov

- Príklad: Výpis súradníc bodov n-uholnika do konzoly pomocou operátoru << :
- Do triedy `Polygon` doplníme preťažený operátor.

```
class Polygon {
        ...
public:
        friend std::ostream& operator <<(std::ostream& output, const Polygon& P);
        ...
};

std::ostream& operator<<(std::ostream& output, const Polygon& P){
        for (int i = 0; i < P.n; i++) {
                output << i << ". vrchol: " << P.B[i].X() << "  " << P.B[i].Y() << std::endl;
        }
        return output;
}
```

Preťažovanie vstupno-výstupných operátorov

- Hlavný program:

```
Point B[4];
B[0].setPoint(0, 0);
B[1].setPoint(1, 0);
B[2].setPoint(1, 1);
B[3].setPoint(0, 1);

Polygon square(4, B);
std::cout << square;
```

- Výstup:

```
0. vrchol: 0  0
1. vrchol: 1  0
2. vrchol: 1  1
3. vrchol: 0  1
```

Preťažovanie vstupno-výstupných operátorov

- Príklad: Priradenie dátumu zo vstupu z konzoly pomocou vstupného operátoru >> :
- Do triedy `Date` doplníme preťažený operátor.

```
class Date {
        ...
public:
        friend std::istream& operator>>(std::istream& input, Date& D);
        ...
};

std::istream& operator>>(std::istream& input, Date& D) {
        input >> D.day >> D.month >> D.year;
        return input;
}
```

Preťažovanie vstupno-výstupných operátorov

- Hlavný program:

```
Date d;
std::cout << "Zadaj postupne den, mesiac a rok:" << std::endl;
std::cin >> d;
d.print();
```

- Výstup:

```
Zadaj postupne den, mesiac a rok:
21
10
2020
21. 10. 2020
```

Preťažovanie operátorov

- Operátory, ktoré môžu byť preťažené:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Operátory, ktoré **nemôžu** byť preťažené:

::	.*	.	?:

Dedičnosť

- V objektovo orientovanom programovaní môže mať každá trieda svojho potomka - odvodenú triedu, ktorá od nej preberie všetky dáta a metódy (okrem konštruktora a deštruktora).
- Odvodená trieda môže mať okrem zdedených dát a metód svoje vlastné, ktorými sa odlíši od svojho rodiča.
- Metódy, ktoré odvodená trieda zdedila, môže mať upravené podľa svojich potrieb.
- Programátor má možnosť ovplyvňovať prístup odvodenej triedy k dátam a metódam svojho rodiča. Teda aj keď odvodená trieda niečo zdedí, nemusí jej byť dovolené to používať.
- Každá trieda môže mať ľubovoľne veľa potomkov aj rodičov.

Chránené dáta a metódy

- Doteraz sme sa stretli len s dvomi typmi dát a metód: verejnými (**public**) a súkromnými (**private**). Vďaka možnosti dedenia ale vzniká nová požiadavka – niektoré dáta a metódy by mali byť síce súkromné, ale odvodené triedy by k nim mali mať prístup. Takéto dáta a metódy sa nazývajú **chránené (protected)**.
- Ak sa teda v programe bude vyskytovať aj dedičnosť, vo všeobecnosti bude štruktúra triedy vyzerať takto:

```
class meno_triedy {
private:
        ...
protected:
        ...
public:
        ...
};
```

Typy dedičnosti

- Tak ako dáta a metódy, aj dedičnosť môže byť troch rôznych druhov:

  1. Súkromná (private)

  2. Chránená (protected)

  3. Verejná (public)

- **Súkromná dedičnosť** znamená, že všetky dáta a metódy zdedené od rodičovskej triedy budú v odvodenej triede súkromné.
- **Chránená dedičnosť** znamená, že verejné a chránené dáta a metódy rodičovskej triedy budú v odvodenej triede chránené. Súkromné dáta a metódy zostávajú súkromnými.
- **Verejná dedičnosť** znamená, že prístupnosť dát a metód sa presne zachováva – verejné budú verejnými, chránené chránenými a súkromné súkromnými.

- **Štandardná dedičnosť v C++ je súkromná**. Ak si programátor praje iný typ, musí to explicitne zadať.

Typy dedičnosti

- Prehľad typov dedičnosti v C++:

DEDIČNOSŤ	SÚKROMNÉ ÚDAJE	CHRÁNENÉ ÚDAJE	VEREJNÉ ÚDAJE
SÚKROMNÁ	SÚKROMNÉ	SÚKROMNÉ	SÚKROMNÉ
CHRÁNENÁ	SÚKROMNÉ	CHRÁNENÉ	CHRÁNENÉ
VEREJNÁ	SÚKROMNÉ	CHRÁNENÉ	VEREJNÉ

Dedičnosť

- Vytvorenie odvodenej triedy:

```
class odvodená_trieda :typ_dedičnosti rodičovská_trieda
{
        ...
};
```

- Príklad: Prístupnosť dát podľa rôzneho typu dedičnosti
- Vytvoríme dve triedy – `BankClient` a `CompanyClient`, pričom `CompanyClient` bude potomkom triedy `BankClient`.
- `BankClient` je trieda na uloženie informácii o klientovi banky a bude mať nasledujúce údaje: meno, adresy, číslo účtu a stav účtu.
- `CompanyClient` je trieda na uloženie konkrétnejšieho typu klienta (v tomto prípade je klientom firma) a bude mať všetky uvedené údaje a navyše ešte IČO a Identifikačné číslo DPH.

- Trieda BankClient :

```
class BankClient {
private:
        int accountBalance = 0;
        std::string accountNumber = "";
protected: //dáta by mali byť síce súkromné aby spĺňala trieda data hiding, ale odvodené triedy by k nim mali mať prístup.
        std::string name = "", address = "";
public:
        BankClient() {}
        BankClient(std::string m, std::string a, std::string cu, int su) : name(m), address(a), accountNumber(cu), accountBalance(su)
        std::string Name() { return name; }
        std::string Address() { return address; }
        std::string AccountNumber() { return accountNumber; }
        int AccountBalance() { return accountBalance; }
};
```

- Trieda CompanyClient:

```
class CompanyClient :private BankClient { //všetko, čo CompanyClient zdedí od triedy BankClient, bude súkromné

private:

        std::string ico, vatNumber;

public:

        CompanyClient(std::string m, std::string a, std::string i, std::string vn);

        std::string Ico() { return ico; }

        std::string VatNumber() { return vatNumber; }

        void printInfo();

};
```

- V tomto prípade je dedičnosť typu private, to znamená, že všetko, čo CompanyClient zdedí od triedy BankClient, bude súkromné. To znamená, že mimo triedy (napr. v hlavnom programe), nemá CompanyClient prístupné žiadne dáta a nemôže použiť ani metódy Name(), Address(), AccountNumber() a AccountBalance().
- Pre svoje súkromné účely však môže všetky tieto metódy použiť. Pridáme teda do triedy CompanyClient metódu printInfo(), pomocou ktorej CompanyClient oznámi svoje meno a adresu.
- Premenné accountBalance a accountNumber nie su pre triedu CompanyClient vôbec prístupné, keďže sú to súkromné údaje triedy BankClient. CompanyClient teda nemôže s týmito premennými priamo nijako pracovať.

- Definujeme metódy triedy `CompanyClient`:

```
CompanyClient::CompanyClient(std::string m, std::string a, std::string i, std::string vn) {
        //k tymto premmennym triedy BankClient mam pristup vo funkciach triedy CompanyClient
        name = m;
        address = a;
        //premenne triedy CompanyClient
        ico = i;
        vatNumber = vn;
        //nemám prístup k premennej accountBalance a accountNumber triedy BankClient, lebo su private
}

void CompanyClient::printInfo() {
        std::cout << "Klient: " << name << " "<< std::endl;
        std::cout << "Adresa: " << address << std::endl;
}
```

- Všimnime si, že `accountBalance a accountNumber` nie sú v konštruktore vôbec inicializované.

- Hlavný program:

```
BankClient client("Pavol Horvath", "Bratislava", "012011/0100", 15000);
std::cout << "Klient: " << client.Name() << std::endl;
std::cout << "Adresa: " << client.Address() << std::endl;
std::cout << "Cislo uctu: "  << client.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << client.AccountBalance() << " Eur" << std::endl;

CompanyClient companyCient("Eset", "Bratislava", "123456", "01/123456");
//V hlavnom programe nemozem pouzit napr funckie companyCient.Name(), companyCient.Surname() ... lebo su uz v triede CompanyClient sukromne
companyCient.printInfo();
std::cout << "ICO: " << companyCient.Ico() << std::endl;
std::cout << "Vat number: " << companyCient.VatNumber() << std::endl;
```

- Výstup:

```
Klient: Pavol Horvath
Adresa: Bratislava
Cislo uctu: 012011/0100
Stav uctu: 15000 Eur

Klient: Eset
Adresa: Bratislava
ICO: 123456
Vat number: 01/123456
```

- Teraz na ten istý prípad použijeme dedičnosť typu `public`. To znamená, že `CompanyClient` nebude potrebovať metódu `printInfo()`, pretože bude môcť oznámiť údaje v hlavnom programe rovnako ako `BankClient`.
- Súkromné dáta triedy `BankClient`, teda `accountBalance a accountNumber`, zostanú však preň naďalej nedostupné. Môže síce použiť metódu `AccountBalance()`, nemá to však zmysel, keďže nemôže túto hodnotu inicializovať ani s ňou inak narábať.

- Príklad: Prístupnosť dát podľa rôzneho typu dedičnosti - Public dedenie

```
class CompanyClient :public BankClient {
        //všetko, čo CompanyClient zdedí od triedy BankClient, bude mat rovnaku
        pristupnost v triede CompanyClient
private:
        std::string ico, vatNumber;
public:
        CompanyClient(std::string m, std::string a, std::string i, std::string vn);
        std::string Ico() { return ico; }
        std::string VatNumber() { return vatNumber; }
};
```

- Hlavný program:

```
BankClient client("Pavol Horvath", "Bratislava", "012011/0100", 15000);
std::cout << "Klient: " << client.Name() << std::endl;
std::cout << "Adresa: " << client.Address() << std::endl;
std::cout << "Cislo uctu: " << client.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << client.AccountBalance() << " Eur" << std::endl;

CompanyClient companyClient("Eset", "Bratislava", "123456", "01/123456");
std::cout << "Klient: " << companyClient.Name() << std::endl;
std::cout << "Adresa: " << companyClient.Address() << std::endl;
std::cout << "ICO: " << companyClient.Ico() << std::endl;
std::cout << "Vat number: " << companyClient.VatNumber() << std::endl;
```

- Výstup:

```
Klient: Pavol Horvath
Adresa: Bratislava
Cislo uctu: 012011/0100
Stav uctu: 15000 Eur

Klient: Eset
Adresa: Bratislava
ICO: 123456
Vat number: 01/123456
```

- Ako upraviť triedu `BankClient` tak aby trieda `CompanyClient` vedela pracovať s premennými `accountBalance` a `accountNumber`?

- Príklad: Trieda BankClient doplnená o metódy pre private premenné. Doplníme triedu `BankClient` o `protected` metódy, ktoré budú nastavovať hodnoty týchto premenných

```
class BankClient {
...
protected:
        void setAccountBalance(int ab) { accountBalance = ab; }
        void setAccountNumber(std::string an) { accountNumber = an; }
...
};
```

- Upravíme konštruktor triedy `CompanyClient`

```
class CompanyClient :public BankClient {
...
public:
CompanyClient(std::string m, std::string a, std::string i, std::string vn, std::string an, int ab);
...
};
```

```
CompanyClient::CompanyClient(std::string m, std::string a, std::string i, std::string vn, std::string an, int ab) {
        //k tymto premmennym triedy BankClient mam pristup vo funkciach triedy CompanyClient
        name = m;
        address = a;
        //premenne triedy CompanyClient
        ico = i;
        vatNumber = vn;
        //protected metody triedy BankClient pomocou ktorých nastavim private premenné
        setAccountBalance(ab);
        setAccountNumber(an);
}
```

- Hlavný kód:

```
BankClient client("Pavol Horvath", "Bratislava", "012011/0100", 15000);
std::cout << "Klient: " << client.Name() << std::endl;
std::cout << "Adresa: " << client.Address() << std::endl;
std::cout << "Cislo uctu: " << client.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << client.AccountBalance() << " Eur" << std::endl;

CompanyClient companyClient("Eset", "Bratislava", "123456", "01/123456", "012011/0100",
150000);
std::cout << "Klient: " << companyClient.Name() << std::endl;
std::cout << "Adresa: " << companyClient.Address() << std::endl;
std::cout << "Cislo uctu: " << companyClient.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << companyClient.AccountBalance() << std::endl;
std::cout << "ICO: " << companyClient.Ico() << std::endl;
std::cout << "Vat number: " << companyClient.VatNumber() << std::endl;
```

Konštruktor a dedičnosť

- V predchádzajúcom príklade obsahovala trieda `BankClient` okrem konštruktora s parametrami aj prázdny konštruktor. Malo to svoje opodstatnenie, ktoré si teraz objasníme.
- Stačí si uvedomiť, aký je mechanizmus vytvárania odvodenej triedy. Keďže odvodená trieda má svojho rodiča, **najprv sa zavolá koštruktor rodiča a až potom konštruktor potomka**. Keďže rodičovská trieda môže mať niekoľko konštruktorov, musíme povedať, ktorý z nich sa má zavolať. Ak priamo neuvedieme, ktorý konštruktor chceme, automaticky sa zavolá prázdny konštruktor, ktorý musí byť k dispozícii.
- Príklad: Konštruktor triedy `CompanyClient` s využitím konštruktora triedy `BankClient`

  Keďže `CompanyClient` má niekoľko parametrov prebratých od svojej rodičovskej triedy, nemusíme ich priamo inicializovať v jeho konštruktore. Môžeme použiť takýto zápis:

```
CompanyClient::CompanyClient(std::string m, std::string a, std::string i, std::string vn, std::string an, int ab) :
BankClient(m,a,an,ab) {

        //nastavujeme iba premenne triedy CompanyClient

        ico = i;

        vatNumber = vn;

}
```

- Týmto spôsobom sa nastavia všetky údaje, ktoré má aj `BankClient`. Zostáva nastaviť už len zvyšné údaje. Všimnime si, že týmto spôsobom sa podarí nastaviť dokonca aj údaje `accountBalance a accountNumber`, keďže jeho inicializácia prebieha v konštruktore rodičovskej triedy.

Konštruktor a dedičnosť

- Hlavný program:

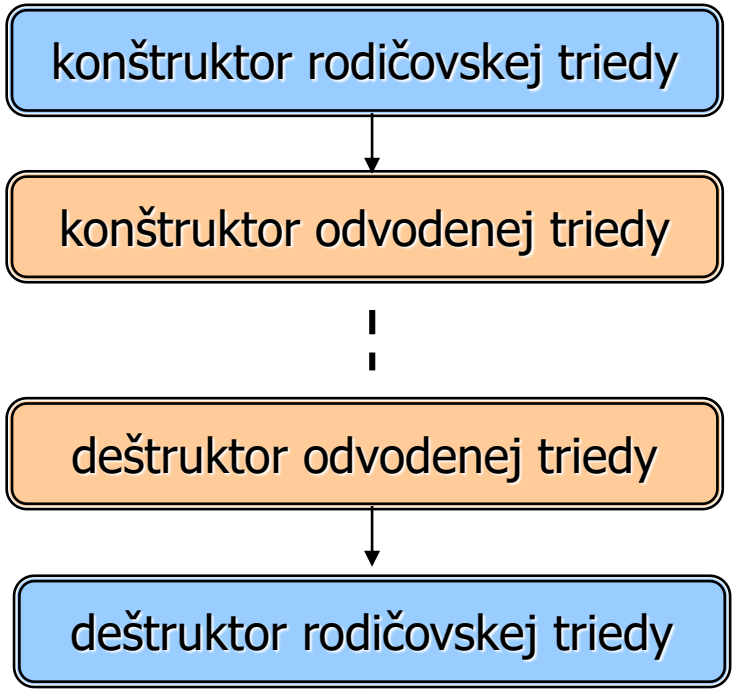
```
CompanyClient companyClient("Eset", "Bratislava", "123456", "01/123456", "012011/0100", 150000);
std::cout << "Klient: " << companyClient.Name() << std::endl;
std::cout << "Adresa: " << companyClient.Address() << std::endl;
std::cout << "Cislo uctu: " << companyClient.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << companyClient.AccountBalance() << std::endl;
std::cout << "ICO: " << companyClient.Ico() << std::endl;
std::cout << "Vat number: " << companyClient.VatNumber() << std::endl;
```

- Výstup:

```
Klient: Eset
Adresa: Bratislava
Cislo uctu: 012011/0100
Stav uctu: 150000
ICO: 123456
Vat number: 01/123456
```

Deštruktor a dedičnosť

- Kým pri vytváraní objektu odvodenej triedy sa najskôr zavolá konštruktor rodičovskej triedy a až potom konštruktor podtriedy, pri zániku objektu je postup presne opačný – najskôr sa zavolá deštruktor potomka a až potom deštruktor rodičovskej triedy.
- Schéma vytvárania a zániku objektu odvodenej triedy je teda takáto:



Deštruktor a dedičnosť

- V predchádzajúcom príklade doplníme do konštruktoru a deštruktoru a výpisy:

```
konstruktor triedy BankClient
konstruktor triedy CompanyClient
Klient: Eset
Adresa: Bratislava
Cislo uctu: 012011/0100
Stav uctu: 150000
ICO: 123456
Vat number: 01/123456
destruktor triedy CompanyClient
destruktor triedy BankClient
```

Viacnásobná dedičnosť

- Rozlišujeme dva typy viacnásobnej dedičnosti:

  1. Viacúrovňová dedičnosť – odvodená trieda sa stane rodičovskou triedou pre ďalšiu odvodenú triedu

  2. Dedičnosť od viacerých rodičov – odvodená trieda dedí naraz od dvoch alebo viacerých tried

  Samozrejme tieto typy dedičnosti môžu byť rôzne kombinované

Viacúrovňová dedičnosť

- V tomto prípade sa odvodená trieda stane rodičom ďalšej triedy, pričom postup sa môže opakovať. Konštruktory sa spúšťajú od triedy, ktorá je v hierarchii najvyššie, po tú, ktorá je najnižšie. Deštruktory sa spúšťajú v opačnom poradí.
- Príklad: Supermarket

  Vytvoríme program, ktorý na základe kódu vyhľadá príslušný tovar a podá o ňom informácie. Budeme uvažovať tri triedy:

  1. Trieda `Goods` – bude obsahovať kód tovaru, názov a cenu

  2. Trieda `Food` – bude potomkom triedy `Goods`, bude navyše obsahovať dátum spotreby

  3. Trieda `Vegetable` – bude potomkom triedy `Food` a navyše bude obsahovať údaj o triede akosti

  Aby sme mohli sledovať priebeh vytvárania a zániku objektov, vytvoríme v týchto triedach aj deštruktory, ktorých úlohou bude len informovať o tom, ktorý deštruktor sa práve spúšťa.

- Naskôr definujeme triedu Goods:

```
class Goods {
protected:
        int code;
        std::string name;
        float price;
public:
        Goods(int k, std::string n, float c) :code(k), name(n), price(c){
                std::cout << "Vytvara sa objekt triedy Goods" << std::endl;
        }
        ~Goods() { std::cout << "Zanika objekt triedy Goods" << std::endl; }
        int Code() { return code; }
        std::string Name() { return name; }
        float Price() { return price; }
};
```

- Trieda Food:

```
class Food :public Goods{
protected:
        std::string expirationDate;
public:
        Food(int k, std::string n, float c, std::string ds) : Goods(k, n, c), expirationDate(ds){
                std::cout << "Vytvara sa objekt triedy Food" << std::endl;
        }
        ~Food() { std::cout << "Zanika objekt triedy Food" << std::endl; }
        std::string ExpirationDate() { return expirationDate; }
};
```

- Trieda Vegetable:

```
class Vegetable :public Food{
private:
        int quality;
public:
        Vegetable(int k, std::string n, float c, std::string ds, int t) :Food(k, n, c, ds), quality(t){
                std::cout << "Vytvara sa objekt triedy Vegetable" << std::endl;
        }
        ~Vegetable() { std::cout << "Zanika objekt triedy Vegetable" << std::endl; }
        int Quality() { return quality; }
};
```

- Hlavný program:

```
Vegetable z[3] = { Vegetable(123, "Paprika", 2.29, "28.11.2021", 1),Vegetable(540, "Uhorka", 1.79, "6.12.2021", 2),
Vegetable(673, "Kapusta", 0.59, "20.11.2021", 1)};
int code;
std::cout << "Zadaj kod: " << std::endl;
std::cin >> code;
int index = -1;
for (int i = 0; i < 3; i++) {
            if (z[i].Code() == code) {
                        index = i;
                        break;
            }
}
if (index == -1) {
            std::cout << "Tovar s takymto kodom nemame na sklade" << std::endl;
} else {
            std::cout << z[index].Name() << std::endl;
            std::cout << z[index].Price() << " Eur/kg" << std::endl;
            std::cout << "Datum spotreby: " << z[index].ExpirationDate() << std::endl;
            std::cout << "Trieda: " << z[index].Quality() << std::endl;
}
```

- Výstup:

  Vytvara sa objekt triedy Goods

  Vytvara sa objekt triedy Food

  Vytvara sa objekt triedy Vegetable

  ...

  Zadaj kod:

  123

  Paprika

  2.29 Eur/kg

  Datum spotreby: 28.11.2020

  Trieda: 1

  ...

  Zanika objekt triedy Zelenina

  Zanika objekt triedy Food

  Zanika objekt triedy Vegetable

Dedičnosť od viacerých rodičov

- V tomto prípade trieda dedí dáta a metódy od viacerých tried naraz.
- Syntax:

```
class odvodená_trieda :typ_dedičnosti1 trieda1, typ_dedičnosti2 trieda2 ... {
        ...
}
```

- Konštruktory sa v tomto prípade spúšťajú v poradí zľava doprava, deštruktory sprava doľava.
- Príklad: Dedičnosť od viacerých rodičov
  Vytvoríme tri triedy:

  `Mammal` – s údajom o dĺžke kŕmenia mláďat

  `WingedAnimal` – s údajom o rozpätí krídiel

  `Bat` – potomok tried `Mammal` a `WingedAnimal`, s údajom o hĺbke jaskýň v ktorých žije

- Trieda `Mammal`:

```
class Mammal{
protected:
        double feedingPeriod = 0;;
public:
        Mammal(double fP) : feedingPeriod(fP) {};
};
```

- Trieda `WingedAnimal`:

```
class WingedAnimal{
protected:
        double wingspan = 0;
public:
        WingedAnimal(double w) : wingspan(w) {};
};
```

- Trieda Bat:

```
class Bat : public Mammal, public WingedAnimal {
protected:
        int caveDepth = 0;
public:
        Bat(double fP, double w, int cD) : Mammal(fP), WingedAnimal(w), caveDepth(cD) {};
        void printInfo() {
                std::cout << "Feeding period: " << feedingPeriod << " months" << std::endl;
                std::cout << "Wingspan: " << wingspan << " cm" << std::endl;
                std::cout << "Living in cave with depth: " << caveDepth << " m" << std::endl;
        }
};
```

- Hlavný program:

```
        Bat b(5., 20., 15);
        b.printInfo();
```

- Výstup:

```
        Feeding period: 5 months
        Wingspan: 20 cm
        Living in cave with depth: 15 m
```

Dedičnosť od viacerých rodičov

- Nie je povolené, aby mali obidve rodičovské funkcie parameter s rovnakým menom alebo funkciu s rovnakým menom:
- Príklad: Trieda `WingedAnimal` s údajom o dĺžke krmenia

  Do triedy `WingedAnimal` doplníme údaj `feedingPeriod:`

```
class WingedAnimal  {
protected:
        double feedingPeriod = 0;;
        double wingspan = 0;
public:
        WingedAnimal(double w) : wingspan(w) {};
};
```

- Chyba počas kompilácie:

```
error C2385: ambiguous access of 'feedingPeriod‘
message : could be the 'feedingPeriod' in base 'Mammal‘
message : or could be the 'feedingPeriod' in base 'WingedAnimal'
```

Dedičnosť a smerníky

- V súvislosti s dedičnosťou prichádza na scénu doteraz nevyužitá vlastnosť smerníkov na objekty – ak je smerník deklarovaný ako smerník na objekt rodičovskej triedy, môže byť použitý aj tak, aby ukazoval na objekt odvodenej triedy.
- Obmedzením v tomto prípade je, že cez takýto smerník môžeme pristupovať len k takým dátam a metódam, ktoré mala aj rodičovská trieda.
- Príklad: Smerník na objekt triedy `Goods`

  Použijeme smerník na objekt triedy `Goods` na prístup k objektu triedy `Vegetable`:

```
Vegetable z[3] = {
        Vegetable(123, "Paprika", 2.29, "28.11.2020", 1),
        Vegetable(540, "Uhorka", 1.79, "6.12.2020", 2),
        Vegetable(673, "Kapusta", 0.59, "20.11.2020", 1)
};
Goods* pointer = z;
std::cout << "Smernik ukazuje na tuto zeleninu: " << std::endl;
std::cout << pointer->Name() << std::endl;
std::cout << pointer->Price() << " Eur/kg" << std::endl;
```

Dedičnosť a smerníky

- Výstup:

```
Smernik ukazuje na tuto zeleninu:
Paprika
2.29 Eur/kg
```

- Fungoval by aj takýto prístup:

```
Goods* g = new Vegetable(540, "Uhorka", 1.79, "6.12.2020", 2);
std::cout << "Smernik ukazuje na tuto zeleninu: " << std::endl;
std::cout << g->Name() << std::endl;
std::cout << g->Price() << " Eur/kg" << std::endl;
```

- Výstup:

```
Smernik ukazuje na tuto zeleninu:
Uhorka
1.79 Eur/kg
```

Virtuálne funkcie

- **Virtuálna funkcia** je funkcia, ktorá je deklarovaná v rámci rodičovskej triedy a je predefinovaná odvodenou triedou. V odvodenej triede môže mať teda funkcia úplne iný obsah.
- Platí pravidlo, že predefinovaná funkcia musí mať rovnaký počet aj typ parametrov aj rovnakú návratovú hodnotu ako funkcia rodičovskej triedy.
- Ak chceme funkciu označiť ako virtuálnu, dosiahneme to vložením kľúčového slova `virtual` pred deklaráciu funkcie. Toto označenie je potrebné len v rodičovskej triede.
- Pri definovaní funkcie (mimo tela triedy) už kľúčové slovo `virtual` neuvádzame.
- Predefinovanie virtuálnej triedy odvodenou triedou sa inak nazýva aj **prehodnotenie (overriding)**.
- Virtuálne funkcie sú vlastne ďalšou formou polymorfizmu, keďže funkcia s jedným menom môže pre každú triedu vykonávať rôzne inštrukcie.
- Príklad: Supermarket s výpisom informácií o tovare

  Do tried `Goods`, `Food` a `Vegetable` doplníme funkciu `info()`, ktorá vypíše informácie o danom tovare. Vypísané informácie sa budú pre každú triedu líšiť. V triede `Goods` a `Food` bude funkcia deklarovaná ako virtuálna, v triede `Vegetable` to už tak nemusí byť.

- Trieda Goods:

```
class Goods {
        ...
public:
        ...
        virtual void info();
};
```

- Trieda Food:

```
class Food :public Goods{
        ...
public:
        ...
        virtual void info();
};
```

- Trieda Vegetable:

```
class Vegetable :public Food{
...
public:
        ...
        void info();
};
```

- Definícia funkcie info() pre každú triedu:

```
void Goods::info() {
        std::cout << "Tovar s kodom " << code << std::endl;
        std::cout << name << std::endl;
        std::cout << price << " Eur" << std::endl;
}
void Food::info() {
        std::cout << "Tovar s kodom " << code << std::endl;
        std::cout << name << std::endl;
        std::cout << price << " Eur" << std::endl;
        std::cout << "Datum spotreby: " << expirationDate << std::endl;
}
void Vegetable::info() {
        std::cout << "Tovar s kodom " << code << std::endl;
        std::cout << name << std::endl;
        std::cout << price << " Eur/kg" << std::endl;
        std::cout << "Datum spotreby: " << expirationDate << std::endl;
        std::cout << "Trieda: " << quality << std::endl;
}
```

- Hlavný program:

```
Goods t(309, "Chladnicka", 452.31);
Food p(598, "Chlieb", 1.19, "5.11.2020");
Vegetable z(439, "Cibula", 20, "10.1.2021", 1);
t.info();
p.info();
z.info();
```

- Výstup:

```
Tovar s kodom 309
Chladnicka
452.31 Eur
Tovar s kodom 598
Chlieb
1.19 Eur
Datum spotreby: 5.11.2020
Tovar s kodom 439
Cibula
20 Eur/kg
Datum spotreby: 10.1.2021
Trieda: 1
```

- Funkcie `info()` pre každú triedu môžeme prepísať aj tak, že využijeme príslušnú funkciu rodičovskej triedy:

```
void Goods::info(){
        std::cout << "Tovar s kodom " << code << std::endl;
        std::cout << name << std::endl;
        std::cout << price << " Eur" << std::endl;
}

void Food::info(){
        Goods::info();
        std::cout << "Datum spotreby: " << expirationDate << std::endl;
}

void Vegetable::info(){
        Food::info();
        std::cout << "Trieda: " << quality << std::endl;
}
```

Virtuálne funkcie a smerníky

- Predchádzajúci príklad by bol fungoval, aj keby funkcia `info()` nebola označená ako virtuálna. Skutočný význam virtuálnych funkcií sa ukáže napr. pri použití smerníkov.
- Už sme si hovorili, že smerník na rodičovskú triedu môže v skutočnosti ukazovať aj na odvodenú triedu. Ak však chceme zavolať metódu, ktorá je v odvodenej triede predefinovaná, musí byť táto metóda v rodičovskej triede označená ako virtuálna, aby program vedel správne rozoznať, ktorú verziu má použiť.
- Majme napr. smerník na triedu `Goods`, ktorý bude ukazovať na objekt triedy `Vegetable`:

```
Goods* g = new Vegetable(540, "Uhorka", 1.79, "6.12.2020", 2);
```

- Ak chceme zavolať funkciu `info()`, t.j.

```
g->info();
```

  program má na výber funkciu `info()` triedy `Goods` a triedy `Vegetable`.

- Správnu verziu (teda funkciu triedy `Vegetable`) zavolá vďaka tomu, že je funkcia `info()` v triede `Goods` označená ako **virtuálna**.

- Príklad: Supermarket s poľom smerníkov

  Vytvoríme pole smerníkov na objekt typu Goods, pričom ale smerníky budú môcť ukazovať aj na objekty typu Food a Vegetable. Potom si dáme vypísať informácie o všetkých prvkoch poľa.

```
Goods* t[3] = { new Goods(309, "Chladnicka", 452.31),
          new Food(598, "Chlieb", 1.19, "5.11.2020"),
          new Vegetable(439, "Cibula", 20, "10.1.2021", 1)
          };
for (int i = 0; i < 3; i++) {
          t[i]->info();}
```

- Ak by funkcia info() nebola deklarovaná ako virtuálna, zavolala by sa vždy funkcia patriaca triede Goods. Dostali by sme nasledujúci výstup:

```
Tovar s kodom 309
Chladnicka
452.31 Eur
Tovar s kodom 598
Chlieb
1.19 Eur
Tovar s kodom 439
Cibula
20 Eur
```

Virtuálne funkcie a smerníky

- Ak však máme funkciu `info()` deklarovanú ako virtuálnu (minimálne v rodičovskej triede `Goods`), program je schopný rozoznať, ktorú funkciu `info()` má zavolať. Dostaneme teda výstup:

```
Tovar s kodom 309
Chladnicka
452.31 Eur
Tovar s kodom 598
Chlieb
1.19 Eur
Datum spotreby: 5.11.2020
Tovar s kodom 439
Cibula
20 Eur
Datum spotreby: 10.1.2021
Trieda: 1
```

Virtuálny deštruktor

- Zatiaľ čo konštruktor nemôže byť virtuálny, deštruktor môže. V niektorých prípadoch je to dokonca žiadúce.
- Príklad: Supermarket s virtuálnymi deštruktormi:

  Ak si v predchádzajúcom príklade dáme vypísať informácie o konštruktoroch a deštruktoroch, dostaneme nasledujúci výstup:

```
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Food
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Food
...
Zanika objekt triedy Goods
Zanika objekt triedy Goods
Zanika objekt triedy Goods
```

- Vidíme, že sa pre objekty všetkých tried zavolal len deštruktor triedy `Goods`.

- Ak chceme, aby sa zavolali správne deštruktory, musíme deštruktor v rodičovskej triede Goods označiť ako virtuálny:

```
class Goods {
        ...
public:
        virtual ~Goods() { std::cout << "Zaniká objekt triedy Goods" << std::endl; }
        ...
};
```

- Výstup:

```
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Food
Vytvara sa objekt triedy Goods
Vytvara sa objekt triedy Food
Vytvara sa objekt triedy Vegetable
...
Zanika objekt triedy Goods
Zanika objekt triedy Food
Zanika objekt triedy Goods
Zanika objekt triedy Vegetable
Zanika objekt triedy Food
Zanika objekt triedy Goods
```

Dedičnosť od viacerých rodičov a virtuálne funkcie

- Príklad: Rozšírme priklad z časti o viacnasobnej dedičnosti o triedu `Animal` ktorú bude rodičovská trieda tried `Mammal` a `WingedAnimal`:

```
class Animal {
protected:
        std::string name = "";
        std::string order = "";
public:
        Animal() {std::cout << "Prazdny konstruktor triedy Animal" << std::endl; };
        Animal(std::string n, std::string o) :name(n), order(o) {
                std::cout << "Konstruktor triedy Animal" << std::endl;
        };
};
```

```
class Mammal : public Animal {
protected:
        double feedingPeriod = 0;;
public:
        Mammal(std::string n, std::string o, double fP) : Animal(n,o), feedingPeriod(fP) {
                std::cout << "Konstruktor triedy Mammal" << std::endl;
        };
};

class WingedAnimal : public Animal {
protected:
        double wingspan = 0;
public:
        WingedAnimal(std::string n, std::string o, double w) : Animal(n, o), wingspan(w) {
                std::cout << "Konstruktor triedy WingedAnimal" << std::endl;
        };
};
```

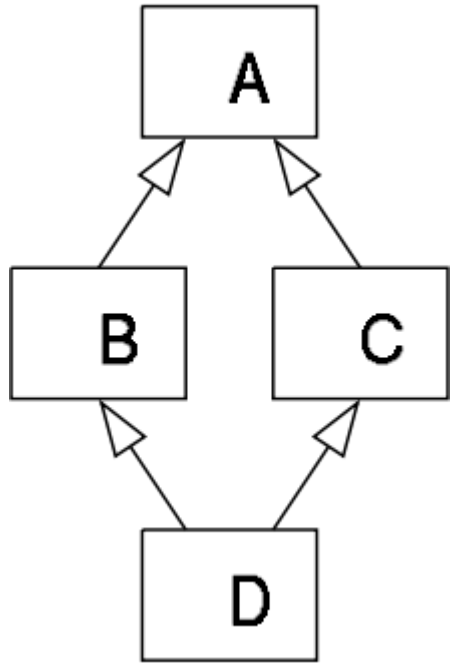
```
class Bat : public Mammal, public WingedAnimal {
protected:
        int caveDepth = 0;
public:
        Bat(std::string n, std::string o, double fP, double w, int cD) : Mammal(n, o, fP),
        WingedAnimal(n, o, cD), caveDepth(cD) {};
        void printInfo() {
                std::cout << "Name: " << name << std::endl;
                std::cout << "Order: " << order << std::endl;
                std::cout << "Feeding period: " << feedingPeriod << " months" << std::endl;
                std::cout << "Wingspan: " << wingspan << " cm" << std::endl;
                std::cout << "Living in cave with depth: " << caveDepth << " m" << std::endl;
        }
};
```

- Hlavný program:

```
Bat b("Netopier obycajny", "netopierovite", 5., 20., 15);
b.printInfo();
```

- Takýto program sa nepodarí ani skompilovať:

```
error C2385: ambiguous access of 'name'
message : could be the 'name' in base 'Animal'
message : or could be the 'name' in base 'Animal'
error C2385: ambiguous access of 'order'
message : could be the 'order' in base 'Animal'
message : or could be the 'order' in base 'Animal‘
```



- Vznikol takzvaný **diamond problem:** ak má trieda dve nadradené triedy, ktoré majú spoločne nadradenú inú triedu, môže vzniknúť nejednoznačnosť (`ambiguous = nejednoznačný`)
- V našom príklade sa chcel konštruktor triedy `Animal` zavolať dva krát keď sa vytváral objekt triedy `Bat`. Ak by objekt zanikal, zavolá sa dva krát deštruktor. Ak by sa program podarilo skompilovať, objekt triedy `Bat` by mal dve kópie premenných triedy `Animal` a tu zaniká už spomínaná nejednoznačnosť.

Dedičnosť od viacerých rodičov a virtuálne funkcie

- Príklad: Zmeňme dedičnosť pre triedy `Mammal` a `WingedAnimal` na virtuálnu:

```
class Mammal : virtual public Animal {... }
class WingedAnimal : virtual public Animal {...}
```

- Program sa skompiluje, ale nedostaneme očakávaný výstup:

```
Prazdny konstruktor triedy Animal
Konstruktor triedy Mammal
Konstruktor triedy WingedAnimal
Name:
Order:
Feeding period: 5 months
Wingspan: 15 cm
Living in cave with depth: 15 m
```

- Vidíme, že aj napriek tomu že bol v konštruktore pre triedu `Mammal` a `WingedAnimal` volaný konštruktor s parametrami pre triedu `Animal`, bol zavolaný prázdny konštruktor triedy `Animal`.

Dedičnosť od viacerých rodičov a virtuálne funkcie

- Ak chceme zavolať konštruktor s parametrami triedy `Animal` z triedy `Bat` , musíme ho explicitne zavolať:

```
class Bat : public Mammal,  public WingedAnimal {
...
public:
        Bat(std::string n, std::string o, double fP, double w, int cD) : Mammal(n, o, fP),
        WingedAnimal(n, o, cD), Animal(n,o), caveDepth(cD) {};
...
};
```

- Výstup:

```
Konstruktor triedy Animal
Konstruktor triedy Mammal
Konstruktor triedy WingedAnimal
Name: Netopier obycajny
Order: netopierovite
Feeding period: 5 months
Wingspan: 15 cm
Living in cave with depth: 15 m
```

Spriatelené triedy

- Ak máme v triede súkromné dáta a chceme, aby iná trieda mala k nim prístup, môžeme to zabezpečiť tým, že ju označíme za spriatelenú (friend).
- Príklad: úprava private premenných pomocou spriatelenej triedy `FriendBankClient:`

```
class BankClient {
friend class FriendBankClient;
private:
        int accountBalance = 0;
        std::string accountNumber = "";
        ...
};
class FriendBankClient:public BankClient {
public:
        //možem pristúpiť k private premenným triedy BankClient
        void changeAccountBalance(int aB) { accountBalance = aB; }
        void changeAccountNumber(std::string aN) {accountNumber = aN;}
};
```

Spriatelené triedy

- Hlavný program:

```
BankClient client("Pavol Horvath", "Bratislava", "012011/0100", 15000);
std::cout << "Klient: " << client.Name() << std::endl;
std::cout << "Adresa: " << client.Address() << std::endl;
std::cout << "Cislo uctu: " << client.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << client.AccountBalance() << " Eur" << std::endl;

FriendBankClient friendClient;
friendClient.changeAccountBalance(30000);
friendClient.changeAccountNumber("012011/1234");

std::cout << "Cislo uctu: " << friendClient.AccountNumber() << std::endl;
std::cout << "Stav uctu: " << friendClient.AccountBalance() << " Eur" << std::endl;
```

Spriatelené funkcie

- Spriatelená funkcia v triede je definovaná mimo menného priestoru tejto triedy, ale má právo prístupu ku všetkým súkromným a chráneným premenným triedy.

```
class BankClient {
          ...
private:
          int accountBalance = 0;
          std::string accountNumber = "";
          ...
public:
          ...
           friend std::ostream& operator <<(std::ostream& output, const BankClient& B);
};
//Spriatelena funckia je mimo BankClient:: ale ma pristup k accountNumber a accountBalance
std::ostream & operator<<(std::ostream & output, const BankClient& B){
          output << B.name <<"  "<< B.address <<"  "<<B.accountNumber <<"  "<< B.accountBalance << std::endl;
          return output;
}
```

Statické členy v triede

- Premenné a funkcie (členy) triedy môžeme definovať ako statické pomocou kľúčového slova `static`. Keď deklarujeme člena triedy ako statického, znamená to, že bez ohľadu na to, koľko objektov v triede je vytvorených, existuje iba jeho jedna kópia.
- Statický člen zdieľajú všetky objekty triedy. Keď sa vytvorí prvý objekt, všetky statické premenné sa inicializované na nulu, ak nie je k dispozícii iná inicializácia (musí byť mimo teľa triedy). **Statická funkcia má prístup iba k statickým premenným**
- Príklad: Počítadlo objektov:

```
class Box {
private:
        double length, breadth, height;
public:
        static int objectCount;
        Box(double l = 2.0, double b = 2.0, double h = 2.0):length(l), breadth(b), height(h) {
                std::cout << "Zavolany konstruktor" << std::endl;
                // staticka premenna sa zvysi vzdy ked je objekt vytvoreny
                objectCount++;
        }
        static int getCount() { return objectCount; } //staticka funkcia
};
// inicializacia statickej premennej
int Box::objectCount = 0;
```

Statické členy v triede

- Hlavný program:

```
std::cout << "Pocet objektov: " << Box::getCount() << std::endl;
Box Box1(3.3, 1.2, 1.5);
Box Box2(8.5, 6.0, 2.0);
std::cout << "Pocet objektov: " << Box::getCount() << std::endl;
```

- Výstup:

```
Pocet objektov: 0
Zavolany konstruktor
Zavolany konstruktor
Pocet objektov: 2
```

- Keďže je premenná `static int objectCount;` typu `public` to isté by sme dostali aj kódom:

```
std::cout << "Pocet objektov: " << Box::objectCount << std::endl;
Box Box1(3.3, 1.2, 1.5);
Box Box2(8.5, 6.0, 2.0);
std::cout << "Pocet objektov: " << Box::objectCount << std::endl;
```

Chyby a výnimky

- Pri vývoji a používaní programu sa môže vyskytnúť niekoľko druhov problémov:

1. Chyby v kóde – tieto chyby sa zachytia už v štádiu kompilácie, to znamená, že ak sa v programe vyskytujú, program sa ani nespustí
2. Logické chyby – program sa spustí, ale kvôli chybe programátora nefunguje správne
3. **Výnimky** – sú to problémy, ktoré nastanú počas behu programu a vznikajú v dôsledku vonkajších okolností - nevhodných dát (delenie nulou, pokus o čítanie neexistujúceho súboru..), hardvérových problémov (nedostatok pamäte, poškodený hardvér) a podobne

- Zatiaľ čo chyby (v kóde a logické) musí programátor odstrániť, výnimky môže len predvídať a môže zahrnúť do programu príkazy, ktoré sa vykonajú, ak výnimka nastane

Spracovanie výnimky

- Spracovanie výnimky v C++ je založené na troch kľúčových slovách: try, throw a catch (skús, oznám a zachyť)

- Blok `try` – obsahuje časť programu, v ktorej sa môže vyskytnúť výnimka. Môžeme ho chápať tak, že program skúsi danú časť programu vykonať a zisťuje, či pri tom nastane výnimka alebo nie.
- Príkaz `throw` – oznámi, že nastala výnimka (a aká). Ak sa nachádza priamo v bloku `try`, vykonávanie príkazov v bloku sa končí, ak sa nachádza vo vnútri nejakej funkcie, program z funkcie odchádza. Dôležitou vlastnosťou tohto príkazu je, že zabezpečuje korektný odchod z bloku alebo funkcie - zruší lokálne premenné, ktoré boli vytvorené, a zavolá deštruktory objektov.
- Funkcia `catch` – táto funkcia zachytí výnimku určitého typu. V jej tele je uvedené, čo sa má spraviť, ak daná výnimka nastala - môže ukončiť program, korektne obísť problém, alebo skúsiť napraviť chybu.

Spracovanie výnimky

- Syntax mechanizmu try-throw-catch je nasledujúca:

```
try {
        ...
        ak nastala výnimka throw výnimka;
        ...
} catch (typ_výnimky výnimka) {
        ...
}
```

- Samozrejme, v bloku `try` môže nastať niekoľko výnimiek, rovnakého alebo rôzneho typu. Takisto príkazov `catch` môže byť za blokom `try` ľubovoľne veľa, jeden pre každý možný typ výnimky.
- Príkaz `throw` musí byť spustený buď z bloku `try` alebo z nejakej funkcie, ktorá sa volá v tomto bloku.
- Výnimka môže byť premenná ľubovoľného dátového typu.

Spracovanie výnimky

- Príklad: Delenie komplexných čísel. Pokúsime sa vydeliť dve komplexné čísla. V prípade, že nastane delenie nulou, program ohlási výnimku.
- Definujeme konštantu, ktorá bude indikátorom výnimky:

```
#define DivisionByZeroException 1
```

- Hlavný program:

```
ComplexNumber x(1, 2), y(0, 0), z;
std::cout << "Komplexne cisla: ";
x.print(); std::cout << ", "; y.print(); std::cout << std::endl;
try{
        if (y.A() == 0 && y.B() == 0) {
                throw DivisionByZeroException;
        }
        z.setA((x.A() * y.A() + x.B() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
        z.setB((x.B() * y.A() - x.A() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
}
```

Spracovanie výnimky

```
catch (int error) {
        if (error == DivisionByZeroException) {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}

std::cout << "vysledok po deleni:" << std::endl;
z.print();
```

- Výstup:

```
Komplexne cisla: 1+2i, 0+0i
Nulou delit neviem!
vysledok po deleni:
0+0i
```

Spracovanie výnimky

- Blok `try` sa v niečom podobá na funkciu – ak sa v bloku vytvoria nejaké lokálne premenné alebo objekty, zaniknú, ak program opustí blok (ak príde na jeho koniec, alebo sa vykoná príkaz `throw`).

- Príklad: Delenie komplexných čísel a zánik objektov:
- Do triedy `ComplexNumber` doplníme deštruktor, ktorý nám oznámi, ktoré komplexné číslo sa ruší.
- Hlavný program:

```
try {
        ComplexNumber z(-1,-1);
        if (y.A() == 0 && y.B() == 0) {
                throw DivisionByZeroException;
        }
        z.setA((x.A() * y.A() + x.B() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
        z.setB((x.B() * y.A() - x.A() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
        std::cout << "vysledok po deleni:" << std::endl;
        z.print();
}
```

Spracovanie výnimky

```
catch (int error) {
        if (error == DivisionByZeroException) {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}
```

- Výstup:

```
Komplexne cisla: 1+2i, 0+0i
Destruktor cisla: -1+-1i
Nulou delit neviem!
Destruktor cisla: 0+0i
Destruktor cisla: 1+2i
```

- Ak teraz y bude nulové, vznikne výnimka a blok `try` sa nedokončí.
- Vďaka `throw` sa však aj teraz zavolá deštruktor čísla z.

Spracovanie výnimky

- Ak by v programe vznikla výnimka nejakého typu a ak by za blokom `try` nenasledoval vhodný príkaz `catch`, ktorý by spracovával práve tento typ, program by zavolal systémovú funkciu **terminate()**. Táto funkcia by po vypísaní štandardnej chybovej hlášky ukončila program. Ukončenie programu pomocou **terminate**() však nie je korektné.

- Príklad: Delenie komplexných čísel bez vhodného príkazu `catch:`

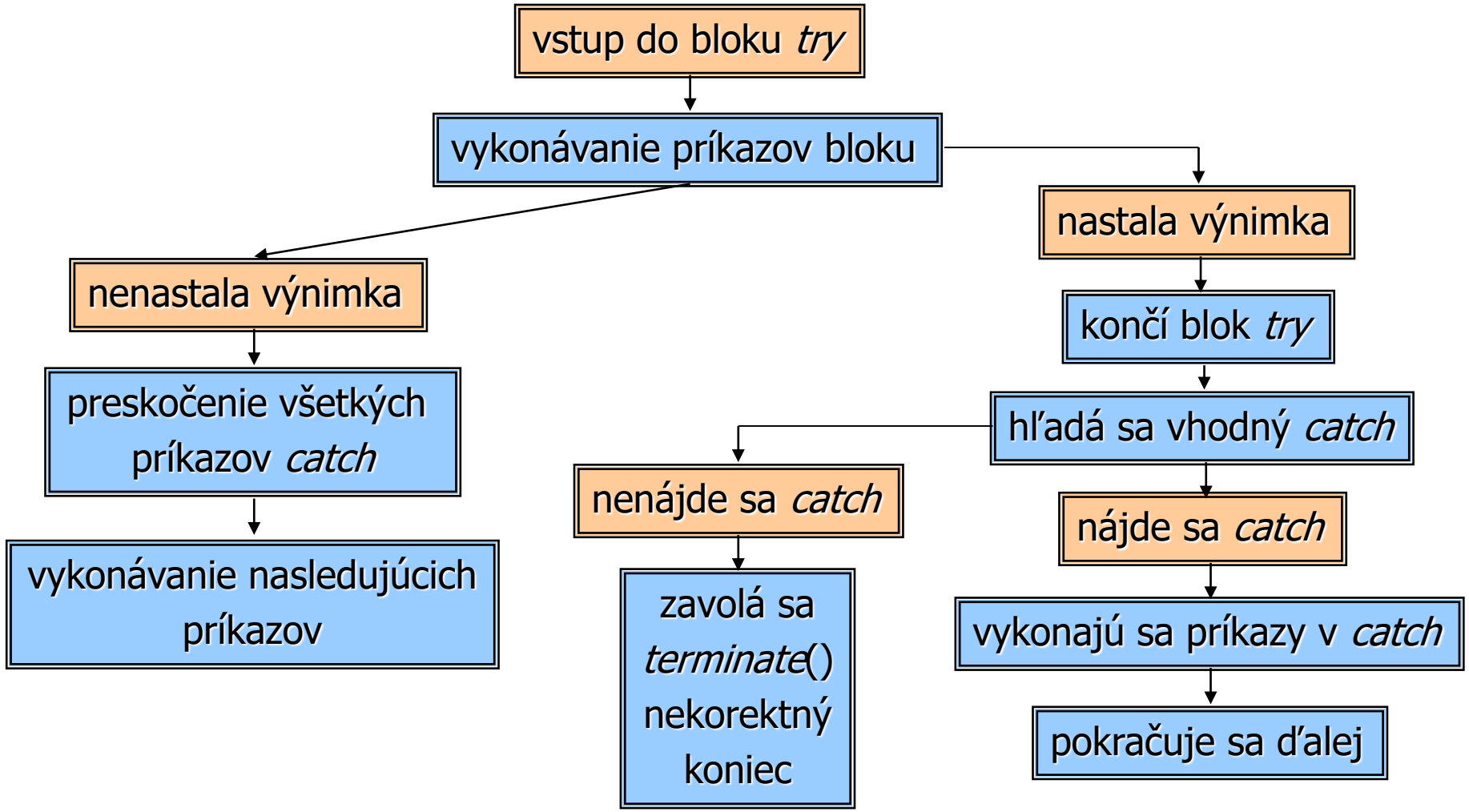
  V predchádzajúcom príklade zmeníme príkaz `catch` tak, aby zachytával výnimku typu `std::string`. Keďže naša výnimka `DivisionByZeroException` je typu `int`, pre tento typ nebudeme mať k dispozícii príkaz `catch`. Nový príkaz `catch` bude vyzerať takto:

```
catch (std::string error) {
        if (error == "DivisionByZeroException") {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}
```

- Výstup: Unhandled exception at 0x75F44662 in … : Microsoft C++ exception: int at memory location 0x0115F8CC.
- **Každý príkaz throw musí mať zodpovedajúci príkaz catch**

Spracovanie výnimky

- Postup pri zachytávaní a spracovaní výnimiek možno znázorniť takto:



Príkaz throw vo funkcii

- Už sme si hovorili, že príkaz `throw` môže byť volaný nielen priamo v bloku `try`, ale aj vo funkcii, ktorá je z tohto bloku volaná. V tomto prípade plní `throw` v podstate úlohu príkazu return – spôsobí korektný návrat z funkcie, teda návrat so zrušením lokálnych premenných.

- Príklad: Operátor / pre komplexné čísla

  Do triedy KomplexnéČíslo doplníme ďalší operátor. V ňom sa bude nachádzať aj príkaz *throw*.

```
ComplexNumber ComplexNumber::operator/(const ComplexNumber& x) {
        ComplexNumber z(-1,-1);
        if (x.a == 0 && x.b == 0){
                throw DivisionByZeroException;
        }
        z.setA((a * x.a + b * x.b) / (x.a * x.a + x.b * x.b));
        z.setB((b * x.a - a * x.b) / (x.a * x.a + x.b * x.b));
        return z;
}
```

Príkaz throw vo funkcii

- V hlavnom programe potom budeme mať v bloku `try` len samotné delenie:

```
ComplexNumber x(1, 2), y(0, 0), z(-2,-2);
...
try {
        z = x/y;
} catch (int error) {
        if (error == DivisionByZeroException) {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}
```

- Výstup: Na výstupe môžeme sledovať, ako vznikajú a zanikajú objekty. Všimnime si aj rušenie kópie potrebnej pri vrátení komplexného čísla funkciou /.

```
Komplexne cisla: 1+2i, 0+0i
Destruktor cisla: -1+-1i
Nulou delit neviem!
Destruktor cisla: -2+-2i
Destruktor cisla: 0+0i
Destruktor cisla: 1+2i
```

Výnimky viacerých typov

- Príklad: Delenie komplexných čísel s dvoma rôznymi príkazmi `catch:`

```
try {
        if (y.A() == 0 && y.B() == 0) {
                //throw std::string("DivisionByZeroException");
                //alebo
                throw DivisionByZeroException;
        }
} catch (int error) {
        if (error == DivisionByZeroException) {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
} catch (std::string error) {
        if (error == "DivisionByZeroException") {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}
```

Dátový typ enum a spracovanie výnimky

- Do triedy doplníme premennú typu enum, ktorú použijeme na spracovanie výnimky:

```
class ComplexNumber {
        ...
public:
        enum exceptions {DivisionByZero};
        ...
};
```

- Hlavný program:

```
try {
        if (y.A() == 0 && y.B() == 0) {
                throw ComplexNumber::DivisionByZero;
        }
        z.setA((x.A() * y.A() + x.B() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
        z.setB((x.B() * y.A() - x.A() * y.B()) / (y.A() * y.A() + y.B() * y.B()));
} catch (ComplexNumber::exceptions error) {
        if (error == ComplexNumber::DivisionByZero) {
                std::cout << "Nulou delit neviem!" << std::endl;
        }
}
```

Alokácia a výnimky

- V C++ je operátor *new* navrhnutý tak, aby oznamoval výnimku, ak je alokácia pamäte neúspešná.
- V bloku `try` už preto nemusíme mať príkaz `throw`, zahŕňa ho samotné *new*. Výnimka, ktorú *new* oznamuje, je typu **bad_alloc**.
- Príklad: Alokácia bloku pamäte:

```
char* data = nullptr;
size_t size;
std::cout << "Kolko bytov mam alokovat?" << std::endl;
std::cin >> size;
try {
        data = new char[size];
        std::cout << "Data uspesne alokovane" << std::endl;
} catch (std::bad_alloc error) {
        std::cout << "Nedostatok miesta" << std::endl;
}
delete[] data;
```

Alokácia a výnimky

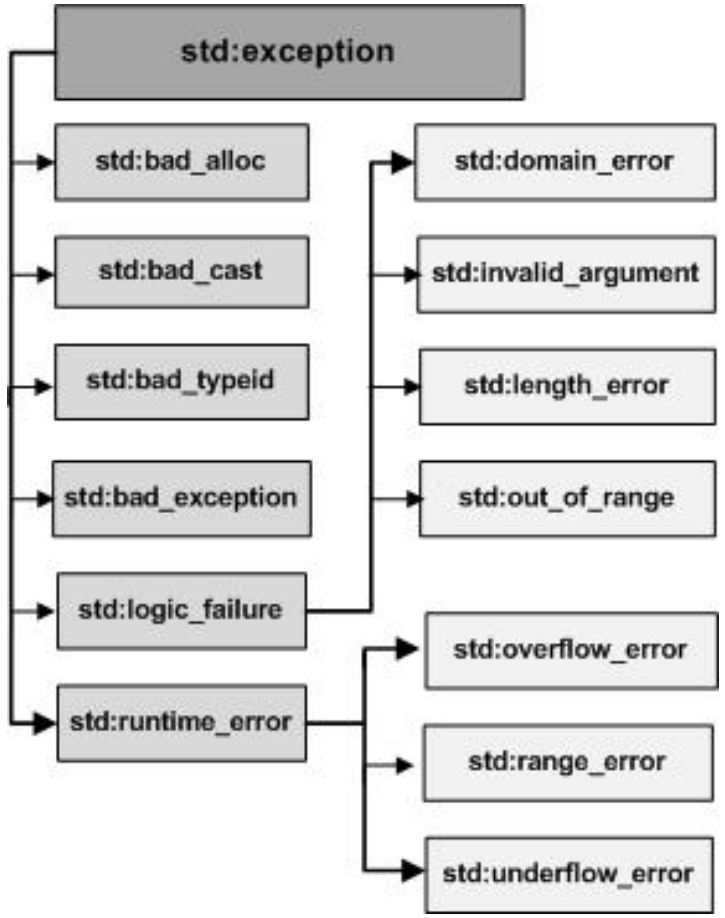
- Výstup 1 (chcem iba málo pamäte - 0.001MB):

```
Kolko bytov mam alokovat?
1000
Data uspesne alokovane
```

- Výstup 2 (chcem príliš veľa pamäte - 1TB (pri 32bitovej konfigurácii je to už veľa)):

```
Kolko bytov mam alokovat?
1000000000000
Nedostatok miesta
```

C++ Standard Exceptions:



Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by **new**.
3	**std::bad_cast** This can be thrown by **dynamic_cast**.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.
5	**std::bad_typeid** This can be thrown by **typeid**.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.
9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error** An exception that theoretically cannot be detected by reading the code.
12	**std::overflow_error** This is thrown if a mathematical overflow occurs.
13	**std::range_error** This is occurred when you try to store a value which is out of range.
14	**std::underflow_error** This is thrown if a mathematical underflow occurs.

Šablóny

- Doteraz, ak sme chceli definovať nejakú triedu alebo funkciu, museli sme presne určiť dátový typ každej premennej, každého parametra a všetkých dát triedy.
- **Šablóny** umožňujú definovať všeobecné triedy a funkcie, pri ktorých sa typy dát, premenných a parametrov určia až v okamihu použitia. Okrem dátových typov je možné mať aj netypové parametre, ktoré sa správajú v podstate podobne ako parametre funkcie.
- Výhoda: prehľadnejší a menej rozsiahly program
- V šablóne je možné meniť len dátové typy a netypové parametre, ktoré sú v hlavičke označené ako všeobecné. Ostatné typy, názvy, príkazy a celková štruktúra ostanú nezmenené.

Generické funkcie

- **Generická (všeobecná) funkcia** je šablóna pre funkcie. Používame ju vtedy, ak chceme mať viacero funkcií, ktoré sa líšia len typom parametra, návratovej hodnoty alebo lokálnej premennej.
- **Syntax**:

```
template <class Typ>
návratový_typ Meno_funkcie(parametre) {
        telo funkcie
}
```

- Úplne rovnocenná je syntax, kde namiesto kľúčového slova `class` použijeme kľúčové slovo `typename`:

```
template <typename Typ>
návratový_typ Meno_funkcie(parametre) {
        telo funkcie
}
```

Generické funkcie

- Príklad: Funkcia na výmenu dvoch hodnôt

  Definujeme funkciu, ktorá vymení hodnoty dvoch premenných. Keďže túto operáciu môžeme chcieť robiť pre premenné rôznych typov, je to ideálna situácia pre použitie šablóny.

  Definujeme funkciu *swap*:

```
template<class T>
void swap(T& a, T& b) {
        T c;
        c = a;
        a = b;
        b = c;
}
```

Generické funkcie

- Hlavný program:

```
int n = 5, m = 7;
std::cout << "n=" << n << ", m=" << m << std::endl;
swap<int>(n, m);
std::cout << "po vymene: n=" << n << ", m=" << m << std::endl;

ComplexNumber c1(1,1), c2(2,2);
std::cout << "Komplexne cisla: " << std::endl;
c1.print(); c2.print();
swap<ComplexNumber>(c1, c2);
std::cout << "Komplexne cisla po vymene: " << std::endl;
c1.print(); c2.print();
```

Generické funkcie

- Výstup:

```
n=5, m=7
po vymene: n=7, m=5
Komplexne cisla:
1+1i
2+2i
Komplexne cisla po vymene:
2+2i
1+1i
```

Generické funkcie

- Samozrejme, v generickej funkcii môžeme mať aj viac ako len jeden všeobecný typ.

- Príklad: Funkcia na nájdenie maxima dvoch čísel.

  Chceme, aby funkcia vrátila maximum z dvoch čísel, pričom tieto čísla môžu byť ľubovoľného číselného typu (nie nutne rovnakého).

  Vytvoríme generickú funkciu *Max*, ktorá bude šablónou s troma všeobecnými typmi:

```
template<class T1, class T2, class T3>
T3 Max(T1 a, T2 b) {
        if (a > b)
                return static_cast<T3>(a);
        else
                return  static_cast<T3>(b);
}
```

Generické funkcie

- Hlavný program:

```
int n = 12;
float x = -2.5;
double y = 3.7;
std::cout << "Max z " << n << " a " << x << " je " << Max<int, float, double>(n, x) << std::endl;
std::cout << "Max z " << x << " a " << y << " je " << Max<float, double, double>(x, y) << std::endl;
```

- Výstup:

```
Max z 12 a -2.5 je 12
Max z -2.5 a 3.7 je 3.7
```

Generické triedy

- Rovnako ako môžeme vytvoriť šablónu pre funkciu, môžeme vytvoriť aj šablónu pre triedu. Takú triedu potom voláme **generická trieda**.
- **Syntax** je veľmi podobná ako pri vytváraní generických funkcií. Generickú triedu vytvoríme takto:

```
template<class T>
class Meno_triedy {
        ...
};
```

- alebo pomocou *typename*:

```
template<typename T>
class Meno_triedy{
        ...
};
```

Generické triedy

- Príklad: Trieda `Vector`

  Definujeme triedu `Vector`, ktorá bude uchovávať pole hodnôt nejakého (všeobecného) typu. Bude vedieť vytvoriť pole s danou dĺžkou, povedať jeho dĺžku, povedať hodnotu konkrétneho prvku a zmeniť hodnotu na konkrétnom mieste.

  Vytvoríme generickú triedu `Vector`:

```
template<class T>
class Vector {
private:
        T* data;
        int length;
public:
        Vector(int n) :length(n) { data = new T[n]; }
        ~Vector() { delete[] data; }
        int Length() { return length; }
        T& at(int i) { return data[i]; }
        void set(int i, T h) { data[i] = h; }
};
```

Generické triedy

- V hlavnom programe vytvoríme vektor celých čísel, v ktorom budú uchované druhé mocniny čísel 0,1,2,3,4 a potom vektor reálnych čísel, v ktorom budú odmocniny tých istých čísel.

```
Vector<int> v(5);
for (int i = 0; i < v.Length(); i++) {
        v.set(i, i * i);
}
for (int i = 0; i < v.Length(); i++) {
        std::cout << i << " na druhu je: " << v.at(i) << std::endl;
}

Vector<double> w(5);
for (int i = 0; i < w.Length(); i++) {
        w.set(i, sqrt(i));
}
for (int i = 0; i < w.Length(); i++) {
        std::cout << "Odmocnina z " << i << " je: " << w.at(i) << std::endl;
}
```

Generické triedy

- Výstup:

```
0 na druhu je: 0
1 na druhu je: 1
2 na druhu je: 4
3 na druhu je: 9
4 na druhu je: 16
Odmocnina z 0 je: 0
Odmocnina z 1 je: 1
Odmocnina z 2 je: 1.41421
Odmocnina z 3 je: 1.73205
Odmocnina z 4 je: 2
```

Generické triedy

- Aj pri generických triedach môžeme za dátový typ dosadiť triedu.

- Príklad: Pole bodov

  Použijeme triedu `Point`, pričom za typ *T* dosadíme triedu `Point`:

```
class Point {
private:
        int x, y;
public:
        Point() {}
        Point(int x1, int y1) :x(x1), y(y1) {}
        int X() { return x; }
        int Y() { return y; }
        void change(int x1, int y1) { x = x1; y = y1; }
};
```

Generické triedy

- V hlavnom programe vytvoríme trojprvkové pole bodov:

```
Vector<Point> b(3);

Point C(0, 0);

b.set(0, C);

C.change(0, 10);

b.set(1, C);

C.change(5, 5);

b.set(2, C);

for (int i = 0; i < b.Length(); i++) {

        std::cout << i + 1 << ". bod ma suradnice: " << b.at(i).X() << ", " << b.at(i).Y() << std::endl;

}
```

- Výstup:

```
1. bod ma suradnice: 0, 0

2. bod ma suradnice: 0, 10

3. bod ma suradnice: 5, 5
```

Netypové parametre

- Už sme si hovorili, že okrem typových parametrov, kvôli ktorým sa šablóny používajú najčastejšie, môžeme mať aj netypové parametre, ktoré vyzerajú podobne ako parametre funkcií.
- Príklad: Trieda `StaticVector` s netypovým parametrom

  Šablónu triedy `Vector` pozmeníme tak, aby mala okrem typového parametra aj netypový, ktorý bude udávať veľkosť poľa. Pole sa teda nebude alokovať dynamicky, ale staticky.

```
template<class T, int N>
class StaticVector {
private:
        T data[N];
        int length;
public:
        StaticVector() { length = N; }
        int Length() { return length; }
        T& at(int i) { return data[i]; }
        void set(int i, T h) { data[i] = h; }
};
```

Netypové parametre

- Hlavný program:

```
StaticVector<Point,3> b;
Point C(0, 0);
b.set(0, C);
C.change(0, 10);
b.set(1, C);
C.change(5, 5);
b.set(2, C);
for (int i = 0; i < b.Length(); i++) {
        std::cout << i + 1 << ". bod ma suradnice: " << b.at(i).X() << ", " << b.at(i).Y() << std::endl;
}
```

- Výstup:

```
1. bod ma suradnice: 0, 0
2. bod ma suradnice: 0, 10
3. bod ma suradnice: 5, 5
```

Standard Template Library

- **STL (Standard Template Library)** je súbor štandardných knižníc, ktoré umožňujú prácu s rôznymi dátovými štruktúrami.

- Jadrom STL sú tri základné prvky:

  1. **Dátové štruktúry (Containers)** – STL umožňuje používať viaceré bežné typy dátových štruktúr. Definícia týchto štruktúr je obsahom väčšiny knižníc STL. Ide vlastne o rôzne typy generických tried. Okrem uchovávania dát majú triedy zabezpečenú aj základnú funkčnosť, napr. vrátenie veľkosti štruktúry, konkrétnej hodnoty, prvej a poslednej hodnoty, pridanie a vymazanie hodnoty a podobne.

  2. **Algoritmy** – v knižniciach STL sú definované aj algoritmy, ktoré dokážu napr. triediť dáta, vyhľadávať v nich, rôzne ich meniť a podobne.

  3. **Iterátory** – iterátory sú objekty, ktoré sú v podstate obohatenými smerníkmi. Ich úlohou je umožňovať prístup k jednotlivým prvkom štruktúr.

Standard Template Library

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

https://www.cplusplus.com/reference/stl/

Sekvenčné dátové štruktúry STL

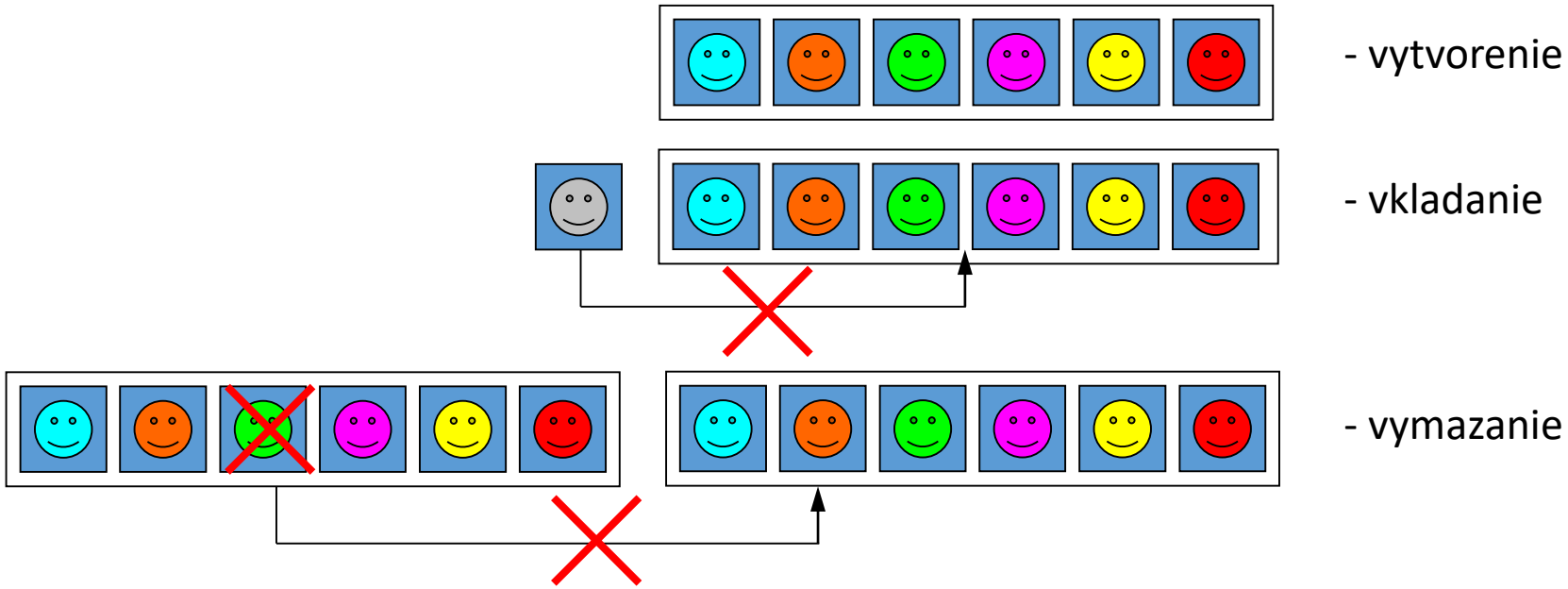
- **Array** (knižnica <array>) C++11

  Je to dátová štruktúra pre statické pole. Obsahujú konkrétny počet prvkov usporiadaných v striktnej postupnosti. Neuchováva žiadne iné údaje okrem prvkov, ktoré obsahuje (dokonca ani veľkosť poľa, ktorá je parametrom šablóny, teda je daná už čase kompilácie). Veľkosť dátovej štruktúry je rovnaká ako pri bežnom poly deklarovanom so syntaxou hranatej zátvorky ([]).

  Použitie: Kedykoľvek, ak chceme použiť štandardné statické pole (napr. ako vstup do funkcie ktorá pracuje s STL dátovými štruktúrami.)

  Výhody: Rýchly prístup k ľubovoľnému prvku poľa

  Nevýhody: Fixná veľkosť



Sekvenčné dátové štruktúry STL

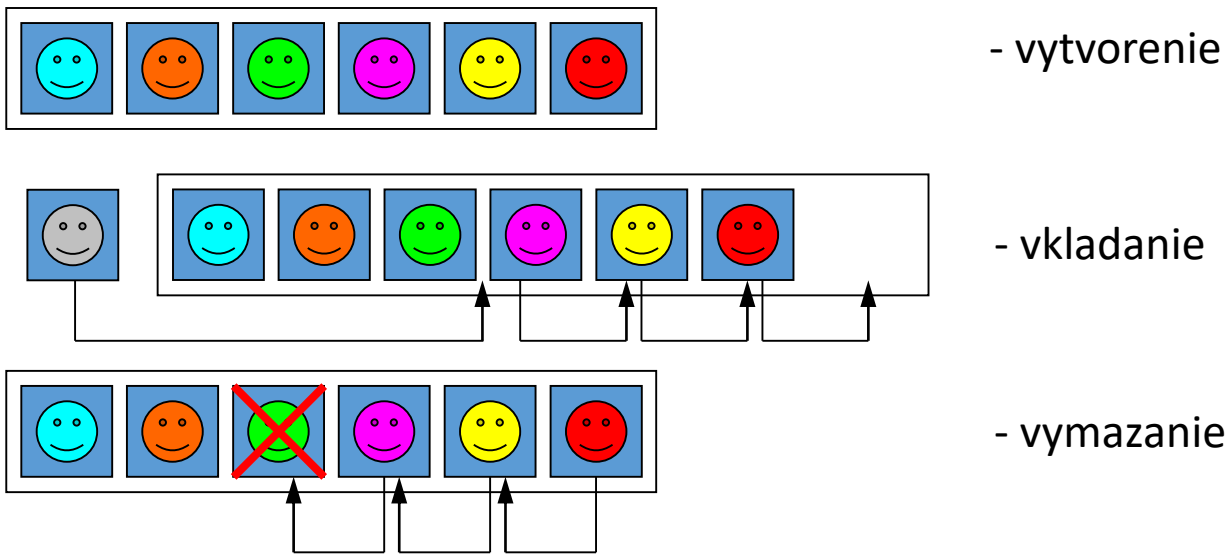
- **Vector** (knižnica <vector>)

  Je to najuniverzálnejšia dátová štruktúra. Ide vlastne o dynamické pole, ktoré môže obsahovať prvky ľubovoľného typu. Rozmery poľa sa môžu meniť podľa potreby, teda kedykoľvek, aj po počiatočnej alokácii. Prístup k prvkom vektora je rovnaký ako k prvkom štandardného poľa.

  Použitie: Kedykoľvek, ak chceme použiť štandardné dynamické pole

  Výhody: Rýchly náhodný prístup

  Nevýhody: Pomalé vkladanie a vymazávanie



Sekvenčné dátové štruktúry STL

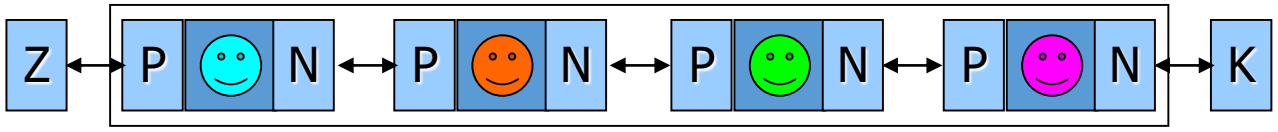
- **List** (knižnica <list>)

  Tu ide o dátovú štruktúru, kde každý prvok okrem hodnoty obsahuje aj **smerník** na nasledujúci a predchádzajúci prvok. Prvky sú takto navzájom previazané, a ku konkrétnemu prvku sa môžeme dostať len postupne od začiatku alebo od konca. Nie je možný priamy náhodný prístup k prvkom.

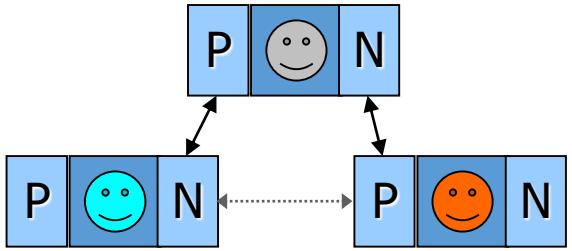
  Použitie: Ak nepotrebujeme k prvkom pristupovať náhodne, ale skôr je dôležité ich poradie, ak vždy spracovávame všetky a ak potrebujeme rýchlo vkladať alebo mazať.

  Výhody: Rýchle vkladanie a mazanie

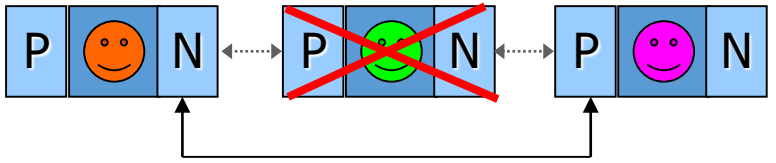
  Nevýhody: Pomalý náhodný prístup



- vytvorenie



- vkladanie



- vymazanie

Sekvenčné dátové štruktúry STL

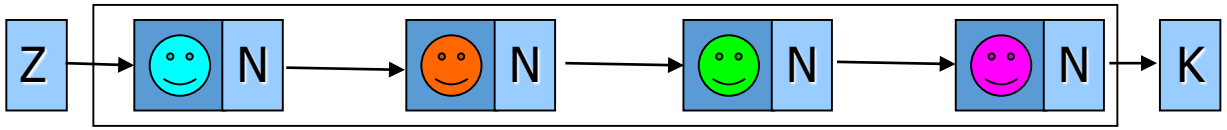
- **Forward list** (knižnica <forward_list>) C++11

  Tu ide o dátovú štruktúru podobnú zoznamu, kde každý prvok okrem hodnoty obsahuje iba **smerník** na nasledujúci prvok (singly-linked lists). Podobne ako pri zozname nie je možný priamy náhodný prístup k prvkom.

  Použitie: Podobné ako v prípade zoznamu, ale iba ak budeme chcieť prechádzať zoznam iba jedným smerom. Vtedy je výpočtovo efektívnejšie použiť forward_list namiesto obyčajného listu.

  Výhody: Rýchle vkladanie a mazanie

  Nevýhody: Pomalý náhodný prístup



- vytvorenie

Sekvenčné dátové štruktúry STL

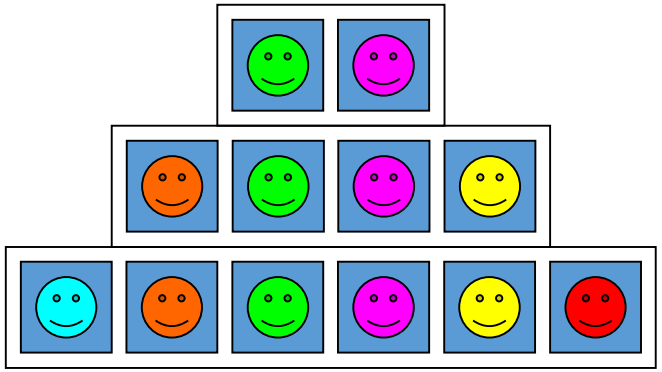
- **Deque** (knižnica <deque>)

  Táto štruktúra sa podobá na vektor, je to teda dynamické pole. Rozdiel je najmä vo vkladaní prvkov. Do vektora sa nové prvky pridávajú len na koniec a ak by sme chceli vložiť prvok inde, musíme celý zvyšok poľa posúvať. Do obojstrannej fronty môžeme prvky pridávať aj na začiatok, bez toho, aby sa celé pole posúvalo. Ďalšie rozdiely sú v spôsobe alokácie a dealokácie pamäte.

  Použitie: Ak potrebujeme použiť dynamické pole, pričom budeme veľakrát vkladať (alebo vyberať) prvky, najmä na koniec a na začiatok.

  Výhody: Rýchle vkladanie na začiatok, rýchly náhodný prístup

  Nevýhody: Pomalé vkladanie a mazanie vo vnútri, časovo náročnejšia dealokácia ako pri vektore



- vytvorenie

- vkladanie

Adaptéry ku sekvenčným dátovým štruktúram STL

- **Adaptéry** rozširujú možnosti využitia základných sekvenčných dátových štruktúr.
- **Queue** (knižnica <queue>)

  Fronta je dátová štruktúra typu **FIFO** (first in, first out). To znamená, že pridávať sa dá len na koniec a odoberať len zo začiatku. Nie je možný priamy náhodný prístup k prvkom.

  Použitie: V prípade, keď potrebujeme postupne spracovávať dáta, ktoré „čakajú" na spracovanie v rade.

- **Stack** (knižnica <stack>)

  Zásobník je štruktúra typu **LIFO** (last in, first out). Teda pridávať sa dá len na koniec a odoberať takisto len z konca. Nie je možný priamy náhodný prístup k prvkom.

  Použitie: Ak potrebujeme odkladať dáta, zatiaľ čo robíme niečo iné, a po ukončení tejto činnosti ich chceme spracovávať, pričom môžeme začať aj posledným uloženým.

- **Priority queue** (knižnica <priority_queue>)

  Zoradená fronta (heap) uchováva dáta vždy zoradené od najväčšieho po najmenší (podľa definovaných pravidiel radenia - priority). Vieme však vždy pristúpiť a vymazať iba posledný prvok – teda ten najmenší (s najmenšou prioritou)

  Použitie: V prípade, že chceme mať zoradené dáta a spracovávať ich postupne.

  Výhody pre všetky adaptéry: Jednoduchá štruktúra

  Nevýhody pre všetky adaptéry : Obmedzená funkčnosť

Asočiačné dátové štruktúry STL

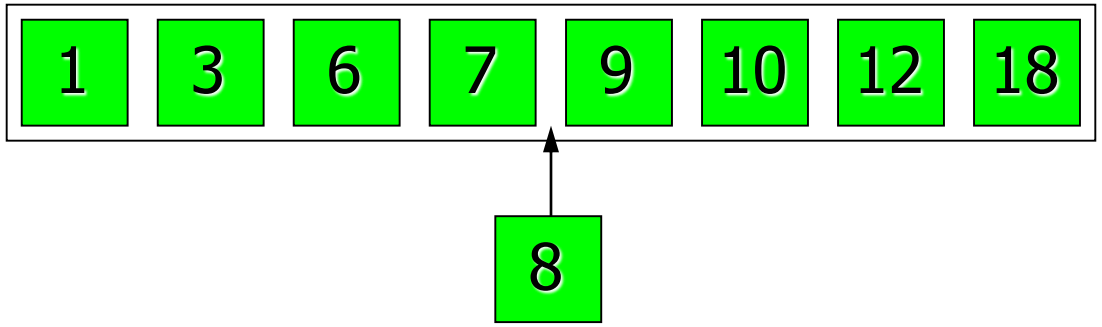
- **Set** (knižnica <set>)

  Množina je usporiadaná dátová štruktúra, ktorá vylučuje duplicitu prvkov. To znamená, že hodnoty sú usporiadané podľa nejakého pravidla, ktoré je možné definovať, a žiadna hodnota sa v štruktúre nevyskytuje dvakrát. Podobne ako zásobník alebo fronta, ani množina neumožňuje priamy náhodný prístup k prvkom. Pri vkladaní sa prvok automaticky zaradí na správne miesto. Ak by sme sa pokúšali vložiť hodnotu, ktorá sa už v množine nachádza, nepodarí sa nám to.

  Použitie: Ak potrebujeme usporiadané hodnoty s vylúčením duplicity

  Výhody: Automatické usporiadanie, vylúčenie duplicity

  Nevýhody: Nemožnosť náhodného prístupu



- **Multiset** (knižnica <set>)

  Multimnožina (mulitset) je množina, ktorá umožňuje duplicitu. Takisto ako množina má svoje prvky usporiadané podľa nejakého pravidla. Použitie, výhody a nevýhody sú podobné ako pri množinách.

Asočiačné dátové štruktúry STL

- **Map** (knižnica <map>)

  Mapa je ďalší utriedený dátový typ, ktorý však namiesto samotných prvkov ukladá dvojice prvkov – tzv. **kľúč** a samotnú hodnotu prvku (napr. poradové číslo a k nemu prislúchajúce meno, meno a k nemu patriace telefónne číslo, reálne číslo a jeho funkčná hodnota atď.). Dvojice v mape sú utriedené podľa kľúča, triediace kritérium sa dá definovať. Pri vkladaní sa nový prvok automaticky zaradí na správne miesto. Mapa nepovoľuje duplicitu kľúčov, hodnoty sa môžu opakovať.

  Použitie: Kedykoľvek potrebujeme uchovať dvojice prvkov, zoradené podľa nejakého kritéria.

  Výhody: Automatické zoradenie hodnôt podľa kľúča, uchovávanie dvojíc, zabraňovanie nejednoznačnosti kľúčov

  Nevýhody: Nemožnosť náhodného prístupu

- **Multimap** (knižnica <map>)

  Multimapa (multimap) je mapa, ktorá umožňuje aj duplicitu kľúčov.

Nezoradene asočiačné dátové štruktúry STL

- **Unordered set** (knižnica <unordered_set>)
- **Unordered multiset** (knižnica <unordered_multiset>)
- **Unordered map** (knižnica <unordered_map>)
- **Unordered multimap** (knižnica <unordered_multimap>)

- Tie isté vlastnosti ako obyčajné asociačné dátové štruktúry, len **nemusia byť zoradené.**

Ostatné dátové štruktúry STL

- **Bitset** (knižnica <bitset>)

  Bitová množina je štruktúra, ktorá umožňuje uchovávať hodnoty 0 a 1. Je to v podstate vektor, ale s tým rozdielom, že jeho veľkosť musí byť daná už pri vytváraní a potom sa už nemôže meniť.
  Použitie: Ak potrebujeme uchovávať vopred známy počet hodnôt 0 a 1

  Výhody: Rýchly náhodný prístup

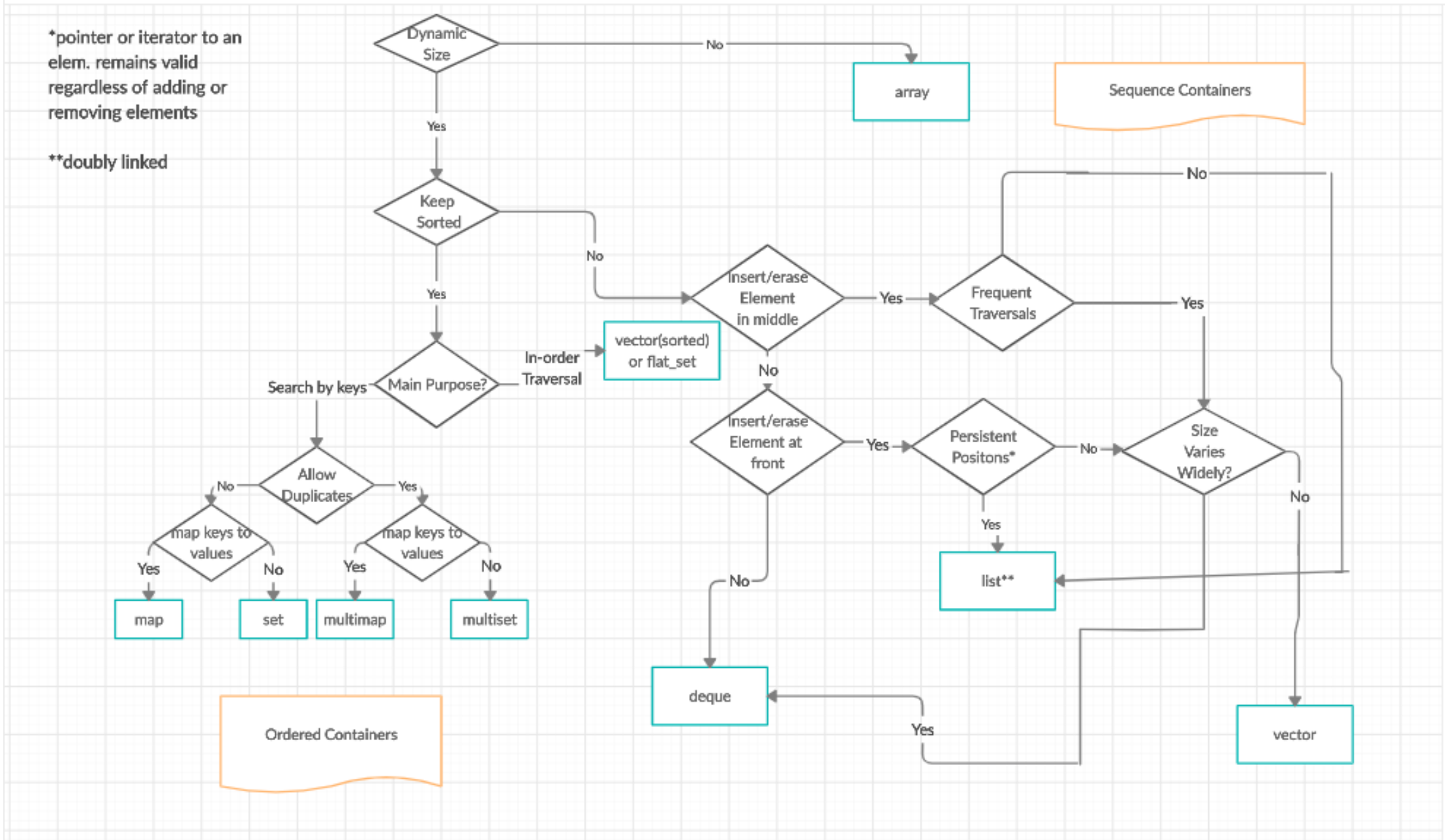
  Nevýhody: Pevná dĺžka

- **Valarray** (knižnica <vallaray>)

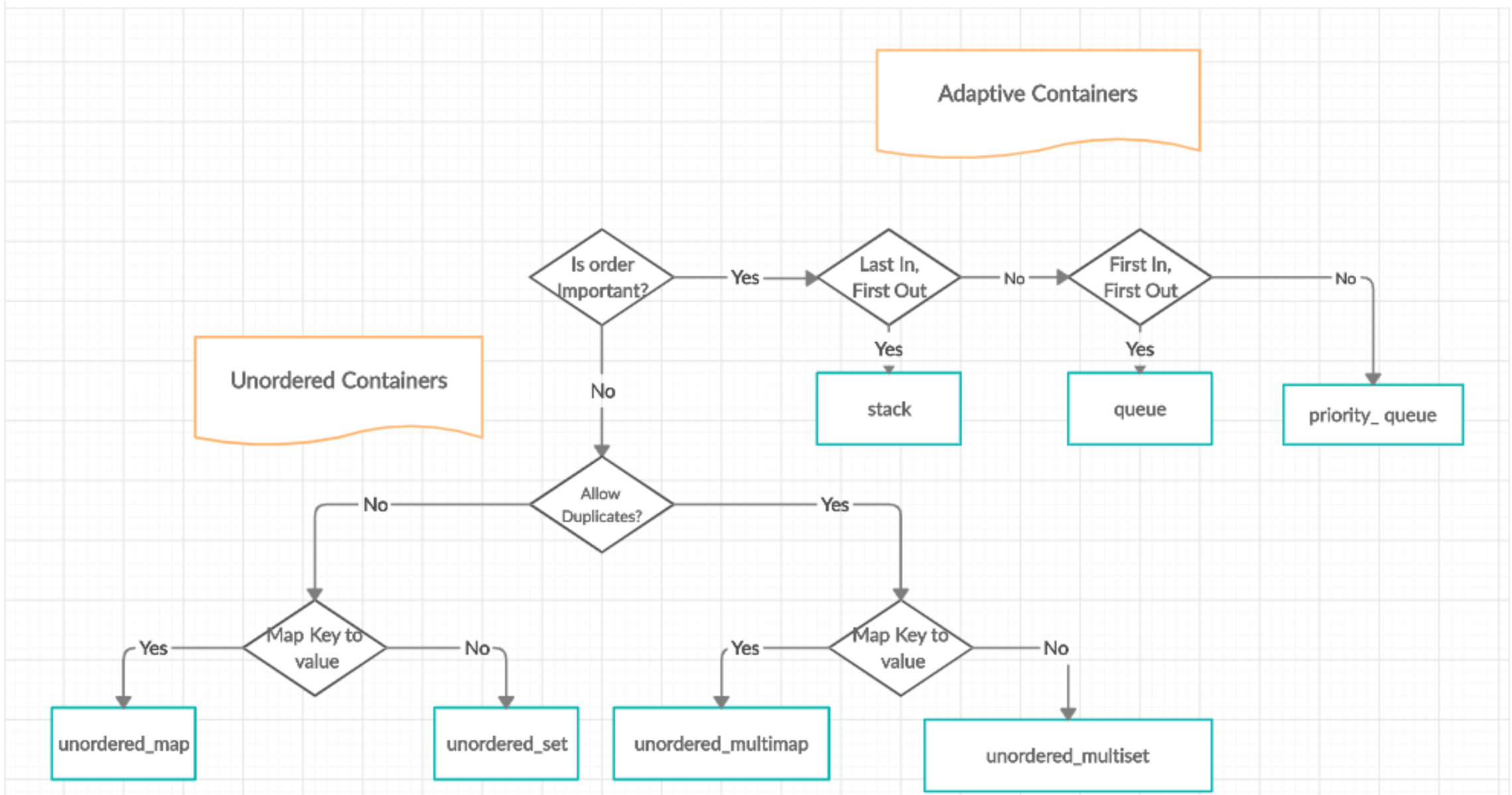
  Objekt valarray je navrhnutý tak, aby mohol obsahovať rôzne dáta nad ktorými sa dajú lahko vykonávať matematické operácie, ktoré sú súčasťou knižnice.

  Výhody: Rýchly náhodný prístup, rýchla aplikácia matematickej operácie na celý objekt

  Nevýhody: Obmedzená funkčnosť



Zdroj: https://www.geeksforgeeks.org/containers-cpp-stl/?ref=lbp



Zdroj: https://www.geeksforgeeks.org/containers-cpp-stl/?ref=lbp

Algoritmy STL

- Súčasťou STL je aj **knižnica <algorithm>**, ktorá obsahuje rôzne algoritmy na prácu s dátovými štruktúrami. Všetky algoritmy sú generické funkcie, fungujú teda pre všetky typy dátových štruktúr, pre ktoré majú zmysel.
- V knižnici <algorithm> je okolo 70 algoritmov (http://www.cplusplus.com/reference/algorithm/), napríklad:

  Hľadanie maxima a minima

  Hľadanie konkrétnej hodnoty v štruktúre

  Triedenie

  Zisťovanie rozsahu štruktúry

  Vkladanie a odstraňovanie prvkov

  Aplikovanie funkcie na prvky

  Hľadanie prieniku a rozdielu

  Odstraňovanie duplicít

  Zmena hodnôt vybraných prvkov

  Výmeny prvkov

  ...

Iterátory

- Ako sme si už hovorili, iterátory sú objekty, ktoré sú rozšírenými smerníkmi na prvky dátových štruktúr. Sú to teda objekty, ktoré v sebe obsahujú smerník na prvok nejakého typu a k tomu ďalšie dáta a metódy (informáciu o type iterátora, preťažené operátory atď.)
- Iterátory slúžia na prístup k prvkom štruktúry. Používajú sa napr. pri prechádzaní štruktúrou, vkladaní a vymazávaní a pod.
- Každá štruktúra ma iný charakter – niektoré štruktúry umožňujú náhodný prístup, iné len postup od začiatku do konca, obojsmerný postup a pod. Preto v STL poznáme niekoľko typov iterátorov.