Krátky úvod do jazyka C++

v1.04

Martin Drozda
2020

Obsah

Úvod

Tento učebný text je určený pre predmet Programovacie techniky, ktorý sa vyučuje v bakalárskom štúdiu v rámci študijného programu Aplikovaná informatika na Fakulte elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.

Predmet Programovacie techniky nadväzuje na predmet Programovanie 2, ktorého cieľom je oboznámenie študentov s programovacím jazykom C. Skriptá sú zamerané na programovací jazyk C++, s dôrazom na tvorbu efektívneho kódu a rôzne dátové štruktúry, ktoré tento jazyk ponúka v rámci Standard template library (STL).

Učebný text ponúka prehľad základných konceptov jazyka C++, a to najmä nasledovných:

- Trieda, objekt, konštruktor, deštruktor.
- Kopírovanie a presúvanie.
- Referencia `&` a univerzálna referencia `&&`.
- Uniformná inicializácia.
- Standard template library (STL), sekvenčné kontajnery, sekvenčné adaptéry, asociatívne kontajnery, neusporiadané asociatívne kontajnery.
- Preťažovanie funkcií, preťažovanie operátorov.
- Výnimky.
- Odvodené triedy, `friend` triedy.
- Šablóny (`template`).
- Chytré smerníky.
- Lambda výrazy, používateľom definované komparátory a hašovacie funkcie.
- `constexpr`, výrazy, ktoré môžu byť vyhodnocované počas kompilácie.
- Rôzne praktické knižnice ako napr. `string`, `iostream`, `fstream`, `sstream`, `algorithm`.

Tento učebný text predpokladá C++ kompilátor s podporou štandardu C++17, pričom pre účel tohto učebného textu bol použitý kompilátor g++ (GNU Compiler Collection) vo verzii 8.3.0 [1].

Spolu s ním je publikovaný aj ďalší učebný text, ktorý vysvetľuje a komentuje najčastejšie chyby, ktorých sa dopúšťajú začínajúci programátori v jazyku C++ [2].

V prípade ambície stať sa odborníkom v oblasti programovania v jazyku C++ je možné odporúčať najmä nasledovnú literatúru:

- Bjarne Stroustrup, The C++ Programming Language, 4th Edition. Addison-Wesley Professional, 2013.
- Scott Meyers, Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition. Addison-Wesley Professional, 2005.
- Scott Meyers, Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014.

V prvom prípade ide o knihu Bjarne Stroustrupa, autora programovacieho jazyka C++. V ostatných dvoch prípadoch ide o knihy Scotta Meyersa, významnej autority v oblasti programovacieho jazyka C++.

V prípade otázok a komentárov je možné kontaktovať autora na emailovej adrese: martin.drozda@stuba.sk a tiež na adrese: Martin Drozda, FEI STU, Ilkovičova 3, 81219 Bratislava.

Bratislava, november 2019 – január 2020.

Poďakovanie

Ďakujem všetkým študentom, ktorí úspešne absolvovali predmet Programovacie techniky.

Terminológia

abstract class	abstraktná trieda
access specifier	špecifikátor prístupu
base class	základná trieda
bucket	kôš
chunk	časť
converting constructor	konverzný konštruktor
comparator	porovnávacia funkcia
copy constructor	kopírovací konštruktor
copy elision	vynechávanie kopírovania
copy semantics	kopírovacia sémantika
default constructor	prednastavený konštruktor
derived class	odvodená trieda
diamond inheritance	diamantové dedenie
double-ended queue	obojsmerná fronta
dynamic binding	dynamické viazanie
generic programming	generické programovanie
getter	prístupová metóda
heap	halda
inheritance	dedenie
move constructor	presúvací konštruktor
move semantics	presúvacia sémantika
overloading	preťažovanie
perfect forwarding	dokonalé preposielanie
pure virtual method	čisto virtuálna metóda
queue	fronta
raw string	primitívny reťazec
reference collapsing	zjednodušenie referencií
return value optimization	optimalizácia návratovej hodnoty
scope	rozsah platnosti
setter	nastavovacia metóda
smart pointer	chytrý smerník
stack	zásobník
stack unwinding	odvinutie zásobníka
stream	prúd
structured binding	štruktúrované viazanie
template	šablóna
trailing return type	koncový typ návratovej hodnoty
type narrowing	zúženie typu
uniform initialization	uniformná inicializácia
universal reference	univerzálna referencia
virtual method	virtuálna metóda

1 Hello, World!

Začneme s tým, že porovnáme krátke kódy v jazykoch C a C++. Ide o štandardný „Hello, World!", ktorý po skompilovaní a spustení vypíše na štandardný výstup zmienenú vetu. Kód v jazyku C je nasledovný:

```
#include <stdio.h>

int main() {
  printf("%s", "Hello,␣World!");

  return 0;
}
```

Porovnateľný kód v jazyku C++ je nasledovný:

```
#include <iostream>

int main() {
  std::cout << "Hello,␣World!";

  return 0;
}
```

Všimnime si, že C++ knižnica `iostream` nie je uvádzaná s príponou .h a pred `cout` sa nachádza špecifikátor štandardného menného priestoru `std::`. V C++ kóde môžeme použiť aj štandardnú C knižnicu, a teda rôzne C a C++ knižnice je možné použiť súčasne. V nasledujúcom príklade použijeme C knižnicu `math.h`, ale v štandardnom mennom priestore. Z tohto dôvodu ju použijeme bez prípony .h a s predponou c, teda `math.h` pretransformujeme na `cmath`:

```
#include <iostream>
#include <cmath> //math.h v std::

int main() {
  std::cout << std::sin(M_PI);

  return 0;
}
```

Na štandardný výstup sa vypíše hodnota $\sin(\pi)$.

2 Menný priestor

Menný priestor zabezpečuje, že nenastane konflikt napr. medzi funkciou zo štandardnej knižnice a s používateľom zadefinovanej knižnice. Pri použití funkcie `printf` z knižnice `cstdio` je stále možné použiť používateľom zadefinovanú funkciu s identickým menom, pričom funkcia zo štandardnej knižnice je pristupovaná pomocou identifikátora štandardného menného priestoru `std::`. Pozrime sa na nasledujúci príklad:

```
1  #include <iostream>
2  #include <cstdio>
3
4  namespace NS {
5    int printf(const char *format, ...) {
6      std::cout << format;
7      return 1;
8    }
9
10   int a = 1;
11 }
12
13 int main() {
14   std::printf("%s", "Hello, World!");
15
16   NS::printf("%s", "Hello, World!");
17
18   std::cout << NS::a;
19
20   return 0;
21 }
```

Na riadku 13 je zavolaná funkcia `printf` zo štandardného menného priestoru a na riadku 15 je zavolaná funkcia z menného priestoru `NS`, ktorý sme si zadefinovali. Na riadku 17 je príklad použitia menného priestoru a premennej, v tomto prípade ide o globálnu premennú typu `int`, ktorá je ale v mennom priestore `NS`.

Použitie menných priestorov rieši tiež problém s funkciami, ktoré majú identické meno, ale sú súčasťou inej entity napr. triedy. Z tohto dôvodu definuje každá trieda v C++ svoj vlastný menný priestor.

Používanie `std::` pred každou funkciou zo štandardnej knižnice sa môže javiť ako zbytočné. Jazyk C++ umožňuje, aby ľubovolný menný priestor bol automaticky prehľadávaný s ohľadom na existenciu definovanej funkcie, premennej alebo iného identifikátora:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    cout << "Hello, World!";
6
7    return 0;
8  }
```

V tomto prípade nie je pred `cout` použitý identifikátor `std::`, pretože na riadku 2 indikujeme, že kompilátor môže prehľadať použité knižnice, v našom prípade knižnicu `iostream`, na existenciu identifikátora `cout`.

V prípade, že si neprajeme používanie `std::` len pred `cout`, je možné nasledovné upresnenie:

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

```
int main() {
  cout << "Hello, World!";

  return 0;
}
```

`using namespace` je možné súčasne použiť aj v prípade viacerých menných priestorov napr.:

```
using namespace std;
using namespace NS;
```

Podobne je možné súčasne použiť:

```
using std::cout;
using std::sin;
```

Druhá možnosť nám poskytuje výrazne väčšiu kontrolu, pretože presne špecifikuje, čo je možné použiť bez upresnenia menného priestoru, a minimalizuje možnosť vzniku konfliktov medzi rôznymi knižnicami, najmä v prípade použitia používateľom definovanej knižnice, ale aj v prípade využitia rôznych knižníc, ktoré sú poskytované tretími stranami.

3 Knižnica iostream

Knižnica `iostream` umožňuje čítanie a zápis zo štandardného vstupu, resp. na štandardný výstup, ako aj zápis do štandardného chybového a logovacieho výstupu.

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

int main() {
  cout << "Hello, World!" << endl;
  cerr << "Error 21" << endl;
  clog << "Parsing completed" << endl;

  int a;
  cin >> a; //čaká na vstup z klávesnice

  return 0;
}
```

`std::cin` je podobné ako C funkcia `scanf` s tým rozdielom, že nie je potrebné špecifikovať vstupný typ (pomocou formátovacieho reťazca). Podobne aj v prípade `std::cout`, `std::cerr` a `std::clog` nie je potrebné špecifikovať výstupný typ na rozdiel od C funkcie `printf`. Vo všetkých prípadoch je

potrebné, aby operácia <<, resp. >> bola pre daný typ definovaná. V prípade používateľom definovaných tried je potrebné tieto operátory dodatočne definovať, čo sa nazýva preťaženie operátora. Týmto problémom sa ďalej zaoberáme v samostatnej časti.

Knižnica `iostream` tiež umožňuje formátovanie. `std::endl` vloží znak EOL (End of line) a zabezpečí vyprázdnenie buffera, a teda okamžitý výpis na štandardný výstup.

```
1  #include <iostream>
2  #include <iomanip>  //std::setprecision
3  #include <cmath>    //M_PI
4
5  int main() {
6    double pi = M_PI;
7
8    std::cout << pi << '\n';                          //3.14159
9    std::cout << std::setprecision(5) << pi << '\n'; //3.1416
10   std::cout << std::setprecision(9) << pi << '\n'; //3.14159265
11   std::cout << std::fixed;
12   std::cout << std::setprecision(5) << pi << '\n'; //3.14159
13   std::cout << std::setprecision(9) << pi << '\n'; //3.141592654
14
15   return 0;
16 }
```

Na riadku 9 sa vypíše hodnota Ludolfovho čísla π s presnosťou 5 číslic a na riadku 10 s presnosťou 9 číslic. Na riadku 12 sa hodnota π vypíše s presnosťou 5 číslic za desatinnou čiarkou a na riadku 13 s presnosťou 9 číslic za desatinnou čiarkou.

```
1  #include <iostream>
2  #include <iomanip>
3
4  int main() {
5    int a = 100;
6
7    std::cout << a << '\n';             //100
8    std::cout << std::hex << a << '\n'; //64
9    std::cout << std::oct << a << '\n'; //144
10
11   return 0;
12 }
```

Na riadku 7 sa vypíše hodnota 100 v desiatkovej číselnej sústave, na riadku 8 v hexadecimálnej číselnej sústave a na riadku 9 v osmičkovej číselnej sústave.

Knižnica `iomanip` obsahuje veľké množstvo iných manipulátorov potrebných pre formátovanie, vzhľadom na ich množstvo je potrebné konzultovať referenciu jazyka C++.

Veľkou výhodou knižnice `iostream` v porovnaní s C knižnicou `stdio.h` je jej typová bezpečnosť, teda schopnosť zachytiť možné chyby už počas kompilácie. Uvažujme nasledujúci príklad pri použití funkcie `printf`:

```
std::printf("%s", 123);
```

V tomto prípade sme pozabudli vložiť reťazec 123 do úvodzoviek, a teda číselná konštanta 123 je interpretovaná ako adresa reťazca. Takýto kód skompiluje, ale nastane nedefinované správanie po spustení (pravdepodobne vo forme pamäťovej chyby). Nedefinované správanie nastane, ak výsledok behu programu je niečo nedefinované štandardom C++.

`std::cout` má zadefinovaný operátor << pre celočíselné konštanty a po spustení kódu sa na štandardný výstup vypíše, čo sme očakávali, teda "123". Pre typovú bezpečnosť sa odporúča využívať C++ knižnicu `iostream` namiesto C knižnice `stdio.h`.

4 Referencia & a univerzálna referencia &&

Referencia `&` je alias na existujúcu premennú príp. na konštantu.

```
int a = 1;
int& b = a;
```

V tomto prípade je premenná `b` alias premennej `a`, pričom použitie jednej z premenných je totožné s použitím druhej premennej.

```
int a = 1;
int& b = a;
b = 2;
```

Po priradení `b = 2` sa obe premenné zhodne rovnajú 2. Adresa premennej `b`, ktorú vráti operátor `&`, je identická s adresou premennej `a`, teda `&a` a `&b` majú rovnaký výsledok.

```
std::cout << &a; //0x7fffbdabc7dc
std::cout << &b; //0x7fffbdabc7dc
```

Využitie referencií je hlavne dôležité v prípade volania funkcií. Referencie je možné pri volaní funkcií nahradiť smerníkmi, ale tento spôsob je syntakticky zložitejší. Referencia neviaže konštanty, v tomto prípade je potrebné použiť `const` referenciu, pričom ale `const` referencia môže byť inicializovaná aj pomocou už existujúcej premennej (vrátane už existujúcej referencie):

```
const int& c = 1;

const int& d = a;
const int& e = b;
```

Pri využití referencie pri volaní funkcie postupujeme nasledovne:

```
void foo(int& a0) {
  ++a0;
}

int main() {
  int a = 100;
  foo(a); //a sa rovná 101
```

```
  return 0;
}
```

V nasledujúcom príklade pri zavolaní funkcie `foo` s konštantou nastane počas kompilácie chyba, pretože referencia neviaže konštanty:

```
1 void foo(int& a0) {
2   ++a0;
3 }
4
5 int main() {
6   foo(100); //chyba
7
8   return 0;
9 }
```

Ak by takáto možnosť bola možná, potom by sme na riadku 2 mohli inkrementovať celočíselnú konštantu 100. Práve pre tento prípad sa zvlášť hodí použitie `const` referencie:

```
void foo(const int& a0) { //const
  //hodnotu a0 nemôžeme zmenit'
}

int main() {
  foo(100); //OK

  int a = 100;
  foo(a);   //OK

  return 0;
}
```

Použitie `const` referencie má tú nevýhodu, že nevieme rozlíšiť, či parametrom funkcie `foo` bola konštanta, alebo premenná. V tomto prípade môžeme využiť univerzálnu referenciu `&&`:

```
void foo(int&& a0) { //foo(100)
  a0 = 10;
}

void foo(int& a0) {  //foo(a)
  ++a0;
}

int main() {
  foo(100); //OK

  int a = 100;
  foo(a);   //OK

  return 0;
}
```

Pre každý z dvoch prípadov môžeme napísať vlastný obslužný kód. Univerzálna referencia teda viaže konštanty a umožňuje ich mutáciu:

```
int&& a = 1;
++a;
```

K ďalšiemu využitiu univerzálnych referencií sa vrátime pri predstavovaní presúvacej (move) sémantiky. Univerzálne referencie viažu dočasné objekty, ktoré pred zavedením štandardu C++11 neefektívne zanikali a nedali sa opätovne využiť.

5 Preťažovanie funkcií

C++ umožňuje rozlišovať medzi funkciami s identickým identifikátorom (názvom), ale s rozdielnymi vstupnými typmi príp. s rozdielnym počtom vstupov:

```
void foo(int a0) {    //int
}

void foo(double b0) { //double
}

void foo(int a0, double b0) { //int, double
}

int main() {
  foo(100); //foo(int)
  foo(3.1); //foo(double)
  foo(100, 3.1); //foo(int, double)

  return 0;
}
```

Pre každý z troch prípadov môžeme napísať špecifický obslužný kód. Preťažovanie funkcií je v C++ zavedené aj na podporu špecifických funkcií v rámci C++ tried tzv. konštruktorov. Trieda môže obsahovať niekoľko konštruktorov, pričom konštruktory majú vždy identický názov, ale často veľmi rozdielne typy vstupných parametrov. Pri preťažovaní funkcií je rozdielny typ návratovej hodnoty nepostačujúci:

```
void foo(int);
int foo(int);
```

V tomto prípade nastane chyba počas kompilácie. Vstupy preťažených funkcií musia mať rozdielne typy.

6 Funkcie s prednastavenými vstupnými hodnotami

Funkcia môže mať prednastavenú hodnotu vstupu, ktorá je využitá v prípade, keď je funkcia volaná bez určenia hodnoty vstupu:

```
#include <iostream>

void foo(int a0 = 1) {
  std::cout << a0;
}

int main() {
  foo();    //1
  foo(100); //100

  return 0;
}
```

Funkcia môže mať viacero prednastavených vstupných parametrov, avšak po vstupe s prednastavenou hodnotou môže nasledovať už len ďalší vstup s prednastavenou hodnotou:

```
void foo(int a0 = 1, int a1 = 2) {} //OK
void foo(int b0, int b1, int b2 = 3) {} //OK
void foo(int a0 = 1, int a1 = 2, int a2, int a3 = 1) {} //chyba
```

Zavolanie funkcie je možné napr. nasledovne:

```
foo();            //foo(1, 2)
foo(10, 10, 10);
```

Pre každý počet vstupných parametrov musí byť zavolanie funkcie jednoznačné, čo v nasledujúcom prípade nie je splnené, pričom s dvoma argumentmi môžeme zavolať obe preťažené funkcie:

```
foo(10, 10);      //chyba
```

Funkcie s prednastavenými hodnotami umožňujú zníženie počtu potrebných deklarácií príp. definícií funkcií, v našom prípade nebolo potrebné zvlášť definovať funkciu `foo()`, pretože táto bezparametrická alternatíva je obsiahnutá v prvej preťaženej funkcii s dvoma prednastavenými hodnotami vstupov.

7 Trieda, objekt, konštruktor, deštruktor

7.1 Trieda, objekt

C štruktúry sú v C++ triedy s verejnými členmi, ktoré je možné zvonku pristupovať. Uvažujme nasledujúci príklad:

```
struct Token {
  int a{1};
};
```

Štruktúra `Token` je trieda s jediným členom, premennou `a`, ktorá je inicializovaná na hodnotu 1. Ide o uniformnú inicializáciu pomocou `{}`, ktorú si presnejšie vysvetlíme neskôr, avšak v našom prípade je ekvivalentná štandardnému spôsobu inicializácie `int a = 1`. Premennú `a` je možné pristupovať zvonka triedy:

```
int main() {
  Token t;
  t.a = 2;

  return 0;
}
```

Všimnime si, že na vytvorenie inštancie štruktúry nie je potrebné:

```
struct Token t;
```

C++ triedy sú deklarované kľúčovým slovom `class` a ich členy sú prednastavené ako privátne, a teda ak vyžadujeme, aby členy triedy boli prístupné zvonka, je potrebné ich označiť ako verejné s použitím kľúčového slova `public`:

```
class Token {
  public:
  int a{1};
};
```

C++ trieda môže obsahovať aj funkcie, ktoré v tomto prípade nazývame metódy:

```
class Token {
  public:
  int a{1};
  int add1() {
    return ++a;
  }
};

int main() {
  Token t;
  t.add1();

  return 0;
}
```

Členy triedy môžu byť tiež označené kľúčovým slovom `private`, čo znamená, že sú privátne, teda zvonka neprístupné:

```
class Token {
  private:
  int a{1};

  public:
  void setA(int a0) {
    a = a0;
```

```
  }
  int getA() {
    return a;
  }
};
```

V tomto prípade obsahuje trieda `Token` metódy `setA` a `getA`, ktoré sú určené na nastavenie, resp. pristúpenie (vrátenie) hodnoty privátnej premennej `a`. Takéto metódy nazývame aj nastavovacie, resp. prístupové (angl. getter a setter).

Inštancie triedy nazývame objekty, ktoré môžu vzniknúť aj dynamicky pomocou `new`:

```
int main() {
  Token* t = new Token;

  delete t;
  return 0;
}
```

Dynamicky vzniknuté objekty je potrebné dealokovať pomocou `delete`. Objekty, ktoré vznikajú dynamicky sú alokované na heap-e, avšak objekty, ktoré vznikajú staticky sú alokované na stack-u (zásobníku) a ich dealokácia nastane po vystúpení z rozsahu platnosti (angl. scope), v ktorom daný objekt vznikol. Dynamicky môžu taktiež vzniknúť inštancie základných typov, vrátane polí:

```
int* a0 = new int;
double* d0 = new double;

int* a1 = new int[100];
double* d1 = new double[100];

delete a0;
delete d0;

delete[] a1;
delete[] d1;
```

Takto vzniknuté objekty je potrebné dealokovať pomocou `delete`, resp. v prípade polí pomocou `delete[]`. Ak pri alokácii pridáme `()` budú numerické základné typy vynulované (inicializované na hodnotu 0):

```
int* a0 = new int();
double* d0 = new double();
```

7.2 Konštruktor

Konštruktor je špeciálna metóda, ktorá po zavolaní „skonštruuje" objekt. Konštruktor môže mať špecifický obslužný kód:

```
class Token {
```

```
  private:
  int a{1};

  public:
  Token() { //konštruktor
    a = 2;
  }
};

int main() {
  Token t;

  return 0;
}
```

V prípade, že trieda neobsahuje žiaden používateľom definovaný konštruktor, potom kompilátor vygeneruje bezparametrický konštruktor. Konštruktor môže byť aj parametrický a trieda môže obsahovať niekoľko konštruktorov:

```
class Token {
  private:
  int a{1};

  public:
  Token() { //konštruktor
    a = 2;
  }
  Token(int a0) { //parametrický konštruktor
    a = a0;
  }
};
```

Konštruktor, ktorý je možné zavolať aj bez parametra sa nazýva prednastavený (angl. default) konštruktor. Členy triedy je možné inicializovať aj pomocou inicializačného zoznamu:

```
class Token {
  private:
  int a{1};
  double d{1.0};

  public:
  Token(int a0, double d0) : a(a0), d(d0) { //inicializačný zoznam
  }
};
```

Výhoda inicializačného zoznamu je, že inicializácia nastane pred spustením tela konštruktora, čo je potrebné pri odvodených triedach, ktoré si predstavíme neskôr. Inicializačný zoznam je jediná možnosť, ako inicializovať `const` členy:

```
class Token {
  private:
  const int a{1}; //const premenná
```

```
  double d{1.0};

  public:
  Token(int a0, double d0) : a(a0), d(d0) { //inicializačný zoznam
  }
};
```

Samozrejme, objekt môže vzniknúť aj dynamicky:

```
int main() {
  Token* t = new Token(10, 3.4);

  delete t;
  return 0;
}
```

7.3 Deštruktor

Deštruktor je zavolaný po `delete`, resp. `delete[]` (aj viacnásobne, pretože pole môže obsahovať veľké množstvo objektov). Deštruktor je tiež zavolaný pri skončení rozsahu platnosti v prípade objektov, ktoré vznikli staticky:

```
class Token {
  private:
  int* a{nullptr};

  public:
  Token() {
    a = new int;
  }

  ~Token() {
    delete a;
  }
};

int main() {
  Token t0;
  Token* t1 = new Token;

  delete t1; //zavolaný deštruktor pre t1

  return 0;
} //zavolaný deštruktor pre t0
```

Deštruktor obsahuje špecifický kód, ktorý je zavolaný pri skončení životnosti objektu. V našom prípade nastane deštrukcia objektu, ktorý vznikol pri zavolaní prednastaveného konštruktora.

7.4 Konverzný konštruktor

V prípade, že nasledovná syntax je povolená, ide o tzv. konverzný konštruktor (angl. converting constructor):

```
Token t = 2;
```

V tomto prípade hľadá kompilátor konštruktor, ktorý umožňuje zavolanie s `int` parametrom (pri zanedbaní možných implicitných konverzií základných numerických typov):

```
class Token {
  private:
  int a{1};

  public:
  Token(int a0) : a(a0) {}
};
```

Konverzné konštruktory často znepriehľadňujú kód, čo je možné ukázať na nasledovnom príklade:

```
1  class TokenA {};
2  class TokenB {
3    public:
4    TokenB(TokenA& t0) {}
5  };
6
7  void foo(TokenB t0) {}
8
9  int main() {
10   TokenA t;
11   foo(t);
12
13   return 0;
14 }
```

Na riadku 7 nastane vznik objektu typu `TokenB` pomocou konverzného konštruktora triedy `TokenB`, teda nastane zavolanie:

```
TokenB t0 = t;
```

kde `t` je premenná typu `TokenA` funkcie `main`. V prípade, že si konverzný konštruktor neprajeme, použijeme označenie `explicit`:

```
explicit TokenB(TokenA& t0) {}
```

Pri konštruktoroch len s jedným argumentom je odporúčané vždy použiť označenie `explicit`, aby sa predišlo neočakávanému vzniku objektu pomocou konverzie.

7.5 this

V prípade, že premenná triedy a metódy (vrátane špeciálnych členov ako konštruktory a deštruktor) má identický názov, kompilátor nemusí vedieť rozoznať o ktorú premennú ide:

```
class Token {
  private:
  int a{1};
```

```
  public:
  Token(int a) {
    a = a; //??
  }
};
```

V tomto prípade je možné použiť `this`, čo je smerník „sám na seba", teda na objekt, ktorý práve používame:

```
class Token {
  private:
  int a{1};

  public:
  Token(int a) {
    this->a = a;
  }
};
```

V prípade `this->a` ide o premennú triedy a v prípade `a` ide o premennú konštruktora. Aby sa predišlo možným chybám pri rozširovaní kódu je odporúčané vždy použiť `this`, aj v prípadoch, keď prekrytie premenných nie je možné:

```
class Token {
  private:
  int a{1};

  public:
  Token(int a0) {
    this->a = a0; //čitateľnejšie s this, ako a = a0
  }
};
```

7.6 = default

V prípade, že trieda obsahuje parametrický konštruktor, nenastane automatické generovanie prednastaveného konštruktora kompilátorom:

```
class Token {
  public:
  Token(int a0) {}
};

int main () {
  Token t; //chyba

  return 0;
}
```

Prednastavený konštruktor je možné pridať, alebo v prípade, že používateľom definovaný prednastavený konštruktor so špecifickým obslužným kódom

nie je potrebný, je možné nechať vygenerovať prázdny prednastavený konštruktor pomocou `= default`:

```
1  class Token {
2    public:
3    Token() = default;
4    Token(int a0) {}
5    ~Token() = default;
6  };
7
8  int main() {
9    Token t; //OK
10
11   return 0;
12 }
```

Na riadku 5 sme dodatočne požiadali kompilátor aj o vygenerovanie (prázdneho) deštruktora, čo v našom prípade nebolo potrebné, ale v niektorých prípadoch to môže byť praktická alternatíva.

7.7 enum trieda

`enum` trieda je špecifický druh triedy, ktorá umožňuje enumeráciu s rozsahom platnosti:

```
enum class Color {
  Red,
  Green,
  Blue
};

int main() {
  Color c = Color::Green;

  return 0;
}
```

`enum` trieda nahrádza `enum` typ jazyka C:

```
enum Color {
  Red,
  Green,
  Blue
};

enum Color2 {
  Red,
  Green,
  Blue
};

int main() {
  Color c = Green; //chyba
```

```
  return 0;
}
```

V tomto prípade nastane pri kompilácii chyba, pretože nie je zrejmé, či bude použitá hodnota Green z `enum` typu `Color` alebo `Color2`. Tento problém je v C++ vyriešený zavedením rozsahu platnosti, v našom prípade špecifikovaním menného priestoru `Color::`.

8 Kopírovací konštruktor a operátor priradenia =

8.1 Kopírovací konštruktor

V nasledujúcom príklade nastane kopírovanie objektu `t0` do objektu `t1`:

```
class Token {};

int main() {
  Token t0;
  Token t1 = t0; //kopírovanie

  return 0;
}
```

Spôsob, akým kopírovanie nastane, je možné určiť pomocou kopírovacieho konštruktora:

```
class Token {
  private:
  int a{1};

  public:
  Token() = default;
  Token(const Token& t0) { //kopírovací konštruktor
    this->a = t0.a;
  }
};

int main() {
  Token t0;
  Token t1 = t0;

  return 0;
}
```

Všimnime si, že kopírovací konštruktor vyžaduje, aby jeho argument bol `const Token&`, teda `const` referencia. V prípade, že neexistuje používateľom definovaný kopírovací konštruktor, kompilátor automaticky vygeneruje kopírovací konštruktor, ktorý prekopíruje všetky premenné triedy. V prípade nášho príkladu by automaticky generovaný kopírovací konštruktor prekopíroval hodnotu premennej `a` v objekte `t0` do objektu `t1`.

8.2 Operátor priradenia =

V nasledujúcom príklade nastane priradenie objektu `t0`:

```
class Token {};

int main() {
  Token t0, t1;
  t1 = t0; //priradenie

  return 0;
}
```

Na rozdiel od kopírovania nebude využitý kopírovací konštruktor. V tomto prípade je potrebné zadefinovanie operátora `=`. Takéto zadefinovanie sa tiež nazýva preťaženie operátora `=`. Uvažujme nasledujúci príklad:

```
class Token {
  private:
  int a{1};

  public:
  Token() = default;
  Token(const Token& t0) {            //kopírovací konštruktor
    this->a = t0.a;
  }
  Token& operator=(const Token& t0) { //pretížený operátor =
    this->a = t0.a;
    return *this;
  }
};

int main() {
  Token t0, t1;
  t1 = t0;

  return 0;
}
```

Existencia kopírovacieho konštruktora a preťaženého operátora `=` nám umožňuje riešiť každú situáciu iným obslužným kódom. Všimnime si, že v prípade použitia kopírovacieho konštruktora objekt `t1` ešte neexistuje, a v prípade preťaženého operátora `=` už objekt `t1` existuje.

Ak používateľom definovaný prednastavený konštruktor, kopírovací konštruktor, preťažený operátor `=`, alebo deštruktor neexistuje, je ľubovolný z nich automaticky generovaný kompilátorom. Prednastavený konštruktor je automaticky generovaný len v prípade, že neexistuje žiaden parametrický konštruktor, vrátane kopírovacieho konštruktora.

8.3 Hlboká kópia

V prípade, že trieda obsahuje dátovú štruktúru, ako napr. pole, je potrebné prvky tohto poľa prekopírovať. Uvažujme nasledujúci príklad:

```
class Token {
private:
  int* p{nullptr};

public:
  Token() {
    p = new int[1000];
  }
  ~Token() {
    delete[] p;
  }

};

int main() {
  Token t0;
  Token t1 = t0;

  return 0;
}
```

V tomto prípade je pri kopírovaní prekopírovaná hodnota premennej `p`, teda adresa poľa, ale nie pole samotné. Pre prekopírovanie všetkých prvkov poľa je potrebný kopírovací konštruktor:

```
#include <algorithm>

class Token {
private:
  int* p{nullptr};

public:
  Token() {
    p = new int[1000];
  }
  Token(const Token& t0) {
    p = new int[1000];
    std::copy(t0.p, t0.p + 1000, p); //kopírovanie poľa
  }
  ~Token() {
    delete[] p;
  }

};

int main() {
  Token t0;
  Token t1 = t0;

  return 0;
}
```

Pre prekopírovanie prvkov poľa sme využili `std::copy` z knižnice `algorithm`. Podobne je potrebné ošetriť aj operátor `=`.

8.4 = delete

V prípade, že si neželáme, aby bolo možné kopírovanie, môžeme označiť kopírovací konštruktor, resp. operátor `=` ako `= delete`:

```
class Token {
  public:
  Token() = default;
  Token(const Token&) = delete;
  Token& operator=(const Token&) = delete;
};

int main() {
  Token t0;
  Token t1 = t0; //chyba

  return 0;
}
```

`= delete` je v tomto prípade potrebné vnímať ako nástroj na vyjadrenie, že zámer programátora je jedinečnosť danej inštancie. Vo všeobecnosti môže byť, avšak `= delete` použité pre označenie ľubovolnej metódy, ktorú už neplánujeme používať:

```
class Token {
  public:
  void foo() = delete;
  void foo2() {}
};
```

V tomto prípade nie je volanie metódy `foo` možné, a to napr. z dôvodu, že bola nahradená metódou `foo2`.

8.5 Vynechávanie kopírovania

Kopírovanie, teda použitie kopírovacieho konštruktora, je častejšie, ako sa na prvý pohľad môže zdať. Z tohto dôvodu bola zavedená optimalizácia kódu s cieľom zamedziť nepotrebnému kopírovaniu. Uvažujme nasledujúci príklad:

```
#include <iostream>

class Token {
public:
  Token() {
    std::cout << "Token()";
  }
  Token(const Token& t) {
    std::cout << "Token(const␣Token&)";
  }
};

Token foo() {
  return Token(); //Token()
}
```

```
int main() {
  Token t = foo();

  return 0;
}
```

V tomto prípade je zavolaný prednastavený konštruktor, ale nie je zavolaný kopírovací konštruktor. Objekt typu `Token`, ktorý vznikne vo funkcii `foo` nie je prekopírovaný do návratovej hodnoty tejto funkcie, a ani do premennej `t`, pričom by sa v oboch prípadoch mal zavolať kopírovací konštruktor. Namiesto toho je využitý už existujúci objekt, ktorý by ináč zanikol pri vystúpení z rozsahu platnosti. Takáto optimalizácia sa nazýva vynechávanie kopírovania (angl. copy elision), a je druhom optimalizácie návratovej hodnoty (angl. return value optimization).

Overiť, že naozaj nastalo vynechanie kopírovania môžeme pomocou prepínača `-fno-elide-constructors`, ktorý pre kompilátor g++ zakáže tento druh optimalizácie. Po použití tohoto prepínača nastane dvojnásobné zavolanie kopírovacieho konštruktora.

Vynechanie kopírovania nastane aj v prípade, že ide o lokálnu premennú, ktorá by podobne ako dočasný objekt v predchádzajúcom prípade po vystúpení z rozsahu platnosti zanikla:

```
Token foo() {
  Token t; //Token()

  return t;
}
```

Takýto spôsob využitia už existujúcej lokálnej premennej, ktorá vznikla na stack-u, je často najefektívnejší spôsob, ako poslať hodnotu lokálnej premennej do volajúcej funkcie.

9 Uniformná inicializácia, automatické odvodenie typu

9.1 Uniformná inicializácia

Uniformná inicializácia zavádza jednotný spôsob inicializácie pre premenné, polia, objekty atď.:

```
int a{};              //prednastavená inicializácia premennej
                      //na hodnotu 0
int b{1};             //inicializácia premennej
int p[]{0, 1, 2, 3};  //inicializácia pol'a

Token t0{};           //prednastavená inicializácia objektu t0
Token t1{1};          //inicializácia objektu t1
Token t2{0, 1, 2, 3}; //inicializácia objektu t2 pomocou
```

```
                        incializačného zoznamu
Token t3{t2};          //kopírovací konštruktor

int* ptr{nullptr};     //inicializácia premennej s nullptr
```

`nullptr` je nulový smerník, ktorý má typ `std::nullptr_t`, a ktorý je zavedený, aby sa odstránila nejednoznačnosť volaní funkcií pri použití `NULL`:

```
void foo(int* a) {}
void foo(int a) {}

int main() {
  foo(nullptr); //zavolá foo(int*)
  foo(NULL);    //chyba

  return 0;
}
```

Cieľom uniformnej inicializácie je zamedzenie zavádzajúcej syntaxe ako napr.:

```
int a();
int b(1);
```

V prvom prípade ide o deklaráciu funkcie `a` a v druhom prípade ide o inicializáciu premennej `b` na hodnotu 1. Uniformná inicializácia tiež zamedzuje nechcenému zúženiu typu (angl. type narrowing):

```
int a{3.4}; //chyba
```

Takýto spôsob inicializácie nebude skompilovaný, pretože dochádza k implicitnej konverzii typu `double` na typ `int`.

Cieľ uniformnej inicializácie je tiež upresnenie zámeru programátora. Uvažujme nasledujúci príklad:

```
std::vector<int> v(10);
```

Je v tomto prípade C++ vektor `v` inicializovaný hodnotou 10, alebo je počiatočná veľkosť vektora 10? Správna odpoveď je, vektor má počiatočnú veľkosť 10. V prípade inicializácie vektora môžeme použiť uniformnú inicializáciu:

```
std::vector<int> v{0, 1, 2};
```

Rozdiel medzi `()` a `{}` je sémantický, použitie `()` napovedá nastavenie parametra pomocou konštruktora a `{}` napovedá inicializáciu so zoznamom hodnôt t. j. s použitím `std::initializer_list`.

9.2 Most vexing parse

Uniformná inicializácia je tiež zavedená, aby sa zamedzilo rôznym prípadom nezrozumiteľnej syntaxe ako napr.:

```
class Token0 {};

class Token1 {
```

```
public:
  Token1(const Token0& t) {};
  void foo() {}
};

int main() {
  Token1 t(Token0()); //??
  t.foo(); //chyba

  return 0;
}
```

Ale čo presne znamená zápis `Token1 t(Token0())`? Ide o premennú, ktorá je inicializovaná inštanciou `Token0`, alebo ide o funkciu, ktorej parameter je smerník na funkciu bez vstupného parametra, ktorá vracia `Token0`? Po zavolaní metódy `foo` nastane kompilačná chyba, a je zrejmé, že `t` nie je premenná typu `Token1`. Kompilátor tento zápis interpretuje ako `Token1(Token0 (*)())`.

„Most vexing parse" [3] je možné eliminovať pomocou uniformnej inicializácie:

```
Token1 t{Token0{}};
```

Kód po tejto zmene skompiluje a metóda `foo` je zavolaná.

9.3 Automatické odvodenie typu

Jazyk C++ umožňuje automatické odvodenie typu pomocou `auto` a `decltype`:

```
auto a = 1;       //a má typ int
decltype(a) b;    //b má rovnaký typ ako a, teda int
auto c = &b;      //c má typ int*
const auto d = 1; //d má typ const int
auto& e0 = a;     //e0 má typ int&
auto&& e1 = 1;    //e1 má typ int&&

auto a0;          //chyba, typ sa nedá odvodit'
```

V poslednom prípade sa typ nedá odvodiť. `decltype` je potrebný na odvodenie typu výrazu, v našom prípade na zistenie typu premennej `a`, pričom premenná `b` má typ `decltype(a)`, teda `int`. Inými slovami, premenná `b` má taký istý typ ako premenná `a`. V niektorých prípadoch môže byť typ odvodený pomocou `auto` zložitý, a preto je potrebný mechanizmus, ktorý umožňuje, aby premenná mala ten istý typ, ako typ odvodený pomocou `auto`. Automatické odvodenie typu pomocou `auto` je tiež možné použiť pre odvodenie návratového typu funkcie:

```
auto foo() {
  return 1;
}
```

Návratový typ funkcie `foo` je v tomto prípade odvodený ako `int`. Uvažujme nasledujúci príklad:

```
const int& foo() {};
int&& foo2() {};

int main() {
  decltype(foo()) a = 1;  //a má typ const int&
  decltype(foo2()) b = 2; //b má typ int&&

  return 0;
}
```

V tomto prípade odvodíme pomocou `decltype` typ premennej `a` a premennej `b`, ktoré sú zhodné s návratovými typmi funkcie `foo`, resp. `foo2`.

Automatické odvodenie typu je samozrejme použiteľné aj s uniformnou inicializáciou:

```
auto a{1};        //a má typ int
auto b{0, 1};     //b má typ std::initializer_list<int>

std::string s{"Hello, World!"};
auto pos = s.find("World");  //pos má typ size_t
auto size = s.size();        //size má typ size_t
```

10 Preťažovanie operátorov

10.1 Preťaženie aritmetického operátora +

V prípade, že potrebujeme spočítať dva objekty, je potrebné preťažiť operátor `+`, aby vykonával to, čo je žiadané:

```
1  class Token {
2    public:
3    int a{1};
4    Token operator+(const Token& t0) { //pretˊažený operátor +
5      Token t;
6      t.a = this->a + t0.a;
7      return t;
8    }
9  };
10
11 int main() {
12   Token t0, t1, t2;
13   t2 = t0 + t1;
14
15   return 0;
16 }
```

Všimnime si, že v prípade preťaženého operátora `+` je na riadku 7 vrátený novovzniknutý objekt, t. j. súčet dvoch objektov typu `Token` je nový objekt typu `Token`.

10.2 Preťaženie operátorov porovnania

V prípade, že potrebujeme použiť niektorý triediaci algoritmus zo štandardnej knižnice, napr. `std::sort` alebo `std::stable_sort`, je potrebné preťažiť operátor <. Takéto preťažovanie je zvyčajne implementované mimo triedu ako samostatná funkcia:

```
1  class Token {
2    public:
3    int a{1};
4  };
5
6  inline bool operator<(const Token& t0, const Token& t1) {
7    return t0.a < t1.a;
8  }
9
10 int main() {
11   Token t0, t1;
12   t0 < t1; //false
13
14   return 0;
15 }
```

V tomto prípade sú objekty `t1` a `t2` porovnávané na základe hodnoty premennej `a`. V prípade, že potrebujeme dodefinovať ďalšie operácie porovnania, je to možné pomocou už definovaného operátora <:

```
class Token {
  public:
  int a{1};
};

inline bool operator<(const Token& t0, const Token& t1) {
  return t0.a < t1.a;
}

inline bool operator>(const Token& t0, const Token& t1) {
  return t1.a < t0.a;
}

inline bool operator<=(const Token& t0, const Token& t1) {
  return !(t0.a > t1.a);
}

inline bool operator>=(const Token& t0, const Token& t1) {
  return !(t0.a < t1.a);
}

int main() {
  Token t0, t1;
  t0 <= t1; //true

  return 0;
}
```

Označením `inline` indikuje používateľ svoje želanie, aby funkcia nebola volaná, ale aby jej skompilovaný kód bol priamo vložený na mieste jeho použitia. Toto môže mať za následok zrýchlenie behu skompilovaného kódu, ale zároveň aj predĺženie samotného skompilovaného kódu, pretože na mieste každého použitia funkcie je vložené skompilované telo funkcie. Kompilátor želanie vyjadrené pomocou `inline` môže, ale nemusí akceptovať.

10.3 Preťaženie operátora vkladania <<

Predpokladajme, že si želáme preťaženie operátora vkladania << tak, aby sme priamo mohli použiť zápis:

```
Token t0;
std::cout << t0;
```

Preťaženie operátora << je možné implementovať nasledovne:

```
1  #include <iostream>
2  #include <ostream>
3
4  class Token {
5    public:
6    int a{1};
7  };
8
9  inline std::ostream& operator<<(std::ostream& stream,
10                                  const Token& t0) {
11   stream << t0.a;
12   return stream;
13 }
14
15 int main() {
16   Token t0;
17   std::cout << t0; //1
18
19   return 0;
20 }
```

Na riadku 17 sa na štandardný výstup vypíše 1, teda hodnota premennej `a` v objekte `t0`.

Pre úplnosť dodáme, že je možné preťažiť operátory priradenia, operátory prefixovej a postfixovej inkrementácie, resp. dekrementácie, aritmetické operátory, logické operátory, operátory porovnania a mnohé ďalšie operátory ako napr. operátor dereferencovania `*`, operátor adresy `&`, operátor prístupu `->`, operátory << a >>.

10.4 friend funkcia

V niektorých prípadoch je praktické označiť preťažený operátor ako `friend`. Takéto funkcie majú prístup k privátnym, ako aj k chráneným členom tried. Uvažujme nasledujúci príklad:

```
1  #include <iostream>
2  #include <ostream>
3
4  class Token {
5    friend std::ostream& operator<<(std::ostream&, const Token&);
6    private:
7    int a{1};
8  };
9
10 inline std::ostream& operator<<(std::ostream& stream,
11                                  const Token& t0) {
12   stream << t0.a;
13   return stream;
14 }
15
16 int main() {
17   Token t0;
18   std::cout << t0;
19
20   return 0;
21 }
```

Na riadku 5 je preťažený operátor << deklarovaný ako `friend`. To znamená, že má prístup k privátnej premennej `a`. Takéto riešenie je potrebné vzhľadom na to, že preťažený operátor je definovaný mimo triedu `Token`. Ďalšia možnosť, ako sprístupniť privátnu premennú `a`, je verejná prístupová funkcia (getter):

```
class Token {
  private:
  int a{1};

  public:
  int getA() const { //const metóda
    return a;
  }
};
```

Preťaženie je potom implementované nasledovne:

```
inline std::ostream& operator<<(std::ostream& stream,
                                const Token& t0) {

  stream << t0.getA();
  return stream;
}
```

Zápis s `friend` sa javí ako jednoduchší, avšak zápis pomocou verejnej metódy `getA` ponúka jednoznačné verejné rozhranie a môže byť výhodné, ak sa v budúcnosti rozhodneme premenovať premennú `a`. Všimnime si, že metóda `getA` je označená ako `const`, čo znamená, že nemôže meniť (mutovať) hodnoty členov triedy. V tomto prípade je to potrebné, pretože túto metódu voláme na `const` referencii `t0`, na ktorej môžeme volať len metódy, ktoré nemenia hodnoty členov triedy.

11 Knižnica string

Knižnica `std::string` umožňuje prácu s reťazcami a na rozdiel od C knižnice `string.h` nevyžaduje explicitnú alokáciu pamäte pre reťazce, alokácia pamäte sa vykoná automaticky pri vzniku reťazca, príp. pri jeho zväčšení:

```
std::string s0{"Hello, "};
std::string s1{"World!"};
std::string s2{s0 + s1}; //s2 = "Hello, World!"
```

Táto knižnica umožňuje základné operácie ako napr. mazanie a vkladanie reťazcov, operácie s podreťazcami, hľadanie a porovnávanie:

```
s2.erase(5, 1); //s2 = "Hello World!"
s2.insert(5, "..."); //s2 = "Hello... World!"

std::string s3{s2.substr(0, 5)};  //s3 = "Hello"

size_t pos0 = s3.find("l");  //pos0 = 2
size_t pos1 = s3.rfind("l"); //pos1 = 3
size_t pos2 = s3.rfind("a"); //pos2 = std::string::npos

s2 == s3; //false

size_t size = s3.size(); //size = 5
```

Metóda `std::string::erase(k, n)` zmaže `n` znakov počínajúc pozíciou `k`, pričom prvý znak v reťazci má pozíciu 0. Metóda `std::string::insert(k, str)` vloží reťazec `str` od pozície `k`. Metóda `std::string::substr(k, n)` vráti podreťazec o dĺžke `n`, ktorý začínal na pozícii `k`. Metóda `std::string::find(str)` vráti pozíciu reťazca `str` pri prechádzaní zľava doprava a metóda `std::string::rfind(str)` vráti pozíciu reťazca `str` pri prechádzaní sprava doľava. V prípade, že reťazec `str` nebolo možné nájsť, je vrátená konštanta `std::string::npos`, ktorá sa rovná max. hodnote `unsigned int`. Porovnanie `s2` a `s3` sa vyhodnotí ako `false`, pretože tieto dva reťazce nie sú identické. Metóda `std::string::size` vráti počet znakov v reťazci. Knižnica `string` taktiež umožňuje iteráciu:

```
for(auto c : s3) {
   std::cout << c; //Hello
}
```

Kontajner knižnice `string` je špecifický tým, že podporuje tzv. Small string optimization (SSO). V prípade, že dĺžka reťazca je max. 16 znakov, je takýto reťazec uložený v premennej triedy (napr. v `std::array<char, 16>`), v opačnom prípade je potrebný pamäťový priestor dynamicky alokovaný na heap-e. Ak ide o krátky" reťazec je na jeho uloženie využitá pamäť ináč potrebná na uloženie adresy dynamicky alokovaného kontajnera a jeho aktuálnej kapacity.

Pre kompatibilitu s C knižnicou `string.h` je pomocou metódy `std::string::c_str` možné získať adresu vnútorného poľa, kde je uložený

reťazec:

```
char[10] s;
std::strcpy(s, s3.c_str());
```

Funkcia `std::strcpy` prekopíruje obsah vnútorného poľa C++ reťazca do reťazca `s`. Metóda `std::string::c_str` vráti `const char*`. Vnútorné pole C++ reťazca teda nie je povolené meniť.

Jazyk C++ taktiež podporuje tzv. primitívny reťazec (angl. raw string), ktorý umožňuje, aby bol reťazec vypísaný vrátane znakov `\n\t\r`, a teda nie je potrebné písať `\\n\\t\\r`:

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

int main() {
  std::string s0 = R"(Hello\tWorld!\n)";
  std::string s1 = "Hello\\tWorld!\\n";

  std::cout << s0; //Hello\tWorld!\n
  std::cout << s1; //Hello\tWorld!\n

  return 0;
}
```

Všimnime si, že primitívny reťazec má syntax `R"()"`.

12 Standard template library (STL)

Štandardná knižnica C++ implementuje veľké množstvo praktických dátových štruktúr, ktoré sú súčasťou STL, a ktoré je možné rozdeliť nasledovne:

1. Sekvenčné kontajnery
   - `std::vector`
   - `std::array`
   - `std::deque`
   - `std::list`
   - `std::forward_list`
2. Sekvenčné adaptéry
   - `std::stack`
   - `std::queue`
   - `std::priority_queue`
3. Asociatívne kontajnery

- `std::map`
- `std::multimap`
- `std::set`
- `std::multiset`

4. Neusporiadané asociatívne kontajnery

   - `std::unordered_map`
   - `std::unorered_multimap`
   - `std::unordered_set`
   - `std::unordered_multiset`

Objekty v týchto dátových štruktúrach sú vkladané do kontajnerov, ktoré sa dynamicky vytvárajú na heap-e, a to aj v prípade, že samotná inštancia dátovej štruktúry môže vzniknúť na stack-u. Kontajner sa dá vnímať ako interné úložisko objektov, ku ktorému je možné pristupovať pomocou verejných metód.

Sekvenčné kontajnery implementujú vlastný spôsob alokácie pamäte pre objekty. Sekvenčné adaptéry využívajú už implementovaný sekvenčný kontajner na ukladanie objektov, štandardne sa pre tento účel využíva `std::deque`, teda obojstranná fronta.

Sekvenčné kontajnery ukladajú objekty do poľa, sekvenčne za sebou. Asociatívne kontajnery ukladajú objekty do vyváženého binárneho stromu, často sa pre tento účel využíva červeno-čierny strom [4]. Na využitie vyváženého binárneho stromu sú objekty triedené podľa zvoleného kritéria, pričom takéto kritérium je možné definovať pomocou porovnávacej funkcie (angl. comparator).

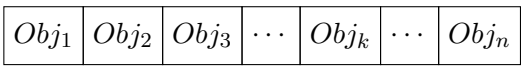
V neusporiadaných asociatívnych kontajneroch nie sú objekty žiadnym spôsobom usporiadané (triedené), objekty sa ukladajú do košov (angl. bucket), pričom členstvo v koši je určené hašovaciou funkciou. Takýto kôš má určenú max. kapacitu a v prípade dosiahnutia jeho kapacity je potrebné vytvoriť väčšie množstvo košov, následne nastane opätovný výpočet členstva objektov pomocou hašovacej funkcie.

Do STL kontajnerov nie je možné vkladať referencie, aby nevznikli neželané vedľajšie efekty (ang. side effect), kde pri zmene hodnoty referencie by nastala aj zmena v kontajneri. Toto obmedzenie je možné obísť pomocou `std::reference_wrapper`, avšak ide o pokročilú techniku jazyka C++, ktorou sa ďalej nezaoberáme.

12.1 std::vector

`std::vector` implementuje dynamické pole, ktorého veľkosť sa zväčšuje podľa počtu vkladaných objektov. Takéto dynamické pole je charakterizované počtom objektov, ktoré obsahuje, a max. počtom objektov, ktoré sa do

neho dajú vložiť bez potreby zväčšenia jeho kapacity. Na Obr. 1 je zobrazený vektor s veľkosťou k a kapacitou n.



Obr. 1. std::vector

Do C++ vektora vkladáme objekt na jeho koniec pomocou metódy `push_back` a vyberáme z jeho konca objekty pomocou metódy `pop_back`:

```
std::vector<int> v; //int

v.push_back(10); //vlož 10
v.push_back(20); //vlož 20
std::cout << v.size(); //2

v.pop_back();    //vyber 20
std::cout << v.size(); //1
```

V našom prípade ide o C++ vektor, do ktorého vkladáme objekty typu `int`. Na rozdiel od C poľa, v prípade C++ vektora nie je možné prekročiť jeho alokovanú kapacitu, pretože táto sa automaticky zvyšuje, avšak podobne ako C pole, vyžaduje C++ vektor spojitú časť pamäte pre alokáciu. V prípade, že spojitá voľná pamäť s danou veľkosťou nie je dostupná, nie je možné ďalej zväčšovať jeho kapacitu príp. jeho použitie vôbec nie je možné.

Ak potrebujeme prechádzať (iterovať) cez všetky objekty v kontajneri, môžeme využiť iterátory t. j. objekty, ktoré ukazujú na začiatok a koniec kontajnera:

```
std::vector<int> v;
std::vector<int>::iterator start = v.begin();
std::vector<int>::iterator end = v.end();

while(start != end) {
  std::cout << *start;

  ++start;
}
```

Ak nepotrebujeme modifikovať žiaden objekt, môžeme využiť `const` iterátory:

```
std::vector<int> v;
std::vector<int>::const_iterator start = v.cbegin();
std::vector<int>::const_iterator end = v.cend();

while(start != end) {
  std::cout << *start;

  ++start;
}
```

V tomto prípade je po dereferencovaní iterátora vrátená `const` referencia na objekt. Vzhľadom na to, že zápis pomocou iterátorov je pomerne neprehľadný, bol v C++11 zavedený tzv. „range-based for", teda `for` slučka s rozsahom:

```
std::vector<int> v;

for(int& i : v) {
  std::cout << i;
}
```

Takýto zápis predpokladá, že je možné zavolať metódy `begin` a `end`. V prípade, že potrebujeme prechádzať objektmi opačným smerom, teda zozadu, je potrebné použiť metódy `rbegin` a `rend`, resp. ich `const` alternatívy `crbegin` a `crend`.

Do vektora je možné tiež vkladať objekty na ľubovolné miesto a mazať objekty z ľubovolného miesta pomocou metód `insert`, resp. `erase`:

```
1  std::vector<int> v;
2  v.push_back(10);
3
4  std::vector<int>::iterator start = v.begin();
5  v.insert(start, 20);
6
7  std::cout << v.front(); //20
8  std::cout << v.back();  //10
9
10 start = v.begin();
11 v.erase(start + 1);
12 std::cout << v.back(); //20
```

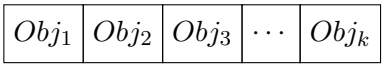
Na riadku 10 je potrebné prenastavenie iterátora, pretože metódy ako `push_back`, a v našom prípade `insert`, môžu spôsobiť jeho neplatnosť. Prvky vo vektore je možné pristupovať a meniť pomocou metódy `at`:

```
std::vector<int> v;
v.push_back(10);
v.push_back(20);

v.at(1) = 30;
```

Metóda `at(k)` umožňuje prístup k prvku na pozícii k, pričom pozície sú číslované od 0. V prípade, že pristúpime k prvku, ktorý vo vektore neexistuje, napr. za koncom C++ vektora, potom nastane tzv. výnimka. Výnimkám a ich významu venujeme oddelenú časť tohto učebného textu.

12.2 std::array

`std::array` je podobná dátová štruktúra ako `std::vector`, avšak kontajner má vopred určenú veľkosť k, ktorú nie je možné zmeniť. Na Obr. 2 je zobrazené takéto C++ pole s veľkosťou k. `std::array` je dátová štruktúra najviac podobná C poľu:



Obr. 2. std::array

```
1 std::array<int, 3> array;
2 array.at(0) = 10;
3 array.at(1) = 20;
4 array.at(2) = 30;
5
6 std::cout << array.at(1);
```

Na riadku 1 deklarujeme `array` s veľkosťou 3, do ktorého je možné vkladať objekty typu `int`.

12.3 std::deque

`std::deque` je obojsmerná fronta (angl. double-ended queue), do ktorej je možné vkladať prvky spredu, ako aj zozadu:

```
std::deque<int> dq;
dq.push_back(10);
dq.push_front(20);

std::cout << dq.at(0); //20
std::cout << dq.at(1); //10
```

Výhodou `std::deque` je podpora prístupu pomocou metódy `at`, teda pomocou pozície prvkov, a zároveň táto dátová štruktúra nevyžaduje spojitú časť pamäte pre kontajner. Kontajner je rozdelený do viacerých častí (angl. chunk), pričom každá časť môže byť implementovaná ako C++ vektor.

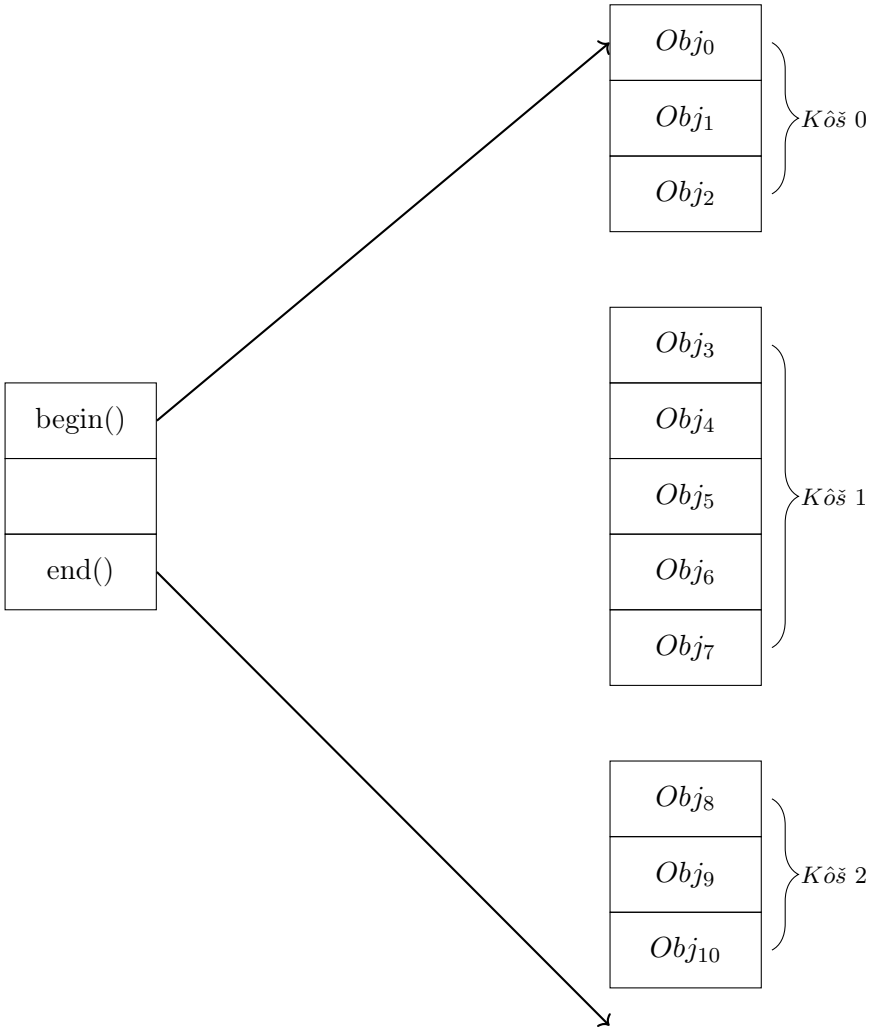
Na Obr. 3 je zobrazená obojsmerná fronta s troma košmi a 11 objektmi. Všimnime si, že po zavolaní metódy `end` získame iterátor, ktorý ukazuje za kontajner (rovnako ako v prípade iných kontajnerov).

12.4 std::list a std::forward_list

`std::list` je implementovaný ako obojsmerne zreťazený zoznam, `std::forward_list` je implementovaný ako (jednosmerne) zreťazený zoznam. `std::list` umožňuje vkladanie prvkov spredu aj zozadu:

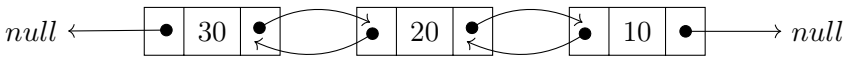
```
std::list<int> list;
list.push_back(10);
list.push_front(20);
list.push_front(30);

std::cout << list.front(); //30
std::cout << list.back(); //10
```

Na Obr. 4 je zobrazený obojsmerne zreťazený zoznam, pričom šípky reprezentujú smerníky ukazujúce na predchádzajúci, resp. nasledujúci prvok.



Obr. 3. std::deque

V prípade, že prvok je prvý alebo posledný, ukazuje smerník v jednom smere na *null*, čo znamená, že prvok neexistuje.



Obr. 4. std::list

`std::list` umožňuje aj zavolanie metód `pop_back` a `pop_front`, ktoré odstránia prvok na konci, resp. na začiatku zreťazeného zoznamu. `std::forward_list` umožňuje vkladanie, resp. odstraňovanie prvkov len spredu, pomocou metódy `push_front`, resp. `pop_front`. `std::list`, ako aj

`std::forward_list` podporujú metódu `insert`, resp. `insert_after`, teda vloženie prvku na ľubovolnú pozíciu, avšak na rozdiel od `push_front` alebo `push_back` takéto vkladanie má za následok prechod cez prvky kontajnera, čo nie je efektívne.

12.5 std::stack a std::queue

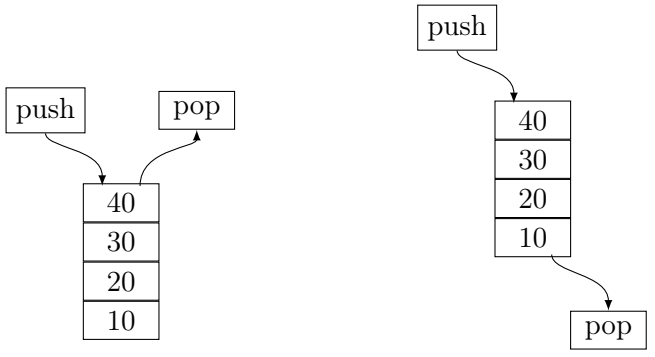
`std::stack` implementuje zásobník a `std::queue` implementuje frontu. Obe dátové štruktúry zvyčajne využívajú v C++ `std::deque` ako kontajner. V oboch prípadoch ide o adaptéry s operáciami, ktoré sú špecifické pre dané dve dátové štruktúry. Tieto dve dátové štruktúry neumožňujú prechádzanie cez prvky, pretože takáto možnosť by narúšala zmysel týchto dvoch dátových štruktúr. `std::stack` podporuje metódy `push`, `pop` a `top`:

```
std::stack<int> stack;
stack.push(10);
stack.push(20);
std::cout << stack.top(); //20

stack.pop();
std::cout << stack.top(); //10
```

`std::queue` podporuje metódy `push`, `pop` a `front`:

```
std::queue<int> q;
q.push(10);
q.push(20);
std::cout << q.front(); //10

q.pop();
std::cout << q.front(); //20
```

Na Obr. 5(a) je zobrazený zásobník a na Obr. 5(b) je zobrazená fronta. Všimnime si, že do fronty sa vkladajú prvky dozadu a odstraňujú spredu.



(a) std::stack (b) std::queue

Obr. 5

12.6 std::priority_queue

`std::priority_queue` implementuje haldu (angl. heap), teda dátovú štruktúru, ktorá umožňuje prístup k max., resp. min. prvku v konštantnom čase, pričom vkladanie a vyberanie prvkov je v logaritmickom čase. Podobne ako v prípade `std::stack` alebo `std::queue` ide o adaptér, cez ktorý nie je možné prechádzať. `std::priority_queue` podporuje metódy `push`, `pop` a `top`:

```
1 std::priority_queue<int> pq;
2 pq.push(10);
3 pq.push(20);
4 std::cout << pq.top(); //20
5
6 pq.pop();
7 std::cout << pq.top(); //10
```

Všimnime si, že zavolanie metódy `top` na riadku 4 vráti hodnotu 20, teda max. prvok. V prípade, že potrebujeme efektívny prístup k min. prvku, je možné použiť preddefinovaný komparátor `std::greater` z knižnice `functional`:

```
std::priority_queue<int, std::vector<int>, std::greater<int> > pq;
pq.push(10);
pq.push(20);
std::cout << pq.top(); //10

pq.pop();
std::cout << pq.top(); //20
```

Ako kontajner je použitý C++ vektor, ktorý umožňuje efektívnu reprezentáciu haldy.

12.7 std::set a std::multiset

`std::set` a `std::multiset` implementujú dátovú štruktúru založenú na vyváženom binárnom strome, ktorý umožňuje vkladanie a vyberanie prvkov v logaritmickom čase. `std::multiset`, na rozdiel od `std::set`, umožňuje viacnásobné vkladanie identických prvkov:

```
std::set<int> set;
set.insert(20);
set.insert(10);
set.insert(10);

for(int i : set) {
  std::cout << i; //10 20
}

set.erase(10);
```

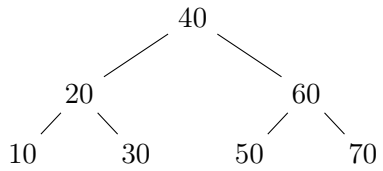
Všimnime si, že prvky sú na štandardný výstup vypísané roztriedené a bez duplicít.

```
std::multiset<int> set;
set.insert(20);
set.insert(10);
set.insert(10);

for(int i : set) {
  std::cout << i; //10 10 20
}

set.erase(10);
```

Všimnime si, že v prípade `std::multiset` sú prvky na štandardný výstup vypísané roztriedené a s duplicitami. Metóda `erase` v prípade `std::multiset` zmaže obe hodnoty 10. Metóda `find` vráti iterátor na prvý prvok s danou hodnotou. V prípade, že prvok nebolo možné nájsť, je vrátený iterátor za posledný prvok v kontajneri, t. j. na `std::set::end`.

```
std::multiset<int> set;
set.insert(20);
set.insert(10);
set.insert(10);

std::multiset<int>::iterator iter = set.find(10);

std::cout << *iter; //10
```

Na Obr. 6 je zobrazená reprezentácia prvkov ako vyvážený binárny strom. Schopnosť hľadať v binárnom strome v logaritmickom čase vyžaduje, aby tento strom bol vyvažovaný po pridaní príp. odstránení prvkov. Častá implementácia vyváženého binárneho stromu je červeno-čierny strom. Na Obr. 7 je zobrazený prípad, keď binárny strom nie je vyvážený.

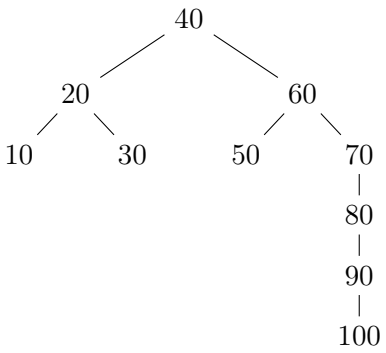


Obr. 6. Vyvážený binárny strom.

Potom, ako je prvok vložený do `std::set` alebo `std::multiset`, nie je možné ho zmeniť. Zavolanie metódy `begin` vráti obojsmerný iterátor, ktorý môže byť inkrementovaný, ale aj dekrementovaný.

12.8 std::map a std::multimap

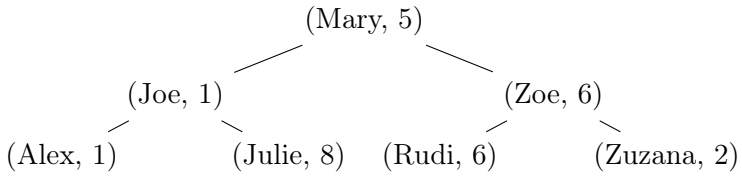
`std::map` a `std::multimap` implementujú podobnú dátovú štruktúru ako, `std::set` resp. `std::multiset`, avšak do kontajnera sa vkladajú usporiadané dvojice hodnôt $(key, value)$, pričom prvá sa nazýva *kľúč* a druhá sa nazýva *hodnota*:



Obr. 7. Nevyvážený binárny strom

```
std::map<std::string, int> m;
m.insert({"Zoe", 6});
m.insert({"Joe", 1});
m.insert({"Mary", 5});

for(auto i : m) {
  std::cout << i.first << "␣"
            << i.second << "␣"; //Joe 1 Mary 5 Zoe 6
}
```

V našom prípade je kľúč typu `std::string` a hodnota typu `int`. Všimnime si, že usporiadané dvojice sú v kontajneri roztriedené podľa kľúča. `first` umožňuje prístup ku kľúču a `second` umožňuje prístup k hodnote usporiadanej dvojice. Na Obr. 8 je zobrazený príklad reprezentácie `std::map` ako vyvážený binárny strom.



Obr. 8. Vyvážený binárny strom pre std::map

Podobne ako `std::multiset` umožňuje `std::multimap` duplicitné hodnoty, v tomto prípade duplicitné usporiadané dvojice s identickým kľúčom.

12.9 std::unordered_set, std::unordered_multiset, std::unordered_map a std::unordered_multimap

Neusporiadané asociatívne kontajnery umožňujú vkladanie, hľadanie a odstraňovanie prvkov v konštantnom čase v priemere, avšak vnútorne sú tieto prvky neroztriedené:

```
std::unordered_map<std::string, int> m;
m.insert({"Joe", 1});
m.insert({"Mary", 5});
m.insert({"Zoe", 6});

for(auto i : m) {
  std::cout << i.first << "␣"
            << i.second << "␣"; //Zoe 6 Joe 1 Mary 5
}

std::cout << m.bucket_count(); //5
```

Vkladanie, hľadanie a odstraňovanie prvkov využíva hašovanie. Výpočet hašu umožní vloženie prvku do špecifického koša (angl. bucket), ktorý okrem práve vloženého prvku môže obsahovať niekoľko ďalších prvkov, avšak ich max. počet je vopred určený a nízky v porovnaní s celkovým počtom prvkov. Pri vkladaní je nový prvok vložený do koša, v prípade hľadania je potrebné prechádzať všetkými prvkami v danom koši a v prípade odstraňovania daný prvok následne zmazať.

Na Obr. 9 je znázornený príklad s 5 košmi, každý s kapacitou 4 prvky. V prípade presiahnutia kapacity koša je potrebné zvýšenie počtu košov. Takéto zvýšenie sa v prípade potreby udeje automaticky a väčšinou sa udeje už pri dosiahnutí prednastavenej kritickej hodnoty. Metóda `bucket_count` vráti aktuálny počet košov.

12.10 Triedenie

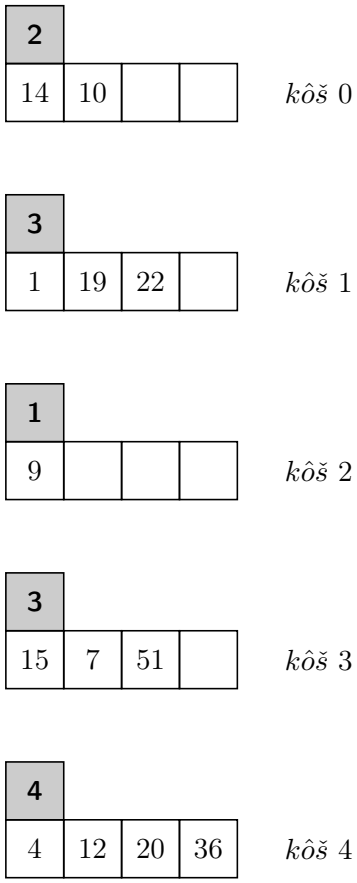
C++ v rámci knižnice `algorithm` ponúka predimplementované triediace algoritmy `std::sort` a `std::stable_sort`, pričom v druhom prípade ide o tzv. stabilné triedenie, pri ktorom relatívne poradie identických prvkov ostane zachované.

Nech P je pole, pričom $P[i]$ a $P[j]$ sú prvky poľa na pozícii i, resp. j, a $<$ je ostré čiastočné usporiadanie, potom triedenie je stabilné, ak platí:

$$i < j \wedge P[i] \equiv P[j] \implies \pi(i) < \pi(j),$$

kde $\pi(i)$, $\pi(j)$ je pozícia prvku $P[i]$, resp. $P[j]$ po triedení. V prípade C++ vektora je implementácia stabilného triedenia nasledovná:

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

int main() {
  std::vector<int> v{0, 1, 2, 10, 7, 22, 32, 7};

  std::stable_sort(v.begin(), v.end());

  return 0;
}
```



Obr. 9. std::unordered_set

Ak použijeme `std::sort` namiesto `std::stable_sort`, výsledok triedenia môže byť identický, avšak nie je garantované, že relatívne poradie identických prvkov ostane zachované. `std::sort` a `std::stable_sort` (v prípade dostatku voľnej pamäte) majú v najhoršom prípade časovú zložitosť $O(n \log n)$.

13 Výnimka

Výnimka nastane v prípade nepovolenej udalosti, ako napr. prístup za rozsah C++ vektora alebo nedostatku pamäte pri vzniku objektu pomocou `new`:

```
std::vector<int> v = {0, 1, 2};
std::cout << v.at(3);
```

V tomto prípade nastane výnimka `std::out_of_range`, pretože pristupujeme na neexistujúcu pozíciu vektora. Výnimku môžeme ošetriť (zachytiť) pomocou `try/catch`:

```
std::vector<int> v = {0, 1, 2};
```

```
try{
  std::cout << v.at(3);
} catch(const std::out_of_range& e) {
  std::cout << "Out of range!";
}
```

Výnimka vzniká pomocou `throw`:

```
double foo(double a, double b) {

  if(b == 0.0) {
    throw std::string("Division by zero!");
  }

  return a / b;
}

int main() {

  try{
    double c = foo(1.0, 0.0);
  } catch(const std::string& e) {
    std::cout << e;
  }

  return 0;
}
```

Po vzniku výnimky nastane odvinutie zásobníka (angl. stack unwinding), čo znamená, že beh programu je na danom mieste prerušený a nastane opustenie funkcie. V prípade, že zachytenie výnimky vôbec nenastane, nasleduje ukončenie behu programu. Odvinutie zásobníka má za následok, že časť kódu v danej funkcii vôbec nebude spustená:

```
void foo() {
  int* p = new int[1000];

  if(p[0] == 0) {
    throw std::string("Bad exception!");
  }

  std::cout << "Call me!";

  delete[] p;
}
```

Zavolanie `delete` nemusí nikdy nastať, pretože po vzniku výnimky nastane prerušenie behu kódu v tejto funkcii. Týmto spôsobíme únik pamäte, v našom prípade stratu adresy poľa `p`. Viacnásobné použitie `catch` je potrebné pre obsluhu viacerých výnimiek:

```
void foo() {
  int* a = new int[1000];
```

```
  if(a[0] != 0) {
    throw std::string("Bad exception!");
  }

  if(a[1] == 0) {
    throw 10;
  }

  std::cout << "Call me!";

  delete a;
}

int main() {

  try {
    foo();
  } catch (const std::string e) {
    std::cout << e;
  } catch (const int& e) {
    std::cout << e;
  } catch (...) {
    std::cout << "Other exception!";
  }

  return 0;
}
```

`catch(...)` umožňuje zachytenie ľubovoľnej výnimky, avšak medzi rôznymi zachytenými výnimkami potom nevieme rozlišovať. Preto sa odporúča vždy zachytávať každý druh výnimky zvlášť, aby bolo možné pre každú použiť špecifický obslužný kód.

Štandardná knižnica C++ používa výnimky definované v triede `std::exception` a v jej odvodených triedach ako napr. `std::bad_alloc`, `std::bad_cast`, `std::logic_error`.

13.1 noexcept

Funkcia môže byť označená ako `noexcept`, čo znamená, že neočakávame, že v nej nastane výnimka. V tomto prípade nemusí kompilátor generovať pomocný kód potrebný na uskutočnenie odvinutia zásobníka. V prípade, že nastane výnimka vo funkcii označenej ako `noexcept` nastane okamžité ukončenie behu programu.

Zvlášť vhodné je označovať funkcie ako `noexcept` v prípade, ak nastane výnimka, ktorú nevieme ošetriť. Napr. v prípade, že nevieme alokovať pamäť pre C++ vektor `tmp`, ako budeme ďalej pokračovať?

```
void foo(double x) noexcept {
  string s = "Anna and Zoe";
  vector<double> tmp(10);
```

```
}
```

Použitie `noexcept` pomáha kompilátoru pri pochopení zámeru programátora, podobne ako je to v prípade používania `const`. Často môže kompilátor optimalizovať kód napr. voľbou vhodného algoritmu, ktorý ťaží z vedomosti, že nie je potrebné brať do úvahy vznik výnimky.

Špecifický prípad nastáva v prípade deštruktorov. V prípade, že nevieme deštruovať objekt, čo by malo nastať, aký obslužný kód by mal byť použitý? Z tohoto dôvodu sú všetky deštruktory prednastavené ako `noexcept`. Ďalší špecifický prípad využitia `noexcept` nastáva v prípade presúvacej sémantiky.

14 Kopírovacia a presúvacia sémantika

14.1 Kopírovacia sémantika

Presúvacia (move) sémantika je súčasťou jazyka C++ od štandardu C++11. Cieľom presúvacej sémantiky je nahradenie neefektívneho kopírovania objektov ich presunom, pričom po presune z objektu `obj0` do objektu `obj1` ostane objekt `obj0` prázdny. Pozrime si nasledujúci príklad:

```
std::vector<Token> v;
Token t;
v.push_back(t);
```

Pri použití kopírovania je objekt `t` prekopírovaný do vektora `v`. Takéto kopírovanie môže byť neefektívne, ak trieda `Token` obsahuje veľké množstvo členov, ktoré je potrebné kopírovať. Uvažujme nasledovnú definíciu triedy `Token`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;
};
```

V tomto prípade pri kopírovaní objektu typu `Token` je potrebné prekopírovanie obsahu vektora `vec`. Lepšie riešenie sa javí presunutie vektora `vec` bez jeho kopírovania (teda bez kopírovania prvkov, ktoré obsahuje). Kopírovaniu premennej `a` nie je možné zabrániť, pretože jej „presun" pomocou smerníkov je rovnako náročný ako jej prekopírovanie.

Pri kopírovaní je zavolaný kopírovací konštruktor, ktorý môže byť implementovaný nasledovne:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;
```

```
  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
};
```

V C++ existuje „nepísané" pravidlo, ktoré v prípade, že trieda definuje kopírovací konštruktor, operátor priradenia alebo deštruktor odporúča definovať všetky tieto tzv. špeciálne metódy. Dôvod je taký, že priraďovanie by malo odzrkadľovať kopírovanie a deštrukcia objektu v tomto prípade by mala byť špecifická [5]. V našom prípade z dôvodu stručnosti definujeme len kopírovací konštruktor.

Pri presúvaní je zavolaný presúvací konštruktor, ktorý môže byť implementovaný nasledovne:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;

  Token(const Token& t0): vec(t0.vec), a(t0.a) {}

  Token(Token&& t0): vec(std::move(t0.vec)), a(t0.a) {}
};
```

Podobne ako v prípade kopírovania je odporúčané definovať presúvací konštruktor, ako aj operátor priradenia pre presúvanie [6].

14.2 Presúvacia sémantika

V prípade presúvania je potrebné, aby kompilátor pochopil náš zámer, teda využitie presúvania namiesto kopírovania. Uvažujme nasledujúci kód:

```
std::vector<Token> v;
Token t;
v.push_back(std::move(t));
```

V tomto prípade sme pretypovali `t` tak, aby nastalo zavolanie presúvacieho konštruktora a nie kopírovacieho konštruktora. Ak by sme naše riešenie otestovali, zistili by sme, že presúvanie napriek našej snahe nenastalo. Presúvanie totiž predpokladá, že presúvací konštruktor, ako aj operátor priradenia pre presúvanie nemôžu generovať výnimku. Tento stav dosiahneme použitím `noexcept`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  Token() = default;

  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
```

```
  Token(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
  }
  Token& operator=(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
    return *this;
  }
};
```

V prípade, že nastane výnimka pri kopírovaní, je situácia zvládnuteľná, pretože stále existujú pôvodné objekty, z ktorých sa kopíruje. Pri presune tieto objekty nemusia existovať v pôvodnom stave, a teda v prípade, že nastane výnimka, nevieme pôvodný stav obnoviť. Z podobného dôvodu sú všetky deštruktory označené ako `noexcept`, pretože ak zlyhá deštrukcia, nastane stav, keď ďalej nevieme pokračovať. Kompletná implementácia triedy `Token` pri dodržaní spomenutých odporúčaní je potom nasledovná:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;

  //default ctor
  Token() = default;

  //copy
  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
  Token& operator=(const Token& t0) {
    vec = t0.vec;
    a = t0.a;
    return *this;
  }

  //move
  Token(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
  }
  Token& operator=(Token&& t0) noexcept {
    vec = std::move(t0.vec);
    a = t0.a;
    return *this;
  }

  //dtor
  ~Token() = default;
};
```

Označenie ctor a dtor je častá skratka anglických výrazov „constructor" a „destructor".

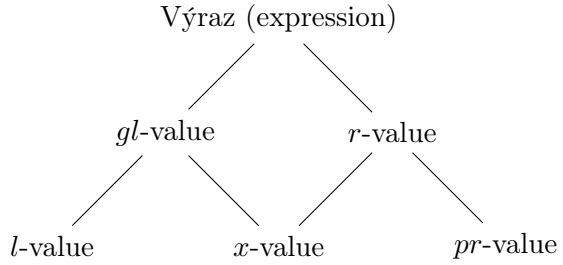
14.3 std::move

`std::move` nič nepresúva, ide len o pretypovanie tak, aby bol zavolaný presúvací konštruktor. `std::move` je implementovaný nasledovne:

```
template <typename T>
typename remove_reference<T>::type&& move(T&& arg) {
  return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Je zrejmé, že `std::move` je implementovaný ako `static_cast` na univerzálnu referenciu `&&`. `std::move` sa využíva na pretypovanie *l*-hodnoty (*l*-value) na *r*-hodnotu (*r*-value). Pretože táto *r*-hodnota nevznikla prirodzene napr. ako konštanta 1 alebo 3.4, takáto pretypovaná hodnota sa nazýva *x*-hodnota (*x*-value).

Pred uvedením štandardu C++11 bola potrebná len *l*-hodnota a *r*-hodnota, pričom *l*-hodnota je uložená na pamäťovom mieste, ktoré vieme určiť napr. pomocou operátora `&` a *r*-hodnota je všetko, čo nie je *l*-hodnota. *r*-hodnota môže tiež mať pamäťové miesto (niekedy existuje len v registri procesora), ale nevieme ho určiť, pretože operátor `&` nie je možné aplikovať na *r*-hodnotu napr. `&`3.4 nie je možné skompilovať.



Obr. 10. Druhy hodnôt zavedené v C++11

Obr. 10 zobrazuje rôzne druhy hodnôt, ktoré boli zavedené v C++11, pričom *gl*-value je generalized *l*-value a *pr*-value je pure *r*-value. *pr*-value je hodnota, ktorá vznikla ako *r*-hodnota, teda napr. už spomínané konštanty 1, 3.4 a tiež dočasné objekty:

```
Token t0, t1;
Token t = t0 + t1;
```

Pri sčítaní objektov `t0` a `t1`, za predpokladu, že sme definovali operátor `+` pre triedu `Token`, vznikne dočasný objekt, ktorý je prekopírovaný, resp. presunutý do `t`.

14.4 Presúvanie pri zmene veľkosti vektora

Kopírovanie vzniká bohužiaľ častejšie ako predpokladáme, a preto je použitie presúvacej sémantiky kľúčové. Majme nasledujúci vektor, ktorý obsahuje dva objekty typu `Token`:

```
std::vector<Token> v(0);
Token t0, t1;
v.push_back(std::move(t0));
v.push_back(std::move(t1));
```

V prípade, že do vektoru `v` pridáme ďalší (tretí) objekt, nastane zmena (alokovanej) veľkosti vektora tak, aby sa tam tento objekt zmestil (pre jednoduchosť predpokladajme, že zmena alokovanej veľkosti v našom prípade naozaj nastane aj pri zmene veľkosti z 2 na 3). Vznikne nové pole, do ktorého sú prekopírované, resp. presunuté pôvodné dva objekty a následovne je ešte pridaný tretí objekt. Pôvodné dva objekty sú deštruované, bude teda dvakrát zavolaný deštruktor. Uvažujme teda nasledovný príklad:

```
std::vector<Token> v(0);
Token t0, t1;
v.push_back(std::move(t0));
v.push_back(std::move(t1));

Token t;
v.push_back(std::move(t));
```

Po pridaní objektu `t` by sme sa pri zmene veľkosti vektora `v` radi vyhli kopírovaniu a využili presúvanie. Z tohto dôvodu je potrebné, aby trieda `Token` obsahovala presúvací konštruktor, v opačnom prípade nastane kopírovanie.

14.5 Automatické generovanie

V prípade, že trieda neobsahuje vlastný prednastavený (default) konštruktor, kopírovací konštruktor, operátor priradenia a deštruktor sú tieto automaticky generované kompilátorom. Presnejšie povedané, sú generované, ak sú potrebné. Ak napr. kód nevyžaduje priraďovanie, kompilátor v tichosti preskočí generovanie operátora priradenia, pretože nie je potrebný.

Podobne je to s presúvacím konštruktorom a operátorom priradenia pre presúvanie, ktoré sú automaticky generované s nasledujúcimi výnimkami. Ak trieda obsahuje kopírovací konštruktor, alebo operátor priradenia, potom nie sú generované. Dôvod je prechod z kompilátorov, ktoré podporujú štandard pred C++11. Pri prechode na novší kompilátor s podporou C++11 by mohlo nastať automatické generovanie presúvacieho konštruktora a operátora priradenia pre presúvanie, čo by mohlo neočakávane pozmeniť pôvodný zmysel kódu. Existencia presúvacieho konštruktora nespôsobí, že kopírovací konštruktor nebude automaticky generovaný.

V prípade, že neplánujeme využiť presúvaciu sémantiku je vhodné, aby presúvací konštruktor a operátor priradenia pre presúvanie boli označené ako `= delete`:

```
class Token {
  public:
  int a{1};
  std::vector<int> vec;
```

```
  //default ctor
  Token() = default;

  //copy
  Token(const Token& t0): vec(t0.vec), a(t0.a) {}
  Token& operator=(const Token& t0) {
    vec = t0.vec;
    a = t0.a;
    return *this;
  }

  //move
  Token(Token&& t0) = delete;
  Token& operator=(Token&& t0) = delete;

  //default dtor
  ~Token() = default;
};
```

14.6 Bez presúvacej sémantiky

Pred zavedením štandardu C++11 bolo možné dosiahnuť efektívnosť presúvacej sémantiky pomocou smerníkov:

```
std::vector<Token*> v;
Token* t = new Token;
v.push_back(t);
```

Objekt vzniká dynamicky pomocou `new` a jeho adresa je prekopírovaná do kontajnera, v našom prípade do vektora `v`. Výhodou tohto prístupu je, že nastane výlučne kopírovanie adries objektov, čo je možné považovať za efektívny spôsob uloženia objektu do kontajnera (resp. adresy objektu do kontajnera). Uvažujme nasledovný prípad zmeny veľkosti vektora:

```
std::vector<Token*> v(0);
Token* t0 = new Token;
Token* t1 = new Token;
v.push_back(t0);
v.push_back(t1);

Token* t = new Token;
v.push_back(t);
```

V tomto prípade pri zmene veľkosti kontajnera nastane prekopírovanie adries dvoch existujúcich objektov a pridanie ďalšej adresy do novovytvoreného poľa s dostatočnou veľkosťou.

Nevýhodou práce so smerníkmi v porovnaní s presúvacou sémantikou je možnosť, že niektorý objekt nebol vytvorený, teda môže nastať vloženie `nullptr`. Existuje veľké množstvo kódu, ktoré bolo napísané pred zavedením C++11 a z rôznych dôvodov pokračoval jeho vývoj bez použitia presúvacej

sémantiky. Z tohto dôvodu je potrebná aj znalosť návrhových vzorov, ktoré presúvaciu sémantiku nevyužívajú.

STL (Standard template library) má plnú podporu presúvacej sémantiky. Z praktického hľadiska to znamená, že ak napr. potrebujeme presunúť objekty z jedného vektora do druhého, za predpokladu existencie presúvacieho konštruktora (hoci aj automaticky generovaného) je možné, že výsledný skompilovaný kód bude efektívnejší.

15 Odvodená trieda

15.1 Základné mechanizmy dedenia

Mechanizmus dedenia v C++ umožňuje odvodiť triedu, nazývanú odvodená trieda (angl. derived class), z inej triedy, ktorú nazývame základná trieda (angl. base class). Odvodená trieda dedí všetky členy základnej triedy, okrem privátnych členov a `friend` členov. Kompilátor odvodenej triede automaticky vygeneruje prednastavený konštruktor, kopírovací konštruktor, operátor `=`, deštruktor, presúvací konštruktor a operátor `=` pre presúvanie:

```
#include <iostream>

class Base {
  protected:
  int a{1};

  public:
  int getA() {
    return a;
  }
};

class Derived : public Base {
  public:
  int b{1};
};

int main() {
  Base b;
  Derived d;

  d.b = 2;

  std::cout << d.getA();

  return 0;
}
```

Odvodená trieda má prístup aj k metóde `getA` triedy `Base`, pretože táto metóda je verejná. Členy triedy označené ako `protected` (tzv. chránené členy) sú prístupné v odvodenej triede, ale nie sú prístupné zvonka:

```
class Derived : public Base {
  public:
  int b{1};
  int foo() {
    return a * b;
  }
};
```

V prípade, že základná a odvodená trieda má člen s rovnakým menom, je potrebné špecifikovať, ktorý člen bude použitý:

```
class Base {
  protected:
  int a{1};
};

class Derived : public Base {
  public:
  int a{1};
  void foo() {
    Base::a = 2;
    a = 3;       //Derived::a
  }
};
```

Špecifikátor prístupu (angl. access specifier) sa používa na nastavenie prístupu k zdedeným členom:

- `class Derived : public Base` špecifikuje, že verejné a chránené členy základnej triedy sú zdedené ako verejné, resp. chránené členy.
- `class Derived : protected Base` špecifikuje, že verejné a chránené členy základnej triedy sú zdedené ako chránené členy.
- `class Derived : private Base` špecifikuje, že verejné a chránené členy základnej triedy sú zdedené ako privátne členy.

```
class Base {
  protected:
  int a{1};

  public:
  void foo() {}
};

class Derived : protected Base {
  public:
  int b{1};
};

int main() {
  Base b;
  Derived d;
```

```
  d.foo(); //chyba

  return 0;
}
```

Pri volaní zdedenej metódy `foo` nastane kompilačná chyba, pretože metóda `foo` je zdedená ako chránená (`protected`), a teda neprístupná mimo triedu.

15.2 Vznik odvodenej triedy

Vznik odvodenej triedy vyžaduje zavolanie konštruktora základnej triedy. Základná trieda buď obsahuje prednastavený konštruktor, alebo je potrebné kompilátor inštruovať, ktorý iný konštruktor bude použitý. Uvažujme nasledujúci príklad:

```
class Base {
 private:
  int a{1};
 public:
  Base(int a0) : a(a0) {}
};

class Derived : public Base {
private:
  int a{1};
public:
  Derived(int a0) : a(a0) {}
};

int main() {
  Derived d(10);

  return 0;
}
```

Tento kód neskompiluje, pretože trieda `Base` neobsahuje prednastavený konštruktor. Existuje niekoľko spôsobov ako sa vysporiadať s touto situáciou. Najjednoduchšia možnosť je pridať prednastavený konštruktor do triedy `Base`:

```
class Base {
 private:
  int a{1};
 public:
  Base() = default;
  Base(int a0) : a(a0) {}
};
```

Ďalšia možnosť je inštruovať kompilátor, aby použil existujúci parametrický konštruktor triedy `Base`:

```
class Base {
 private:
```

```
  int a{1};
 public:
  Base(int a0) : a(a0) {}
};

class Derived : public Base {
private:
  int a{1};
public:
  Derived(int a0) : Base(a0), a(a0) {}
};
```

15.3 Virtuálna metóda

Metóda označená ako `virtual` je deklarovaná, príp. aj definovaná v základnej triede, a znova definovaná v odvodenej triede:

```
class Base {
  protected:
  virtual int foo() = 0; //pure virtual
};

class Derived : protected Base {
private:
  int a{1};
  public:
  int foo() {
    return a;
  }
};

int main() {
  Derived d;

  d.foo();

  return 0;
}
```

Virtuálna metóda deklarovaná pomocou `= 0`, tiež nazývaná čisto virtuálna metóda (angl. pure virtual method), musí byť definovaná v odvodenej triede. Trieda, ktorá obsahuje čisto virtuálnu metódu sa nazýva abstraktná trieda. Virtuálne metódy sú dôležitým nástrojom pre špecifikovanie ktoré metódy odvodená trieda musí obsahovať. Abstraktné triedy často navrhuje architekt kódu a odvodené triedy implementuje iný programátor.

Virtuálna metóda implementuje dynamické viazanie (angl. dynamic binding), ktoré umožňuje, aby bola zavolaná metóda, ktorá zodpovedá odvodenej triede. Uvažujme nasledujúci príklad:

```
#include <iostream>

class Base {
```

```
  public:
  virtual void foo() {
    std::cout << "Base";
  };
};

class Derived0 : public Base {
private:
  public:
  void foo() {
    std::cout << "Derived0";
  }
};

class Derived1 : public Base {
private:
  public:
  void foo() {
    std::cout << "Derived1";
  }
};

int main() {
  Derived0 d0;
  Derived1 d1;

  Base* b0 = &d0;
  Base* b1 = &d1;

  b0->foo(); //Derived0
  b1->foo(); //Derived1

  return 0;
}
```

Ako virtuálne sú často deklarované deštruktory:

```
#include <iostream>

class Base {
public:
  virtual ~Base() {
    std::cout << "~Base";
  }
};

class Derived : public Base {
private:
  int* p{nullptr};

public:
  Derived() {
    p = new int[1000];
  }
  ~Derived() {
```

```
    delete[] p;
    std::cout << "~Derived";
  }
};


int main() {
  Base* b = new Derived;

  delete b;

  return 0;
}
```

Všimnime si, že je zavolaný deštruktor odvodenej triedy, ako aj deštruktor základnej triedy. V prípade, že deštruktor základnej triedy nie je virtuálny, bude zavolaný len deštruktor základnej triedy, čo môže viesť k úniku pamäte, napr. v našom prípade deštruktor odvodenej triedy obsahuje kód pre dealokáciu pamäte poľa `p`.

Pre zdôraznenie, že definovaná metóda v odvodenej triede je znova definovaná virtuálna metóda, používame kľúčové slovo `override`:

```
class Base {
public:
  virtual void foo() {}
};

class Derived : public Base {
public:
  void foo() override {}
  void foo(int a) {}
};

int main() {
  Derived d;
  d.foo();

  return 0;
}
```

V tomto prípade je metóda `foo()` v odvodenej triede znova definovaná metóda a metóda `foo(int)` je nová metóda odvodenej triedy. Pre možnosť preťažovania funkcií môže nastať zámena znova definovanej metódy a novej metódy, a preto sa odporúča znova definované metódy označovať pomocou `override`.

V prípade, že metóda už nebude ďalej znova definovaná, v našom prípade v triede, ktorá by dedila z triedy `Derived`, je takúto metódu vhodné označiť ako `final`:

```
void foo() final {}
```

Kľúčové slová `override` a `final` sú silné nástroje architektúry kódu tak,

aby nedochádzalo k zbytočným nedorozumeniam vo vývojárskych tímoch ohľadom účelu použitých metód.

Virtuálne metódy v jazyku C++ implementujú polymorfizmus (grécky „veľa foriem"), ktorý umožňuje, že rôzne funkcie sú spustené v závislosti od vstupného typu. Preťažovanie funkcií je tiež druh polymorfizmu.

15.4 Viacnásobné dedenie

Trieda môže dediť z viacerých tried:

```
class Base0 {};

class Base1 {};

class Derived : public Base0, public Base1 {};


int main() {
  Derived d;

  return 0;
}
```

Viacnásobné dedenie znamená, že členy oboch základných tried sú zdedené (podľa použitého špecifikátora prístupu). Viacnásobné dedenie má obmedzenie známe ako diamantové dedenie (angl. diamond inheritance). Uvažujme nasledujúci príklad, v ktorom triedy `B` a `C` dedia z triedy `A`, a nasledovne trieda `D` dedí z tried `B` a `C`:

```
class A {
public:
  int a{1};
};

class B : public A {};

class C : public A {};

class D : public B, public C {};

int main() {
  D d;

  d.a = 2; //chyba

  return 0;
}
```

Trieda `B`, ako aj trieda `C` zdedia premennú `a`. Nasledovne trieda `D` zdedí premennú `a` z triedy `B`, a tiež z triedy `C`. Pri pristúpení k premennej a nie je zrejmé, o ktorú premennú ide, a preto tento kód neskompiluje. Z tohto dôvodu je potrebné špecifikovať, že ide o virtuálne dedenie:

```
class A {
public:
  int a{1};
};

class B : virtual public A {};

class C : virtual public A {};

class D : public B, public C {};

int main() {
  D d;

  d.a = 2; //OK

  return 0;
}
```

`class B : virtual public A` znamená, že vytvorenie členov triedy odvodenej z triedy `B` je odložené na neskoršie.

15.5 friend trieda

Dedenie tak, ako sme si ho doteraz predstavili predpokladá, že medzi základnou a odvodenou triedou existuje logický vzťah. Napr. medzi triedami `Animal` a `Bird` existuje logický vzťah, pretože vták je druh zvieraťa, čo môžeme vyjadriť nasledovne:

```
class Animal {};

class Bird : public Animal {};
```

Ako ale postupovať v prípadoch, keď trieda potrebuje prístup k členom inej triedy, ale takýto logický vzťah neexistuje? Prečo by trieda `Cage` mala dediť z triedy `Animal`, pričom je zrejmé, že klietka nie je druh zvieraťa. Z tohoto dôvodu existuje možnosť označiť triedu ako `friend`, a takáto trieda má potom prístup k privátnym a chráneným členom inej triedy:

```
class Animal {
  friend class Cage;

private:
  int a{1};
};

class Cage {
public:
  void foo(const Animal& mAnimal) {
    std::cout << mAnimal.a;
  }
};
```

```
int main() {
  Animal a;
  Cage b;

  b.foo(a);

  return 0;
}
```

V tomto prípade má trieda `Cage` prístup k privátnej premennej `a` triedy `Animal`. Všimnime si, že priateľstvo nie je symetrické, teda trieda `Cage` má prístup k privátnym a chráneným členom triedy `Animal`, ale neplatí to naopak.

Priateľstvo tried sa nededí. Trieda, ktorá by dedila z triedy `Animal`, neposkytne prístup triede `Cage` k svojim privátnym a chráneným členom.

16 Lambda výraz

Lambda výraz umožňuje definovať lokálnu funkciu ako objekt, ktorý má prístup k premenným v danom rozsahu platnosti:

```
#include <iostream>

int main() {

  auto f = [](int a, int b) -> bool {
    return a < b;
  };

  std::cout << f(2, 1); //0

  return 0;
}
```

Lambda výraz sa skladá z nasledovných častí:

- `[]`, pričom `[&]` označuje zachytenie všetkých premenných v rozsahu platnosti podľa referencie, `[=]` označuje zachytenie všetkých premenných v rozsahu platnosti pomocou kópie (by value), `[&x]` označuje zachytenie premennej `x` podľa referencie a `[]` znamená, že premenné nie sú zachytené.
- () Vstupné premenné.
- `->` Typ návratovej hodnoty lambda výrazu (angl. trailing return type). Takýto spôsob zápisu je potrebný, aby návratový typ lambda výrazu mohol byť určený až za vstupnými parametrami.
- {} Telo lambda výrazu.

V nasledujúcom príklade je zachytená premenná `x` podľa referencie:

```
#include <iostream>

int main() {

  int x = 10;

  auto f = [&x](int a, int b) -> int {
    return a < b ? x : -x;
  };

  std::cout << f(2, 1);

  return 0;
}
```

V prípade lambda výrazov je zvlášť vhodné použiť automatické odvodenie typu pomocou `auto`, pretože typ `f` môže byť zložitý. Je potrebné upozorniť, že v našom prípade je návratový typ lambda výrazu `int`, ale typ `f` je iný zložitý typ.

16.1 Komparátor

Lambda výrazy sa využívajú na definovanie komparátorov, ktoré sú potrebné napr. pre usporiadanie objektov, ktoré sú vkladané do STL kontajnera:

```
class Token {
public:
  int a{1};
  Token(int a0) : a(a0) {}
};

int main() {

  Token t0(2), t1(1);

  auto comp = [](const Token& t0, const Token& t1) -> bool {
    return t0.a < t1.a;
  };

  std::set<Token, decltype(comp)> mSet(comp);

  mSet.insert(t0);
  mSet.insert(t1);

  return 0;
}
```

V tomto prípade sú objekty `t0` a `t1` vložené do `mSet` a usporiadané podľa hodnoty premennej `a`. `decltype(comp)` je typ lambda výrazu a samotné aplikovanie komparátora je pomocou konštruktora, ktorý má ako vstupný parameter samotný komparátor `comp`.

16.2 Hašovacia funkcia

Lambda výrazy sa tiež používajú na definovanie hašovacej funkcie pre neusporiadané asociatívne kontajnery. V nasledujúcom príklade si ukážeme, ako definovať hašovaciu funkciu v prípade `std::unordered_set`:

```
#include <unordered_set>

class Token {
public:
  int a{1};
  Token(int a0) : a(a0) {}
};

int main() {

  Token t0(2), t1(1);

  auto mHash = [](const Token& t0) {
    return std::hash<int>()(t0.a);
  };

  auto equal = [](const Token& t0, const Token& t1) {
    return t0.a == t1.a;
  };

  std::unordered_set<Token,
                     decltype(mHash),
                     decltype(equal)>
                     mSet(10, mHash, equal);

  mSet.insert(t0);
  mSet.insert(t1);

  return 0;
}
```

Pripomíname, že hašovacia funkcia je potrebná na určenie, do ktorého koša bude umiestnený objekt. V tomto prípade pomocou lambda výrazu `equal` taktiež definujeme porovnanie, ktoré je potrebné na vyhodnotenie, či vkladaný objekt je ekvivalentný s už vloženým objektom v danom koši. Toto je potrebné, pretože `unordered_set` nemôže obsahovať duplicitné objekty.

Pre základné numerické typy, ako aj pre `std::string`, existuje už definovaná hašovacia funkcia. Druh hašovacej funkcie môže byť rozdielny pre rôzne kompilátory, avšak pre tento účel je v prípade numerických typov často využívaná hašovacia funkcia MurmurHash [7] a v prípade `std::string` hašovacia funkcia djb2.[1]

Použitie kľúčového slova `using` ponúka zjednodušenie zápisu zložitejších deklarácii:

[1] Autor tejto hašovacej funkcie je Daniel J. Bernstein (DJB).

```
using mUSet = std::unordered_set<Token,
                                 decltype(mHash),
                                 decltype(equal)>;

mUSet mSet(10, mHash, equal);
```

`using` v jazyku C++ nahrádza používanie `typedef` a sprehľadňuje zápis, pričom alias je na ľavej strane, a to čo je nahradené je na pravej strane.

17 Šablóna (template)

Šablóna umožňuje tzv. generické programovanie (angl. generic programming), teda jediný kód pre viaceré vstupné typy. Uvažujme nasledujúci príklad:

```
template <typename T>
T const& max(T const& a, T const& b) {
  return a < b ? b : a;
}

int main() {
  int a = max(1, 2);
  double b = max(2.0, 1.0);

  return 0;
}
```

Generická funkcia max vráti maximum pre každý typ `T`, ktorý má definovaný operátor <. Podobným spôsobom môže byť definovaná aj trieda:

```
template <typename T>
class Token {
public:
  T const& max(T const& a, T const& b) {
    return a < b ? b : a;
  }
};

int main() {
  Token<int> t;

  t.max(1, 2);

  return 0;
}
```

S generickým programovaním sme sa už stretli pri prezentácii STL kontajnerov. Napr. `std::vector<int>` je C++ vektor, ktorý obsahuje numerické typy `int` a `std::unordered_set<int>` je neusporiadaný asociatívny kontajner, ktorý tiež obsahuje len numerické typy `int`. Tieto C++ kontajnery sú deklarované nasledovne:

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

Všimnime si, že C++ vektor vyžaduje zvolenie typu `T` a prípadne aj zvolenie alokátora, pričom ale môžeme použiť aj prednastavený alokátor `std::allocator<T>`. Účelom alokátora je manažment pamäte a v prípade C++ vektora je jeho úloha napr. alokácia pamäte potrebnej pri zmene veľkosti vektora. Ako napovedá deklarácia, alokátor môže byť používateľom zmenený.

```
template <class Key,
          class Hash = hash<Key>,
          class Pred = equal_to<Key>,
          class Alloc = allocator<Key>
          > class unordered_set;
```

`std::unordered_set` vyžaduje zvolenie typu `T`, a prípadne aj hašovacej funkcie `Hash`, funkcie ekvivalentnosti prvkov `Pred` a alokátora `Alloc`.

Použitie `class T`, ako aj `typename T` je možné. Pôvodne existovala len možnosť `class T`, avšak kľúčové slovo `class` takto získalo nové použitie, dodatočne k deklarácii triedy. Neskoršie bolo preto navrhnuté, aby sa v prípade šablón používalo `typename`. Avšak z dôvodu spätnej kompatibility je naďalej podporovaná aj pôvodná možnosť.

V C++ nie je možné vytvoriť typ počas behu kódu. Z tohoto dôvodu je C++ šablóna vyhodnocovaná počas kompilácie pre špecifický typ. Inak povedané, šablóna neexistuje v skompilovanom kóde, šablóna umožňuje flexibilitu pri návrhu kódu tak, aby nebol potrebný identický kód pre rôzne typy.

Šablóna tiež umožňuje polymorfizmus, avšak vzhľadom na vyhodnocovanie šablón počas kompilácie sa tento druh polymorfizmu nazýva statický polymorfizmus. Virtuálne metódy sú príkladom polymorfizmu, ktorý je možný počas behu kódu.

18 Pretypovanie

Pretypovanie v C++ je možné rovnakým spôsobom ako v jazyku C, avšak v prípade použitia C-pretypovania nie je zrejmé, či pretypovanie je naozaj zámerom programátora, alebo ide len o prehliadnutie pri tvorbe kódu. C++ zavádza nasledovné možnosti pre pretypovanie:

- `static_cast`
- `reinterpret_cast`
- `const_cast`
- `dynamic_cast`

Uvažujme nasledujúce príklady:

```
char c = 10;
int* p = (int*) &c

const int a = 3;
int b = (int) a;
b = 4;
```

V prvom prípade by bolo vhodné porozumieť, či zámer programátora je skutočne interpretovať danú oblasť pamäte ako pamäť obsahujúcu `int`, a v druhom prípade by bolo vhodné porozumieť, či zámer programátora je skutočne odstrániť kvalifikátor `const`. V prvom prípade môže nastať prístup do nevyhradenej pamäte, ak predpokladáme, že veľkosť `char` je 1 a veľkosť `int` je 4.

Jazyk C++ dáva programátorovi jednoznačnú možnosť vyjadriť svoj zámer:

```
char c = 10;

int* p = static_cast<int*>(&c);      //chyba
int* p = reinterpret_cast<int*>(&c); //OK
```

Účelom `static_cast` je „bezpečné" pretypovanie a účelom `reinterpret_cast` je pretypovanie s jasne vyjadreným zámerom, pričom je zrejmé, že programátor si bol (vo väčšej miere) vedomý možných dôsledkov. V nasledujúcom kóde nastane v prípade `static_cast` kompilačná chyba, pretože medzi triedami `Token0` a `Token1` neexistuje vzťah dedenia, ani žiadna možnosť konverzie. Pri použití `reinterpret_cast` už chyba nenastane, pretože sme zreteľne vyjadrili, že práve takéto pretypovanie si želáme:

```
class Token0 {};
class Token1 {};

int main() {
  Token0* t0;
  Token1* t1 = new Token1;

  t0 = static_cast<Token0*>(t1);      //chyba
  t0 = reinterpret_cast<Token0*>(t1); //OK

  delete t1;
  return 0;
}
```

`static_cast` umožňuje implicitné konverzie ako napr. z `int` do `double`, taktiež umožňuje pretypovanie v prípade, že existuje príslušný konverzný konštruktor:

```
class Token1;

class Token0 {
  public:
  Token0() = default;
  Token0(const Token1& t0) {} //konverzný konštruktor
```

```
};

class Token1 {};

int main() {
  Token0 t0;
  Token1 t1;

  t0 = static_cast<Token0>(t1); //OK

  return 0;
}
```

const_cast umožňuje zrušenie const (a súčasne žiadné iné pretypovanie):

```
int a = 1;
const int& b = a;
const_cast<int&>(b) = 2;
```

Nasledujúce použitie const_cast má za následok nedefinované správanie, pretože b je referencia na premennú a, ktorá neumožňuje zmenu hodnoty.

```
const int a = 1;
const int& b = a;
const_cast<int&>(b) = 2;
```

const_cast je možné využiť aj v prípade, že potrebujeme zavolať funkciu so smerníkom, ktorý nie je const:

```
int foo(int* p) {
  return *p;
}

int main() {
  const int a = 10;
  const int* p0 = &a;

  int* p1 = const_cast<int*>(p0);
  foo(p1);

  return 0;
}
```

dynamic_cast je praktický v prípadoch, že nevieme, aký typ ma daný objekt. V prípade, že pretypovanie je možné, je vrátený platný smerník, v opačnom prípade je vrátený nullptr. Na rozdiel od static_cast je dynamic_cast vykonaný počas behu programu. Uvažujme nasledovný príklad:

```
class Base {
public:
  virtual ~Base() {};
};

class Derived : public Base {};

int main() {
```

```
  Derived* d0 = new Derived();
  Base* b = dynamic_cast<Base*>(d0); //OK

  Derived* d1 = dynamic_cast<Derived*>(b); //OK

  delete d0;
  return 0;
}
```

V tomto prípade môžeme `b` pretypovať späť na `Derived`, pretože `b` ukazuje na platný objekt typu `Derived`. Takýto postup sa často využíva na overenie počas behu programu, či daný objekt je tiež inštanciou inej triedy. Všimnime si, že trieda `Base` má virtuálny deštruktor. Virtuálny člen je podmienka pre úspešné pretypovanie na odvodenú triedu pomocou `dynamic_cast`, pretože je vyžadované uloženie informácie o type, s akým objekt vznikol.

19 Dokonalé preposielanie

Uvažujme nasledujúci príklad, keď do C++ vektora vkladáme objekt typu `Token` pomocou metódy `push_back`:

```
class Token {
 public:
  int a{1};
  Token(int a0) : a(a0) {
    std::cout << "Token(int)";
  }
  Token(const Token& t0) {
    std::cout << "Token(const Token&)";
  }
  Token(Token&& t0) noexcept {
    std::cout << "Token(Token&&)";
  }
};

int main() {
  std::vector<Token> v;

  v.push_back(Token(10)); //vznikne dočasný objekt

  return 0;
}
```

Po spustení skompilovaného kódu sa pri vkladaní do C++ vektora zavolá parametrický konštruktor a následne presúvací konštruktor. Presnejšie povedané, nastane vznik dočasného objektu typu `Token`, následne je zavolaný presúvací konštruktor, vznikne prázdny objekt a do neho je presunutý dočasný objekt (a pôvodný objekt už nie je potrebný, a preto je ešte zavolaný aj deštruktor).

Je možné vložiť do C++ vektora objekt bez presúvania príp. kopírovania? Na tento účel môžeme použiť metódu `emplace_back`, ktorá vytvorí objekt

priamo vo vektore (v prípade asociatívnych kontajnerov a neusporiadaných asociatívnych kontajnerov v STL je potrebné použiť `emplace`):

```
v.emplace_back(10);
```

V tomto prípade sa zavolá len parametrický konštruktor `Token(int)`. Čo potrebujeme, aby sme mohli definovať metódy ako `emplace_back`? Takéto riešenie z pohľadu jazyka C++ vyžaduje, že metóda môže prijať parameter ľubovolného typu, aby mohol byť zavolaný zodpovedajúci konštruktor. Uvažujme nasledujúci príklad s funkciou `wrapper` s dvoma vstupnými parametrami:

```
template <typename T1, typename T2>
void wrapper(T1& a0, T2& a1) {
    foo(a0, a1);
}
```

V tomto prípade môžeme zavolať funkciu `wrapper`, pričom jej vstupné parametre sú referencie `&`, čo znamená, že túto funkciu nemôžeme volať napr. s konštantami:

```
wrapper(1, 2);
```

Ak by sme potrebovali pokryť všetky možnosti volania s `const` referenciami a s referenciami, potrebovali by sme implementovať nasledujúce možnosti:

```
template <typename T1, typename T2>
void wrapper(T1& a0, T2& a1) { foo(a0, a1); }

template <typename T1, typename T2>
void wrapper(const T1& a0, T2& a1) { foo(a0, a1); }

template <typename T1, typename T2>
void wrapper(T1& a0, const T2& a1) { foo(a0, a1); }

template <typename T1, typename T2>
void wrapper(const T1& a0, const T2& a1) { foo(a0, a1); }
```

Takýto únavný prístup hrubou silou je nepraktický, a je zrejmé, že iný prístup je potrebný. Tento prístup sa v jazyku C++ nazýva dokonalé preposielanie (angl. perfect forwarding), pričom je potrebné napísanie jedinej funkcie:

```
template <typename T1, typename T2>
void wrapper(T1&& a0, T2&& a1) {
    foo(std::forward<T1>(a0), std::forward<T2>(a1));
}
```

19.1 Reference collapsing

Pri dokonalom preposielaní sa využívajú pravidlá zjednodušenia referencií (angl. reference collapsing). Definícia funkcie `wrapper` určuje, že vstupné parametre sú univerzálne referencie `&&`. Čo ale nastane, ak je funkcia wrapper zavolaná s `const` referenciou `&` alebo s referenciou `&`?

```
int a = 1;
wrapper(a, 2);
```

Názov „univerzálna" referencia napovedá, že v určitých prípadoch môže takáto referencia byť referencia na r-hodnotu a v iných referencia na l-hodnotu. V prípade, že zavoláme funkciu `wrapper` s `&` referenciou a vstupný parameter je `&&` referencia, čo presne nastane? Z tohoto dôvodu bolo potrebné určiť pravidla pre konverziu referencií. Pravidlá sú jednoduché, referencia `&` vždy „zvíťazí":

```
& & = &
&& & = &
& && = &
&& && = &&
```

Jediná výnimka nastane, ak zavoláme funkciu `wrapper` s `&&` referenciou, v tomto prípade je ďalej preposlaná referencia `&&`, a teda funkcia `foo` je zavolaná akoby sme použili priamo `&&` referenciu.

V našom prípade je typ `T1` je odvodený ako `int&` a typ `T2` je odvodený ako `int&&`. Funkcia `foo` je nakoniec zavolaná ako `foo(int&, int&&)`. Ako presne je definovaný `std::forward`?

```
template<class T>
T&& forward(typename std::remove_reference<T>::type& t) noexcept {
  return static_cast<T&&>(t);
}

template <class T>
T&& forward(typename std::remove_reference<T>::type&& t) noexcept {
  return static_cast<T&&>(t);
}
```

`std::forward` je pretypovanie pomocou `static_cast`, pričom existuje preťaženie pre `T&` a `T&&`. Všimnime si, že pri použití `static_cast` nastane zjednodušenie referencií napr. ak je `T` odvodené ako `int&` potom môžeme `std::forward` prepísať ako:

```
int& forward(int& t) noexcept {
  return static_cast<int&>(t);
}
```

Pre úplnosť vysvetlenia, `std::remove_reference` znemožňuje, aby zjednodušenie referencií nastalo pri volaní `std::forward`, a aby bola zavolaná správna preťažená verzia `std::forward`. Z pohľadu dokonalého preposielania ide o menej dôležitý technický detail.

20 constexpr

`constexpr` označuje výraz, ktorý je inicializovaný počas kompilácie, t. j. jeho hodnota musí byť vyhodnotená tiež počas kompilácie:

```
constexpr double twoPi = 2 * M_PI;
```

kde `M_PI` je hodnota π z knižnice `cmath`. Výrazy označené ako `const` sú na rozdiel od `constexpr` inicializované počas behu kódu. Uvažujme nasledujúci príklad:

```
int a = 1;

constexpr double twoPi = 2 * M_PI;
twoPi = a; //chyba
```

V tomto prípade nastane pri priradení novej hodnoty premennej `twoPi` kompilačná chyba, pretože `twoPi` je `constexpr`, a teda jej vyhodnotenie musí nastať počas kompilácie. Avšak hodnota premennej `a` je známa až počas behu kódu. Ako `constexpr` môžu byť označené aj funkcie, nielen premenné:

```
constexpr double twoPi() {
  return 2 * M_PI;
}
```

`constexpr` je možné použiť aj na vyhodnotenie zložitých výrazov ako napr. faktoriál:

```
constexpr unsigned long long factorial(unsigned long long n) {
  return n > 0 ? n * factorial(n - 1) : 1;
}

int main() {
  constexpr unsigned long long f = factorial(10);

  return 0;
}
```

Lambda výrazy môžu byť taktiež vyhodnotené počas kompilácie:

```
auto add = [](int a, int b) constexpr {
    return a + b;
};
```

`constexpr` funkcie sú vyhodnotené počas kompilácie, ak ich vstupné parametre sú `const`:

```
constexpr int a = add(1, 2); //OK

int k = 1;
constexpr int b = add(k, 2); //chyba
```

`constexpr` výraz plne nahrádza makro `#define` v prípade potreby zavedenia konštantného výrazu a `#define` by mal byť výlučne použitý (spolu s ďalšími makrami) pre podmienečnú kompiláciu kódu:

```
#define MAX 4

int main() {
  constexpr int max = 4; //lepšie ako #define

  return 0;
}
```

21 Chytrý smerník

Chytrý smerník manažuje ďalší objekt, ktorý je automaticky dealokovaný pri vystúpení z rozsahu platnosti bez potreby explicitnej dealokácie s využitím `delete`:

```
#include <memory>
#include <iostream>

class Token {
public:
  ~Token() {
    std::cout << "~Token";
  }
};

int main() {
  std::unique_ptr<Token> t(new Token);

  return 0;
} //zavolaný deštruktor ~Token
```

Jazyk C++ obsahuje niekoľko druhov chytrých smerníkov definovaných v knižnici `memory`:

- `unique_ptr`, ktorý umožňuje len jedinečné vlastníctvo objektu,
- `shared_ptr`, ktorý umožňuje zdielané vlastníctvo objektu,
- `weak_ptr`, ktorý umožňuje prístup k objektu, pokým tento objekt existuje.

21.1 unique_ptr

`unique_ptr` zabezpečuje, že vlastníctvo manažovaného objektu je jedinečné, teda nemôže nastať kopírovanie. `unique_ptr` sa často využíva na manažovanie privátneho kontajnera triedy, v našom prípade C++ vektora. Naším zámerom je, aby tento vektor nemohol byť kopírovaný do inej inštancie triedy `Token`:

```
class Token {
private:
  std::unique_ptr<std::vector<int> > v{new std::vector<int>};
public:
  Token() = default;
  Token(const Token& t0) {
    this->v = t0.v; //chyba
  };
};
```

V tomto prípade nastane kompilačná chyba, pretože sa pokúšame kopírovať `std::unique_ptr`. Bez možnosti takéhoto kopírovania je zabezpečené jedinečné vlastníctvo inštancie C++ vektora. V tomto prípade môžeme uvažovať o označení kopírovacie konštruktora ako `= delete` (ako aj operátora `=`).

```
class Token {
private:
  std::unique_ptr<std::vector<int> > v{new std::vector<int>};
public:
  Token() = default;
  Token(const Token& t0) = delete;
};
```

Pri zaniknutí inštancie triedy `Token`, automaticky zanikne aj inštancia chytrého smerníka, a teda nie je potrebná dealokácia C++ vektora v deštruktore.

21.2 shared_ptr

`shared_ptr` umožňuje zdielaný prístup k manažovanému objektu, v prípade, že tento objekt nie je vlastnený žiadnym `shared_ptr`, potom tento objekt zanikne. Uvažujme nasledujúci príklad:

```
class Token {
public:
  ~Token() {
    std::cout << "~Token";
  }
};

void foo(std::shared_ptr<Token> ptr0) { //kopírovanie
  std::cout << ptr0.use_count(); //2
}

int main() {
  std::shared_ptr<Token> ptr(new Token);
  std::cout << ptr.use_count(); //1

  foo(ptr);

  return 0;
} //zavolaný deštruktor ~Token
```

Metóda `use_count` vráti počet zdielaní manažovaného objektu. Pri prvom zavolaní vráti hodnotu 1, pretože existuje len jedna inštancia chytrého smerníka. Pri druhom zavolaní vo funkcii `foo` vráti hodnotu 2, pretože nastane kopírovanie do premennej `ptr0`. Nakoniec je zavolaný deštruktor triedy `Token`, pretože chytrý smerník vychádza zo svojho rozsahu platnosti, a teda manažovaný objekt je automaticky dealokovaný. Prístup k smerníku manažovaného objektu je možné získať pomocou metódy `get`:

```
Token* t0 = t.get();
```

Vymeniť manažovaný objekt za iný objekt je možné pomocou metódy `reset`:

```
ptr.reset(new Token);
```

V prípade, že `ptr` je jediný (posledný) chytrý smerník, ktorý manažuje vymieňaný objekt, je zavolaný jeho deštruktor.

Jazyk C++ tiež umožňuje zjednodušený zápis pomocou `std::make_shared` bez potreby písať `new`:

```
std::shared_ptr<Token> ptr = std::make_shared<Token>();
```

Podobný zjednodušený zápis je možný aj v prípade `std::unique_ptr` pomocou `std::make_unique`.

21.3 weak_ptr

`weak_ptr` umožňuje prístup k manažovanému objektu v prípade, že tento objekt ešte existuje, pričom samotný objekt je manažovaný pomocou `shared_ptr`. Uvažujme nasledujúci príklad:

```
class Token {
public:
  ~Token() {
    std::cout << "~Token";
  }
};

int main() {
  std::shared_ptr<Token> ptr(new Token);
  std::cout << ptr.use_count(); //1

  std::weak_ptr<Token> ptr_w = ptr;
  std::cout << ptr.use_count(); //1

  ptr.reset(); //zavolaný deštruktor ~Token

  std::cout << ptr_w.expired(); //1
  return 0;
}
```

`weak_ptr` nezvyšuje počet využití manažovaného objektu, preto metóda `use_count` v našom prípade vždy vráti hodnotu 1. Metóda `reset` odstráni manažovaný objekt, a preto je zavolaný deštruktor triedy `Token`. Pomocou metódy `expired` môžeme zistiť, či manažovaný objekt je neplatný, v našom prípade táto metóda vráti hodnotu `true`, teda manažovaný objekt je neplatný.

`weak_ptr` je užitočný pri vzájomnom vlastníctve manažovaného objektu:

```
class Token1;

class Token0 {
public:
  std::shared_ptr<Token1> ptr;
};
```

```
class Token1 {
public:
  std::shared_ptr<Token0> ptr;
};

int main() {
  std::shared_ptr<Token0> ptr0(new Token0);
  std::shared_ptr<Token1> ptr1(new Token1);

  ptr0->ptr = ptr1;
  ptr1->ptr = ptr0;

  return 0;
}
```

V tomto prípade `ptr0` nemôže zaniknúť, pokým nezanikne `ptr1` a naopak. Riešením tohto problému je použitie `weak_ptr` v jednej z týchto tried:

```
class Token1 {
public:
  std::weak_ptr<Token0> ptr;
};
```

22 I/O

Vstup a výstup (angl. input, output) do/z programu je v jazyku C++ založený na sekvencii bajtov, ktorú nazývame prúd (angl. stream). S príkladom prúdov sme sa už stretli pri predstavovaní knižnice `iostream`, ktorá umožňuje vkladanie do štandardného výstupu (napr. obrazovka), resp. čítanie zo štandardného vstupu (napr. klávesnica).

Základné vlastnosti prúdov sú:

- Typová bezpečnosť, pričom ak I/O operácia nie je definovaná pre daný typ, potom kompilátor vygeneruje chybu,
- Operácie na prúdoch sú nezávislé od druhu vstupu a výstupu, tie isté operácie môžeme použiť pre rôzne zariadenia.

Prúdy, resp. triedy, ktoré definujú rôzne druhy prúdov, je možné rozdeliť nasledovne:

- Knižnice `ios`, `istream`, `ostream`, `streambuf` a `iosfwd` definujú základné triedy a zvyčajne nie sú priamo používané v používateľskom kóde.
- Knižnica `iostream` definuje triedy potrebné na komunikáciu so štandardným vstupom a výstupom.
- Knižnica `fstream` definuje triedy potrebné pre čítanie zo súborov, zápis do súborov, ako aj vznik a zmazanie súborov.

- Knižnica `sstream` umožňuje narábanie s reťazcami, ako keby boli prúdy.

22.1 Typová bezpečnosť

V takomto prípade kompilácie neprebehne úspešne, pretože operácia vkladania << nie je definovaná pre triedu `Token`:

```
class Token {
  public:
  int a{1};
};

int main() {
  Token t;
  std::cout << t; //chyba

  return 0;
}
```

Pre úspešné použitie operácie vkladania << je potrebné túto operáciu preťažiť napr. takýmto spôsobom:

```
inline std::ostream& operator<<(std::ostream& stream,
                                const Token& t0) {
  stream << t0.a;
  return stream;
}
```

22.2 fstream

Knižnica `fstream` umožňuje zápis do súboru a čítanie zo súboru. V nasledujúcom príklade vytvoríme súbor `test.txt` a následne do neho zapíšeme „Hello, World!":

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

int main() {
  std::ofstream fout("test.txt");

  if (!fout) {
    std::cerr << "Error!";
  } else {

    fout << "Hello,␣World!";
    fout.close();
  }

  return 0;
}
```

V prípade, že sa súbor nepodarí otvoriť je na štandardnom výstupe vypísané chybové hlásenie. Súbor je zatvorený zavolaním metódy `close`. Čítanie zo súboru je podobné:

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

int main() {
  std::ifstream fin("test.txt");

  if (!fin) {
    std::cerr << "Error!";
  } else {

    char ch;
    while (fin.get(ch)) {
    std::cout << ch;
    }

  fin.close();
  }

  return 0;
}
```

Zmazanie súboru je možné zavolaním `std::remove`:

```
std::remove("test.txt");
```

Súbor je možné otvoriť na zápis a čítanie zároveň, ako aj na čítanie a zápis v binárnom móde:

```
fstream file("test.txt", ios::in | ios::out | ios::binary);
```

Spôsob otvorenia súboru je určený tzv. módom:

- `ios::in`, súbor je otvorený na čítanie,
- `ios::out`, súbor je otvorený na zápis,
- `ios::binary`, súbor je otvorený v binárnom móde, tento mód zabezpečuje, že znak „nový riadok" nie je interpretovaný ako `\r\n`, inými slovami binárny mód zabezpečuje, že to čo je poslané do prúdu vyjde nezmenené z prúdu von,
- `ios::app`, zapisovanie na koniec súboru,
- `ios::trunc`, obsah súboru je po otvorení na zápis zmazaný.

Stav prúdu je zaznamenávaný pomocou niekoľkých príznakov, resp. bitov. Po otvorení je prúd v stave `good`, avšak práca s prúdom môže zmeniť tento stav na niektorý z ďalších stavov. Čítanie zo súboru a zápis do súboru sú možné len v prípade, že stav prúdu je `good`. Prúd ostane v nedobrom stave až pokým nie je zavolaná metóda `clear`.

- `goodbit` je nastavený v prípade, že prúd je v normálnom stave,
- `eofbit` je nastavený po dosiahnutí konca súboru,
- `failbit` je nastavený v prípade logického zlyhania, napr. operácia na prúde nie je možná, pričom ostatné operácie môžu naďalej pokračovať,
- `badbit` je nastavený v prípade, že nastalo zlyhanie, ktoré znemožňuje ďalšie používanie prúdu.

Tieto stavy je možné zistiť zavolaním príslušných metód:

```
void print_state (const std::ios& stream) {
  std::cout << "␣good()=" << stream.good();
  std::cout << "␣eof()=" << stream.eof();
  std::cout << "␣fail()=" << stream.fail();
  std::cout << "␣bad()=" << stream.bad();
}
```

22.3 sstream

Knižnica `sstream` umožňuje narábanie s reťazcom tak ako s prúdom. V nasledujúcom príklade použijeme `std::getline` na rozdelenie reťazca `str` podľa rozdeľovacieho znaku, v našom prípade použijeme medzeru `' '`:

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

int main() {
  std::stringstream s("Hello,␣World!");

  std::string str;
  while(std::getline(s, str, '␣')) {
    std::cout << str;
  }

  return 0;
}
```

Program vypíše „Hello, World!" bez medzery, pretože rozdeľovací znak nie je vložený do reťazca `str`. Všimnime si, že nakoniec sú `eofbit`, ako aj `failbit` nastavené, pretože sme dosiahli koniec prúdu, resp. operácia čítania z prúdu nie je naďalej možná.

Od C++17 je súčasťou jazyka knižnica `std::filesystem`, ktorá rozširuje možnosti manipulácie so súbormi a priečinkami.

23 Ďalšie možnosti štandardu C++17

23.1 if a switch s inicializáciou

Štandard C++17 umožňuje inicializáciu vo vnútri `if` a `switch` nasledovným spôsobom:

```
#include <iostream>

enum class Color {
  Red,
  Green,
  Blue
};

auto getColor() {
  return Color::Red;
}

int main() {
  if(auto c = getColor(); c == Color::Red) { //inicializácia
    std::cout << "Color is red!";
  }

  return 0;
}
```

Podobným spôsobom je tiež možná inicializácia vo vnútri `switch`:

```
int main() {

  switch(auto c = getColor(); c) { //inicializácia

  case Color::Red:
    std::cout << "Color is red!";
    break;

  case Color::Green:
    std::cout << "Color is green!";
    break;

  case Color::Blue:
    std::cout << "Color is blue!";
    break;

  default:
    std::cout << "Other color!";
  }

  return 0;
}
```

V prípade zápisu `if` a `switch` s inicializáciou je obmedzenie rozsahu platnosti inicializovanej premennej; po opustení rozsahu platnosti táto premenná zanikne.

23.2 Štruktúrované viazanie

Štruktúrované viazanie (angl. structured binding) umožňuje návrat viacerých hodnôt z funkcií. Uvažujme nasledujúci príklad:

```
#include <tuple>

auto foo() {
  int a{0};

  return std::make_tuple(a, 1, 2);
}

int main() {
  auto[a, b, c] = foo();

  return 0;
}
```

V prípade iterácie kontajnerom môže štruktúrované viazanie výrazne zjednodušiť zápis:

```
#include <iostream>
#include <map>

int main() {
  std::map<int, double> m = {{1, 1.0}, {2, 2.0}, {3, 3.0}};

  for(auto[key, value] : m) {
    std::cout << key << "\t" << value << std::endl;
  }

  return 0;
}
```

V tomto prípade eliminuje použitie štruktúrovaného viazania potrebu použiť `first` a `second` na prístup k usporiadaným dvojiciam $(key, value)$, ktoré sú uložené v `std::map`.

Register

Literatúra

[1] GNU Compiler Collection. [cit. 2020-03-02] Dostupné na: https://gcc.gnu.org

[2] M. Drozda, *Zbierka príkladov v jazyku C++*. Spektrum STU, 2019.

[3] S. Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, Reading, MA, 2001.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[5] Rule of three (C++ programming). [cit. 2020-03-02] Dostupné na: https://en.wikipedia.org/wiki/Rule_of_three_(C++_programming)

[6] The rule of five. [cit. 2020-03-02] Dostupné na: https://cpppatterns.com/patterns/rule-of-five.html

[7] A. Appleby. MurmurHash. [cit. 2020-03-02] Dostupné na: https://sites.google.com/site/murmurhash