Zbierka príkladov v jazyku C++

v1.05

Martin Drozda
2019

Obsah

Úvod

Tento učebný text na príkladoch, z ktorých mnohé boli zadané ako skúškové úlohy na predmete Programovacie techniky (B-PT) na Fakulte elektrotechniky a informatiky (FEI) Slovenskej technickej univerzity v Bratislave (STU) v rokoch 2015–2018, vysvetľuje a komentuje najčastejšie chyby, ktorých sa dopúšťajú začínajúci programátori v jazyku C++.

Tento učebný text vznikol na podnet študentov, ktorí navrhovali, aby tieto úlohy boli vydané spolu s komentárom. Úlohy v tomto učebnom texte boli testované s kompilátorom gcc/g++ vo verzii 7.3.[1] Niektoré tu uvedené príklady vyžadujú kompiláciu v štandarde C++17, a teda je potrebné kompilovať ich s prepínačom -std=c++17. Príklady neboli testované s inými kompilátormi a pri ich použití je potrebné informovať sa, do akej miery podporujú štandardy jazyka C++.

Ako odporúčanú literatúru z oblasti programovania v jazyku C++ je vhodné spomenúť najmä:

- 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 významnej autority v oblasti programovacieho jazyka C++, Scotta Meyersa. Tieto knihy sú odporúčané aj ako referencie pre kohokoľvek, koho ambíciou je stať sa odborníkom pre tento programovací jazyk.

Tento učebný text nie je základný kurz programovacieho jazyka C++, ale doplnkovou učebnou pomôckou pre absolvovanie predmetu Programovacie techniky.

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, august 2018 – január 2019.

[1] `https://gcc.gnu.org/`

Poďakovanie

Ďakujem všetkým študentom, ktorí úspešne absolvovali predmet Programovacie techniky.

GNU Compiler Collection

GNU Compiler Collection sa skladá z niekoľkých kompilátorov, kde gcc (GNU C Compiler) sa používa na kompilovanie C kódu a g++ sa používa na kompilovanie C++ kódu. Počiatky gcc siahajú do roku 1987, keď bol vydaný kompilátor gcc v1.0. Kompilátor g++ podporuje rôzne štandardy C++ ako napr. C++03, C++11, C++14 a C++17, kde C++03 je štandard z roku 2003 a pod. Pre najnovší štandard je vždy vhodné overiť, ktoré časti štandardu sú už implementované v danej verzii kompilátora. Aj keď kompilátor g++ podporuje štandard C++17, neznamená to, že všetky časti štandardu sú už implementované. Implementácia C++14 bola ukončená v g++ v5, ktorý bol vydaný koncom roka 2014.

Pre skompilovanie C++ kódu, ktorý je v súbore file.C stačí použiť:

```
g++ -std=c++11 file.C
```

pričom kompilácia nastane v štandarde C++11. Najrelevantnejšie prípustné hodnoty z pohľadu tohto učebného textu pre prepínač -std sú {c++03, c++11, c++14, c++17}. Skompilovaný kód je v súbore a.out, ktorý môžeme spustiť:

```
./a.out
```

Zmena prednastaveného mena tohoto súbora je možná pomocou prepínača -o:

```
g++ -std=c++11 file.C -o myfile
```

kde v tomto prípade sa skompiluje do súboru s názvom *myfile* a ktorý spustíme nasledovne:

```
./myfile
```

Pri kompilácii je odporúčaný výpis varovaní kompilátora, ktorý je možný pomocou prepínačov -Wall a -Wextra.

```
g++ -std=c++11 file.C -o myfile -Wall -Wextra
```

Pri používaní týchto prepínačov je vhodné overiť si, aké varovania budú kompilátorom vypisované. Ak skompilujeme nasledovný kód:

```
1 int main() { //g++ -std=c++11 -Wall -Wextra
2   int a = 3.3;
3
4   return 0;
5 }
```

kompilátor vypíše varovanie *"warning: unused variable 'a' [-Wunused-variable]"*, t. j. premenná `a` je nepoužitá. V prípade, že použijeme aj prepínač -Wconversion dostaneme aj varovanie *"warning: conversion to 'int'*

alters 'double' constant value [-Wfloat-conversion]", t. j. prepínače -Wall a -Wextra nepokrývajú všetky varovania.

Pre kompiláciu je možné zapnúť rôzne optimalizácie a to pomocou prepínača -O s úrovňou optimalizácie napr. -O2, -O3 alebo -Os.

```
g++ –std=c++11 file.C –o myfile –Wall –Wextra –O2
```

V prípade, že by sme chceli kód optimalizovať na chvostovú rekurziu (tail recursion) môžeme použiť prepínač -foptimize-sibling-calls. Úrovne optimalizácie ako napr. -O2, -O3 alebo -Os združujú väčšie množstvo optimalizácií, ktoré budú použité.[2] Cieľom optimalizácie -Os je minimalizácia veľkosti skompilovaného kódu, kde -Os používa všetky optimalizácie ako v -O2 okrem tých, kde je známe, že predlžujú skompilovaný kód. Pri ladení kódu napr. s gdb (GNU Project Debugger)[3] je potrebné generovať informáciu pre tento debugger. V tomto prípade je potrebné použitie prepínača -ggdb.

```
g++ –std=c++11 file.C –o myfile –Wall –Wextra –ggdb
```

Tento krátky úvod pokrýva len použitie relevantné pre tento učebný text. Kompletná dokumentácia je dostupná online na `https://gcc.gnu.org/onlinedocs/`.

[2] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
[3] `https://www.gnu.org/software/gdb/`

1 string: operátor +

Tento kód neskompiluje, kompilátor g++ vypíše *"error: invalid operands of types 'const char [6]' and 'const char [8]' to binary 'operator+'"*. Problém je použitie operátora + na riadku 5 s dvoma reťazcami typu `const char*`. Operátor + v knižnici `string` pripojí reťazec typu `const char*` do už existujúceho reťazca typu `std::string`, preto kompilácia riadkov 6 a 7 prebehne bez problémov. Problém nastane opäť na riadku 8, kde nastane evaluácia výrazu zľava doprava, a teda operátor + bude použitý na spojenie `"Hello"+ " world"`, čo nie je možné.

```
1  #include <string>
2  using namespace std;
3
4  int main() {
5    string s0 = "Hello" + "␣world!"; //chyba
6    string s1 = "Hello" + string("␣world!"); //ok
7    string s2 = string("Hello") + "␣world!"; //ok
8    string s3 = "Hello" + "␣world" + string("!"); //chyba
9
10   return 0;
11 }
```

2 Binárne literály

C++ od verzie C++14 podporuje binárne literály. Pre komplexnejšiu manipuláciu s číslami v binárnom zápise je odporúčaná knižnica `std::bitset`.

```
1  int main() {
2    int d = 42;
3    int o = 052;
4    int x = 0x2a;
5    int X = 0X2A;
6
7    int b = 0b101010; //C++14
8    int B = 0B101010; //C++14
9
10   return 0;
11 }
```

Literály sú konštanty, s ktorými kompilátor nemôže manipulovať, napr. horeuvedené numerické literály, `float` a `double` literály ako napr. `3.0` alebo `-144.15`, `bool` literály `true` a `false`, `char` literály ako napr. `'a'` alebo `'Z'` a reťazcové literály `"Hello world!\n"`.

3 Globálne a static premenné

Globálne a static premenné `a`, `b` sú inicializované na hodnotu `0`. Lokálna premenná `c` má nedefinovanú hodnotu (garbage). Miesto v pamäti, kde vznikne

lokálna premenná môže byť z nejakého dôvodu vynulované, a preto sa stáva, že programátor vníma hodnotu lokálnej premennej ako nastavenú na hodnotu `0` – zhoda okolností.

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

int a;
static int b;

int main() {
  int c;

  cout << a << endl; //0
  cout << b << endl; //0
  cout << c << endl; //?

  return 0;
}
```

4 Inicializácia premenných

Premenná `a0` nie je inicializovaná, teda môže mať ľubovolnú hodnotu. Premenná `a1` je inicializovaná na hodnotu `5` pomocou univerzálnej inicializácie. Premenná `a2` je inicializovaná na hodnotu `0` pomocou `int()`, ktorý vytvorí objekt typu `int` a nastane inicializácia na prednastavenú hodnotu `0`. Premenná `a3` je inicializovaná na hodnotu `5` pomocou `int(5)`, ktorý vytvorí objekt typu `int` a nastane inicializácia na hodnotu `5`.

Premenná `a4` je inicializovaná na hodnotu `5`. Hodnota `5.5` je reprezentovaná ako double a následovne nastane implicitná konverzia na `int`. Kompilátor g++ s prepínačom -Wconversion vypíše *"warning: conversion to 'int' alters 'double' constant value [-Wfloat-conversion]"*.

V prípade premennej `a5` kód neskompiluje, pretože nastane zúženie typu `double` na `int`, čo v prípade univerzálnej inicializácie nie je možné. Kompilátor g++ vypíše *"error: narrowing conversion of '5.5e+0' from 'double' to 'int'"*. Pole `a6` je inicializované na hodnoty `0`. Pole `a7` je tiež inicializované na hodnoty `0`.

```
int main() {
  int a0; //garbage
  int a1{5}; //C++11
  int a2 = int();
  int a3(5);
  int a4 = 5.5;
  int a5{5.5}; //C++11, chyba
  int a6[10] = {};
  int a7[10]{}; //C++11

```

```
11    return 0;
12  }
```

Premenná `d0` je adresa, ktorej obsah nie je inicializovaný. Premenná `d1` je adresa, ktorej obsah je inicializovaný na hodnotu `0`. Pole `arr0` je inicializované na hodnoty `0`. Pole `arr1` je inicializované na hodnoty `{1, 2, 3, 4, 5}`.

```
1  int main() {
2    double* d0 = new double;
3    double* d1 = new double();
4
5    int* arr0 = new int[5]();
6    int* arr1 = new int[5]{1, 2, 3, 4, 5}; //C++11
7
8    return 0;
9  }
```

V C/C++ premenná typu void neexistuje (a teda ju nie je možné ani inicializovať).

```
  void a = 3;
```

Kompilátor g++ vypíše *"error: variable or field 'a' declared void"*.

V prípade inicializácie referencií je situácia nasledovná. Premenná `a1` je inicializovaná na hodnotu `5`. Premenná `a2` typu `int&` musí byť inicializovaná. Kompilátor g++ vypíše *"7:8: error: 'a2' declared as reference but not initialized"*. Premenná `a3` je inicializovaná literálom. Kompilátor g++ vypíše *"8:13: error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'"*. Premenná `a4` je inicializovaná na hodnotu `5`, pretože inicializácia je v prípade `const int&` prípustná. Opätovné priradenie na riadku 9 skončí chybou *"12:8: error: redeclaration of 'int& a1'"*. Opätovná deklarácia referencií nie je povolená.

```
1  int main() {
2    int a0 = 5;
3    int& a1 = a0; //OK
4    int& a2; //chyba
5    int& a3 = 5; //chyba
6    const int& a4 = 5; //OK
7
8    int a5 = 5;
9    int& a1 = a5; //chyba
10
11   return 0;
12 }
```

5 Premenná a funkcia

Kód skompiluje s varovaním *"warning: the address of 'int a1()' will always evaluate as 'true' [-Waddress]"* a na obrazovku sa vypíše "5 1 0". Tu je dôležité

všimnúť si, že na riadku 5 prebehne inicializácia premennej `a0` na hodnotu `5`. **Na riadku 6 sa nachádza deklarácia funkcie `a1()`,** ktorá má nenulovú adresu a tá je prekonvertovaná na `bool`. Premenná `a2` je inicializovaná na hodnotu `0`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int a0(5);
6    int a1(); //funkcia!
7    int a2{};
8
9    cout << a0 << "\t"; //5
10   cout << a1 << "\t"; //1
11   cout << a2 << endl; //0
12
13   return 0;
14 }
```

Na riadku 6 sa nachádza deklarácia funkcie a nie zavolanie `int()`. Zachovanie tejto syntaxe je potrebné pre spätnú kompatibilitu s jazykom C. Scott Meyers nazýva podobné syntaktické anomálie "most vexing parse".[4]

6 Pretečenie typu

Pri násobení na riadku 8 sa vynásobia dva literály typu `int`. Výsledkom násobenia je dočasný objekt typu `int`, ktorý pretečie a následne sa prekopíruje do premennej typu `long long`. Po prenásobení s `1LL` je problém vyriešený; násobenie je vyhodnocované zľava doprava. V testovanom prípade sa `sizeof(int)` rovná 4 a `sizeof(long long)` sa rovná 8.[5]

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    int x = 1000000;
6    int y = 1000000;
7
8    long long z = x * y; //long long z = 1LL * x * y;
9    cout << z; //-727379968
10
11   return 0;
12 }
```

[4] `https://en.wikipedia.org/wiki/Most_vexing_parse`

[5] Tento príklad je prevzatý z online zdroja Geeks for Geeks: `https://www.geeksforgeeks.org/common-mistakes-avoided-competitive-programming-c-beginners/`.

7 Menné priestory

Tento kód neskompiluje, pretože premenná `x` je v oboch menných priestoroch `N1` a `N2` a pri inicializácii `x` nie je použitý špecifikátor menného priestoru `N1::` alebo `N2::`. Kompilátor g++ vypíše *"error: reference to 'x' is ambiguous"*. Menné priestory boli zavedené, aby nenastávali konflikty medzi identifikátormi ako napr. premennými, názvami funkcií a metód.

```
1  namespace N1 {int x;}
2  namespace N2 {int x;}
3  using namespace N1;
4  using namespace N2;
5
6  int main() {
7     x = 1; //N1::x
8     x = 2; //N2::x
9
10    return 0;
11 }
```

Menný priestor `std` obsahuje identifikátory, ktoré sú súčasťou jazyka C++. Napr. `std::string` je typ pre znakové reťazce a `std::cout` označuje štandardný výstup (standard output stream).

8 nullptr

Tento kód neskompiluje a kompilátor g++ vypíše *"error: call of overloaded 'foo(NULL)' is ambiguous"*. C++11 zavádza `nullptr`, ktorý má typ `std::nullptr_t` a vyjadruje presne to, čo je potrebné, teda nulový smerník. Vždy je potrebné použiť `nullptr` namiesto `NULL`. Mimochodom `NULL` je `#define NULL 0`, teda `0`, a nie nulový smerník. V našom prípade je nejasné, či sa pri zápise `foo(NULL)` má zavolať funkcia `foo(int)` alebo `foo(int*)`. Staršie verzie kompilátorov v tomto prípade volali funkciu `foo(int)`, pretože `NULL` je `0`, a teda `int`.

```
1  #include <iostream>
2  using namespace std;
3
4  void foo(int a) {
5    cout << "foo(int)" << endl;
6  }
7
8  void foo(int* a) {
9    cout << "foo(int*)" << endl;
10 }
11
12 int main() {
13   foo(0); //foo(int)
14   foo(NULL); //?
15   foo(nullptr); //foo(int*)
```

```
16
17   return 0;
18 }
```

9 Návrat adresy lokálnej premennej

Jedná sa o jednu z najčastejších chýb začínajúcich programátorov v C/C++.

Return na riadku 7 vráti adresu lokálnej premennej `b`, ktorá po skončení funkcie `add5()` prestane existovať. Na riadku 14 nastane pokus dereferencovať premennú `a`, ktorá obsahuje adresu neexistujúcej premennej, a preto nastane pamäťová chyba. Presnejšie povedané nastane podľa použitého kompilátora nedefinované správanie, ktoré sa väčšinou prejaví ako pamäťová chyba (segmentation fault), ale v prípade niektorých kompilátorov sa táto chyba môže prejaviť aj iným spôsobom príp. (výrazne) neskôr.

```
1  #include <iostream>
2  using namespace std;
3
4  int* add5(int* a) {
5    int b = *a + 5;
6
7    return &b;
8  }
9
10 int main() {
11   int* a;
12   *a = 3;
13   a = add5(a);
14   cout << *a;
15
16   return 0;
17 }
```

Return na riadku 11 vracia adresu lokálnej premennej `d` z riadku 10. Pri pokuse dereferencovať na riadku 23 nastane pamäťová chyba.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5   private:
6    double d;
7
8   public:
9    double* get() {
10     double d;
11     return &d;
12   }
13   void set(double d) {
```

```
14      this->d = d;
15    }
16  };
17
18  int main() {
19    Token* t = new Token();
20    t->set(3.14);
21    double* d = t->get();
22
23    cout << *d << endl;
24
25    delete t;
26    return 0;
27  }
```

10 Návrat referencie na lokálnu premennú

V tomto prípade vraciame referenciu na lokálnu premennú. Premenná `t` po skončení funkcie `foo()` zanikne, a teda referenciu na túto premennú nie je možné použiť. Kompilátor g++ vypíše *"7:9: warning: reference to local variable 't' returned [-Wreturn-local-addr]"*. Na riadku 13 nastane pamäťová chyba.

```
1  class Token {
2  public:
3    int size = 0;
4  };
5
6  Token& foo() {
7    Token t;
8
9    return t;
10 }
11
12 int main() {
13   Token& t = foo();
14   t.size = 1;
15
16   return 0;
17 }
```

11 Návrat adresy existujúcej premennej

Premenná a vznikne na riadku 11 a referencia na túto premennú je vstupný parameter funkcie `add5()`. Return na riadku 7 teda vráti adresu existujúcej premennej. `cout` na riadku 12 vypíše 3.

```
1  #include <iostream>
2  using namespace std;
3
```

```
4  int* add5(int& a) {
5    int b = a + 5;
6
7    return &a;
8  }
9
10 int main() {
11   int a = 3;
12   cout << *add5(a); //3
13
14   return 0;
15 }
```

12 Presne špecifikovaný vstupný parameter funkcie

Vstupná premenná funkcie `add()` má typ `int const * const`, čo znamená "konštantný smerník na konštantný `int`". Takéto presné špecifikovanie vstupu je časté pre knižnice. Na obrazovku sa vypíše 2. Všimnime si, že samotná premenná `b` zanikne po ukončení funkcie `add()`, ale jej hodnota je vrátená.

```
1  #include <iostream>
2  using namespace std;
3
4  int add(int const * const a) {
5    int b = *a + 1;
6
7    return b;
8  }
9
10 int main() {
11   int  a = 1;
12   cout << add(&a) << endl; //2
13
14   return 0;
15 }
```

Pre úplnosť:

- `const int a;`
  `int const a;` je to isté.
  V oboch prípadoch ide o konštantný `int`. Druhá možnosť zápisu je lepšia, pretože pri čítaní sprava doľava je typ zrejmejší.
- `int * const a;`
  je konštantný smerník na `int`.
- `int const * a;`
  je smerník na konštantný `int`.
- `int const * const a;`
  je konštantný smerník na konštantný `int`.

13 Nesprávne dereferencovanie

Tento kód nie je možné skompilovať. Na riadku 12 sa pokúšame dereferencovať premennú `a`, ktorá je typu `int`, teda nie je smerník. Všimnime si, že funkcia `add()` vracia adresu lokálnej premennej `b`, avšak tento nedostatok sa nijako neprejaví, pretože neprebehne kompilácia. Všimnime si tiež, že kompilátor g++ vypíše varovanie *"5:7: warning: address of local variable 'b' returned [-Wreturn-local-addr]"*, t. j. na riadku 5 je definovaná lokálna premenná, ktorej adresa je vrátená na riadku 7. Všetky varovania kompilátora je potrebné vyriešiť, príp. rozumieť prečo ich kompilátor použil.

```
1  #include <iostream>
2  using namespace std;
3
4  int* add(int* a) {
5    int b = *a + 1;
6
7    return &b;
8  }
9
10 int main() {
11   int a = 1;
12   cout << add(*a) << endl;
13
14   return 0;
15 }
```

14 Typová nezhoda

Na riadku 4 má funkcia `add()` vstupný parameter typu `int&`, ale na riadku 13 voláme funkciu `add()` s adresou premennej `a`. Tento kód nie je možné skompilovať. `&` na riadku 4 a `&` na riadku 13 vyzerajú podobne, ale v prvom prípade ide o zápis referencie a v druhom prípade o zápis operátora, ktorý vráti adresu premennej.

```
1  #include <iostream>
2  using namespace std;
3
4  int* add(int& a) {
5    int* b = new int();
6    *b = a + 1;
7
8    return b;
9  }
10
11 int main() {
12   int a = 1;
13   cout << add(&a) << endl;
14
15   return 0;
```

```
16 }
```

15 Funkcia so statickou premennou

Na obrazovku sa vypíše adresa uložená v premennej `b` (teda adresa premennej `a`), napr. 0x7ffc404967ec. Funkcia `add()` obsahuje statickú premennú. Vzhľadom na to, že táto funkcia je zavolaná len raz, nemá v tomto prípade použitie statickej premennej žiadne opodstatnenie. Skúste zavolať funkciu `add()` ešte raz!

```
1  #include <iostream>
2  using namespace std;
3
4  int* add(int& a) {
5    static int i = 0;
6    a += i;
7    ++i;
8
9    return &a;
10 }
11
12 int main() {
13   int a = 10;
14   int* b = add(a);
15
16   cout << b;
17
18   return 0;
19 }
```

Na obrazovku sa vypíše 10. Na skúške predmetu Programovacie techniky sa často vyskytujú úlohy, ktoré sú skoro identické (porovnajme s predchádzajúcim príkladom). Je zaujímavé pozorovať, že takéto podobné úlohy vedú k predstave, že v jednom prípade je kód kompilovateľný, nespôsobuje žiadnu pamäťovú chybu a v druhom prípade musí teda kód obsahovať chybu (alebo naopak).

```
1  #include <iostream>
2  using namespace std;
3
4  int* add(int& a) {
5    static int i = 0;
6    a += i;
7    ++i;
8
9    return &a;
10 }
11
12 int main() {
13   int a = 10;
14   int* b = add(a);
```

```
15
16   cout << *b;
17
18   return 0;
19 }
```

16 Únik pamäte (memory leak)

Na riadku 7 vznikne nový objekt a jeho adresa je uložená v premennej `t`. Na riadku 8 vznikne ďalší objekt a jeho adresa je uložená v premennej `t`, pričom adresa objektu z riadku 7 je prepísaná, t. j. nevieme kde sa prvý objekt nachádza v pamäti, a teda ani ho nemôžeme deštruovať pomocou `delete`. Po skončení behu kódu väčšina operačných systémov uvoľní pamäť, ktorá bola alokovaná. Pred skončením behu kódu napr. v prípade služieb, ktoré dlhodobo bežia na serveri, môže byť takéto zabúdanie adries fatálne, pretože vytvorené a stratené objekty zaberajú pamäť a jej uvoľnenie pomocou `delete` nie je možné. Všimnime si, že veľkosť poľa `array` je nezanedbateľná.

```
1  class Token {
2  public:
3    int array[1000000000]; //very big!
4  };
5
6  int main() {
7    Token* t = new Token;
8    t = new Token; //memory leak
9
10   delete t;
11   return 0;
12 }
```

17 Návrat adresy dynamicky vytvoreného objektu

Na riadku 5 sa dynamicky vytvorí objekt. Adresa, ktorá je vrátená na riadku 8 ukazuje na tento objekt. Dereferencovanie prebehne bez problémov a na obrazovku sa vypíše 2. Samotná premenná `b`, ktorá vznikne na riadku 5 zaniká vystúpením z funkcie `add()`, ale adresa, ktorú uchovávala sa cez return vráti do funkcie `main()`. Všimnime si, že na riadku 13 nastane memory leak, pretože stratíme adresu objektu `int`, ktorá bola vrátená funkciou `add()`.

```
1  #include <iostream>
2  using namespace std;
3
4  int* add(int a) {
5    int* b = new int();
6    *b = a + 1;
7
8    return b;
```

```
9  }
10
11 int main() {
12   int a = 1;
13   cout << *add(a) << endl; //memory leak
14
15   return 0;
16 }
```

Pamäť počítača z pohľadu C++ kódu:

- Stack (zásobník) pre statickú alokáciu napr. pre lokálne premenné, ktoré automaticky zanikajú pri vystúpení z daného kontextu {...}.
- Free store resp. heap pre dynamickú alokáciu pomocou `new` resp. `malloc()` alebo `realloc()`, kde pre C++ je viacej zaužívaný pojem free store a pre C je viacej zaužívaný pojem heap.
- Statické dáta ako napr. globálne a static premenné, ktoré existujú počas celého behu kódu t. j. na rozdiel od dát na stacku nezanikajú.
- Spustiteľný kód t. j. to čo sme skompilovali.

18 Pretečenie zásobníka (stack overflow)

Opätovným volaním funkcie `f()` na riadku 9 nastane pretečenie zásobníka. Zásobník je použitý na ukladanie lokálnych premenných funkcií ako napr. poľa `p`. Pri každom ďalšom zavolaní sa voľný priestor na zásobníku zmenší o min. 10000B (plus priestor potrebný na uloženie návratových adries atď.). Veľkosť zásobníka je závislá od kompilátora a operačného systému, v našom prípade nastala pamäťová chyba po 209 zavolaniach funkcie `f()`. Z tohto je možné odvodiť, že v našom prípade (`sizeof(int)` je 4) veľkosť zásobníka bola 8MB (čo si tiež môžeme overiť pomocou linuxovského príkazu `ulimit -s`). Je to veľa alebo málo? Našťastie vieme dynamicky alokovať priestor pomocou `new` a `malloc()`.

```
1  #include <iostream>
2  using namespace std;
3
4  void f() {
5    static int i = 0;
6    cout << i << endl;
7    int p[10000];
8    ++i;
9    f();
10 }
11
12 int main() {
13   f();
```

```
14
15   return 0;
16 }
```

Všimnime si tiež, že ide o chvostovú rekurziu (tail recursion), keďže po zavolaní funkcie `f()` na riadku 9 a návrate sa pole `p` už nevyužíva. Z toho dôvodu nie je potrebné pole `p` držať v stacku. Kompilátor g++ vie detegovať chvostovú rekurziu. Skúste kód skompilovať s prepínačom optimalizácie -O2!

19 Čiarka a bodka

Ako hovoria Česi, "chybička se vloudí". A tak namiesto bodky je na riadku 5 použitá čiarka, čo je možné ľahko prehliadnuť. Kód skompiluje bez problémov a na obrazovku sa vypíše 140. Ako je to možné? Výraz `(3,14)` sa vyhodnotí zľava doprava. Výsledok vyhodnotenia je teda `14` a po prenásobení s `10` je z toho `140`.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5   double d = (3,14) * 10;
6   cout << d;
7
8   return 0;
9 }
```

20 class a struct

Rozdiel v C++ medzi triedou a štruktúrou je nasledovný:

- `struct` má členy prednastavené ako `public`. Špecifikátor dedenia je prednastavený ako `public`. `struct` môže obsahovať konštruktory, deštruktor atď.
- `class` má členy prednastavené ako `private`. Špecifikátor dedenia je prednastavený ako `private`. `class` môže (samozrejme) obsahovať konštruktory, deštruktor atď.

Štruktúra `Token1` na riadku 9 je definovaná ako odvodená štruktúra. Test "triedy" na riadkoch 39 až 42 vráti hodnotu `true`, teda `Token`, `Token1`, `Token2` a `Token3` sú triedy. Na riadku 37 nastane pri kompilácii chyba. Premenná `size2` je `private`, pretože špecifikátor dedenia na riadku 20 je prednastavený `private`. Použitie `struct` namiesto `class` môže v niektorých prípadoch zjednodušiť zápis:

```
struct Token {int size = 0;};
```

kde `Token` má len jednu premennú. V prípade `class` je pre rovnaký výsledok potrebné použiť špecifikátor `public`:

```
class Token {public: int size = 0;};
```

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

struct Token {
  int size = 0;
  Token() {}
};

struct Token1: Token { //public dedenie
  int size1 = 0;
  Token1() {}
};

class Token2 {
public:
  int size2 = 0;
  Token2() {}
};

class Token3: Token2 { //private dedenie
public:
  int size3 = 0;
  Token3() {}
};

int main() {
  Token t;
  t.size = 1;

  Token1 t1;
  t1.size1 = 1;

  Token2 t2;
  t2.size2 = 1;

  Token3 t3;
  t3.size2 = 2; //chyba

  cout << std::is_class<Token>::value << endl; //1
  cout << std::is_class<Token1>::value << endl; //1
  cout << std::is_class<Token2>::value << endl; //1
  cout << std::is_class<Token3>::value << endl; //1

  return 0;
}
```

Mimochodom, nasledovný `typedef`:

```
typedef Token struct Token;
```

je v C++ nepotrebný vďaka potrebe kompatibilnej deklarácie s `class`, teda aj bez `typedef` môžeme deklarovať:

```
Token t;
```

21 malloc

Zavolanie `malloc()` na riadku 17 nevytvorí objekt t. j. nezavolá sa konštruktor na riadku 8. `malloc()` alokuje pamäť o veľkosti `sizeof(PT)`. Zavolanie metódy `getStudents()` na riadku 18 má za následok nedefinované správanie, ktoré je špecifické pre rôzne kompilátory. Na vytvorenie objektu potrebujeme použiť `new`!

```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  class PT {
6  public:
7    int students = 1;
8    PT() {
9      students = 2;
10   }
11   int getStudents() {
12     return students;
13   }
14 };
15
16 int main() {
17   PT* pt = (PT*) malloc(sizeof(PT)); //new
18   cout << pt->getStudents();
19
20   return 0;
21 }
```

22 Referencia & a &&

Konštruktor na riadku 3 predpisuje, že jeho vstupný parameter je referencia na `int`. Na riadku 8 je konštruktor volaný s premennou `a`. Na riadkoch 9 a 10 je tento konštruktor volaný s hodnotou `3`. Kompilátor g++ vypíše *"9:13: error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'"*, čo znamená, že kompilátor nevie ako z "*r*-hodnoty" inicializovať referenciu na `int`.

```
1  class Token {
2  public:
3    Token(int& a) {}
4  };
5
```

```
6  int main() {
7    int a = 3;
8    Token t0(a); //OK
9    Token t1(3); //chyba
10   Token t2 = 3; //chyba
11
12   return 0;
13 }
```

Prečo nastáva tento problém? Pozrime sa na nasledujúci kód, z ktorého je zrejmé, že v prípade, ak by takáto inicializácia bola možná, potom nie je zrejmé, ako by sa inkrementovala r-hodnota, teda literál `3`.

```
1  class Token {
2  public:
3    Token(int& i) {
4      ++i; //ako inkremetovat'?
5    }
6  };
7
8  int main() {
9    Token t(3);
10
11   return 0;
12 }
```

Iná situácia nastane, keď vstupný parameter, v našom prípade referencia na `int`, je aj `const`. V tomto prípade nemôže nastať zmena premennej `i`, t. j. mutácia, a preto je takýto kód možné skompilovať.

```
1  class Token {
2  public:
3    Token(const int& i) {} //i je const
4  };
5
6  int main() {
7    int a = 3;
8    Token t0(a); //OK
9    Token t1(3); //OK
10
11   return 0;
12 }
```

Naša séria problémov sa ale týmto nekončí. Vieme skompilovať v oboch prípadoch, ale nevieme medzi týmito dvoma prípadmi rozlišovať. Je zrejmé, že na takéto rozlišovanie potrebujeme druh referencie, ktorý viaže r-hodnotu, ale neviaže l-hodnotu. Tento druh referencie bol zavedený v C++11 a nazýva sa univerzálna referencia. Syntaktický rozdiel oproti "normálnej" referencii je použitie `&&`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
```

```
5  public:
6    Token(int& i) { //& referencia
7      cout << "Token(const␣int&)" << endl;
8    }
9    Token(int&& i) { //&& referencia
10     cout << "Token(const␣int&&)" << endl;
11   }
12 };
13
14 int main() {
15   int a = 3;
16   Token t0(a); //Token(int&)
17   Token t1(3); //Token(int&&)
18
19   return 0;
20 }
```

Referencia na *r*-hodnotu `&&` umožňuje aj to, čo s "normálnou" referenciou nebolo možné, teda mutáciu vstupnej premennej.

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

Mimochodom referencia na referenciu v jazyku C++ neexistuje. Všimnime si medzeru medzi `&` a `&`!

```
int a;
int& & b = a; //chyba
```

Kompilátor g++ vypíše *"error: cannot declare reference to 'int&'"*. Oprava tejto "chyby" nám výrazne nepomôže, pretože premenná `a` je tzv. *l*-hodnota a `int&&` očakáva *r*-hodnotu:

- *l*-hodnota (lvalue = left value) zaberá *identifikovateľné* miesto v pamäti, t. j. niečo s adresou. Napr. premenná `a` je *l*-hodnota, pretože pomocou operátora `&` vieme zistiť jej adresu v pamäti (`&a` vráti adresu premennej `a`).
- *r*-hodnota (rvalue = right value) je všetko, čo nie je *l*-hodnota. *r*-hodnota tiež zaberá miesto v pamäti, ale nie je identifikovateľné.[6]

[6] Viacej o lvalue a rvalue sa môžete dočítať v článku od Scotta Meyersa: Universal References in C++11. `https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers`

Pre úplnosť je potrebné uviesť, že rozdelenie na l-hodnoty a r-hodnoty je výrazne zjednodušené. Po zavedení presúvacej (move) sémantiky v C++11 bolo potrebné zvlášť rozlišovať napr. r-hodnoty, ktoré vznikli pretypovaním, a teda bolo potrebné zaviesť presnejšiu kategorizáciu druhov hodnôt.

Horeuvedené nevylučuje existenciu referencie na smerník. Referencia `&` môže byť referencia na ľubovolný typ a `int*` je typ, a teda aj referencia na `int*` t. j. `int*&` je v poriadku.

```
int* a;
int*& b = a; //OK
```

23 Deštruktor

Deštruktor je vykonaný len pre objekt vytvorený staticky na riadku 14. Pre objekt vytvorený dynamicky na riadku 15 sa nevykoná, pretože tento objekt je potrebné deštruovať pomocou `delete` (pozri riadok 17). Na obrazovku sa vypíše "~Token 0".

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    int id;
7    Token(int id): id(id) {};
8    ~Token() {
9      cout << "~Token " << id;
10   }
11 };
12
13 int main() {
14   Token t0(0);
15   Token* t1 = new Token(1);
16
17   //delete t1;
18   return 0;
19 }
```

Všimnime si, že `delete` nemá žiaden vstupný parameter (príp. parametre), napr. `delete(3, "now")`, pretože by to zbytočne komplikovalo syntax. Deštruktor má zničiť objekt a nič iné. Mimochodom, viacnásobné zavolanie `delete` spôsobí nedefinované správanie (pravdepodobne okamžitú pamäťovú chybu).

```
1  int main() {
2    int* a = new int;
3
4    delete a; //OK
5    delete a; //chyba, nedefinované správanie
6
```

```
7   return 0;
8 }
```

24 Inicializácia premenných tried

Premenná `x` na riadku 6 nie je inicializovaná. Štandard C++11 umožňuje statickú inicializáciu premenných triedy, t.j. umožňuje použitie `int x = 1` na riadku 6, kde v tomto prípade je premenná `x` inicializovaná na hodnotu `1`. Všimnime si, že v prípade, ak na riadku 18 nastane zmena na `Token* t = new Token()`, potom je premenná `x` vynulovaná. Napriek tomu je dobrým zvykom v C/C++ vždy inicializovať premenné! V závislosti od použitého operačného systému a kompilátora je možné, že premenné tried sú vždy vynulované. Dôvodov môže byť viac, napr. neinicializovaná premenná môže obsahovať citlivé informácie iného procesu, ktorý dané pamäťové miesto využíval v minulosti.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5   private:
6    int x; //int x = 1
7
8   public: //len implicitne generovaný konštruktor
9    int get() { //getter
10     return x;
11   }
12   void set(int x = 2) { //setter
13     this->x = x;
14   }
15 };
16
17 int main() {
18   Token* t = new Token; //Token* t = new Token()
19   cout <<  t->get();
20
21   delete t;
22   return 0;
23 }
```

Tento spôsob inicializácie tiež umožňuje inicializáciu pri vytvorení objektu (pred štandardom C++11 to bol jediný spôsob ako inicializovať premennú pri vytvorení objektu). Takto je napr. možné inicializovať premennú `y`, ktorej typ je `const int`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  private:
```

```
6   int x;
7   const int y;
8
9  public:
10  Token(int x0, int y0): x(x0), y(y0) {}
11 };
12
13 int main() {
14  Token t(5,10);
15
16  return 0;
17 }
```

Premennú `y` nie je možné inicializovať po vytvorení objektu, pretože je `const`. Kompilátor g++ vypíše *"12:9: error: assignment of read-only member 'Token::y'"*.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  private:
6    int x;
7    const int y;
8
9  public:
10   Token(int x0, int y0) {
11     this->x = x0;
12     this->y = y0; //chyba
13   }
14 };
15
16 int main() {
17   Token t(5,10);
18
19   return 0;
20 }
```

25 Konverzný konštruktor

Konverzný konštruktor umožňuje inicializáciu aj nasledovným spôsobom:

```
Token t = 1;
```

V prípade, že konštruktor nie je konverzný môžeme inicializovať len priamo:

```
Token t(1);
```

Na riadkoch 11 až 13 sa zavolá konštruktor `Token(int)`, pretože ide o tzv. konverzný konštruktor. V našom prípade pri inicializácii `t0` s 3 sa bude hľadať konštruktor, ktorý má `int` ako vstupný parameter. Na riadku 12 nastane implicitná konverzia na `int`. Na riadku 13 nastane rozšírenie typu `char` na

`int` (type promotion). **Rozšírenie typu na rozdiel od konverzie neznižuje presnosť.**

```
#include <iostream>

class Token {
public:
  Token(int a) {
    std::cout << "Token(int)" << std::endl;
  }
};

int main() {
  Token t0 = 3; //OK
  Token t1 = 3.3; //OK
  Token t2 = 'x'; //OK

  return 0;
}
```

V tomto prípade je konštruktor označený ako `explicit`. g++ vypíše *"11:14: error: conversion from 'int' to non-scalar type 'Token' requested"*.

```
#include <iostream>

class Token {
public:
  explicit Token(int a) {
    std::cout << "Token(int)" << std::endl;
  }
};

int main() {
  Token t0 = 3; //chyba
  Token t1 = 3.3; //chyba
  Token t2 = 'x'; //chyba

  return 0;
}
```

Priamy spôsob inicializácie je naďalej možný. `explicit` teda zakáže len inicializáciu z r-hodnoty.

```
#include <iostream>

class Token {
public:
  explicit Token(int a) {
    std::cout << "Token(int)" << std::endl;
  }
};

int main() {
  Token t0(3); //OK
  Token t1(3.3); //OK
```

```
13   Token t2('x'); //OK
14
15   return 0;
16 }
```

26 Kopírovací konštruktor bez využitia referencie

Tento kód neskompiluje a kompilátor g++ vypíše *"error: invalid constructor; you probably meant 'Token (const Token&)'"*. Kopírovací konštruktor má teda byť deklarovaný ako `Token (const Token&)`, čo je dôležité, aby nedochádzalo ku kopírovaniu premennej `t`, ale k využitiu `t` ako referencie na `t0`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    Token() {}
7    Token(Token t) {
8      cout << "Kopirovaci konstruktor" << endl;
9    }
10 };
11
12 int main() {
13   Token t0;
14
15   return 0;
16 }
```

27 Kopírovací konštruktor a priradenie

Na riadku 20 je dvakrát zavolaný prednastavený (defaultný) konštruktor. Na riadku 21 je zavolaný kopírovací konštruktor. Na riadku 22 je zavolaný preťažený operátor = na riadku 12, t. j. nie je zavolaný kopírovací konštruktor, pretože objekt `t0` už existuje. Kopírovací konštruktor na riadku 21 je zavolaný, pretože objekt `t` ešte neexistuje.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    Token() {
7      cout << "Prednastaveny konstruktor" << endl;
8    }
9    Token(const Token& t) {
10     cout << "Kopirovaci konstruktor" << endl;
11   }
```

```
12    Token& operator=(const Token& t) {
13      cout << "Priradenie" << endl;
14
15      return *this;
16    }
17  };
18
19  int main() {
20    Token t0, t1;
21    Token t = t0;
22    t0 = t1;
23
24    return 0;
25  }
```

28 Kopírovací konštruktor a funkcia

Na riadku 15 sa zavolá kopírovací konštruktor, pretože premenná `t0` je vytvorená prekopírovaním obsahu premennej `t`, ktorá je vstupným parametrom funkcie `foo()`. Funkcia `foo()` vráti `t0` a následovne nastane prekopírovanie `t0` do premennej `t1`. Takže nastane zavolanie kopírovacieho konštruktora dvakrát. Po zmene `Token t0` na `Token& t0` (pozri riadok 14) je kopírovací konštruktor stále zavolaný na riadku 21. Zavolať kopírovací konštruktor len raz je lepšie ako dvakrát; ešte lepšie riešenie ponúka presúvacia (move) sémantika.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   public:
6     Token() {
7       cout << "Token()" << endl;
8     }
9     Token(const Token& t) {
10      cout << "Token(const␣Token&)" << endl;
11    }
12  };
13
14  //Token& foo(Token& t0) {
15  Token foo(Token t0) { //Token(const Token&)
16    return t0;
17  }
18
19  int main() {
20    Token t; //Token()
21    Token t1 = foo(t); //Token(const Token&)
22
23    return 0;
24  }
```

29 Chýbajúci prednastavený konštruktor

Každá písomná skúška obsahuje aspoň jeden príklad na chýbajúci prednastavený (defaultný) konštruktor.

Prednastavený konštruktor je konštruktor, ktorý môže byť zavolaný bez parametrov, napr. `Token()`. Chyba nastane na riadku 8, pretože neexistuje prednastavený konštruktor.

```
1  class Token {
2  public:
3    int size = 0;
4    Token(const Token& t) {} //kopírovací konštruktor
5  };
6
7  int main() {
8    Token t; //chyba
9    Token t1 = t;
10
11   return 0;
12 }
```

V prípade, že neexistuje žiaden konštruktor vygeneruje kompilátor implicitný prednastavený konštruktor.

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

Všimnime si, že chyba nastane aj na riadku 18, pretože odvodená trieda `Token1` tiež vyžaduje prednastavený konštruktor triedy `Token`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    int size = 0;
7    Token(int s) {}
8  };
9
10 class Token1: public Token {
11 public:
12   int size = 0;
13   Token1() {}
14 };
15
16 int main() {
```

```
17    Token t; //chyba
18    Token1 mT; //chyba
19
20    return 0;
21  }
```

30 Odvodená trieda

Tento kód nie je možné skompilovať, pretože trieda `Planet` neobsahuje prednastavený konštruktor `Planet()`. A v konštruktore triedy `Mars` nie je uvedené, ktorý konštruktor triedy `Planet` sa má použiť (pri neexistencii prednastaveného konštruktora). Pri kompilácii kompilátorom g++ sa na obrazovku vypíše *"20:17: error: no matching function for call to 'Planet::Planet()'"*, t. j. na riadku 20 pri kompilácii triedy `Mars` nebolo možné nájsť potrebný prednastavený konštruktor `Planet()`. Všimnime si, že v tomto prípade trieda `Planet` obsahuje konštruktor `Planet(int)`, avšak pre úspešnú kompiláciu je potrebný konštruktor `Planet()` a nie konštruktor s rovnakým vstupným parametrom.

```
1   #include <iostream>
2   using namespace std;
3
4   class Planet {
5   public:
6     int size = 0;
7     int color = 1;
8
9     Planet(int size) {
10      this->size = size;
11    }
12  };
13
14  class Mars: public Planet {
15  private:
16    long time = 1001010;
17    int galaxy = 16;
18
19  public:
20    Mars(int size) {
21      this->size = size;
22    }
23    int getColor() {
24      return color;
25    }
26  };
27
28  int main() {
29    Mars* mars = new Mars(10);
30    cout << mars->getColor();
31
```

```
32    delete mars;
33    return 0;
34  }
```

Trieda `Planet` v tomto prípade obsahuje prednastavený konštruktor `Planet()`. Na obrazovku sa vypíše 1. Všimnime si riadok 35, ktorý zabezpečuje dealokáciu vytvoreného objektu. Každý `new` musí mať zodpovedajúci `delete` tak, aby pri dlhom behu kódu nenastali problémy s nedostatkom pamäte.

```
1   #include <iostream>
2   using namespace std;
3
4   class Planet {
5   public:
6     int size = 0;
7     int color = 1;
8
9   protected:
10    Planet() {
11      this->size = size;
12    }
13  };
14
15  class Mars: public Planet {
16  private:
17    long time = 1001010;
18    int galaxy = 16;
19
20  public:
21    Mars(int& size) { //referencia int&
22      this->size = size;
23    }
24
25    int getColor() {
26      return color;
27    }
28  };
29
30  int main() {
31    int a = 10;
32    Mars* mars = new Mars(a);
33    cout << mars->getColor();
34
35    delete mars;
36    return 0;
37  }
```

Nasledujúce dve úlohy sú variácie predošlých a boli zadané na opravnom termíne. Tento kód nie je možné skompilovať, pretože trieda `Planet` neobsahuje prednastavený konštruktor `Planet()`.

```
1   #include <iostream>
```

```
2  using namespace std;
3
4  class Planet {
5  public:
6    int size = 0;
7    int color = 1;
8
9    Planet(int size) {
10     this->size = size;
11   }
12 };
13
14 class Mars: public Planet {
15 private:
16   long time = 1001010;
17   int galaxy = 16;
18
19 public:
20   Mars(int size, int color) {
21     this->size = size;
22     this->color = color;
23   }
24   int getColor() {
25     return color;
26   }
27 };
28
29 int main() {
30   Mars* mars = new Mars(10, 1);
31   cout << mars->getColor();
32
33   delete mars;
34   return 0;
35 }
```

Trieda `Planet` obsahuje prednastavený konštruktor `Planet()`. Na obrazovku sa vypíše 1.

```
1  #include <iostream>
2  using namespace std;
3
4  class Planet {
5  public:
6    int size = 0;
7    int color = 1;
8
9    Planet() {
10     this->size = size;
11   }
12 };
13
14 class Mars: public Planet {
15 private:
16   long time = 1001010;
```

```
17    int galaxy = 16;
18
19  public:
20    Mars(int size) {
21      this->size = size;
22    }
23
24    int getColor() {
25      return color;
26    }
27  };
28
29  int main() {
30    int a = 10;
31    Mars* mars = new Mars(a);
32    cout << mars->getColor();
33
34    delete mars;
35    return 0;
36  }
```

31 Chýbajúci prednastavený konštruktor ešte raz

Tento kód nie je možné skompilovať, pretože trieda `STU` neobsahuje prednastavený konštruktor `STU()`. Pri kompilácii kompilátorom g++ sa na obrazovku vypíše *"26:29: error: no matching function for call to 'STU::STU()'"*, t. j. pri kompilácii triedy `FEI` na riadku 26 nebolo možné v triede `STU` nájsť potrebný prednastavený konštruktor `STU()`.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class STU {
6   private:
7     string rektor;
8     string prorektor;
9
10  public:
11    STU(string rektor) {
12      this->rektor = rektor;
13    }
14    string getRektor(string rektor = "Redhammer") {
15      this->rektor = rektor;
16      return this->rektor;
17    }
18  };
19
20  class FEI: public STU {
21  private:
22    string dekan;
23    string prodekan;
```

```
24
25 public:
26   FEI(string dekan = "Oravec") {
27     this->dekan = dekan;
28   }
29   string getDekan() {
30     return this->dekan;
31   }
32 };
33
34 int main() {
35   FEI fei;
36
37   cout << fei.getDekan();
38
39   return 0;
40 }
```

Je to ako v tom westerne: "zopakujem ešte raz, na prvýkrát sa to dobre nepochopí".[7] Trieda `Token` nemá prednastavený konštruktor a tak tento kód nie je možné skompilovať. Kompilátor g++ vypíše *"24:33: error: no matching function for call to 'Token::Token()'"*, t. j. na riadku 24 pri kompilácii triedy `Token1` nebolo možné nájsť v triede `Token` prednastavený konštruktor `Token()`.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Token {
6  private:
7    string s;
8
9  public:
10   Token(string s) {
11     this->s = s;
12   }
13   string getString(string s = "I␣love␣PT!") {
14     this->s = s;
15     return this->s;
16   }
17 };
18
19 class Token1: public Token {
20 private:
21   string s;
22
23 public:
24   Token1(string s = "I␣love␣C++!") {
25     this->s = s;
26   }
```

[7] Malý unavený Joe (...continuavano a chiamarlo Trinità, v tal. orig.), 1972

```
27    string getString() {
28      return this->s;
29    }
30  };
31
32  int main() {
33    Token1 token;
34
35    cout << token.getString();
36
37    return 0;
38  }
```

32 = default

Na riadku 13 nastane chyba, pretože neexistuje prednastavený konštruktor.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   public:
6     Token(int a) {
7       cout << "Token(int)" << endl;
8     }
9   };
10
11  int main() {
12    Token t0(3); //OK
13    Token t1; //chyba
14
15    return 0;
16  }
```

V tomto prípade kompilátor vygeneruje prednastavený konštruktor. Na riadku 6 je konštruktor `Token()` označený ako `=default`, čo znamená, že tento konštruktor musí kompilátor vygenerovať. Dôvod prečo prednastavený konštruktor nebol vygenerovaný je pravidlo, podľa ktorého sa prednastavený konštruktor vygeneruje, len ak neexistuje žiadny iný konštruktor.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   public:
6     Token() = default;
7     Token(int a) {
8       cout << "Token(int)" << endl;
9     }
10  };
11
12  int main() {
```

```
13    Token t0(3); //OK
14    Token t1; //OK
15
16    return 0;
17  }
```

33 Preťažený operátor ! v odvodenej triede s rovnakým menom

Trieda `STU` obsahuje prednastavený konštruktor na riadku 10. Otázkou ostáva, či bude použitý preťažený operátor ! na riadku 15, alebo na riadku 28. Bude použitý preťažený operátor na riadku 28 v odvodenej triede a na obrazovku sa vypíše "Redhammer!". Metódy v odvodenej triede prekryjú metódy s identickým názvom v materskej triede.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class STU {
6   protected:
7     string rektor;
8
9   public:
10    STU() {}
11    string getRektor(string rektor = "Redhammer") {
12      this->rektor = rektor;
13      return this->rektor;
14    }
15    string operator!() {
16      return (this->rektor).append(".");
17    }
18  };
19
20  class FEI: public STU {
21  private:
22    string dekan;
23
24  public:
25    FEI(string dekan = "Oravec") {
26      this->dekan = dekan;
27    }
28    string operator!() {
29      return (this->rektor).append("!");
30    }
31  };
32
33  int main() {
34    FEI fei;
35    string s = fei.getRektor();
36
```

```
37    cout << !fei;
38
39    return 0;
40  }
```

34 Prednastavený konštruktor s implicitnou hodnotou

Po kompilácii a spustení sa na obrazovku vypíše "I love C++!". Trieda `Token` má prednastavený konštruktor s implicitnou hodnotou. Všimnime si, že trieda `Token1` obsahuje premennú `s`, ktorá sa prekrýva s premennou `s` triedy `Token`. Na obrazovku sa vypíše "I love C++!", pretože je zavolaná metóda `getString()` triedy `Token1` na riadku 27.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   class Token {
6   private:
7     string s;
8
9   public:
10    Token(string s = "I␣love␣C!") {
11      this->s = s;
12    }
13    string getString(string s = "I␣love␣PT!") {
14      this->s = s;
15      return this->s;
16    }
17  };
18
19  class Token1: public Token {
20  private:
21    string s;
22
23  public:
24    Token1(string s = "I␣love␣C++!") {
25      this->s = s;
26    }
27    string getString() {
28      return this->s;
29    }
30  };
31
32  int main() {
33    Token1 token;
34
35    cout << token.getString();
36    return 0;
37  }
```

35 Private a protected dedenie

Špecifikátory prístupu `private`, `protected` a `public` pre triedy:

- `class Token1: public Token {};` Trieda `Token1` zdedí všetky `public` a `protected` členy triedy `Token`.
- `class Token1: protected Token {};` Trieda `Token1` zdedí všetky `public` a `protected` členy triedy `Token`. Zdedené `public` členy sa zdedia ako `protected`.
- `class Token1: private Token {};` Trieda `Token1` zdedí všetky `public` a `protected` členy triedy `Token`. Zdedené `public` a `protected` členy sa zdedia ako `private`.

Premenná `x` je v triede `Token1` zdedená ako `private`, a teda v triede `Token2` nie je zdedená. Kompilátor g++ vypíše *"23:12: error: 'int Token::x' is inaccessible within this context"*.

```
1  class Token {
2  private:
3    int time = 0;
4
5  public:
6    int x = 0;
7    Token(int t): time(t) {};
8  };
9
10 class Token1: private Token {
11 private:
12   int size = 0;
13
14 public:
15   Token1(int s, int t): Token(t), size(s) {
16     x = 10;
17   }
18 };
19
20 class Token2: public Token1 {
21 public:
22   int getX() {
23     return x; //chyba
24   }
25 };
26
27 int main() {
28   return 0;
29 }
```

Po zmene na `protected` na riadku 10 kód skompiluje. Premenná `x` je zdedená ako `protected` v triede `Token2`.

```
1  class Token {
2  private:
3    int time = 0;
4
5  public:
6    int x = 0;
7    Token(int t): time(t) {};
8  };
9
10 class Token1: protected Token {
11 private:
12   int size = 0;
13
14 public:
15   Token1(int s, int t): Token(t), size(s) {
16     x = 10;
17   }
18 };
19
20 class Token2: public Token1 {
21 public:
22   int getX() {
23     return x; //OK
24   }
25 };
26
27 int main() {
28   return 0;
29 }
```

36 Private premenná

Kód neskompiluje, pretože premenná `getMeno` je `private` a na riadku 24 sa k nej pokúšame pristúpiť. Všimnime si, že funkcie a metódy majú vždy () a teda `getMeno` nemôže byť funkcia alebo metóda. Kompilátor g++ vypíše *"12:3: error: 'std::__cxx11::string Auto::getMeno()' conflicts with a previous declaration"*.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Auto {
6   private:
7    string getMeno;
8
9   public:
10   string getMeno() {
11     return this->getMeno;
12   }
13   void setMeno(string _meno) {
14     this->getMeno = _meno;
```

```
15    }
16  };
17
18  int main() {
19    Auto* avto = new Auto();
20
21    avto->setMeno("Mercedes");
22    avto->setMeno("Lada");
23
24    cout <<  avto->getMeno << endl;
25
26    return 0;
27  }
```

Tento kód neskompiluje, pretože na riadku 22 sa pokúšame použiť premennú `a`, ktorá je `private` premenná triedy `Token`. Trieda `Token` nemá prednastavený konštruktor, ale na riadku 21 presne špecifikujeme, ktorý konštruktor triedy `Token` je potrebné využiť. Kompilátor g++ vypíše *"6:7: error: 'int Token::a' is private"*.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   private:
6     int a = 1;
7   protected:
8     int b = 1;
9   public:
10    int c = 1;
11    Token(int x, int y, int z) {
12      this->a = x;
13      this->b = y;
14      this->c = z;
15    }
16  };
17
18  class Token1: protected Token {
19  public:
20    int result;
21    Token1(int x, int y, int z): Token(x,y,z) {
22      this->result = a * b * c;
23      cout << this->result;
24    }
25  };
26
27  int main() {
28    Token1 t(0,10,100);
29
30    return 0;
31  }
```

37 Private konštruktor a špecifikátor protected

V tomto príklade nastane prekrytie premenných `a` v triede `Token1` a `Token`. Navyše je trieda `Token1` na riadku 9 odvodená pomocou špecifikátora `protected`. Aký to má vplyv? Žiadny! Kód neskompiluje, pretože konštruktor `Token1()` na riadku 13 je `private`. Táto úloha je čiastočne založená na neistote, či po odriadkovaní na riadku 12 je naďalej zachovaná špecifikácia `private` z riadku 10. `private` konštruktory sa využívajú (najmä pred zavedením `=delete` v C++11), ako spôsob znefunkčnenia konštruktora bez toho, aby bol kód úplne odstránený. Z vnútra triedy je `private` konštruktor stále možné zavolať, takže využitie `=delete` je v mnohých prípadoch vhodnejšie.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  private:
6    int a;
7  };
8
9  class Token1: protected Token {
10 private:
11   int a;
12
13   Token1(int a = 10) {
14     cout << a;
15   }
16 };
17
18 int main() {
19   Token t;
20   Token1 tt;
21
22   return 0;
23 }
```

38 = delete

Na riadku 8 nastane implicitná konverzia `3.3` na `int`, a preto sa zavolá konštruktor na riadku 3.

```
1  class Token {
2  public:
3    Token(int a) {}
4  };
5
6  int main() {
7    Token t0(3);
8    Token t1(3.3);
9
10   return 0;
```

```
11 }
```

V tomto prípade je konštruktor `Token(double)` označený ako `=delete`. Tento konštruktor nie je možné zavolať, a teda na riadkoch 9 a 10 nastane chyba. Kompilátor g++ vypíše *"9:15: error: use of deleted function 'Token::Token(double)'"*. Ako `=delete` je možné označiť ľubovolnú metódu triedy, nielen konštruktor.

```
1  class Token {
2  public:
3    Token(int a) {}
4    Token(double a) = delete;
5  };
6
7  int main() {
8    Token t0(3); //OK
9    Token t1(3.3); //chyba
10   Token t2(3.3f); //chyba
11
12   return 0;
13 }
```

V tomto prípade sú ako `=delete` označené kopírovací konštruktor a operátor priradenia. Objekty typu `Token` nie je možné kopírovať ani priraďovať, čo môže byť praktické napr. v prípade, ak nie je implementované hlboké kopírovanie.

```
1  class Token {
2  public:
3    Token() {}
4    Token(const Token& t) = delete;
5    Token& operator=(const Token& t) = delete;
6  };
7
8  int main() {
9    Token t0, t1; //OK
10   Token t2 = t0; //chyba
11   t1 = t0; //chyba
12
13   return 0;
14 }
```

39 Virtuálny deštruktor

Pri zavolaní `delete` na riadku 29 nastane len zavolanie deštruktora triedy `Token` a nie deštruktora triedy `Token1`. Pole `array` teda ostane v pamäti. Na odstránenie tohto problému je potrebné deklarovať deštruktor triedy `Token` ako `virtual`. Problém nastal, keď sme adresu vytvoreného objektu typu `Token1` uložili do premennej typu `Token*`. `delete` na riadku 29 vykoná presne, čo sa žiada a zavolá len deštruktor na riadku 9.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    //virtual ~Token() {
7    //  cout << "~Token()";
8    //}
9    ~Token() {
10     cout << "~Token()";
11   }
12 };
13
14 class Token1: public Token {
15 public:
16   int* array;
17   Token1() {
18     array = new int[100000000];
19   }
20   ~Token1() {
21     delete array;
22     cout << "~Token1()";
23   }
24 };
25
26 int main() {
27   Token* t = new Token1;
28
29   delete t;
30   return 0;
31 }
```

40 >> je operátor

Kompilátor g++ vypíše *"error: '>>' should be '> >' within a nested template argument list"*. Problém tu nastáva so zámenou >> s možným preťaženým operátorom príp. bitovým posunom, pričom v tomto prípade sa jedná o vnorenie typu `std::vector<int>` do `std::stack<>`. Na riadku 6 je vnorenie správne zapísané s medzerou "> >". Tento problém je špecifický pre niektoré kompilátory ako napr. g++.

```
1  #include <stack>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6    stack<vector<int> >* myv0 = new stack<vector<int> >;
7    stack<vector<int>>* myv1 = new stack<vector<int>>;
8
9    return 0;
10 }
```

41 Typová nezhoda: jednoduchý príklad

Kód neskompiluje, pretože `v` má typ `vector<int>` a na riadku 10 sa pokúšame vložiť položku s typom `int*`.

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6    vector<int> v;
7    int* a = new int();
8    int* b = new int();
9
10   v.push_back(a);
11   v.push_back(b);
12   cout << v.at(0);
13
14   delete a;
15   delete b;
16   return 0;
17 }
```

Tento kód je variácia predchádzajúceho príkladu. Kód skompiluje a na obrazovku sa vypíše adresa vložená do `v`.

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6    vector<int*> v;
7    int* a = new int();
8    int* b = new int();
9
10   v.push_back(a);
11   v.push_back(b);
12   cout << v.at(0);
13
14   delete a;
15   delete b;
16   return 0;
17 }
```

42 Typová nezhoda: zložitejší príklad

Premenná `l` je typu `list<int*>*`. Vektor `v` predpokladá položky typu `list<int*>`. Tento kód neskompiluje pre typovú nezhodu na riadku 9. Kompilátor g++ vypíše *"9:16: error: no matching function for call to 'std::vector<std::__cxx11::list<int*>*

>::push_back(std::__cxx11::list<int>*&)'"*, t. j. "hľadal som typ `list<int*>`, ale našiel som iný typ".

```
1  #include <iostream>
2  #include <vector>
3  #include <list>
4  using namespace std;
5
6  int main() {
7    vector<list<int*> > v;
8    list<int*>* l = new list<int*>();
9    v.push_back(l);
10
11   cout << (v.at(0)).front();
12
13   delete l;
14   return 0;
15 }
```

Tento kód nie je možné skompilovať, pretože na riadku 8 má premenná `m` typ `map<int, list<string>*>*`, ale `new` sa pokúša vytvoriť objekt typu `map<int, list<string> >`. Kód pod riadkom 8 je už len dekorácia.

```
1  #include <map>
2  #include <list>
3  #include <string>
4  #include <iostream>
5  using namespace std;
6
7  int main() {
8    map<int, list<string>*>* m = new map<int, list<string> >();
9    list<string>* l = new list<string>();
10   l->push_back("I love PT!");
11   (*m)[0] = l;
12
13   cout << (m->at(0))->front();
14   return 0;
15 }
```

43 Návrat adresy lokálnej premennej

Kód skompiluje s varovaním *"8:20: warning: address of local variable 'm' returned [-Wreturn-local-addr]"*, pretože na riadku 11 vraciame adresu lokálnej premennej `m`. Pri pokuse o dereferencovanie na riadku 17 nastane pamäťová chyba (segmentation fault).

```
1  #include <map>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5
6  map<int, string>* insert(string s) {
```

```
7   static int i = 0;
8   map<int, string> m;
9   m[i] = s;
10  ++i;
11  return &m;
12 }
13
14 int main() {
15  map<int, string>* m;
16  m = insert("I love PT!");
17  cout << (*m)[0];
18
19  return 0;
20 }
```

Kód neskompiluje, pretože na riadku 17 je pokus pristúpiť k položke 0, avšak `m` je typu `map<int, string>*`, teda smerník. Všimnime si, že aj v tomto prípade kompilátor g++ varuje, že `m` je lokálna premenná.

```
1  #include <map>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5
6  map<int, string>* insert(string s) {
7   static int i = 0;
8   map<int, string> m;
9   m[i] = s;
10  ++i;
11  return &m;
12 }
13
14 int main() {
15  map<int, string>* m;
16  m = insert("I love PT!");
17  cout << m[0];
18  return 0;
19 }
```

44 STL set

Obsah `v` je vložený do `s`, ktorý má typ `set<long>`. Pretože sa jedná o set, vložené položky sa zoradia podľa veľkosti a duplikátne položky nie sú vložené. Na obrazovku sa vypíše "1 7 19 100 209".

```
1  #include <set>
2  #include <vector>
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7   vector<long> v = {100, 1, 7, 19, 209, 1};
```

```
8   set<long> s;
9
10  for(long i: v) {
11    s.insert(i);
12  }
13
14  for(long i: s) {
15    cout << i << "␣";
16  }
17
18  return 0;
19 }
```

Tento príklad je podobný predchádzajúcemu príkladu. Na obrazovku sa vypíše "1 7 19 100 209". Všimnime si, že premenná `i` vo `for` cykloch na riadkoch 10 a 14, je typu `long&`, t. j. v tomto prípade je premenná `i` referencia. To znamená, že nenastane prekopírovanie hodnoty z vektora `v`.

```
1  #include <set>
2  #include <vector>
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7    vector<long> v = {100, 1, 7, 19, 209, 1};
8    set<long> s;
9
10   for(long& i: v) {
11     s.insert(i);
12   }
13
14   for(const long& i: s) {
15     cout << i << "␣";
16   }
17
18   return 0;
19 }
```

45 Chytrý smerník

Čo sa vypíše na obrazovku? Vypíše sa "5 5 5". `shared_ptr<>` spočítava počet svojich kópii. Na riadku 9 sa vytvorí prvý objekt typu `shared_ptr<int>`, t. j. prvá kópia. Na riadkoch 11 až 13 sa vložením do vec vytvoria ďalšie 3 kópie. Vo `for` cykle na riadku 15 sa do premennej `item` vloží ďalšia kópia. `use_count()` vráti počet kópii a tých je 5, `for` cyklus zbehne 3-krát, pretože vo `vec` sú 3 položky.

```
1  #include <iostream>
2  #include <memory>
3  #include <vector>
4  using namespace std;
```

```
5
6  int main() {
7    vector<shared_ptr<int> > vec;
8
9    shared_ptr<int> p = make_shared<int>();
10
11   vec.push_back(p);
12   vec.push_back(p);
13   vec.push_back(p);
14
15   for(auto item: vec) {
16     cout<< item.use_count() <<"␣";
17   }
18
19   return 0;
20 }
```

Čo sa vypíše na obrazovku? Vypíše sa "4 4 4". V tomto prípade nedôjde k prekopírovanie do premennej `item`, pretože táto je referencia na už existujúci objekt typu `shared_ptr<int>`.

```
1  #include <iostream>
2  #include <memory>
3  #include <vector>
4  using namespace std;
5
6  int main() {
7    vector<shared_ptr<int> > vec;
8
9    shared_ptr<int> p = make_shared<int>();
10
11   vec.push_back(p);
12   vec.push_back(p);
13   vec.push_back(p);
14
15   for(auto& item: vec) {
16     cout<< item.use_count() <<"␣";
17   }
18
19   return 0;
20 }
```

Na obrazovku sa vypíše "2 2". Všimnime si, že premenná `item` na riadku 25 je lokálna premenná, teda pri iterácii cez vektor `vec` nastane prekopírovanie hodnôt tohto vektora do tejto premennej, čo má za následok zvýšenie počtu kópií každého `shared_ptr<>` (inkrementuje sa `use_count`). Tiež si všimnime, že premenná `myPtr` na riadku 21 je lokálna premenná a po ukončení `for` cyklu zanikne.

```
1  #include <memory>
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
```

```
6
7  class PT{
8  private:
9    int students;
10
11 public:
12   PT() {
13     students = 0;
14   }
15 };
16
17 int main() {
18   vector<shared_ptr<PT> > vec;
19
20   for(int i = 0; i < 2 ; i++) {
21       shared_ptr<PT> myPtr = make_shared<PT>();
22       vec.push_back(myPtr);
23   }
24
25   for(auto item: vec) {
26       cout<< item.use_count() <<"␣"; //2 2
27   }
28
29   return 0;
30 }
```

Na obrazovku sa vypíše "1 1". Tento príklad je podobný ako predchádzajúci s tým rozdielom, že na riadku 25 je premenná `item` referencia. To znamená, že hodnoty vektora `vec` nie sú kopírované, ale referencované, čo nemá za následok zvýšenie `use_count`.

```
1  #include <memory>
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  class PT{
8  private:
9    int students;
10
11 public:
12   PT() {
13     students = 0;
14   }
15 };
16
17 int main() {
18   vector<shared_ptr<PT> > vec;
19
20   for(int i = 0; i < 2 ; i++) {
21       shared_ptr<PT> myPtr = make_shared<PT>();
22       vec.push_back(myPtr);
23   }
```

```
24
25    for(auto& item: vec) {
26        cout<< item.use_count() <<"␣"; //1 1
27    }
28
29    return 0;
30  }
```

46 catch

Kód skompiluje, ale na riadku 13 pristupujeme k neexistujúcemu prvku, čo má za následok nedefinované správanie, teda čokoľvek sa môže stať v závislosti od použitého kompilátora. Ak by sme riadok 13 modifikovali na `v.at(3)`, potom by nastala výnimka typu `std::out_of_range`, ktorú by sme mohli zachytiť.

```
1   #include <iostream>
2   #include <string>
3   #include <vector>
4   using namespace std;
5
6   int main() {
7     vector<string> v;
8     v.push_back("Imro");
9     v.push_back("Fero");
10    v.push_back("");
11
12    try{
13      cout <<  v[3];
14    } catch(...) { //zachytí všetky C++ výnimky
15      cout <<  v[1];
16    }
17
18    return 0;
19  }
```

Zachytávanie viacerých výnimiek je možné, pričom `catch(...)` musí byť posledné v poradí.

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main() {
6     vector<int> v(10);
7
8     try{
9       int a = v.at(100);
10    } catch(const std::out_of_range& e) {
11      cout << "catch(const␣std::out_of_range&)" << endl;
12    } catch(const std::bad_alloc& e) {
13      cout << "catch(std::bad_alloc&)" << endl;
```

```
14    } catch(...) {
15      cout << "catch(...)" << endl;
16    }
17
18    return 0;
19  }
```

Použitie `catch(...)` je vhodné najmä pri skúšaní jazyka C++, pretože nie je možné predpokladať, že si niekto pamätá akú C++ výnimku môže daná entita vyhodiť. V reálnych prípadoch je odporúčané zachytávať špecifické výnimky tak, aby sa dal pre ne napísať špecifický obslužný kód. Ak zachytíme výnimku pomocou `catch(...)`, aký druh výnimky sme zachytili?

47 catch: spočítaj počet položiek

Na obrazovku sa vypíše "2 1 1 1 1", pretože v obsahuje duplicitnú položku 1 a ďalšie 4 položky. Na riadku 12 sa vždy pokúsime vložiť položku z `v` do `m`. Ak táto položka do `m` ešte nebola vložená spôsobí to výnimku typu `std::out_of_range`. Táto výnimka je zachytená pomocou `catch(...)`, ktorý zachytí všetky výnimky. Táto položka sa vloží na riadku 14. Ak sa pokúsime vložiť položku, ktorú `m` už obsahuje, výnimka nenastane a inkrementuje sa hodnota `m` pre túto položku. Tento kód funguje ako jednoduché počítadlo duplicity položiek.

```
1   #include <map>
2   #include <vector>
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7     vector<int> v = {100, 1, 7, 19, 209, 1};
8     map<int, int> m;
9
10    for(int i: v) {
11      try{
12        ++m.at(i);
13      } catch(...) {
14        m[i] = 1;
15      }
16    }
17
18    for(auto &i: m) {
19      cout << i.second << "␣";
20    }
21
22    return 0;
23  }
```

48 static_cast<>, reinterpret_cast<>

Na rozdiel od C castu na riadku 7, `static_cast<>` nevie pretypovať `t1`, pretože trieda Token0 nie je odvodená od triedy Token1. Kompilátor g++ vypíše *"8:31: error: invalid static_cast from type 'Token1*' to type 'Token0*'"*.

```
1  class Token0 {};
2  class Token1 {};
3
4  int main() {
5    Token0* t0;
6    Token1* t1 = new Token1;
7    t0 = (Token0*) t1; //OK
8    t0 = static_cast<Token0*>(t1); //chyba
9
10   delete t1;
11   return 0;
12 }
```

V tomto prípade je pretypovanie úspešné, pretože trieda `Token1` je odvodená od triedy `Token0`. Všimnime si, že pretypovanie je uskutočnené z odvodenej triedy na základnú triedu. V prípade opačného pretypovania, teda zo základnej triedy na odvodenú môže nastať vytvorenie neúplného objektu, t. j. kompilátor vám dôveruje, že takéto pretypovanie je bezpečné, počas kompilácie overí, že `Token1` je odvodená triedy `Token0`, ale nedokáže overiť, či objekt v pamäti je naozaj platný objekt typu `Token1`. Takéto overenie počas behu kódu je možné pomocou `dynamic_cast<>`, ktorý je ale menej efektívny ako `static_cast<>`.

```
1  class Token0 {};
2  class Token1: public Token0 {};
3
4  int main() {
5    Token0* t0;
6    Token1* t1 = new Token1;
7    t0 = (Token0*) t1; //OK
8    t0 = static_cast<Token0*>(t1); //OK
9
10   delete t1;
11   return 0;
12 }
```

V prípade, že potrebujeme pretypovať objekty, keď trieda `Token1` nie je odvodená od triedy `Token0`, potom je potrebné použiť `reinterpret_cast<>`. Programátor takto zreteľne vyjadrí svoj zámer pretypovať takéto rozdielne objekty.

```
1  class Token0 {};
2  class Token1 {};
3
4  int main() {
5    Token0* t0;
6    Token1* t1 = new Token1;
```

```
7    t0 = (Token0*) t1; //OK
8    t0 = reinterpret_cast<Token0*>(t1); //OK
9
10   delete t1
11   return 0;
12 }
```

49 const_cast<>

const_cast<> je pretypovanie s odstránením const. Po použití const_cast<> na riadku 11 získame ptr1, ktorý má typ int*, a ktorý ukazuje na to isté pamäťové miesto ako ptr0. Všimnime si, že premennú s typom int* potrebujeme na zavolanie funkcie foo(int*). Pokus zmeniť hodnotu na riadku 14 je ale nedefinované správanie, pretože premenná val je const. Nedefinované správanie znamená, že cout na riadku 15 môže, ale nemusí vždy vypísať 30.

```
1  #include <iostream>
2  using namespace std;
3
4  int foo(int* ptr) {
5    return (*ptr + 10);
6  }
7
8  int main() {
9    const int val = 10;
10   const int* ptr0 = &val;
11   int* ptr1 = const_cast <int*>(ptr0); //OK
12   cout << foo(ptr1);
13
14   *ptr1 = 30; //nedefinované správanie
15   cout << *ptr1 << endl; //?
16
17   return 0;
18 }
```

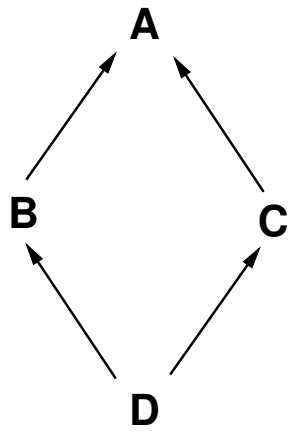
const_cast<> nemôže byť použitý na odstránenie const a zároveň na pretypovanie na iný typ. V tomto prípade vypíše kompilátor g++ *"11:38: error: invalid const_ cast from type 'const int*' to type 'long int*'"*.

```
1  #include <iostream>
2  using namespace std;
3
4  int foo(int* ptr) {
5      return (*ptr + 10);
6  }
7
8  int main() {
9    const int val = 10;
10   const int *ptr0 = &val;
11   int* ptr1 = const_cast <long*>(ptr0); //chyba
12   cout << foo(ptr1);
13
```

```
14    return 0;
15  }
```

50 Diamantové dedenie a virtuálne dedenie

V tomto prípade sa jedná o tzv. diamantové dedenie. Triedy `B` a `C` dedia z triedy `A`. Trieda `D` dedí z tried `B` a `C`. Pri dedení zdedia triedy `B` a `C` premennú `a`. Pri dedení triedy `D` z tried `B` a `C` nie je jasné, či sa má zdediť premenná `a` z triedy `B` alebo `C`. Kompilátor g++ vypíše "27:15: error: request for member 'a' is ambiguous".



Obr. 1. Diamantové dedenie

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    int a;
7  };
8
9  class B: public A {
10 public:
11   int b;
12 };
13
14 class C: public A {
15 public:
16   int c;
17 };
18
19 class D: public B,C {
20 public:
21   int d;
22 };
```

```
23
24 int main() {
25   D ddd;
26
27   cout << ddd.a << endl;
28
29   return 0;
30 }
```

Aby sme sa vyhli problému s diamantovým dedením je potrebné tzv. virtuálne dedenie. V tomto prípade zdedí trieda `D` len jednu kópiu premennej `a`.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    int a;
7  };
8
9  class B: virtual public A {
10 public:
11   int b;
12 };
13
14 class C: virtual public A {
15 public:
16   int c;
17 };
18
19 class D: public B,C {
20 public:
21   int d;
22 };
23
24 int main() {
25   D ddd;
26
27   cout << ddd.a << endl;
28
29   return 0;
30 }
```

51 auto a decltype

Premenná `start` má typ `std::vector<int>::iterator`, ktorý bol odvodený z metódy `begin()` t. j. `begin()` vracia iterátor typu `std::vector<int>::iterator`. Premenná `a` má typ `int`, ktorý bol odvodený z literálu 1. Premenná `b` má taký istý typ ako `a`. Typ premennej `c` nie je možné odvodiť, pretože premenná `c` nie je inicializovaná. Kompilátor

g++ vypíše *"11:8: error: declaration of 'auto c' has no initializer"*.

```
1  #include <vector>
2  using namespace std;
3
4  int main() {
5    vector<int> v;
6    auto start = v.begin(); //std::vector<int>::iterator
7
8    auto a = 1;
9    decltype(a) b; //b je typu int
10
11   auto c; //chyba
12
13   return 0;
14 }
```

Od C++14 je možné auto použiť aj na odvodenie typu návratovej hodnoty funkcie. Kompilátor v tomto prípade odvodí, že návratový typ funkcie `foo()` je `std::size_t`, ktorý pre mnohé kompilátory je alias (typedef) pre `unsigned int`, alebo `long unsigned int`. V prípade, že použijete prepínač -Wconversion vypíše kompilátor g++ varovanie *"11:19: warning: conversion to 'int' from 'long unsigned int' may alter its value [-Wconversion]"*.

```
1  #include <vector>
2  using namespace std;
3
4  auto foo(vector<int>& v) {
5    return v.size();
6  }
7
8  int main() {
9    vector<int> v;
10
11   int size = foo(v);
12
13   return 0;
14 }
```

52 Lambda výraz

Lambda výraz umožňuje definovať funkciu ako objekt. Výhodou lambda výrazu je, že definovanie funkcie je možné v tej časti kódu, kde bude využitý.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    auto lambda = [](int x, int y) { //C++11
6      return x + y;
7    };
8
```

```
9   cout << lambda(1, 1) << endl; //2
10
11  return 0;
12 }
```

C++14 umožňuje aby vstupné premenné boli generické, teda bez špecifikovaného typu.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    auto lambda = [](auto x, auto y) { //C++14
6      return x + y;
7    };
8
9    cout << lambda(1, 1) << endl; //2
10
11   return 0;
12 }
```

V tomto prípade použijeme lambda výraz na modifikáciu kontajnera, kde všetky položky budú nezáporné. Využitie referencie na riadku 9 umožňuje zmenu obsahu kontajnera. V prípade použitia `int x` by sme zmenili len hodnotu lokálnej premennej lambda výrazu.

```
1  #include <iostream>
2  #include <algorithm> //for_each
3  #include <vector>
4  using namespace std;
5
6  int main() {
7    vector<int> v = {1, -1, -99};
8
9    auto lambda = [](int& x) { //referencia int& !
10     x = abs(x);
11   };
12
13   std::for_each(v.begin(), v.end(), lambda);
14
15   for(auto& x: v) {
16     cout << x << endl;
17   } //1 1 99
18
19   return 0;
20 }
```

Výhodou lambda výrazov je možnosť použiť ich anonymne, teda bez názvu. V tomto prípade je lambda výraz použitý priamo ako vstupný parameter `std::for_each()`.

```
1  #include <iostream>
2  #include <algorithm> //for_each
3  #include <vector>
4  using namespace std;
```

```
5
6  int main() {
7    vector<int> v = {1, -1, -99};
8
9    std::for_each(v.begin(), v.end(), [](int& x) {
10     x = abs(x);
11   });
12
13   for(auto& x: v) {
14     cout << x << endl;
15   } //1 1 99
16
17   return 0;
18 }
```

Lambda výraz sa skladá z nasledovných častí:

- `[ ]` Zachytené premenné, kde `[&]` označuje zachytenie všetkých premenných podľa referencie, `[=]` označuje zachytenie všetkých premenných pomocou kópie (by value), `[&x]` označuje zachytenie premennej `x` podľa referencie a `[x]` označuje zachytenie premennej `x` pomocou kópie.
- `()` Vstupné premenné.
- `{}` Telo lambda výrazu.

V tomto prípade sme zachytili premennú `a` ako referenciu. Túto premennú je možné využiť vo vnútri lambda výrazu.

```
1  #include <iostream>
2  #include <algorithm> //for_each
3  #include <vector>
4  using namespace std;
5
6  int main() {
7    vector<int> v = {1, -1, -99};
8    int a = 10;
9
10   std::for_each(v.begin(), v.end(), [&a](int& x) { //[a]
11     x = a * x;
12   });
13
14   for(auto& x: v) {
15     cout << x << endl;
16   } //10 -10 -990
17
18   return 0;
19 }
```

53 Lambda komparátor

Komparátor je potrebný na porovnávanie objektov, ktoré sú vkladané do usporiadaných kontajnerov ako napr. map, multimap, set alebo multiset. V tomto prípade používame komparátory `std::less<>` a `std::greater<>`, ktoré usporiadajú objekty (podľa ich kľúčov) od najmenšieho po najväčší v prípade `std::less<>` a od najväčšieho po najmenší v prípade `std::greater<>`. Komparátor `std::less<>` je implicitný komparátor, t. j. v prípade `map<string, int> m;` sa použije komparátor `std::less<>`.

```
1  #include <map>
2  #include <functional> //less<>, greater<>
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7    map<string, int, less<string> > m0;
8    map<string, int, greater<string> > m1;
9
10   m0["a"] = 5;
11   m0["b"] = 10;
12
13   m1["a"] = 5;
14   m1["b"] = 10;
15
16   for(auto& x: m0) {
17     cout << x.second << endl;
18   } //5 10
19
20   for(auto& x: m1) {
21     cout << x.second << endl;
22   } //10 5
23
24   return 0;
25 }
```

V prípade, že dva objekty nie je možné porovnať pomocou operátora $<$ alebo $>$, pretože nie sú definované (preťažené), je potrebné zadefinovať vlastný komparátor napr. pomocou lambda výrazu. V našom prípade sú dva objekty porovnávané vzhľadom na hodnotu premennej `t`. Všimnime si, že na riadku 11 je použité `auto` odvodenie typu. Typ premennej `my_comp` je v tomto prípade `bool(*)(const Token&, const Token&)` a je zrejmé, že zápis pomocou `auto` je stručnejší a čitateľnejší. Typ komparátora na riadku 15 je odvodený pomocou `decltype`, t. j. typ je taký istý ako typ `my_comp`.

```
1  #include <map>
2  #include <iostream>
3  using namespace std;
4
5  class Token {
6  public:
7    int t;
```

```
};

int main() {
  auto my_comp = [](const Token& a, const Token& b) {
    return a.t < b.t;
  }; //lambda

  map<Token, int, decltype(my_comp)> m(my_comp);
  Token t0, t1;
  t0.t = 0;
  t1.t = 1;

  m[t0] = 1;
  m[t1] = 2;

  for(auto& x: m) {
    cout << x.second << endl;
  } //1 2

  return 0;
}
```

Typ `bool(*)(const Token&, const Token&)` znamená, že `my_comp` je smerník na funkciu `(*)`, ktorá má `(const Token&, const Token&)` ako vstupné typy a `bool` ako výstupný typ.

54 Knižnica algorithm

Knižnica `algorithm` obsahuje veľké množstvo efektívne implementovaných algoritmov. `std:accumulate()` spočíta sumu položiek v kontajneri, v našom prípade v `std::vector<>`.

```
#include <iostream>
#include <numeric> //accumulate
#include <vector>
using namespace std;

int main() {
  vector<int> v = {1, -1, -99};

  //0 je hodnota pripočítaná k sume
  auto sum = std::accumulate(v.begin(), v.end(), 0);
  cout << sum << endl; //-99

  return 0;
}
```

`std::all_of()` overí, či všetky prvky spĺňajú danú podmienku, `none_of()` overí, či žiaden nespĺňa danú podmienku a `std::any_of()` overí, či aspoň jeden prvok spĺňa danú podmienku. Všimnime si, že logické funkcie sú implementované ako lambdy.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
  vector<int> v = {1, -1, -99};

  auto even = std::all_of(v.begin(), v.end(), [](int x){
      return x % 2 == 0;
    });

  auto odd = std::none_of(v.begin(), v.end(), [](int x){
      return x % 2 == 0;
    });

  auto one_even = std::any_of(v.begin(), v.end(), [](int x){
      return x % 2 == 0;
    });

  cout << even << endl; //0
  cout << odd << endl; //1
  cout << one_even << endl; //0

  return 0;
}
```

`std::find()` overí, či sa daný prvok nachádza v kontajneri, v pozitívnom prípade vráti iterátor na prvý nájdený prvok. `std::count()` spočíta početnosť výskytu daného prvku v kontajneri.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
  vector<int> v = {1, -1, -99};

  int k = -99;

  auto result = std::find(v.begin(), v.end(), k);
  auto count = std::count(v.begin(), v.end(), k);

  if(result != v.end()) {
    cout << k << endl; //-99
  }

  cout << count << endl; //1

  return 0;
}
```

`std::sort()` a `std::stable_sort()` implementujú triedenie resp. stabilné

triedenie. `std::binary_search()` implementuje binárne hľadanie pre daný prvok (v našom prípade `k`). `std::max_element()` nájde najväčší prvok a `std::min_element()` nájde najmenší prvok; pre nájdené prvky vráti iterátor.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
  vector<int> v = {1, -1, -99};

  std::sort(v.begin(), v.end(), std::greater<int>());

  for(auto x: v) {
    cout << x << endl;
  } //1 -1 -99

  std::sort(v.begin(), v.end(), std::less<int>());

  for(auto x: v) {
    cout << x << endl;
  } //-99 -1 1

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

  for(auto x: v) {
    cout << x << endl;
  } //-99 -1 1

  int k = -99;
  auto found = std::binary_search(v.begin(), v.end(), k);
  cout << found << endl; //1

  auto maximum = std::max_element(v.begin(), v.end());
  auto minimum = std::min_element(v.begin(), v.end());

  cout << *maximum << endl; //1
  cout << *minimum << endl; //-99

  return 0;
}
```

55 friend

`friend` umožňuje prístup k `private` a `protected` členom triedy na základe explicitného povolenia, a nie ako dôsledok dedenia (`protected` členy).

- `friend` funkcia je funkcia, ktorá má prístup k `private` a `protected` členom triedy.

- `friend` trieda je trieda, ktorá má prístup k `private` a `protected` členom inej triedy. `friend` trieda na rozdiel od odvodenej triedy nie je rozšírenie inej triedy, ale nezávislá trieda, ktorá má prístup k `private` a `protected` členom inej triedy. Napr. trieda `Automobil` by nemala dediť z triedy `Clovek`, pretože `Automobil` nie je druh Clovek-a. Ale `Automobil` môže byť „priateľ“ Clovek-a.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  class Automobil {
6
7  friend class Clovek; //friend trieda
8  private:
9    string spz;
10
11 public:
12   string getSPZ() {
13     return spz;
14   }
15
16 protected:
17   void setSPZ(string& spz) {
18     this->spz = spz;
19   }
20 };
21
22 class Clovek {
23 public:
24   void setMyAutomobilSPZ(Automobil& automobil, string& spz) {
25     automobil.setSPZ(spz);
26   }
27 };
28
29 int main() {
30   Automobil automobil;
31   Clovek c;
32   string spz = "BT101AB";
33
34   c.setMyAutomobilSPZ(automobil, spz);
35
36   return 0;
37 }
```

V prípade, že na riadku 7 zakomentujeme `friend class Clovek` kompilátor g++ vypíše *“25:25: error: ‘void Automobil::setSPZ(std::__cxx11::string&)’ is protected within this context”*.

Všimnime si, že “priateľstvo” sa nededí. Kompilátor g++ vypíše *“27:25: error: ‘void Automobil::setSPZ(std::__cxx11::string&)’ is protected*

within this context". Na riadku je 7 je potrebná zmena na `friend class Student`.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  class Automobil {
6
7  friend class Clovek;
8  private:
9    string spz;
10
11 public:
12   string getSPZ() {
13     return spz;
14   }
15
16 protected:
17   void setSPZ(string& spz) {
18     this->spz = spz;
19   }
20 };
21
22 class Clovek {};
23
24 class Student: public Clovek {
25 public:
26 void setMyAutomobilSPZ(Automobil& automobil, string& spz) {
27     automobil.setSPZ(spz);
28   }
29 };
30
31 int main() {
32   Automobil automobil;
33   Student s;
34   string spz = "BT101AB";
35
36   s.setMyAutomobilSPZ(automobil, spz);
37
38   return 0;
39 }
```

V tomto prípade používame friend funkciu resp. preťažený operátor <<, ktorý potrebuje prístup k `private` premenným `x` a `y` triedy `Token`. Ďalšie možné riešenie je zmena `private` na `public`, avšak takéto riešenie nie je odporúčané, pretože to umožní priamy prístup k premenným triedy `Token`. Ako ďalšie riešenie sa ponúkajú `public` "gettery", teda metódy, ktoré vrátia hodnotu `private` premenných `x` a `y`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
```

```
5   friend std::ostream& operator<<(std::ostream& stream,
6                                   const Token& t);
7
8 private:
9   int x = 0;
10  const int y = 0;
11 };
12
13 std::ostream& operator<<(std::ostream& stream,
14                          const Token& t) {
15   stream << t.x << endl << t.y;
16
17   return stream;
18 }
19
20 int main() {
21   Token t;
22   cout << t; //0 0
23
24   return 0;
25 }
```

56 template

`template` umožňuje písanie kódu, ktorý je nezávislý od vstupného typu. V tomto prípade máme funkciu `max()`, ktorá vráti max hodnotu pre každý typ, ktorý má definovaný operátor $<$. V opačnom prípade je potrebné pre danú triedu tento operátor definovať resp. preťažiť. Funkcia `max()` je v mennom priestore `NS`, aby nenastal konflikt s `std::max()`.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  namespace NS {
6    template <typename T>
7    T const& max(T const& a, T const& b)  {
8      return a < b ? b : a;
9    }
10 };
11
12 int main() {
13   int i = 39;
14   int j = 20;
15   cout << "max(i,␣j):␣" << NS::max(i, j) << endl;
16
17   double f1 = 13.5;
18   double f2 = 20.7;
19   cout << "max(f1,␣f2):␣" << NS::max(f1, f2) << endl;
20
21   string s1 = "Hello";
22   string s2 = "World";
```

```
23    cout << "max(s1,␣s2):␣" << NS::max(s1, s2) << endl;
24
25    return 0;
26  }
```

`template` tiež umožňuje `template` triedy, ktoré sú nezávislé od vstupného typu. V našom prípade máme triedu `MyMath`, ktorá obsahuje metódy `max()` a `min()`. Všimnime si na riadku 5, že typ `int` je implicitný, čo potom umožňuje zápis `MyMath<>` na riadku 19. Ak by implicitný typ neexistoval, bol by potrebný zápis `MyMath<int>`.

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   template <typename T = int> //int je implicitný typ
6   class MyMath {
7   public:
8     T const& max(T const& a, T const& b)  {
9       return a < b ? b : a;
10    }
11    T const& min(T const& a, T const& b)  {
12      return a < b ? a : b;
13    }
14  };
15
16  int main() {
17    int i = 39;
18    int j = 20;
19    MyMath<> mm; //MyMath<int> mm;
20
21    cout << "max(i,␣j):␣" << mm.max(i, j) << endl;
22    cout << "min(i,␣j):␣" << mm.min(i, j) << endl;
23
24    return 0;
25  }
```

57 Implicitné generovanie

Kompilátor v prípade potreby vygeneruje:

- Prednastavený (defaultný) konštruktor.
- Operátor priradenia =.
- Kopírovací (copy) konštruktor.
- Deštruktor.
- Presúvací (move) konštruktor (C++11).
- Operátor priradenia = pre presúvanie (C++11).

V prípade, že to nie je potrebné napr. ak by bol riadok 8 zakomentovaný, nie je generovaný kopírovací konštruktor, ak by bol riadok 9 zakomentovaný nie je potrebný operátor priradenia =. Prednastavený konštruktor teda tiež nie je generovaný, ak objekt vzniká zavolaním konštruktora so vstupným parametrom (a preto sa môže stať, že prednastavený konštruktor je potrebné dodatočne pridať príp. použiť `=default` na vynútenie jeho generovania).

```
1  class Token {
2  public:
3    int size = 0;
4  };
5
6  int main() {
7    Token mt0;
8    Token mt1 = mt0;
9    mt1 = mt0;
10
11   return 0;
12 }
```

Generátor generuje implicitný operátor priradenia = aj pre odvodenú triedu `Token1`. `operator+(const Token&)` je zdedený z triedy `Token`. Kompilátor g++ vypíše *"20:15: error: no match for 'operator=' (operand types are 'Token1' and 'Token')"*. Na riadku 20 je zavolaný operátor + na riadku 6, ktorý vráti objekt typu `Token`. Operátor = na riadku 20 nemôže skonvertovať objekt typu `Token` na objekt typu `Token1` a nastane chyba. Operátor = na riadku 20 je implicitne generovaný kompilátorom.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    Token operator+(const Token& t) {
7      cout << "operator+(const␣Token&)";
8
9      return Token();
10   }
11 };
12
13 class Token1: public Token {
14 public:
15   int size = 0;
16 };
17
18 int main() {
19   Token1 mt0, mt1, mt2;
20   mt0 = mt1 + mt2;
21
22   return 0;
23 }
```

Operátor + na riadku 17 prekryje zdedený operátor + z triedy `Token`. Tento

operátor + vráti objekt typu `Token1`, ktorý vie implicitne generovaný operátor = na riadku 26 priradiť.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    Token operator+(const Token& t) {
7      cout << "operator+(const␣Token&)";
8
9      return Token();
10   }
11 };
12
13 class Token1: public Token {
14 public:
15   int size = 0;
16
17   Token1 operator+(const Token1& t) {
18     cout << "operator+(const␣Token1&)";
19
20     return Token1();
21   }
22 };
23
24 int main() {
25   Token1 mt0, mt1, mt2;
26   mt0 = mt1 + mt2;
27
28   return 0;
29 }
```

58 constexpr

`constexpr` deklaruje, že funkciu alebo premennú je možné vyhodnotiť počas kompilácie.

```
1  #include <iostream>
2  #include <cmath> //M_PI
3  using namespace std;
4
5  constexpr double pi2() {
6    return M_PI * M_PI;
7  }
8
9  int main() {
10   constexpr double a = 4 * M_PI;
11   cout << a << "␣" << pi2() << endl; //12.5664 9.8696
12
13   return 0;
14 }
```

Vyhodnotenie počas kompilácie je možné, ak vstupný parameter funkcie je konštanta. V tomto prípade sa počas kompilácie vypočíta hodnota `10!`. Všimnime si, že je použitý typ `long long` (resp. `unsigned long long`) zavedený v C++11, ktorý má šírku min. 64 bitov.

```
1  #include <iostream>
2  using namespace std;
3
4  constexpr unsigned long long factorial(unsigned long long n) {
5    return n > 0 ? n * factorial(n - 1) : 1;
6  }
7
8  int main() {
9    constexpr unsigned long long f = factorial(10);
10   cout << f; //3628800
11
12   return 0;
13 }
```

V prípade, že vstupný parameter nie je `const` vypíše g++ *"10:47: error: the value of 'k' is not usable in a constant expression"*. Ak zmeníme premennú `k` na riadku 9 na `const` kompilácia prebehne. V prípade, že vstupný parameter nie je `const` má kompilátor problém usúdiť, či sa premenná `k` môže zmeniť a teda, či je vyhodnotenie počas kompilácie možné.

```
1  #include <iostream>
2  using namespace std;
3
4  constexpr unsigned long long factorial(unsigned n) {
5    return n > 0 ? n * factorial(n - 1) : 1;
6  }
7
8  int main() {
9    int k = 10; //chyba, const int k = 10;
10   constexpr unsigned long long f = factorial(k);
11   cout << f; //3628800
12
13   return 0;
14 }
```

C++14 umožňuje aj vyhodnocovanie `constexpr` funkcií, ktoré obsahujú `for` slučku.

```
1  #include <iostream>
2  using namespace std;
3
4  constexpr unsigned long long factorial (unsigned long long n) {
5    unsigned long long f = 1;
6
7    for(unsigned i = 1; i <= n; ++i) { //od C++14
8      f *= i;
9    }
10
11   return f;
```

```
12 }
13
14 int main() {
15   constexpr unsigned long long f = factorial(10);
16   cout << f; //3628800
17
18   return 0;
19 }
```

`constexpr` lambda výrazy sú podporované od C++17 (a v g++ implementované od v7).

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    auto lambda = [](auto x, auto y) constexpr { //C++17
6      return x + y;
7    };
8
9    constexpr int sum = lambda(1, 1);
10   cout << sum << endl; //2
11
12   return 0;
13 }
```

Tu je potrebné si uvedomiť, že po vyhodnotení lambda výrazu počas kompilácie sa kód zredukuje, ako keby mal nasledovnú formu.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5    cout << 2 << endl; //2
6
7    return 0;
8  }
```

59 using

Od C++11 nahrádza using alias typu cez `typedef`. Syntakticky sa jedná o lepší zápis, pretože jeho forma je štandardná `using new_id = type_id`, t. j. akoby priradenie. Pretože tvorcovia jazyka C++ nechceli zaviesť nové kľúčové slovo, využíva sa using, ktoré v tomto prípade má iný význam ako napr. v `using namespace std`. Výhoda `using` a dôvod jeho zavedenia je podpora `template`.

```
1  #include <deque>
2  #include <queue>
3  using namespace std;
4
5  using Token = struct Token {
```

```
6   int size = 1;
7 };
8
9 //typedef struct Token Token;
10 using Token = struct Token;
11
12 template <typename T>
13 using ptr = T*;
14
15 //MyQ je alias queue s deque ako alokátorom
16 template <class T>
17 using MyQ = std::queue<T, deque<T> >;
18
19 int main() {
20   Token t;
21   ptr<int> x;
22   MyQ<int> myq;
23
24   int i;
25   //I4 je alias pre int
26   using I4 = decltype(i);
27
28   return 0;
29 }
```

60 [[deprecated]]

Ako deprecated môžeme označiť premennú, funkciu, triedu, typedef, enum, using atď. Deprecated znamená, že danú entitu je možné použiť, ale jej použitie nie je odporúčané, napr. z dôvodu ukončenia podpory v nasledujúcej verzii kódu. Pri použití deprecated je možné nastaviť aj správu pre používateľa `[[deprecated(string-literal)]]`. Kompilátor g++ vypíše pre každé použitie deprecated entity varovanie, napr. *"1:7: warning: 'Token::c' is deprecated [-Wdeprecated-declarations]"* alebo *"25:23: warning: 'int Token::compute2()' is deprecated: Hello, unsupported method! [-Wdeprecated-declarations]"* v prípade `[[deprecated()]]` s nastavenou správou.

```
1  class Token {
2  private:
3    int a = 1;
4    int b = 2;
5    [[deprecated]] int c;
6
7  public:
8    [[deprecated]]
9    int compute() {
10     return a - b;
11   }
12   [[deprecated("Hello, unsupported method!")]]
13   int compute2() {
14     return a * b;
```

```
15    }
16  };
17
18  class [[deprecated("Use␣class␣Token␣instead!")]] Token1 {};
19
20  using TokenPtr [[deprecated]] = Token*;
21
22  int main() {
23    Token t;
24    int c = t.compute();
25    int c1 = t.compute2();
26
27    Token1 t1;
28    TokenPtr tPtr;
29
30    return 0;
31  }
```

61 override, final

Kľúčové slovo `override` zabezpečí, že takto označená metóda je implementácia virtuálnej metódy deklarovanej v materskej triede. Na riadku 15 nastane chyba, pretože metóda `bar()` na riadku 7 nie je virtuálna metóda. Ak by sme na riadku 15 nepoužili označenie `override`, potom by táto metóda prekryla zdedenú metódu s rovnakým menom. A presne toto je význam označenia `override`, t. j. uistenie, že naozaj implementujeme metódu, ktorá bola označená ako `virtual` v materskej triede, a nie novú metódu. Kompilátor g++ vypíše *"15:8: error: 'void Token1::bar()' marked 'override', but does not override"*.

Metóda `foo()` na riadku 12 je úspešná implementácia virtuálnej metódy materskej triedy. Metóda `foo()` na riadku 20 je označená ako `final`, čo znamená, že táto implementácia virtuálnej triedy je posledná. V odvodenej triede z triedy `Token2` ďalšia reimplementácia už nebude možná.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   public:
6     virtual void foo() = 0;
7     void bar() {};
8   };
9
10  class Token1: public Token {
11  public:
12    void foo() override { //OK
13      cout << "Token1::foo()" << endl;
14    }
15    void bar() override {} //chyba
```

```
16 };
17
18 class Token2: public Token1 {
19 public:
20   void foo() final { //final = posledný override
21     cout << "Token2::foo()" << endl;
22   }
23 };
24
25
26 int main() {
27   Token2 t;
28   t.foo(); //Token2::foo()
29
30   return 0;
31 }
```

62 Kopírovacia a presúvacia sémantika

Rozdiely medzi kopírovacou a presúvacou sémantikou sú nasledovné:

- Kopírovanie (copy) znamená, že obsah objektu `t0` je *prekopírovaný* do `t`, pričom objekt `t0` ostane zachovaný v pôvodnom stave.
- Presúvanie (move) znamená, že obsah objektu `t0` je *presunutý* do `t`, pričom objekt `t0` bude "vyprázdnený".

Pre implementáciu presúvacej sémantiky potrebujeme aspoň jedno z nasledujúcich:

- Presúvací konštruktor: `Token(Token&& t)`.
- Operátor priradenia pre presúvanie: `Token& operator=(Token&& t)`.

Na riadku 29 je zavolaný kopírovací konštruktor, pretože vyžadujeme aby premenná `t` bola kópiou premennej `t0`. Na riadku 30 je zavolaný presúvací konštruktor, pretože sme použili funkciu `std:move()`, ktorá pretypuje výstup funkcie `foo()` z `Token&` na `Token&&`. Tento príklad ešte neimplementuje samotné presunutie, pretože presúvací konštruktor, ako aj operátor priradenia pre presúvanie, obsahujú len zavolanie `std::cout` resp. vrátenie `*this`.

```
1 #include <iostream>
2 using namespace std;
3
4 class Token {
5 public:
6   Token() { //prednastavený konštruktor
7     cout << "Token()" << endl;
8   }
9   Token(const Token& t) { //kopírovací konštruktor
```

```
10      cout << "Token(const␣Token&)" << endl;
11    }
12    Token& operator=(const Token& t) { //operátor priradenia =
13      cout << "operator=(const␣Token&)" << endl;
14
15      return *this;
16    }
17    Token(Token&& t) { //move konštruktor
18      cout << "Token(Token&&)" << endl;
19    }
20    Token& operator=(Token&& t) { //operátor priradenia = pre presúvanie
21      cout << "operator=(Token&&)" << endl;
22
23      return *this;
24    }
25  };
26
27  int main() {
28    Token t0; //Token()
29    Token t = t0; //Token(const Token&)
30    Token t1 = std::move(t0); //Token(Token&&)
31
32    return 0;
33  }
```

STL `std::vector<>` (ako aj map, list atď.) implementuje presúvaciu sémantiku. Po presune na riadku 7 obsahuje vektor `b` to, čo predtým obsahoval vektor `a`. Vektor `a` je po presune prázdny, avšak toto správanie závisí od použitého kompilátora; vektor `a` môže byť následne použitý, jeho obsah môže byť nedefinovaný t. j. ľubovolný.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6    vector<int> a = {1, 2, 3, 4, 5};
7    auto b = std::move(a);
8
9    cout << a.size() << endl; //0
10   cout << b.size() << endl; //5
11
12   return 0;
13 }
```

Na nasledujúcom príklade ukážeme ako použiť presúvaciu sémantiku v prípade, že trieda obsahuje vektor, ktorý má dĺžku 10 000 položiek typu `int`. Na riadku 43 nastane zavolanie kopírovacieho konštruktora a všetky položky vektora `v` sú prekopírované, t. j. `t` teraz obsahuje kópiu vektora `v` z `t0`. Na riadku 45 nastane presunutie položiek vektora `v`, po ich presunutí obsahuje vektor `v` to, čo predtým obsahoval vektor `v` z `t0`. Presunutie je možné, pretože `std::vector<>` implementuje presúvaciu sémantiku.

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

class Token {
private:
  vector<int> v;
public:
  Token(): v(10000) { //v je inicializovaný na vel'kost' 10000
    cout << "Token()" << endl;
  }
  Token(const Token& t) {
    v = t.v; //kopírovanie

    cout << "Token(const␣Token&)" << endl;
  }
  Token& operator=(const Token& t) {
    v = t.v; //kopírovanie

    cout << "operator=(const␣Token&)" << endl;
    return *this;
  }
  Token(Token&& t) {
    v = std::move(t.v); //presunutie

    cout << "Token(Token&&)" << endl;
  }
  Token& operator=(Token&& t) {
    v = std::move(t.v); //presunutie

    cout << "operator=(Token&&)" << endl;
    return *this;
  }
  auto getSize() {
    return v.size();
  }
};

int main() {
  Token t0; //Token()
  cout << t0.getSize() << endl; //10000

  Token t = t0; //Token(const Token&)

  Token t1 = std::move(t0); //Token(Token&&)
  cout << t1.getSize() << endl; //10000
  cout << t0.getSize() << endl; //0

  return 0;
}
```

Všimnime si, že presun nie je možný z `const` objektu. `const` znamená "len na čítanie", a teda nie je zrejmé ako po presune vyprázdniť objekt, ktorý bol deklarovaný "len na čítanie". Narážame tu na sémantiku `const`, kde pri

použití `const` sme deklarovali, že objekt ostane taký aký je, teda nebude sa meniť. Všimnime si, že namiesto presunu nastalo kopírovanie.

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

int main() {
  std::vector<int> const a = {1, 2, 3, 4, 5}; //const
  auto b = std::move(a); //kopírovanie

  cout << a.size() << endl; //5
  cout << b.size() << endl; //5

  return 0;
}
```

Z rovnakého dôvodu si nemôžeme zjednodušiť zápis a použiť len kopírovací konštruktor, pretože tento má ako vstupný parameter `const Token&`. V tomto prípade teda nastane kopírovanie.

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

class Token {
private:
  vector<int> v;
public:
  Token(): v(10000) {
    cout << "Token()" << endl;
  }
  Token(const Token& t) {
    v = std::move(t.v);
    cout << "Token(const Token&)" << endl;
  }
  auto getSize() {
    return v.size();
  }
};

int main() {
  Token t0; //Token()
  cout << t0.getSize() << endl; //10000

  Token t1 = std::move(t0); //Token(const Token&)
  cout << t1.getSize() << endl; //10000
  cout << t0.getSize() << endl; //10000

  return 0;
}
```

Čo vlastne spôsobí zavolanie funkcie `std::move()`? `std::move()` nič nepresunie, len pretypuje objekt tak, že ak je z neho presun možný, tak sa vykoná. V našom prípade je `move` pretypovane:

```
static_cast<std::vector<int>&&>(t.v)
```

Presun je taktiež vhodný pri inicializácii objektu. Všimnime si, že veľkosť vektora `v0` po presune je `0`.

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

class Token {
private:
  vector<int> v;
public:
  Token(vector<int>&& v0): v(std::move(v0)) {}
  auto getSize() {
    return v.size();
  }
};

int main() {
  vector<int> v0 = {0, 1, 2, 3, 4, 5, 6, 7 , 8, 9, 10};
  cout << v0.size() << endl; //11

  Token t = std::move(v0);
  cout << v0.size() << endl; //0, po presune
  cout << t.getSize() << endl; //11

  return 0;
}
```

Zaujímavá možnosť je zakázať kopírovanie, a to tak, že kopírovací konštruktor aj operátor priradenia = pre kopírovanie sú označené ako `=delete`. V tomto prípade je možný len presun, ale nie kopírovanie. Sémanticky ide o zabezpečenie, že určitý zdroj, v našom prípade vektor `v` je jedinečný, t.j. existuje len ako originál bez možnosti kopírovania.

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

class Token {
private:
  vector<int> v;
public:
  Token(): v(10000) {
    cout << "Token()" << endl;
  }
  Token(const Token& t) = delete; //bez kopírovania
  Token& operator=(const Token& t) = delete; //bez kopírovania
  Token(Token&& t) { //len presun
    v = std::move(t.v);

    cout << "Token(Token&&)" << endl;
  }
```

```
19    auto getSize() {
20      return v.size();
21    }
22  };
23
24  int main() {
25    Token t0; //Token()
26    Token t1 = std::move(t0); //OK, Token(Token&&)
27
28    Token t2 = t0; //chyba
29    t2 = t0; //chyba
30
31    return 0;
32  }
```

V prípade, že potrebujeme implementovať presúvaciu sémantiku, môžeme dosiahnuť žiadané správanie výmenou smerníkov a deštrukciou nepotrebného zdroja. V tomto prípade existujúce pole `data` zmažeme pomocou `delete[]` a následovne prevezmeme `data` z objektu, z ktorého robíme presun. Všimnime si, že na riadku 53 nastane chyba, pretože sa pokúšame kopírovať z toho objektu, z ktorého už bol urobený presun, a teda je prázdny; pole `data` je `nullptr`.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   private:
6     int* data;
7     size_t size = 10000;
8   public:
9     Token() {
10      data = new int[size];
11    }
12    Token(const Token& t) { //kopírovanie
13      if(this != &t) {
14        std::copy(t.data, t.data + size, this->data);
15      }
16    }
17    Token& operator=(const Token& t) { //kopírovanie
18      if(this != &t) {
19        std::copy(t.data, t.data + size, this->data);
20      }
21
22      return *this;
23    }
24    Token(Token&& t) { //presun
25      if(this != &t) {
26        delete[] this->data;
27        this->data = t.data;
28        t.data = nullptr;
29        t.size = 0;
30      }
```

```
31    }
32    Token& operator=(Token&& t) { //presun
33      if(this != &t) {
34        delete[] this->data;
35        this->data = t.data;
36        t.data = nullptr;
37        t.size = 0;
38      }
39
40      return *this;
41    }
42    ~Token() {
43      if(data != nullptr) {
44        delete[] data;
45      }
46    }
47  };
48
49  int main() {
50    Token t0;
51    Token t1 = std::move(t0); //OK
52
53    Token t2 = t0; //chyba
54    t2 = t0; //chyba
55
56    return 0;
57  }
```

Ak implementujeme presúvací konštruktor, ako aj operátor priradenia = pre presunutie, potom z presúvacieho konštruktora môžeme zavolať operátor priradenia a tak zabrániť duplicite kódu.

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   private:
6     int* data;
7     size_t size = 10000;
8   public:
9     Token() {
10      data = new int[size];
11    }
12    Token(const Token& t) = delete;
13    Token& operator=(const Token& t) = delete;
14    Token(Token&& t) {
15      cout << "Token(Token&&)" << endl;
16
17      *this = std::move(t); //operátor priradenia
18    }
19    Token& operator=(Token&& t) {
20      cout << "operator=(Token&&)" << endl;
21
22      if(this != &t) {
```

```
23        delete[] this->data;
24        this->data = t.data;
25        t.data = nullptr;
26        t.size = 0;
27      }
28
29      return *this;
30    }
31    ~Token() {
32      if(data != nullptr) {
33        delete[] data;
34      }
35    }
36  };
37
38  int main() {
39    Token t0;
40    Token t1 = std::move(t0);
41
42    return 0;
43  }
```

63 noexcept

Po tom ako je vyhodená výnimka na riadku 7, nastane okamžité odstránenie lokálnych premenných zo stacku (angl. stack unwinding), v našom prípade bude odstránená premenná `array` typu `int*`. `delete[] array` nebude nikdy vykonané a zároveň stratíme adresu poľa `array`.

```
1  #include <iostream>
2  using namespace std;
3
4  void foo() {
5    int* array = new int[10000000];
6
7    throw 21; //memory leak
8
9    delete[] array; //nebude nikdy zavolané
10   cout << "delete[] array" << endl;
11 }
12
13 int main() {
14   try{
15     foo();
16   } catch(const int& e) {
17     cout << "catch(int&)" << endl; //catch(int&)
18   }
19
20   return 0;
21 }
```

Príklad s `throw 21` sa môže zdať umelý, ale presne rovnaká situácia nastane

v prípade, že nemáme dosť pamäte pre vytvorenie veľkého množstva objektov. Tak ako v predchádzajúcom prípade nastane memory leak, pretože sme stratili adresu poľa `array`.

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

class Token {
  int array[100000];
};

void foo() {
  Token* array = new Token[10000000];

  delete[] array; //nebude nikdy zavolané
  cout << "delete[] array" << endl;
}

int main() {
  try{
    foo();
  } catch(std::bad_alloc& e) {
    cout << "catch(std::bad_alloc&)" << endl; //catch(std::bad_alloc&)
  }

  return 0;
}
```

V prípade, že označíme funkciu, metódu, konštruktor atď. ako `noexcept`, znamená to, že sme spravili sľub, že táto entita nevyhodí výnimku. Tento sľub umožní kompilátoru optimalizáciu, pretože stack unwinding nebude možný; v prípade, že napriek tomu nastane výnimka je beh kódu ukončený zavolaním `std::terminate()`. Počas behu kódu sa v našom prípade vypíše *"terminate called after throwing an instance of 'std::bad_ alloc' what(): std::bad_ alloc"*. Všeobecné ponaučenie z tohto prípadu je, že výnimky potrebujeme zachytávať, čo najskôr a zabrániť ich šíreniu v kóde, pretože to môže viesť k rôznym problémom.

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

class Token {
  int array[100000];
};

void foo() noexcept { //noexcept
  Token* array = new Token[10000000];

  delete[] array;
  cout << "delete[] array" << endl;
}

int main() {
```

```
16    try{
17      foo();
18    } catch(std::bad_alloc& e) {
19      cout << "catch(std::bad_alloc&)" << endl;
20    }
21
22    return 0;
23  }
```

Zaujímavý prípad použitia `noexcept` nastane v prípade presúvacej (move) sémantiky. V prípade kopírovacej (copy) sémantiky, ak nastane výnimka počas kopírovania, pretože napr. nebolo možné alokovať miesto pre kopírované objekty, je možné kopírovanie prerušiť; pôvodné objekty sú na pôvodnom mieste v pamäti a v pôvodnom stave. V prípade presúvania, ak nastane výnimka pri presune, môžeme pôvodný stav obnoviť presunom späť, ale čo ak nastane výnimka aj pri presune späť? V tomto prípade je najvhodnejšie použiť `noexcept` t. j. sľúbiť, čo sa sľúbiť nedá (nebude žiadna výnimka), ale v prípade, že výnimku zachytíme, aj tak nám to nepomôže (pretože nevieme ako obnoviť pôvodný stav), takže nemáme na výber.

Ak nepredpokladáme, že výnimku spôsobí `cout` na riadku 13, ostáva ďalšia možnosť, že výnimka nastane pri `delete` na riadku 16. Vzhľadom na to, že `int` je základný typ a deštrukcia `int` výnimku nespôsobí, môžeme v tomto prípade sľúbiť, že výnimka (s veľkou pravdepodobnosťou) nenastane...

```
1   #include <iostream>
2   using namespace std;
3
4   class Token {
5   private:
6     int* data;
7     size_t size = 10000;
8   public:
9     Token() {
10      data = new int[size];
11    }
12    Token(Token&& t) noexcept { //noexcept
13      cout << "Token(Token&&)" << endl;
14
15      if(this != &t) {
16        delete[] this->data;
17        this->data = t.data;
18        t.data = nullptr;
19        t.size = 0;
20      }
21    }
22    ~Token() {
23      if(data != nullptr) {
24        delete[] data;
25      }
26    }
27  };
```

```
28
29 int main() {
30   Token t0;
31   Token t1 = std::move(t0);
32
33   return 0;
34 }
```

Čo ak by pole neobsahovalo `int`, ale iný používateľom definovaný typ? V tom prípade sa pri deštrukcii poľa zavolá deštruktor každého objektu, pričom výnimka už nastať môže. To nás vedie k myšlienke, že deštruktory by mali byť vždy `noexcept`; a naozaj v C++ implicitne generované, ako aj používateľom definované deštruktory sú `noexcept`. Po spustení sa vypíše *"terminate called after throwing an instance of 'int'"*.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    ~Token() {
7      throw 21;
8    }
9  };
10
11 int main() {
12   Token* t0 = new Token;
13
14   try{
15     delete t0;
16   } catch(const int& e) {
17     cout << "catch(const int&)" << endl;
18   }
19
20   return 0;
21 }
```

Deštruktor môžeme označiť ako nie-noexcept pomocou `noexcept(false)`. V tomto prípade je výnimka šírená a zachytená v `main()`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Token {
5  public:
6    ~Token() noexcept(false) { //noexcept(false)
7      throw 21;
8    }
9  };
10
11 int main() {
12   Token* t0 = new Token;
13
14   try{
```

```
15      delete t0;
16    } catch(const int& e) {
17      cout << "catch(const int&)" << endl; //catch(const int&)
18    }
19
20    return 0;
21  }
```

64 dynamic_cast<>

`dynamic_cast<>` na rozdiel od `static_cast<>` vykoná pretypovanie počas behu kódu (run-time type identification), naproti tomu `static_cast<>` vykoná pretypovanie počas kompilácie. Z tohto dôvodu je `dynamic_cast<>` menej efektívny, a preto, ak je to možné, `static_cast<>` by mal byť uprednostnený. V niektorých prípadoch nie je možné vykonať `static_cast<>`, preto ostáva `dynamic_cast<>` ako jediná možnosť. V našom prípade funkcia `foo()` náhodne vygeneruje objekt typu `Token1` alebo `Token2`. Po zistení, či bol vygenerovaný `Token1` alebo `Token2` je potom možné vykonať špecifický obslužný kód. Ak je pretypovanie pomocou `dynamic_cast<>` úspešné, vráti platnú adresu, v opačnom prípade vráti `nullptr`. Po spustení sa vypíše "I am Token2", pretože funkcia `foo()` vygenerovala objekt typu `Token2`. Všimnime si, že triedy `Token1` a `Token2` sú `template` triedy.

```
1   #include <iostream>
2   #include <cstdlib> //rand()
3   using namespace std;
4
5   class Token {
6   public:
7     virtual ~Token() = default;
8   };
9
10  template <typename T>
11  class Token1: public Token {};
12
13  template <typename T>
14  class Token2: public Token {};
15
16  Token* foo() {
17    if((rand()%2)==0) {
18      return new Token1<int>;
19    } else {
20      return new Token2<long>;
21    }
22  }
23
24  int main() {
25    Token* t = foo();
26
27    Token1<int>* t1 = dynamic_cast<Token1<int>*>(t);
```

```
28    Token2<long>* t2 = dynamic_cast<Token2<long>*>(t);
29
30    if(t1) {
31      cout << "I am Token1" << endl;
32    } else if(t2) {
33      cout << "I am Token2" << endl;
34    } else {
35      cout << "I am UFO" << endl;
36    }
37
38    delete t;
39    return 0;
40  }
```

V nasledujúcom prípade máme `std::list<Token*>*`, do ktorého vkladáme objekty typu `Token1*` a `Token2*`. Pri iterácii cez tento list potrebujeme zavolať metódy špecifické pre `Token1` a `Token2`. Ako možná implementácia sa ponúka `dynamic_cast<>`. V prípade, že je takéto pretypovanie úspešné, môžeme zavolať metódy `Token1` a `Token2`, ktoré môžu byť rozdielne, napr. `special1()` a `special2()`. Všimnime si, že na riadku 61 voláme `delete l`, avšak objekty referencované v tomto list-e ostávajú nedeštruované. V kóde tiež predpokladáme, že `T` má implementovanú operáciu *, v opačnom prípade je potrebné preťaženie.

```
1   #include <iostream>
2   #include <list>
3   using namespace std;
4
5   class Token {
6   public:
7     virtual ~Token() = default;
8     virtual string getName() = 0;
9   };
10
11  template <typename T>
12  class Token1: public Token {
13  public:
14    string getName() override {
15      return string("I am Token1");
16    }
17    T compute(T x) {
18      return x * x;
19    }
20    void special1() {
21      cout << "special1()" << endl;
22    }
23  };
24
25  template <typename T>
26  class Token2: public Token {
27    public:
28    string getName() override {
29      return string("I am Token2");
```

```
30    }
31    T compute(T x) {
32      return x * x;
33    }
34    void special2() {
35      cout << "special2()" << endl;
36    }
37  };
38
39  int main() {
40    list<Token*>* l = new list<Token*>;
41    l->push_back(new Token1<int>);
42    l->push_back(new Token2<long>);
43
44    for(auto x: *l) {
45      Token1<int>* t1 = dynamic_cast<Token1<int>*>(x);
46      Token2<long>* t2 = dynamic_cast<Token2<long>*>(x);
47
48      if(t1) {
49        cout << t1->getName() << endl;
50        cout << t1->compute(1000000) << endl; //-727379968, pretečenie typu int
51        t1->special1();
52      } else if(t2) {
53        cout << t2->getName() << endl;
54        cout << t2->compute(1000000) << endl; //1000000000000
55        t2->special2();
56      } else {
57        cout << "I am UFO" << endl;
58      }
59    }
60
61    delete l; //objekty v liste ostávajú, memory leak
62    return 0;
63  }
```

Register