STU FEI

Programovacie techniky

12. constexpr, C++ pretypovanie, chytré smerníky, emplace/emplace_back, záver

constexpr

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

constexpr double f_pi2() {
  return M_PI * M_PI;
}

int main() {
  constexpr double a = 4 * M_PI;
  cout << a << " " << f_pi2() << endl;
  return 0;
}
```

constexpr = evaluácia počas kompilácie

constexpr

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

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

  return 0;
}
```

constexpr

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


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

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

C pretypovanie

```
int main() {
  char c = 10;       //lokálna premenná na stacku
  int *p0 = (int*) &c; //char=1 byte, int=8 byte
  *p0 = 20; //dereferencovanie, chyba?

  return 0;
}
```

C cast neumožňuje pochopenie zámeru programátora

C++ pretypovanie

static_cast: odvodené typy príp. s konverziou

reinterpret_cast: ľubovolný typ

dynamic_cast: počas behu programu

const_cast: odstránenie const

C++ static_cast<>

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

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

  t1 = (Token1*) t0; //OK
  t1 = static_cast<Token1*>(t0); //chyba

  delete t0;
  return 0;
}
```

error: invalid static_cast from type ‘Token0*’ to type ‘Token1*’

C++ static_cast<>

```
class Token0 {};
class Token1: public Token0 {};

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

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

  return 0;
}
```

Zámer bol aby Token1 bola odvodená trieda z Token0.

C++ static_cast<>

```
class Token1;

class Token0 {
  public:
  Token0() = default;
  Token0(const Token1& t0) {}
};

class Token1 {};

int main() {
  Token0 t0;
  Token1 t1;
  t0 = static_cast<Token0>(t1); //OK, únik pamäte

  return 0; }
```

C++ dynamic_cast<>

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

class Derived : public Base {};

int main() {
  Derived* d0 = new Derived();
  Base* b = dynamic_cast<Base*>(d0); //OK

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

  delete d0;
  return 0;
}
```

Pretypovanie počas behu kódu. V prípade neúspechu nullptr.

C++ reinterpret_cast<>

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

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

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

  delete t1;
  return 0;
}
```

C++ const_cast<>

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

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

C++ const_cast<>

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

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

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

  return 0;
}
```

Chytrý smerník (C++11)

Chytrý smerník je zmazaný, ak už nie je používaný (referencovaný), bez potreby delete

- unique_ptr<T> myPtr(new T);
  myPtr má priradený jeden objekt typu T

  Po vystúpení zo scopu je object typu T uvoľnený

- shared_ptr<T> myPtr(new T);
  Niekoľko shared_ptr<T> má priradený ten istý objekt typu T

  Po tom ako posledný shared_ptr prestane byť priradený tomu istému objektu typu T, nastane uvoľnenie

Memory leak

```
void foo() {
    int* ptr = new int(15);
    int x = 45;
    // ...
    if (x == 45)
        return;   //nenastane delete ptr;
    // ...
    delete ptr;
}

int main() {
  foo();
  return 0;
}
```

std::unique_ptr<T>

```
#include <memory>

void foo() {
    std::unique_ptr<int> valuePtr(new int(15));
    int x = 45;
    // ...
    if (x == 45)
        return;   //OK
    // ...
}

int main() {
  foo();
  return 0;
}
```

unique_prt vs shared_ptr

Kopírovanie nie je možné:

```
unique_ptr<T> myPtr(new T);
unique_ptr<T> myOtherPtr = myPtr; //chyba
```

Kopírovanie povolené:

```
shared_ptr<T> myPtr(new T);
shared_ptr<T> myOtherPtr = myPtr; //OK
```

myOtherPtr aj myPtr manažujú ten istý objekt

unique_ptr

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

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

  ~Token() {
    cout << "~Token()" << endl;
  }
};
```

unique_ptr

```
int main() {
  unique_ptr<Token> t0(new Token()); //Token()
  Token* t1 = new Token(); //Token()

  return 0;
} //~Token() zavolaný pre t0 (ale nie pre t1)
```

unique_ptr: release

```
int main(){
  unique_ptr<Token> t0(new Token());

  Token* t1 = t0.release(); //release objektu Token
  delete t1; //jeho okamžité odstránenie

  return 0;
}
```

unique_ptr: get

```
int main(){
  unique_ptr<Token> t0(new Token());

  Token* t1 = t0.get(); //smerník na objekt Token

  return 0;
}
```

unique_ptr: pole

```
int main(){
  unique_ptr<Token[]> t0(new Token[5]);

  return 0;
}
```

```
Token()
Token()
Token()
Token()
Token()
~Token()
~Token()
~Token()
~Token()
~Token()
```

unique_ptr

```
class Token {
private:
  std::unique_ptr<std::vector<int> > v{new
std::vector<int>};
};
```

Vektor je vo vlastníctve objektu typu Token, nemôže byť kopírovaný

shared_ptr

```
void foo(shared_ptr<Token> p) { //vznik kópie
  cout << p.use_count() << endl; //2
}

int main() {
  shared_ptr<Token> t0 = make_shared<Token>();

  cout << t0.use_count() << endl; //1
  foo(t0);
  cout << t0.use_count() << endl; //1

  return 0;
}
```

use_count(): početnosť využitia (kópií) shared_ptr<>

shared_ptr

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

using namespace std;

class Token {
public:
  Token() = default;
};
```

shared_ptr

```
int main() {
  vector<shared_ptr<Token> > vec;

  for(int i=0; i < 2 ; i++){
    //lokálna premenná
    shared_ptr<Token> t0 = make_shared<Token>();
    //kópia do vec
    vec.push_back(t0);
  } //lokálna premenná t0 zanikne

  for(auto item: vec){ //item je lokálna premenná
    cout << item.use_count() << " "; //2 2
  }

  return 0;
}
```

shared_ptr

```
int main() {
  vector<shared_ptr<Token> > vec;

  for(int i=0; i < 2 ; i++){
    //lokálna premenná
    shared_ptr<Token> t0 = make_shared<Token>();
    //kópia do vec
    vec.push_back(t0);
  } //lokálna premenná t0 zanikne

  for(auto& item: vec){ //item je referencia
    cout << item.use_count() << " "; //1 1
  }

  return 0;
}
```

emplace_back

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

emplace_back

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

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

  v.emplace_back(10);  //objekt vznikne priamo vo vektore
                       //bez kopírovania
  return 0;
}
```

Kopírovacia sémantika

```
int main() {
  std::vector<Token*> v;
  Token* t = new Token(10);

  v.push_back(t); //kopírovanie adresy

  delete t;
  return 0;
}
```

Presúvacia sémantika

```
int main() {
  std::vector<Token> v;
  Token t(10);

  v.push_back(std::move(t)); //presúvanie

  v.emplace_back(10); //dokonalé preposielanie

  return 0;
}
```

Dokonalé preposielanie = perfect forwarding, parametre konštruktora sú preposlané konštruktoru

algorithm: min, max

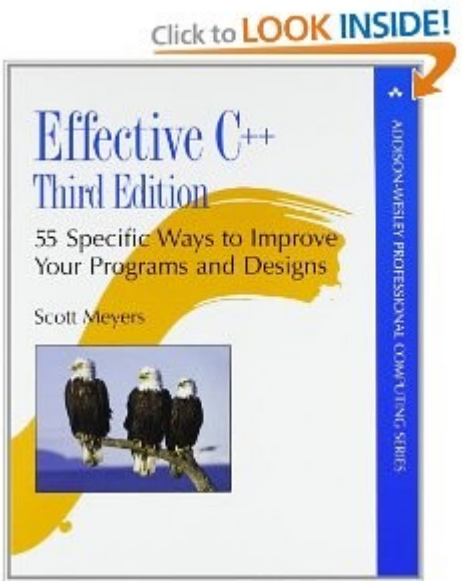
```
#include <algorithm>

int main() {
  int a = std::max(1, 2);
  int b = std::min(2, 3);

  return 0;
}
```

numeric: accumulate

```
#include <iostream>
#include <vector>
#include <numeric>

int main() {
  std::vector<int> v{1, 2, 3};

  int a = std::accumulate(v.begin(), v.end(), 0);

  std::cout << a; //6

  return 0;
}
```

Meyers: Effective C++

Odporúčaná ďalšia literatúra:



Meyers: Effective C++

Item 1: const je lepšie ako #define

Item 2: iostream je lepšie ako stdio.h

Item 20: žiadne public premenné

Item 21: použi const, vždy keď je to možné

Item 32: definuj premenné čo najneskôr

Item 33: inline funkcie predlžujú kód

Item 48: vyrieš všetky varovania kompilátora

Moje odporúčania

Používajte referencie, nekopírujte (ak je to možné).

Kopírujte adresy objektov, objekty je potrebné presúvať.

Zvoľte vhodný kontajner std::vector, std::set, std::map …

Vždy párujte new a delete, pri písaní new

Používajte const premenné, const referencie, const metódy

Používajte C++11 for pre iterovanie cez kontajner

Používajte unique_ptr pre vnútorné kontajnery objektov

Používajte knižnicu algorithm