STU FEI

Programovacie techniky

11. = delete, virtuálna trieda, friend trieda, komparátor, polymorfizmus, template, virtuálny deštruktor, override, final

= delete

Konštruktor alebo metóda označená ako = delete neexistuje.

```
Token(const Token& t) = delete;
```

Kopírovací konštruktor neexistuje.

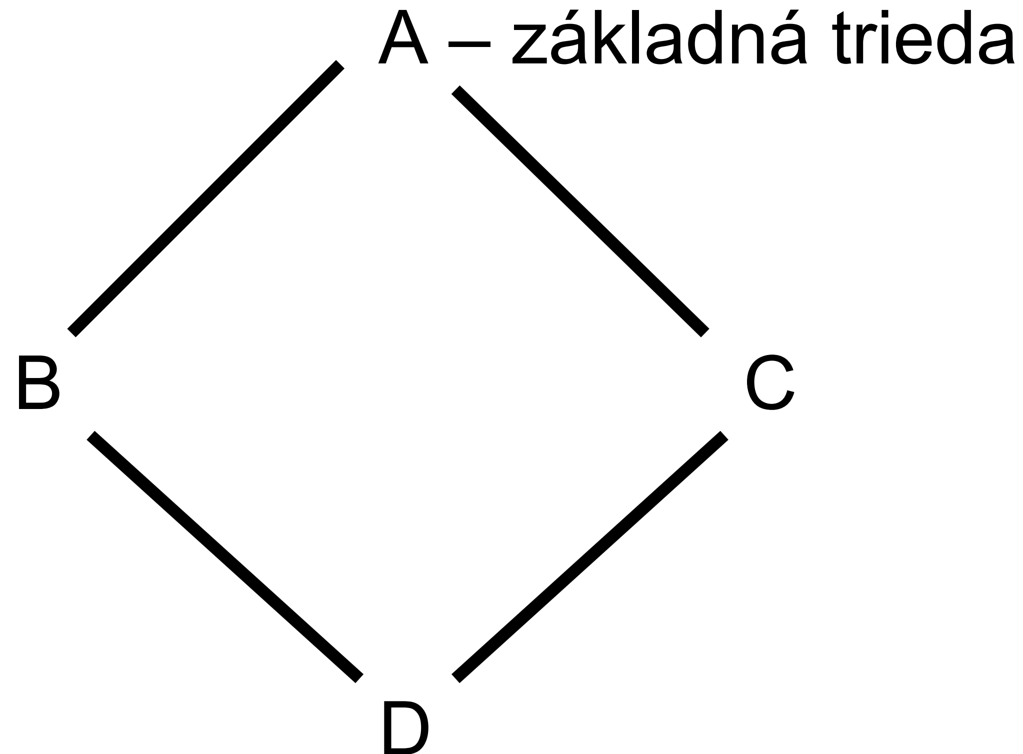
STU FEI

= delete

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

int main(){
  Token t0, t1; //OK
  Token t2 = t0; //chyba, copy constructor
  t1 = t0; //chyba, operator=

  return 0; }
```

error: use of deleted function ‘Token::Token(const Token&)’

Diamantové dedenie



Trieda D dedí z tried B a C, triedy B a C dedia z triedy A

STU FEI

Diamantové dedenie

```
#include <iostream>

using namespace std;

class A {
public:
  int a;
};

class B: public A {
public:
  int b;
};
```

STU FEI

Diamantové dedenie

```
class C: public A {

public:
  int c;
};

class D: public B,C {

public:
  int d;
};
```

STU FEI

Diamantové dedenie

```
int main () {

  D ddd;

  cout << ddd.a << endl;
  //ktoré a? a z triedy B, alebo a z triedy C?

  return 0;
}
```

```
g++: error: request for member ‘a’ is ambiguous
   cout << ddd.a << endl;
                      ^
```

Ktorá premenná a má byť použitá? Z triedy B alebo C?

Virtuálna trieda

```
class B: virtual public A {
public:
  int b;
};

class C: virtual public A {
public:
  int c;
};

class D: public B,C {
public:
  int d;
};
```

STU FEI

Virtuálna trieda

```
int main () {
  D ddd;

  cout << ddd.a << endl; //OK, kompilátor vytvorí
len jednu kópiu a

  return 0;
}
```

friend trieda

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/protected členom inej triedy.

Napr. trieda Automobil by nemala dediť z triedy Clovek, pretože Automobil nie je druh Clovek-a. Ale Automobil môže byť „priatel'" Clovek-a.

friend trieda

```
class Child {

friend class Mother;

public:

  string getName(); //každý môže získať meno

protected:

  void setName(string meno); //Mother ho môže aj
nastaviť
};
```

STU FEI

friend trieda

```
class Automobil {

friend class Clovek; //friend trieda
private:
  std::string spz;

public:
  std::string getSPZ() {
    return spz;
  }

protected:
  void setSPZ(std::string& spz) {
    this->spz = spz;
  }
};
```

friend trieda

```
class Clovek {
public:
  void setMyAutomobilSPZ(Automobil& automobil,
std::string& spz) {
    automobil.setSPZ(spz);
  }
};

int main() {
  Automobil automobil;
  Clovek c;
  std::string spz = "BT101AB";

  c.setMyAutomobilSPZ(automobil, spz);

  return 0; }
```

STU FEI

Komparátor

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional> //greater

using namespace std;

int main () {
  priority_queue<int, vector<int>, greater<int> > pq;

  return 0;
}
```

STU FEI

Vlastný komparátor: template

map <Clovek, int, porovnaj<Clovek> >;

- Porovnávanie objektov typu Clovek potrebujeme pri vkladaní týchto objektov do stromu.
- Ako porovnať dva Clovek objekty? Napr. podľa veku.
- Nestačí preťaženie operátora < ? Preťaženie operátora < nie je možné meniť. Čo ak potrebujeme porovnávať podľa rôznych atribútov?

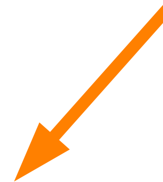
STU FEI

Lambda výraz

```
int main() {

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

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

  return 0;
}
```

Lambda výraz

```
int main() {

  int x = 10;

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

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

  return 0;
}
```



Výstupný typ

STU FEI

Lambda výraz

[&x] : zachytenie premennej x pomocou referencie

[x] : zachytenie premennej x pomocou kópie

[] : bez zachytenia premenných

[&] : zachytenie všetkých premenných v danom scope pomocou referencie

[=] : zachytenie všetkých premenných v danom scope pomocou kópie

Lambda výraz

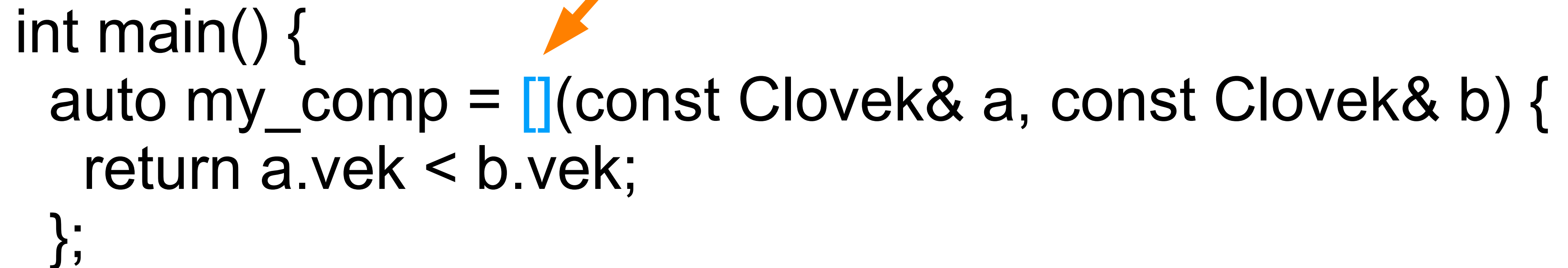
```
class Token {
public:
  int a{1};
  Token(int a0) : a(a0) {}
};

int main() {
  Token t0(2), t1(1);

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

Lambda výraz

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

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

    return 0;
}
```

Komparátor: lambda výraz

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

class Clovek {
public:
  int vek;
};
```



```
int main() {
  auto my_comp = [](const Clovek& a, const Clovek& b) {
    return a.vek < b.vek;
  };
```

Komparátor: lambda výraz

```
    map<Clovek, int, decltype(my_comp)> st(my_comp);
    Clovek a, b;

    st[a] = 1;
    st[b] = 2;

    return 0;
}
```

STU FEI

Polymorfizmus

Jediný kód pre viacej typov:

- preťažovanie funkcií/operátorov
- templates
- virtuálne metódy

Template

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

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

Template

```
int main () {
    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;

    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;

    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

std::vector

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

Class to isté ako typename, pôvodne bolo použité slovo class, neskôr sa začalo používať typename, aby class malo len význam deklarácie triedy. Z dôvodu spätnej kompatibility je možné používať class aj typename skoro zameniteľne.

std::vector<int> v; //int je T

allocator je trieda pre manažovanie pamäte napr. riadi alokáciu pamäte pri zväčšení vektora

STU FEI

std::map

```
template < class Key,
           class T,
           class Compare = less<Key>,
           class Alloc = allocator<pair<const Key,T> >
         > class map;
```

STU FEI

std::unordered_set

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

Hash = hašovacia funkcia
Pred = predikát rovnosti prvkov, potrebný pri pokuse o vloženie identického prvku do unordered_set.

Základné numerické typy std::string majú definovanú hašovaciu funkciu a predikát rovnosti prvkov.

Virtuálny deštruktor

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

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

STU FEI

Virtuálny deštruktor

```
int main() {
  Derived* d0 = new Derived();
  Base* b = (Base*) d0;

  delete b;
  return 0;
}
```

Zavolaný len deštruktor triedy Base.

Virtuálny deštruktor

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

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

Virtuálny deštruktor

```
int main() {
  Derived* d0 = new Derived();
  Base* b = (Base*) d0;

  delete b;
  return 0;
}
```

Zavolaný len deštruktor triedy Base, ako aj deštruktor triedy Derived.

Virtuálna metóda

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

class Derived : public Base {
public:
  ~Derived() {}
  void foo() override {
    std::cout << "Derived";
  }
};
```

Virtuálna metóda

```
int main() {
  Derived* d0 = new Derived();
  Base* b = (Base*) d0;

  b->foo();

  delete b;
  return 0;
}
```

Zavolaná je metóda foo() triedy Derived.

STU FEI

override, final

Znova definujem virtuálnu metódu:

```
void foo() override {
    std::cout << "Derived";
 }
```

Znova definujem virtuálnu metódu, pričom táto metóda už nebude ďalej dedená:

```
void foo() final {
    std::cout << "Derived";
 }
```