STU FEI

Programovacie techniky

6. string, uniformná inicializácia, konverzný konštruktor, preťažovanie operátorov, kopírovanie (copy), presúvanie (move)

Pre niekoho ovocie

“Durian”, William Farquhar, Singapore 1819-1823.

Pre niekoho trieda

```
#include <string>

class Fruit {

private:
   int weight;
   int color;
   std::string name;
   bool isEdible;

public:
   void setName(const std::string& name) {
      this->name = name;
   }
   //ďalšie metódy
};
```

string

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

int main() {
   std::string str1 = "Hello", str2 = "World", str3;
   int  len ;

   str3 = str1;   //prekopíruj str1 do str3
   cout << "str3 : " << str3 << endl;

   str3 = str1 + str2;  //spojenie str1 a str2
   cout << "str1 + str2 : " << str3 << endl;

   len = str3.size();  //veľkosť str3
   cout << "str3.size() :  " << len << endl;

   return 0; }
```

string

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

int main() {
    string str1 = "remove aaa";
    str1.erase(7, 3); //zmaže aaa

    string str2 = "ade";
    str2.insert(1, "bc"); //abcde

    return 0;
}
```

string

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

int main() {
  std::string str = "We think in generalities, but we live in details.";

  std::string str2 = str.substr(12,12);   // "generalities"

  unsigned pos = str.find("live");          // pozícia "live" v str (33). Prvý výskyt reťazca.

  std::string str3 = str.substr(pos);     //od "live" až po koniec

  return 0;
}
```

string

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

int main() {
  string str = "password";

  if(str == "password") {
     cout << "ur using a weak password";
  }

  return 0;
}
```

string: c_str()

```
#include <string>
#include <cstring> //string.h, C knižnica

using namespace std;

int main() {
   string str1 = "I love PT!";
   char str[100];

   strcpy (str, str1.c_str());
   return 0;
}
```

c_str() potrebné pre spätnú kompatibilitu s C
c_str() vráti const char*

string: c_str()

```
#include <string>
#include <cstdio>

using namespace std;

int main() {
   string str1 = "1 8 16 23 78";
   int a1, a2, a3, a4, a5;

   sscanf(str1.c_str(), "%d %d %d %d %d", &a1, &a2,
&a3, &a4, &a5);

   return 0;
}
```

Inicializácia premenných

„C“ inicializácia:

```
int a = 1;
double b = 3.4;
```

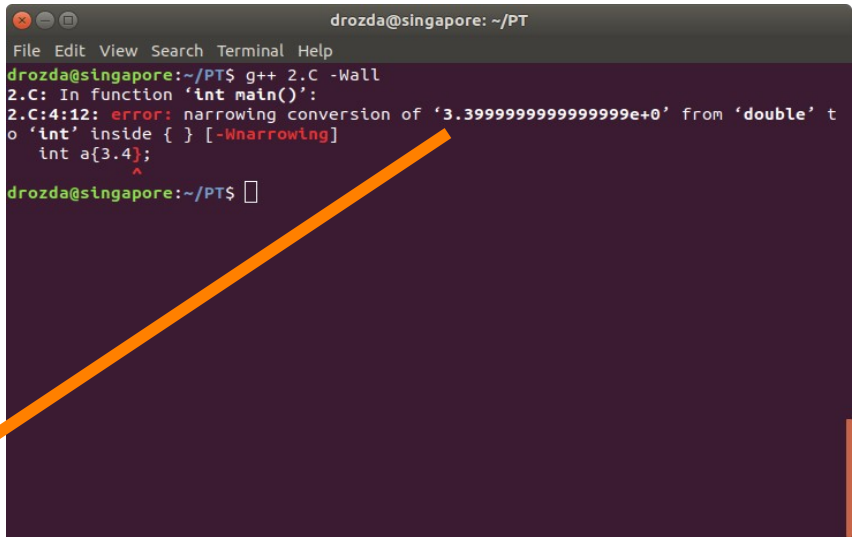
C++ (ako štandardný objekt):

```
int a(1);
double b(3.4);

int a = int(3); //dočasný objekt, ktorý sa prekopíruje
double b = double(3.4);

int a();        //funkcia!
double b();  //funkcia!
```

Inicializácia premenných {}

Uniformná inicializácia:

```
int a{1};
double b{3.4};

int a = int{3};
double b = double{3};

int a{};          //OK, a = 0
double b{};  //OK, b = 0.0

int a{3.4};   //chyba, zúženie typu
```

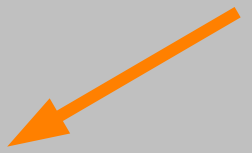


{} inicializácia zvyšuje typovú bezpečnosť

Inicializácia

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

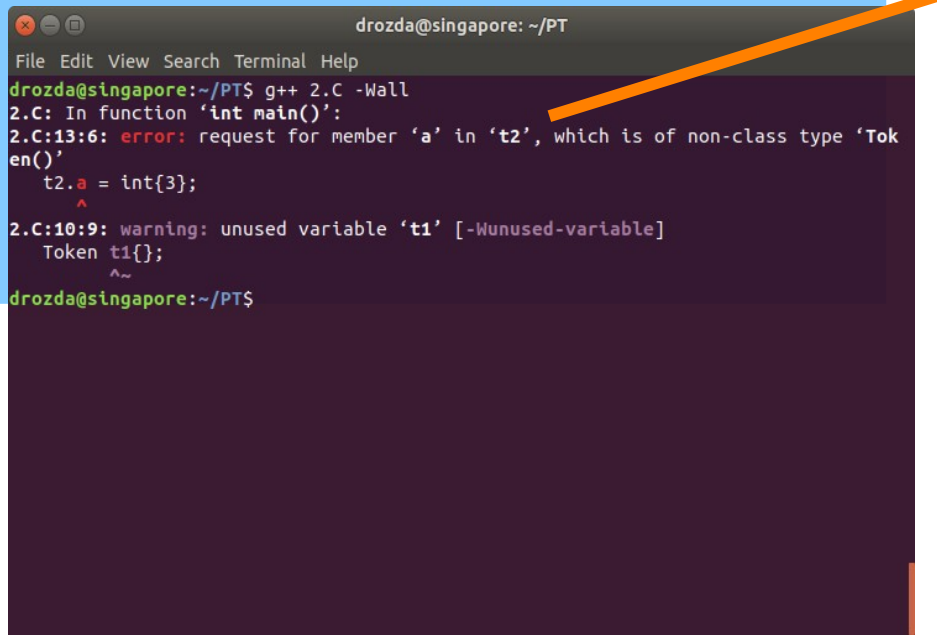
```
int main() {
  Token t0;
  Token t1{};

  Token t2(); //funkcia!

  t0.a = int{3};

  return 0;
}
```



Inicializácia

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

```
int main() {
  Token t0;
  Token t1{};

  Token t2(); //funkcia!

  t2.a = int{3};

  return 0;
}
```



Konverzný konštruktor

```
class Token {
public:
  int a{1};
  Token(int t0): a(t0) {}
};
```

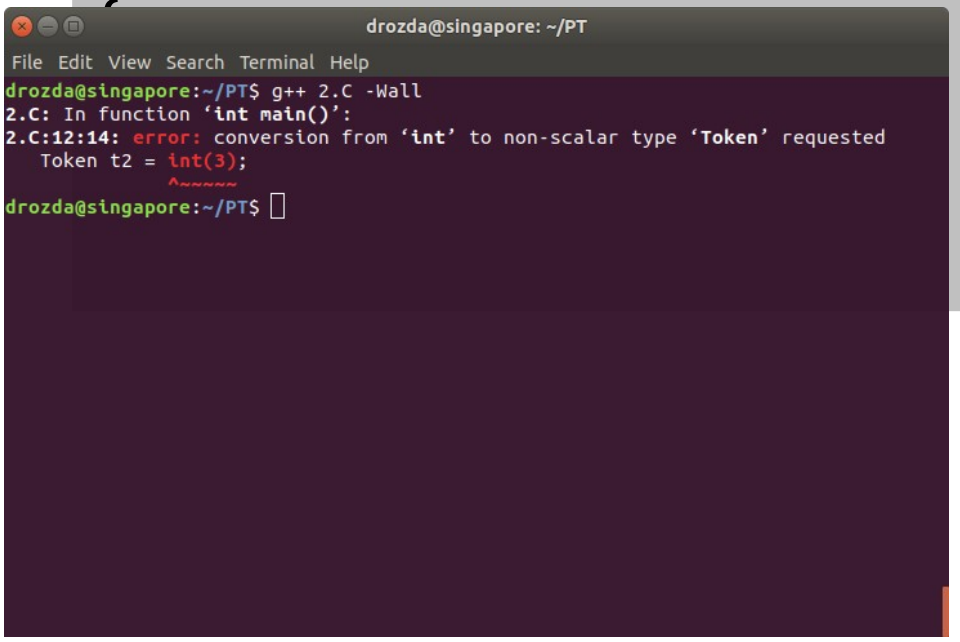
```
int main() {
  Token t0(3);
  Token t1{3};

  Token t2 = 3;

  return 0;
}
```

Token t2 = 3;
Na pravej strane je int.

explicit

```
class Token {
public:
  int a{1};
  explicit Token(int t0):
a(t0) {}
};
```

```
int main() {
  Token t0(3);
  Token t1{3};

  Token t2 = int(3);

  return 0;
}
```



Token(int) už nie je konverzný

explicit

```
class A {};

class B {
public:
    B() {}
    //explicit B(A const& a) {}
    B(A const& a) {}
};

void f(B const& b) {}
```

```
int main() {
    A obj;
    f(obj);

    return 0;
}
```

B const& b = obj;

Vždy používať explicit pre konštruktory s 1 parametrom

Nastane zavolanie B(A const&).

Preťažený operátor =

```
class Token {
public:
  int a{1};
  Token& operator=(const
Token& t0) {

    this->a = t0.a;
    return *this;
  }
};
```

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

  return 0;
}
```

Zavolá sa preťažený operátor =.

Preťaženie, kopírovanie

```
class Token {
public:
  Token() = default;
  Token(const Token& t0) { }
  Token& operator=(const Token& t0) {
    return *this;
  }
};

int main() {
  Token t0, t1;    //zavolá sa defaultný konštruktor 2x
  Token t2 = t0;  //zavolá sa kopírovací konštruktor
  t1 = t2;            //zavolá sa preťažený operátor =

  return 0; }
```

Preťažovanie

Operátor + je pre základné numerické typy jednoznačne definovaný:

```
double a = 3.4 + 1.8;
```

Ako je operátor + definovaný pre triedu?

```
class Student {
public:
  double weight = 82.3;
};

Student s1, s2;
Student s = s1 + s2; //s = ??
```

Preťažený operátor +

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

  Token operator+(const
Token& t0) {
    Token t;
    t.a = this->a + t0.a;

    return t;
  }
};
```

```
int main() {
  Token t0, t1, t2;
  t2 = t1 + t0;

  return 0;
}
```

Zavolá sa preťažený operátor + na objekte t1.

Preťaženie =, +

Nepotrebujeme nový objekt:

```
Token& operator=(const Token& t0) {
    this->a = t0.a;
    return *this;
}
```

Potrebujeme nový objekt:

```
Token operator+(const Token& t0) {
    Token t;
    t.a = this->a + t0.a;
    return t;
}
```

Môžeme preťažiť

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

new delete new[] delete[]: preťaženie je tiež možné

Je možné preťažiť < tak, aby < vykonávalo > ?
Áno, ale nie je to odporúčané ;)

Referencia &

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

  Token(int& a0): a(a0) {} //referencia neviaže konštantu
};

int main() {
  Token t(3); //chyba

  return 0;
}
```

const referencia &

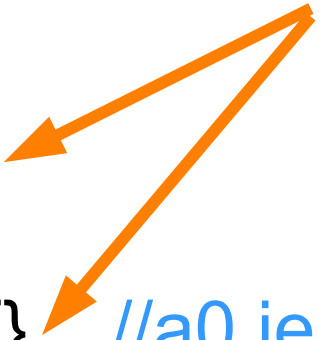
```
class Token {
public:
  int a{1};

  Token(const int& a0): a(a0) {} //OK
};

int main() {
  Token t(3); //OK

  int a = 3;
  Token t1(a); //OK

  return 0;
}
```

Neviem rozlíšiť medzi konštantou 3 a premennou a

Referencia &&

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

  Token(int& a0): a(a0) {}

  Token(int&& a0): a(a0) {}  //a0 je prekopírované do a
};

int main() {
  Token t(3); //zavolá sa Token(int&&)

  int a = 3;
  Token t1(a); //zavolá sa Token(int&)

  return 0; }
```

Viem rozlíšiť medzi konštantou 3 a premennou a



std::move

```
#include <utility>

class Token {
public:
  int a{1};

  Token(const int& a0): a(a0) {}
  Token(int&& a0) {}  //&a0: 0x7ffde6d1e010
};

int main() {
  int a = 3; //&a: 0x7ffde6d1e010

  Token t(std::move(a)); //zavolá sa Token(int&&)
                         //std::move pretypuje a na xvalue
  return 0; }
```

lvalue, rvalue

Pred C++11:

Adresa l-hodnoty je známa.

r-hodnota je všetko, čo nie je l-hodnota.

lvalue, rvalue, prvalue, xvalue

Od C++11:

prvalue („pure“ rvalue)

- 3, true, nullptr, this
- To čo vráti funkcia:

```
int foo(int a, int b) {
    return a + b;
}
```

- & adresa
- a++, a--, a + b, a % b, a & b, a << b, a && b, a || b, !a, a < b, a == b, a >= b

lvalue, rvalue, prvalue, xvalue

Od C++11:

lvalue

- premenná (pretože má adresu)
- To čo vráti funkcia:

```
int& foo(int& a) {
    return a;
}
```

- a = b, a += b, a %= b //priradenie
- ++a, --a //prefixová inkrementácia a dekrementácia
- a ? b : c, ak b a c sú lvalue a majú rovnaký type

lvalue, rvalue, prvalue, xvalue

Od C++11:

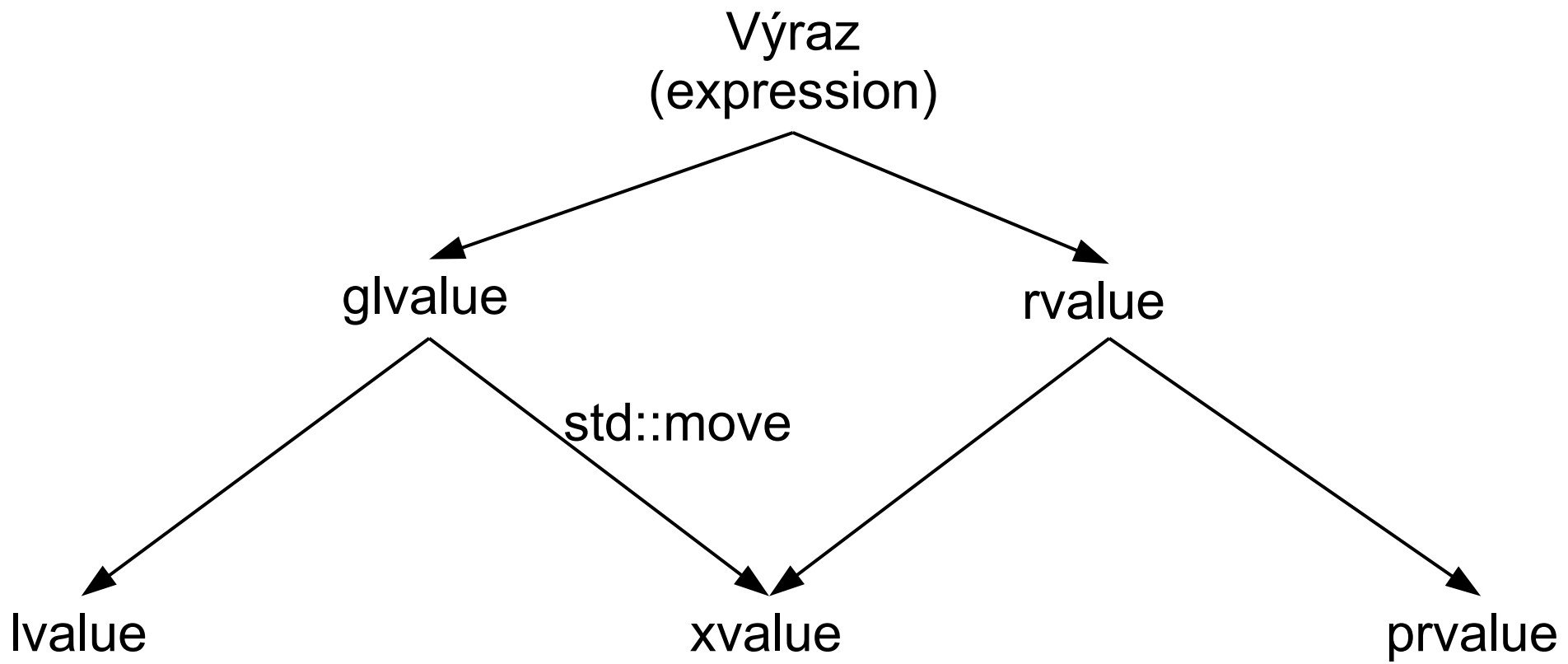
xvalue

- std::move(a)

rvalue je buď prvalue alebo xvalue

- &int(), &i++[3], &42, &std::move(x) : použitie operátora & nie je možné
- rvalue nemôže byť na ľavej strane, napr. 3 = a;

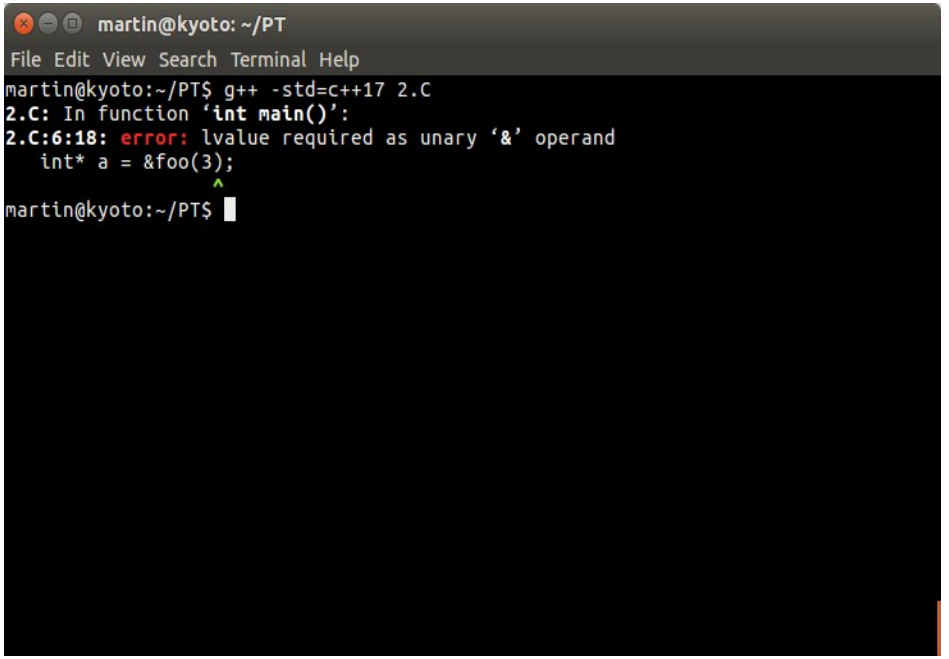
lvalue, rvalue, prvalue, xvalue



glvalue = generalized lvalue

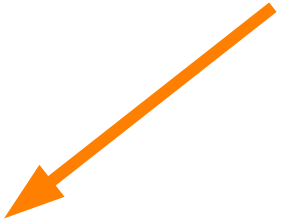
rvalue

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

int main() {
  int* a = &foo(3); //chyba, foo(int) vráti rvalue

  return 0;
}
```



rvalue

```
#include <utility>

class Token {
public:
  int a{1};

  Token(const int& a0): a(a0) {}
  Token(int&& a0) {}
};

int main() {
  int a = 3;
  int&& b = std::move(a);
  Token t(b); //zavolá sa Token(const int&)

  return 0; }
```



move(a) pretypuje a na xvalue, ale b je referencia &, pretože vznik rvalue referencie && by vyžadoval kopírovanie

rvalue

```
#include <utility>

class Token {
public:
  int a{1};

  Token(const int& a0): a(a0) {}
  Token(int&& a0) {}
};

int main() {
  int a = 3;

  Token t(std::move(a)); //zavolá sa Token(int&&)

  return 0; }
```

std::move je pretypovanie

```
#include <iostream>
#include <utility>

int main() {
  int a = 1;
  std::cout << &a; //0x7ffdfda2c874

  int& b = a;
  std::cout << &b; //0x7ffdfda2c874

  int&& c = std::move(a); //len pretypovanie, c je lvalue
  std::cout << &c; //0x7ffdfda2c874

  return 0; }
```

Kopírovanie, presúvanie

Kopírovanie (copy) znamená, že obsah objektu obj0 je prekopírovaný do obj1, pričom objekt obj0 ostane zachovaný v pôvodnom stave.

Presúvanie (move) znamená, že obsah objektu obj0 je presunutý do obj1, pričom objekt obj0 bude "vyprázdnený".

Kopírovanie, presúvanie

```
#include <iostream>
#include <string>
#include <utility>
using std::string;

class Token {
public:
  string s{};
  Token(const string& s0) : s{s0} {} //konštruktor

  explicit Token(const Token& t0) : s{t0.s} { } //copy konš.

  explicit Token(Token&& t0) { //move konštruktor
    this->s = std::move(t0.s);
  }
};
```

Kopírovanie, presúvanie

```
int main() {
  Token t0{"Hello, World!"}; //zavolá Token(const string&)

  Token t1{t0}; //zavolá Token(const Token&)

  std::cout << t0.s;              //Hello, World!

  Token t2(std::move(t0)); //zavolá Token(Token&&)

  std::cout << t0.s;              //nevypíše nič
  std::cout << t0.s.size();    //0

  return 0;
}
```

Presúvanie

```
explicit Token(Token&& t0) { //move konštruktor
    this->s = std::move(t0.s);
  }
```

s existuje, zavolá sa preťažený operátor = pre triedu std::string.

Zavolá sa string& operator=(string&&), pretože vstupná hodnota je xvalue (pretypovanie pomocou std::move)

Príklad:

```
string& operator=(string && str) {
    std::swap(data, str.data); //výmena smerníkov
    std::swap(m_size, str.m_size);
    return *this;
}
```

Presúvanie

```
#include <iostream>
#include <string>
#include <utility>
using std::string;

class Token {
public:
  string s{};
  Token() = default;
  Token(const string& s0) : s{s0} {}
  Token(Token&& t0) {
    this->s = std::move(t0.s);
  }
  Token& operator=(Token&& t0) {
    this->s = std::move(t0.s);
    return *this;
  }
};
```

Presúvanie

```
#include <iostream>
#include <string>
#include <utility>
using std::string;

class Token {
public:
  string s{};
  Token() = default;
  Token(const string& s0) : s{s0} {}
  Token(Token&& t0) {
    *this = std::move(t0); //zjednodušená implementácia
  }
  Token& operator=(Token&& t0) {
    this->s = std::move(t0.s);
    return *this;
  }
};
```

Presúvanie

```
int main() {
  Token t0;
  t0 = Token("Hello, World!"); //operator=(Token&&)
                               //t0 už existuje
                               //"Hello, World!" je prvalue

  std::cout << t0.s;           //Hello, World!
  std::cout << t0.s.size();    //13

  return 0;
}
```

Automatické generovanie

Vždy sa vygeneruje:

- Defaultný konštruktor
- Copy konštruktor
- Operátor priradenia
- Deštruktor

A tiež:

- Move konštruktor
- Operátor priradenia pre move

Automatické generovania nastane len v prípade potreby, napr. ak naozaj je copy konštruktor potrebný.

Úloha

Naimplementuj merge sort pomocou presúvania (move), namiesto kopírovania