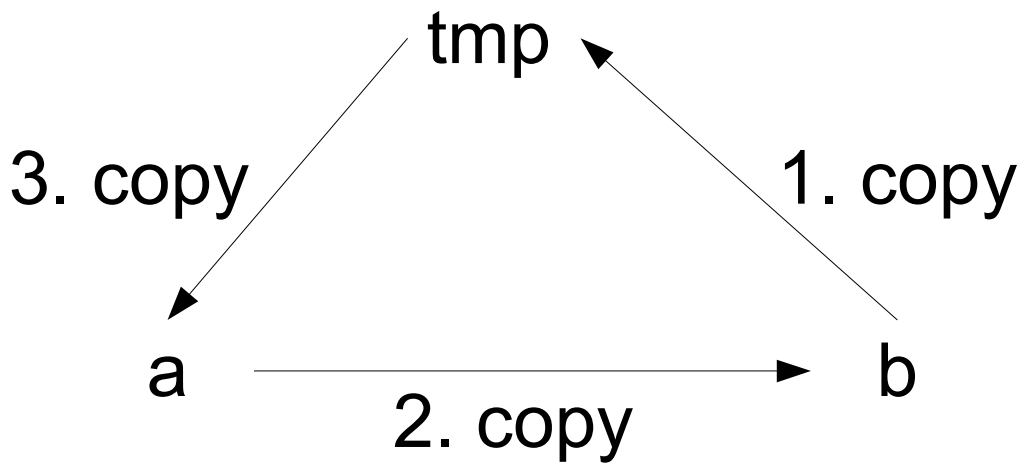
STU FEI

Programovacie techniky

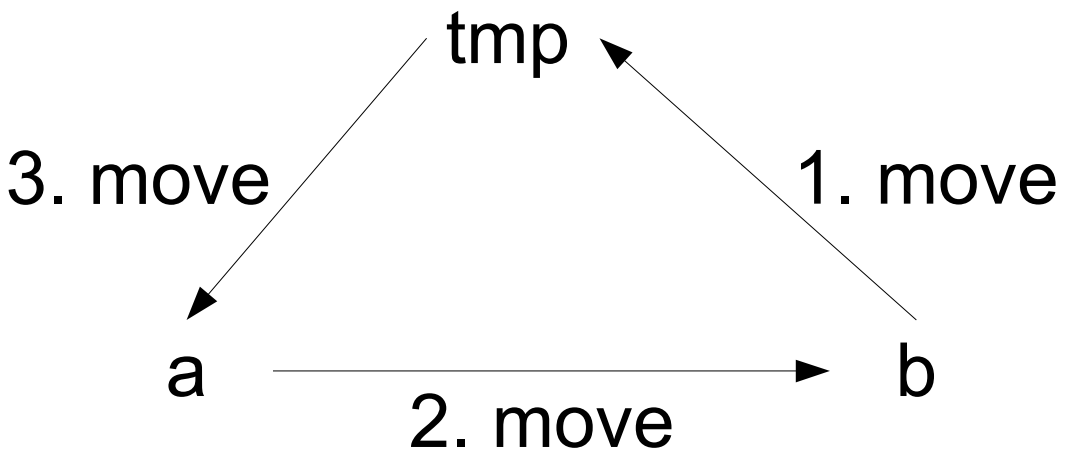
7. auto, decltype, STL, vector, iterátory, výnimky

Swap (copy)



Výmena hodnôt a, b.
V tmp ostane kópia b.

Swap (move)



Výmena hodnôt a, b pomocou move.
tmp ostane prázdne.

Move nastane, len ak je podporovaný.

```
T tmp(std::move(b));
b = std::move(a);
a = std::move(tmp);
```

auto, decltype

auto: odvodenie typu
decltype: typ je rovnaký ako odvodený typ

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

auto a0;              //chyba, typ sa nedá odvodiť
```

auto

```
auto a = 1;
auto&& b = a;  //b má typ int&, pretože a je l-hodnota

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

auto

Odvodenie návratového typu funkcie:

```
auto foo(const std::string& s) {
  return s.size();
}
```

auto môže zabezpečiť použitie správneho typu.

size_t je unsigned int. V nasledovnom sme ale “omylom” použili int.

```
int foo(const std::string& s) {
  return s.size();
}
```

STL

STL = Standard template library

STL obsahuje efektívne implementácie bežných dátových štruktúr ako napr. vector, array, list, forward_list, stack, queue, deque, map, set, multimap, multiset.

STL podporuje move sémantiku, presun (move) sa udeje, ak je to možné.

Dátová štruktúra (STL) využíva kontajner, kde sú uložené objekty.

Stack, queue môžu využiť deque (double ended queue), vector a array sú implementované bez využitia inej dátovej štruktúry.

vector

Vector zaberá v pamäti spojitú oblasť.

Veľkosť sa automaticky prispôsobuje počtu vložených prvkov, pričom nastane realokácia pamäte.

Umožňuje pridávanie prvkov na koniec, ich odstraňovanie z konca pri zložitosti O(1).

Ale aj pridávanie a odstraňovanie prvkov na ľubovolnej pozícii, ale zložitosť je potom O(n).

vector

```
std::vector<int> v0;          //v kontajneri sú int
std::vector<double> v1;   //v kontajneri sú double

std::vector<int*> v2;         //v kontajneri sú int*
std::vector<double*> v3;  //v kontajneri sú double*

std::vector<Token> v4;  //v kontajneri sú objekty typu Token
std::vector<Token*> v5; //v kontajneri sú objekty typu Token*

std::vector<std::vector<int> > v6;
std::vector<std::vector<int>* > v7;

vector<vector<int> >* v8 = new vector<vector<int> >;
vector<vector<int>* >* v9 = new vector<vector<int>* >;
auto v10 = new vector<vector<int> >;
```

vector

Metódy:

size: počet prvkov
max_size: max. povolená veľkosť

empty: je prázdny?

push_back: vlož dozadu
pop_back: odstráň zozadu
clear: odstráň všetky prvky (size=0)

resize: zmeň veľkosť

vector

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

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

  for (int i = 0; i < 5; ++i) {
    v.push_back(i);
  }

  v.pop_back();

  std::cout << v.front(); //0
  std::cout << v.back(); //3

  return 0; }
```

vector

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

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

  for (int i = 0; i < 5; ++i) {
    v.push_back(i);
  }

  std::cout << v.size();          //5, počet prvkov
  std::cout << v.capacity();      //8, alokovaná veľkosť
  std::cout << v.max_size(); //4611686018427387903
                                  //max. veľkosť
  return 0;
}
```

vector

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

int main () {
  std::vector<int> v(100); //100 je počiatočná veľkosť

  for (int i = 0; i < 5; ++i) {
    v.push_back(i);
  }

  std::cout << v.front();       //0
  std::cout << v.size();        //105
  std::cout << v.capacity(); //200

  return 0;
}
```

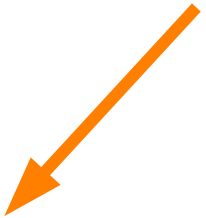
vector

```
#include <vector>

int main () {
  std::vector<int> v(); //funkcia!

  return 0;
}
```

vector

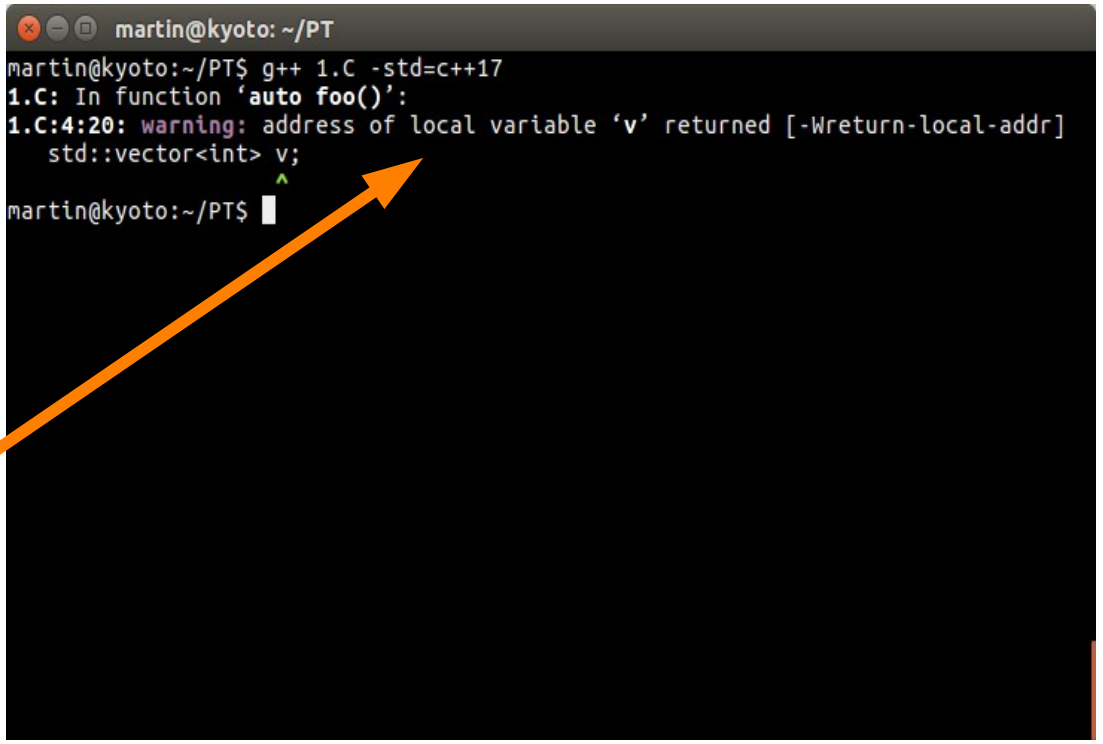
```
#include <vector>
#include <iostream>

class Token {
  int a{1};
};

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

  return 0;
}
```

V STL kontaineroch vznikajú položky kopírovaním



vector

```
#include <vector>

auto foo() {
  std::vector<int> v;

  return &v;
}

int main () {
  std::vector<int>* v = foo();

  return 0;
}
```



Kopírovanie, presúvanie

Pri vkladaní smerníkov (adries) nastane kopírovanie:

```
auto v0 = new vector<Token*>;
Token* t = new Token;
v0->push_back(t);
```

Na vloženie objektov do vektora je vhodné použiť presúvanie (move):

```
auto v1 = new vector<Token>;
Token t;
v1->push_back(std::move(t));
```

Výhoda je vkladanie objektu, ktorý určite existuje (vyhneme sa nullptr). Nevýhoda je nižšia výkonosť.

vector: at

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

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

  for (unsigned i = 0; i < v.size(); i++) {
    v.at(i) = i; //na pozícii i je hodnota i
  }

  for (unsigned i = 0; i < v.size(); i++) {
    std::cout << ' ' << v.at(i);
  }

  return 0; }
```

vector: [], at

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


int a = myvector.at(100);  //out_of_range, C++

int a = myvector[100];  //bez overenia rozsahu, “C” štýl
```

Pokus čítať/zapisovať mimo rozsah s at možné ošetriť try/catch.

Mimo rozsah: try...catch

```
#include <iostream>
#include <stdexcept>
#include <vector>

int main () {
  std::vector<int> v(10);
  try {
    v.at(20)=100;

  } catch (std::out_of_range& oor) {
    std::cerr << "Out of range: " << oor.what();
  }

  return 0; }
```

Out of range:
vector::_M_range_check
: __n (which is 20) >=
this->size() (which is 10)

Vygenerovanie výnimky

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

int main () {
  try {
    throw 20;

  } catch (int e) {
    cout << "An exception occurred. Exception Nr. " << e;
  }

  return 0;
}
```

try...catch

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

double div(int a, int b) {
   if(b == 0) {
      throw "Division by zero
condition!";
   }

   return (a/b);
}
```

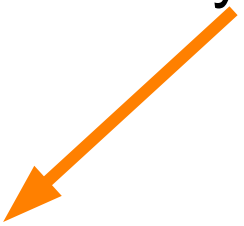
```
int main () {
   double z;

   try {
     z = div(50, 0);
     cout << z << endl;
   } catch (const char* msg {
     cerr << msg;
   }
   return 0;
}
```

Po throw je beh funkcie div(int, int) ukončený!

try...catch

```
#include <iostream>
#include <new>

int main () {
  int* pole = nullptr;

  try {
    pole = new int[10000];

  } catch (std::bad_alloc& ba) {
    std::cerr << "bad_alloc caught: " << ba.what();
  }

  delete[] pole;
  return 0;
}
```

Alokácia pamäte môže byť neúspešná



Odvinutie zásobníka

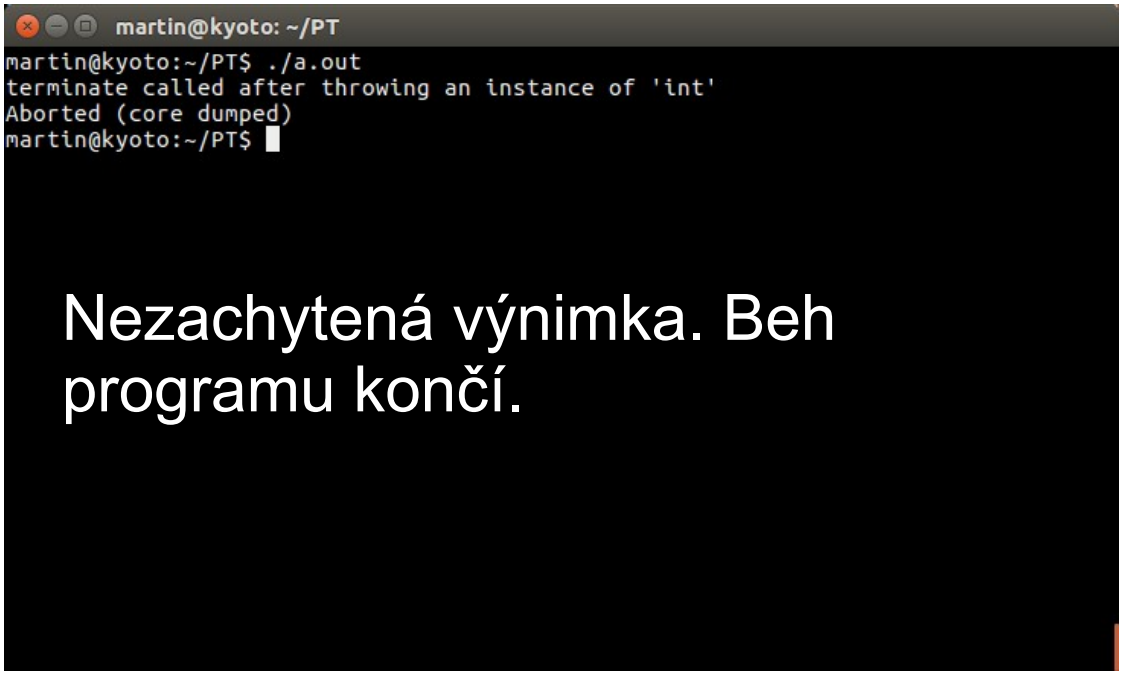
```
void foo() {
  int* array = new int[10000000];

  throw 21; //vygeneruje („vyhodí“) výnimku

  delete[] array; //nebude nikdy zavolané
}
```

Po throw nastane odvinutie zásobníka (stack unwinding), nastane odstránenie lokálnych premenných zo stacku, vrátane premennej array.

delete[] nebude zavolaný, nastane únik pamäte (memory leak).

Výnimka

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

  throw 21;

  delete[] array;
}

int main() {
  foo();

  return 0;
}
```



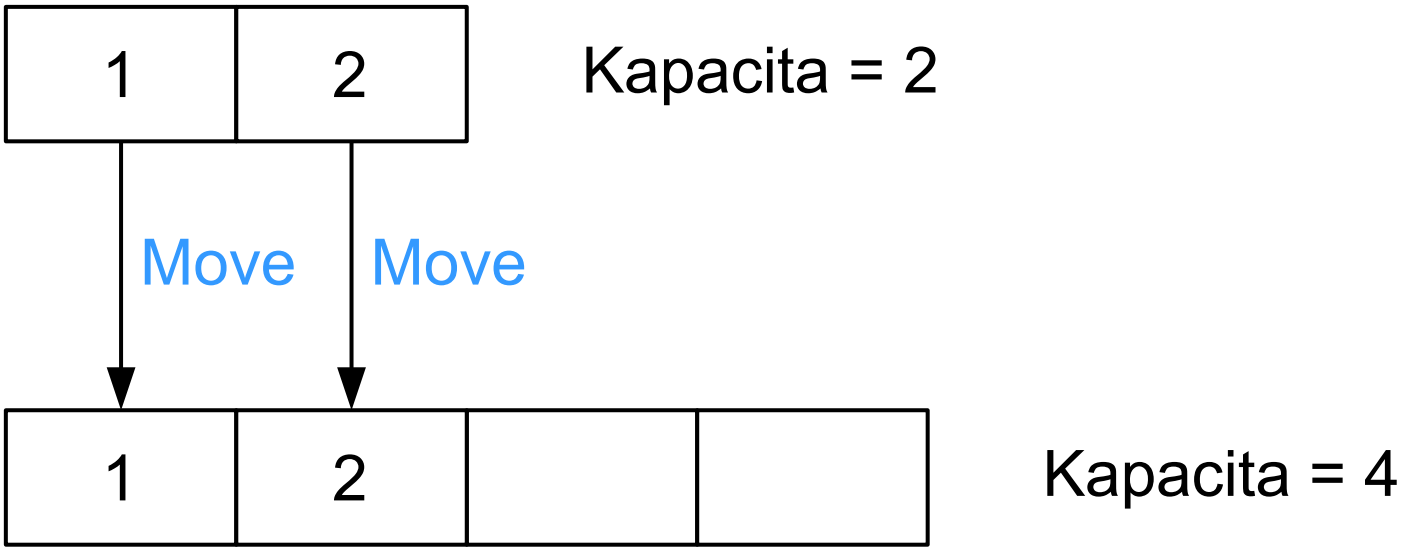
Nezachytená výnimka. Beh programu končí.

Výnimka

```
#include <iostream>

void foo() {
  int* array = new int[10000000];
  throw 21;
  delete[] array; //nebude nikdy zavolané
}

int main() {
  try {
    foo();
  } catch (int& e) {
    std::cout << "Exception!";
  }

  return 0; }
```

Zachytená výnimka

Vector, zmena veľkosti

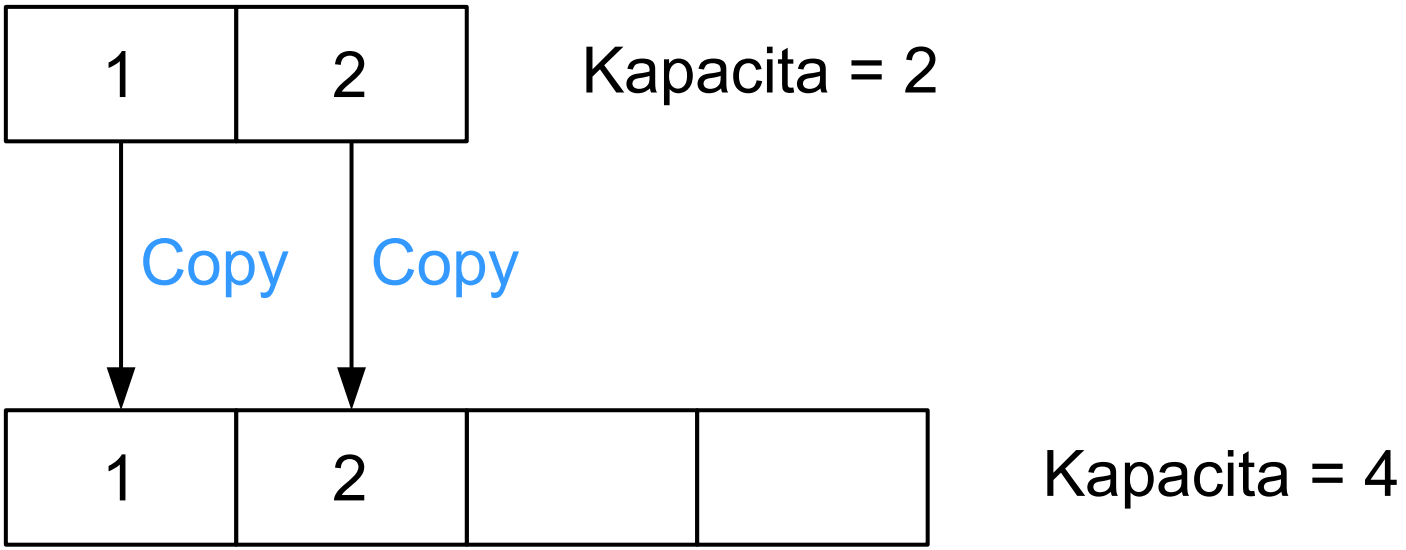


Veľkosť vektora (alokovaná kapacita) sa zvýšila z 2 na 4.

Existujúce prvky sú presunuté pomocou move.

Maybe.

Vector, zmena veľkosti



Kompilátor použije move, ak move konštruktor nemôže vyhodiť výnimku, ináč použije copy.

Copy, move

Ak výnimka nastane počas copy, stále existuje pôvodný objekt.

Ak výnimka nastane počas move, pôvodný objekt už nemusí existovať v pôvodnom stave.

Riešenie: move konštruktor a move priradenie nesmú generovať výnimky

Ako: zakážeme im to.

```
Token(Token&& t) noexcept {
   //implementácia move
}
```

noexcept

noexcept označuje metódu, ktorá negeneruje (nevyhadzuje) výnimky.

noexcept umožňuje optimimalizáciu kódu, pretože odvinutie zásobníka nebude potrebné.

Ak nastane výnimka v metóde označenej ako noexcept, beh programu končí. Kód bol skompilovaný bez podpory odvinutia zásobníka a teda takúto vynimku ani nevieme zachytiť.

Príklad 1

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

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

int main() {
  Token t0, t1;
  std::vector<Token> v(0);

  v.push_back(t0);
  v.push_back(t1); //~Token(), resize vektora

  return 0; }          //~Token() 4x
```

Príklad 2

```
#include <iostream>
#include <vector>
class Token {
public:
  Token() {
    std::cout << "Token()";
  }
  Token(const Token& t) {
    std::cout << "Token(const Token&)";
  }
  Token(Token&& t) {
    std::cout << "Token(Token&&)";
  }
  ~Token() {
    std::cout << "~Token()";
  }
};
```

Príklad 2

```
int main () {
  std::vector<Token> v(0);    //veľkosť vektora je 0
  Token t;                    //Token()

  std::cout << v.capacity();  //0
  v.push_back(std::move(t)); //Token(Token&&)
  std::cout << v.capacity();  //1
  v.push_back(t);              //Token(const Token&)
                               //Token(const Token&)
                               //~Token()
  std::cout << v.capacity();  //2

  return 0;
}                             //~Token() 3x
```

Príklad 3

```
#include <iostream>
#include <vector>
class Token {
public:
  Token() {
    std::cout << "Token()";
  }
  Token(const Token& t) {
    std::cout << "Token(const Token&)";
  }
  Token(Token&& t) noexcept {
    std::cout << "Token(Token&&)";
  }
  ~Token() {
    std::cout << "~Token()";
  }
};
```

Príklad 3

```
int main () {
  std::vector<Token> v(0);     //veľkosť vektora je 0
  Token t;                     //Token()

  std::cout << v.capacity();   //0
  v.push_back(std::move(t)); //Token(Token&&)
  std::cout << v.capacity();   //1
  v.push_back(t);               //Token(const Token&)
                               //Token(Token&&)
                               //~Token()
  std::cout << v.capacity();   //2

  return 0;
}                              //~Token() 3x
```

noexcept

Move konštruktor a move priradenie musia byť označené ako noexcept pre využitie move sémantiky v STL.

```
Token(Token&& t) noexcept {
    std::cout << "Token(Token&&)";
    //implementácia move
}

Token& operator=(Token&& t) noexcept {
    std::cout << "operator=(Token&&)";
    //implementácia move
    return *this;
}
```

noexcept

Deštruktory sú prednastavené noexcept.

Ak nevieme dealokovať/deštruovať objekt, pravdepodobne nevieme pokračovať. Deštruktor preto nepotrebuje odvinutie zásobníka.

Iterátory: begin, end

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

int main() {
  vector<int> v;
  vector<int>::iterator itv;
  v.push_back(22); v.push_back(22997845);

  itv = v.begin();
  while(itv != v.end()) {
    cout << *itv << ' ';
    itv++;
  }

return 0; }
```

Const iterátory: cbegin, cend

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

int main() {
  vector<int> v;
  vector<int>::const_iterator itv;
  v.push_back(22); v.push_back(22997845);

  itv = v.cbegin();
  while(itv != v.cend()) {
    cout << *itv << ' ';
    itv++;
  }

return 0; }
```

Iterátory

Iterátor:

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

Const iterátor:

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

Ak prvky v kontajneri nemením je lepšie použiť const iterátor a umožniť kompilátoru optimalizovať (ak sa dá).

C++11 for

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

int main() {
  vector<int> v;
  v.push_back(22); v.push_back(22997845);

  for(int x : v) { //v x je kópia
    cout << x << ' ';
  }

  return 0;
}
```

C++11 for

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

int main() {
  vector<int> v;
  v.push_back(22); v.push_back(22997845);

  for(auto x : v) { //v x je kópia
    cout << x << ' ';
  }

  return 0;
}
```

C++11 for

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

int main() {
  vector<int> v;
  v.push_back(22); v.push_back(22997845);

  for(auto& x : v) { //x je referencia
    cout << x << ' ';
  }

  return 0;
}
```

sort

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

int main() {
  std::vector<int> v = {1, 1, 3, 22, 4, 123};

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

  for(auto x : v) {
    std::cout << x << ' ';  //1 1 3 4 22 123
  }

  return 0;
}
```

stable_sort

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

int main() {
  std::vector<int> v = {1, 1, 3, 22, 4, 123};

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

  for(auto x : v) {
    std::cout << x << ' ';  //1 1 3 4 22 123
  }

  return 0;
}
```

stable_sort

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

int main() {
  std::vector<int> v = {1, 1, 3, 22, 4, 123};

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

  for(auto x : v) {
    std::cout << x << ' ';  //123 22 4 3 1 1
  }

  return 0;
}
```

max_element

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

int main() {
  std::vector<int> v = {1, 1, 3, 22, 4, 123};

  auto result = std::max_element(v.cbegin(), v.cend());

  std::cout << *result; //result je iterátor na max element

  return 0;
}
```