NPRG041 – C++

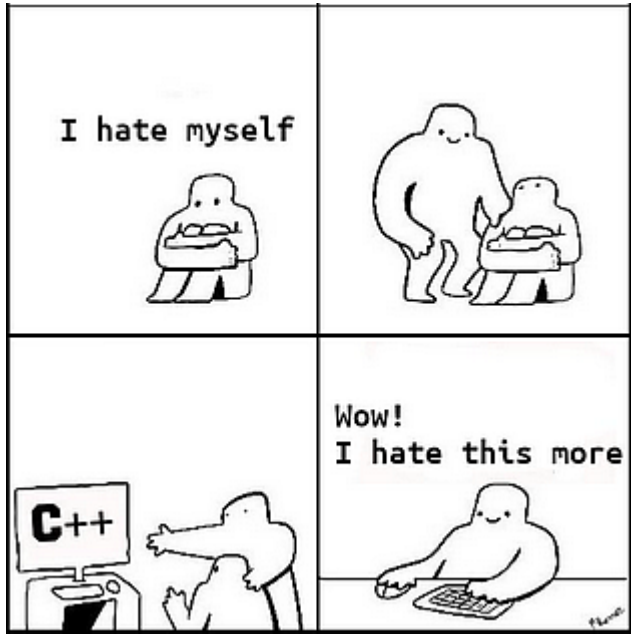
cvičení – Jiří Klepl

mattermost: ulita/2425ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

Klepl@d3s.mff.cuni.cz

Agenda

- Výjimky
  - std::exception, std::runtime_error, std::logic_error
  - noexcept a exception safety
- Assertions
- Atributy
- Tooly



Výjimky

```
class DivisionError : public exception
{
public:
  virtual const char* what() const
    noexcept override {
      return "Division by zero";
  }
};

int divide(int a, int b) {
  if (b == 0) {
    throw DivisionError();
  }
  return a / b;
}
```

```
int main() try {
  divide(10, 0);
} catch (const DivisionError& e) {
  cout << "An arithmetic error occurred: "
       << e.what() << endl;
} catch (const exception& e) {
  cout << "An error occurred: "
       << e.what() << endl;
} catch (...) {
  cout << "Unknown error" << endl;
}
```

Výjimky

Výjimka dědí od std::exception

Overriduje metodu what()

```
class DivisionError : public exception
{
public:
  virtual const char* what() const
    noexcept override {
      return "Division by zero";
  }
};

int divide(int a, int b) {
  if (b == 0) {
    throw DivisionError();
  }
  return a / b;
}
```

```
int main() try {
  divide(10, 0);
} catch (const DivisionError& e) {
  cout << "An arithmetic error occurred: "
       << e.what() << endl;
} catch (const exception& e) {
  cout << "An error occurred: "
       << e.what() << endl;
} catch (...) {
  cout << "Unknown error" << endl;
}
```

Výjimky

Výjimka dědí od std::exception

Overriduje metodu what()

```
class DivisionError : public exception
{
public:
  virtual const char* what() const
    noexcept override {
      return "Division by zero";
  }
};

int divide(int a, int b) {
  if (b == 0) {
    throw DivisionError();
  }
  return a / b;
}
```

Házení hodnotou

Chytání referencí

```
int main() try {
  divide(10, 0);
} catch (const DivisionError& e) {
  cout << "An arithmetic error occurred: "
       << e.what() << endl;
} catch (const exception& e) {
  cout << "An error occurred: "
       << e.what() << endl;
} catch (...) {
  cout << "Unknown error" << endl;
}
```

RAII a výjimky

```
class C {
    public: ~C() { cout << "C::~C()" <<
endl; }
};
class D {
    public: ~D() { cout << "D::~D()" <<
endl; }
};

int f() {
    vector<C> v(4, C());
    {
        D d;
        throw runtime_error("error");
    }
    C c;
}
```

RAII a výjimky

```
class C {
    public: ~C() { cout << "C::~C()" <<
endl; }
};
class D {
    public: ~D() { cout << "D::~D()" <<
endl; }
};

int f() {
    vector<C> v(4, C());
    {
        D d;
        throw runtime_error("error");
    }
    C c;
}
```

1. Začne unrollovat stack

RAII a výjimky

```
class C {
    public: ~C() { cout << "C::~C()" <<
endl; }
};
class D {
    public: ~D() { cout << "D::~D()" <<
endl; }
};

int f() {
    vector<C> v(4, C());
    {
        D d;
        throw runtime_error("error");
    }
    C c;
}
```

2. Uklidí d

1. Začne unrollovat stack

RAII a výjimky

```
class C {
    public: ~C() { cout << "C::~C()" <<
endl; }
};
class D {
    public: ~D() { cout << "D::~D()" <<
endl; }
};

int f() {
    vector<C> v(4, C());
    {
        D d;
        throw runtime_error("error");
    }
    C c;
}
```

3. Uklidí všechny C

2. Uklidí d

1. Začne unrollovat stack

Exception safety (když funkce skončí výjimkou)

1. Funkce dává „no exception guarantee“

   ☠️ Program se mohl rozbít, můžeme ho postupně ukončovat (funkce nenapravuje chyby související s výjimkou, hrozí memory corruption atd.)

Exception safety (když funkce skončí výjimkou)

1. Funkce dává „no exception guarantee“

   ☠️ Program se mohl rozbít, můžeme ho postupně ukončovat (funkce nenapravuje chyby související s výjimkou, hrozí memory corruption atd.)

2. Basic exception guarantee

   ⚠️ Funkce mohla být přerušena uprostřed provádění změn (hrozí např ztráta dat)

   ✅ Funkce napravila všechny chyby související s výjimkou, program stále validní

Exception safety (když funkce skončí výjimkou)

1. Funkce dává „no exception guarantee“

   ☠️ Program se mohl rozbít, můžeme ho postupně ukončovat (funkce nenapravuje chyby související s výjimkou, hrozí memory corruption atd.)

2. Basic exception guarantee

   ⚠️ Funkce mohla být přerušena uprostřed provádění změn (hrozí např ztráta dat)

   ✅ Funkce napravila všechny chyby související s výjimkou, program stále validní

3. Strong exception guarantee

   ✅ Pokud funkce byla přerušena uprostřed práce, rollbackne všechny změny

   ℹ️ Tohle guarantee nám dávají např. standardní kontejnery (**vector.push_back**)

Exception safety (když funkce skončí výjimkou)

1. Funkce dává „no exception guarantee“

   ☠️ Program se mohl rozbít, můžeme ho postupně ukončovat (funkce nenapravuje chyby související s výjimkou, hrozí memory corruption atd.)

2. Basic exception guarantee

   ⚠️ Funkce mohla být přerušena uprostřed provádění změn (hrozí např ztráta dat)

   ✅ Funkce napravila všechny chyby související s výjimkou, program stále validní

3. Strong exception guarantee

   ✅ Pokud funkce byla přerušena uprostřed práce, rollbackne všechny změny

   ℹ️ Tohle guarantee nám dávají např. standardní kontejnery (**vector.push_back**)

4. Nothrow exception guarantee

   ✅ Funkce neskončí výjimkou; keywordem **noexcept** to můžem slíbit

noexcept

- Označuje, že funkce neskončí výjimkou
  - ℹ️ Pokud funkce není označena **noexcept**, tak potenciálně vždy háže
  - ⚠️ Pokud se pleteme a nastane nechycená výjimka, program se ukončí

noexcept

- Označuje, že funkce neskončí výjimkou
  - ℹ️ Pokud funkce není označena **noexcept**, tak potenciálně vždy háže
  - ⚠️ Pokud se pleteme a nastane nechycená výjimka, program se ukončí

Zavzpomínáme na The Rule of Five

```
class MyValue {
public:
    MyValue(int v) : value(v) { /* ... */ }

    MyValue(const MyValue& other) : value(other.value) { /* ... */ }
    MyValue(MyValue&& other) noexcept : value(other.value) { /* ... */ }

    MyValue& operator=(const MyValue& other) { /* ... */ return *this; }
    MyValue& operator=(MyValue&& other) noexcept { /* ... */ return *this; }

    int value;
};
```

Jak zajistit strong guarantee typicky

Naivní implementace

```
MyContainer& operator=(const MyContainer&
                           other) {
    values.clear();
    values.reserve(other.values.size());

    for (const auto& v : other.values) {
        values.push_back(v);
    }
}
```

Jak zajistit strong guarantee typicky

Naivní implementace

```
MyContainer& operator=(const MyContainer&
                       other) {
    values.clear();
    values.reserve(other.values.size());

    for (const auto& v : other.values) {
        values.push_back(v);
    }
}
```

Strong exception gurantee

```
MyContainer& operator=(const MyContainer&
                             other) {
    if (this == &other) {
        // skip self-assignment
        return *this;
    }

    MyContainer temp(other);
    // we can swap the vectors
    std::swap(values, temp.values);

    return *this;
}
```

Assertions (na usnadnění debuggingu)

static_assert(cond, "message")

- Kontrola prováděná za překladu
  - ➡️ Žádný runtime overhead
- Typicky dává readable errory
  - Pro jednoduché podmínky vypíše hodnoty
  - Vypisuje requirementy conceptu
- **Zastaví compilaci** při nálezu chyby

assert(cond)

- Runtimový check v DEBUG verzi
  - ℹ️ V Releasu automaticky smazaný
- Typicky na kontrolu následujících
  - **Preconditions** (věci, co funkce očekává – např. Binary sort např. očekává setříděné pole)
  - **Invariants** (podle logiky programu)
  - **Postconditions** (Stav zařízený funkcí – např. sort vrací sesortěné pole)
- **Vypne program** při nálezu chyby

[[Attributes]]

- C++ nabízí několik standardních atributů a compilery řadu dalších
- Typicky nemění význam kódu, ale řídí kontroly/optimalizace překladače

- Užitečné na předcházení chyb a dokumentaci kódu:
  - **[[nodiscard("reason")]]** – výsledek funkce by neměl být zahazován
    - **[[maybe_unused]]** – označuje, že proměnná/parametr funkce není použit úmyslně
  - **[[noreturn]]** – funkce **nevrátí** hodnotu (nekonečná smyčka, throw, exit, ...)

```
[[nodiscard]] int computeSquare(int number)
{
    return number * number;
}
```

```
[[noreturn]] void fatalError(const string& msg)
{
    cerr << "Fatal error: " << msg << endl;
    exit(EXIT_FAILURE); // exit the program
}
```

Co můžeme použít na předcházení/řešení chyb mimo výjimky

1. **if** statementy
   - Typické běžně nastávající corner-casy, hranice polí, atd.
2. **std::optional<Value>**, **std::expected<Value, Error>**
   - Nějaký krok procesu mohl selhat, nejde vrátit hledanou hodnotu, atd.
   - Vlastně tim můžeme líp nahradit **nullptr** (hlavně u typů bez nulové hodnoty)
3. **assert**
   - Na kontrolu, že je program správně naprogramovaný (preconditions, invariants)
4. **static_assert**
   - Na věci, co umíme rozhodnout za překladu
5. **concept**
   - Na restrikci typů, se kterými může algoritmus pracovat, atd.

Nesnadno odhalitelné chyby a různé compilery

```
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```

Output of x86-64 gcc (trunk) (Compiler #1) | x86-64 gcc (trunk) (Editor #1)

Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  0
```

g++ -Og -Wall -Wextra -pedantic -DDEBUG

Output of x64 msvc v19.latest (Compiler #2) | x64 msvc v19.latest (Editor #1)

Wrap lines | Select all

```
example.cpp
ASM generation compiler returned: 0
example.cpp
Execution build compiler returned: 0
Program returned: 0
  0
```

cl.exe /std:c++latest /W4 /permissive- /DDEBUG /RTC1

Output of x86-64 clang (trunk) (Compiler #3) | x86-64 clang (trunk) (Editor #1)

Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  0
```

clang++ -Og -Wall -Wextra -pedantic -DDEBUG

Nesnadno odhalitelné chyby a různé compilery

```
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```

Program je obviously bez bugů :D

Output of x86-64 gcc (trunk) (Compiler #1) | x86-64 gcc (trunk) (Editor #1)

A | Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  0
```

g++ -Og -Wall -Wextra -pedantic -DDEBUG

Output of x64 msvc v19.latest (Compiler #2) | x64 msvc v19.latest (Editor #1)

A | Wrap lines | Select all

```
example.cpp
ASM generation compiler returned: 0
example.cpp
Execution build compiler returned: 0
Program returned: 0
  0
```

cl.exe /std:c++latest /W4 /permissive- /DDEBUG /RTC1

Output of x86-64 clang (trunk) (Compiler #3) | x86-64 clang (trunk) (Editor #1)

A | Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  0
```

clang++ -Og -Wall -Wextra -pedantic -DDEBUG

Tak uděláme release verze

```cpp
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```

Output of x86-64 gcc (trunk) (Compiler #1) | x86-64 gcc (trunk) (Editor #1)

Wrap lines | Select all

```
<source>: In function 'int main()':
<source>:8:21: warning: array subscript 5 is outside array boun
    8 |     std::cout << v[5] << std::endl;
```

g++ -O2 -g -Wall -Wextra -pedantic -DNDEBUG

Output of x64 msvc v19.latest (Compiler #2) | x64 msvc v19.latest (Editor #1)

Wrap lines | Select all

```
example.cpp
ASM generation compiler returned: 0
example.cpp
Execution build compiler returned: 0
Program returned: 0
  0
```

cl.exe /std:c++latest /O2 /Zi /W4 /permissive- /DNDEBUG

Output of x86-64 clang (trunk) (Compiler #3) | x86-64 clang (trunk) (Editor #1)

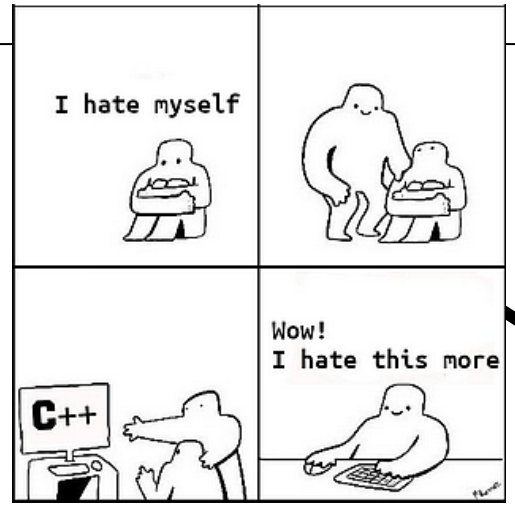
Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  -1776932536
```

clang++ -O2 -g -flto -Wall -Wextra -pedantic -DNDEBUG

Tak uděláme release verze

```cpp
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```



Output of x86-64 gcc (trunk) (Compiler #1) | x86-64 gcc (trunk) (Editor #1)

Wrap lines | Select all

```
<source>: In function 'int main()':
<source>:8:21: warning: array subscript 5 is outside array boun
    8 |     std::cout << v[5] << std::endl;
```

g++ -O2 -g -Wall -Wextra -pedantic -DNDEBUG

Output of x64 msvc v19.latest (Compiler #2) | x64 msvc v19.latest (Editor #1)

Wrap lines | Select all

```
example.cpp
ASM generation compiler returned: 0
example.cpp
Execution build compiler returned: 0
Program returned: 0
  0
```

cl.exe /std:c++latest /O2 /Zi /W4 /permissive- /DNDEBUG

Output of x86-64 clang (trunk) (Compiler #3) | x86-64 clang (trunk) (Editor #1)

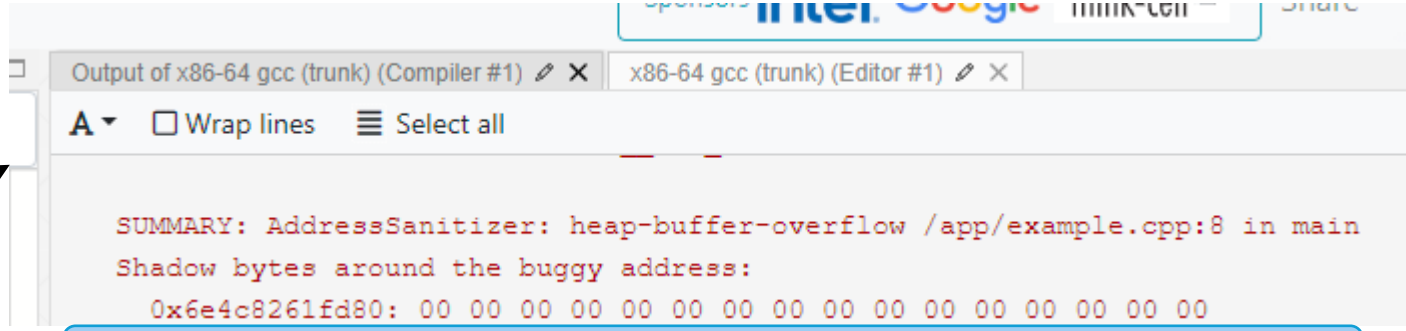
Wrap lines | Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  -1776932536
```

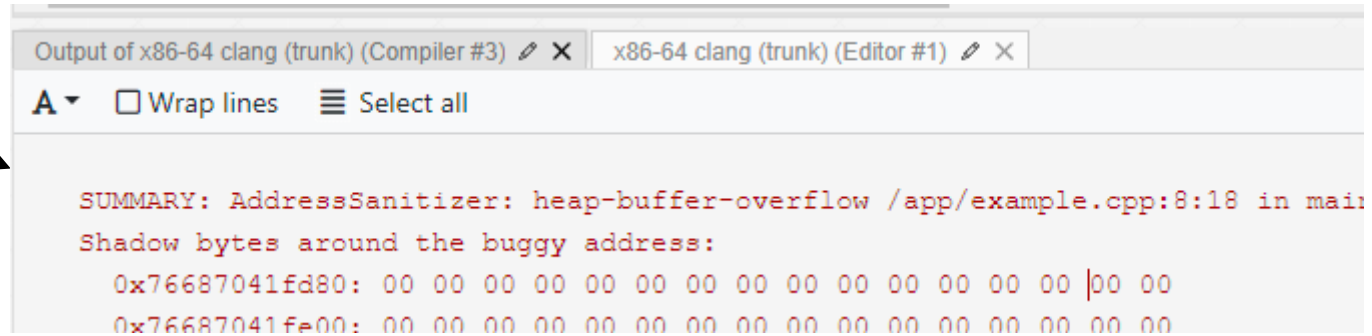
clang++ -O2 -g -flto -Wall -Wextra -pedantic -DNDEBUG

Naštěstí existují tooly: např. sanitizéry

```
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```



g++ -Og -g -DDEBUG -fsanitize=undefined -fsanitize=address



clang++ -Og -g -DDEBUG -fsanitize=undefined -fsanitize=address

Naštěstí existují tooly: nebo třeba clang-tidy

```
int main() {
    std::vector v{0,1,2,3,4,};
    std::cout << v[5] << std::endl;
}
```

clang-tidy -checks=cppcoreguidelines*,performance*,bugprone*,readability*,clang-analyzer-* test.cpp

```
/home/jirka/teaching/nprg041/web/test.cpp:7:17: warning: variable name 'v' is too short
    7 |     std::vector v{0,1,2,3,4,};
      |                 ^
/home/jirka/teaching/nprg041/web/test.cpp:8:20: warning: 5 is a magic number; consider
    8 |     std::cout << v[5] << std::endl;
      |                    ^
/home/jirka/teaching/nprg041/web/test.cpp:8:26: warning: do not use 'std::endl' with st
    8 |     std::cout << v[5] << std::endl;
      |                          ^~~~~~~~~
      |                          '\n'
```

Úloha: checked_expressions

- Vycházejte z jedné úloh pro expressions (třeba lab-06 nebo 07)
- Pokud máte implementaci s polymorfní Value
  - **Vytvořte koncept** kontrolující, že jde typ T sčítat, odečítat, atd. (a printit); prostě semicolon-separated list výrazů, ve kterých můžete proměnnou toho typu použít
  - **Definujte ValueExpression** s tímhle conceptem; u sčítání atd stačí podporovat jen operace s hodnotami stejných typů (abychom to zjednodušili)
- Pokud máte implementaci s variantem
  - **Vytvořte koncept** kontrolující, že jde typ T sčítat, odečítat, atd. (a printit); prostě semicolon-separated list výrazů, ve kterých můžete proměnnou toho typu použít
  - **Dovolte někde definovat, jaké typy má variant podporovat** (třeba jeden templatový parametr všech expressionů); u sčítání atd stačí podporovat jen operace s hodnotami stejných typů
- Kód rozumně proložte různými **asserty** (a **static_asserty**) a funkce označte atributy podle toho, jestli nevyužití hodnoty nedává smysl
- Vytvořte příklady implementací (třeba bonusové cpp fily), které postupně vše výše poruší