# Wyjątki w C++

CODERS.SCHOOL

http://coders.school


- Kamil Szatkowski, kamil.szatkowski@nokia.com
- Łukasz Ziobroń, lukasz@coders.school

# About authors

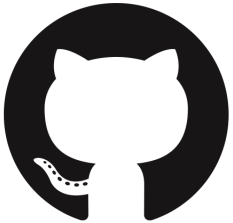
## Kamil Szatkowski

- Work at Nokia:
  - C++ software engineer @ CCH
  - C++ software engineer @ LTE CPlane
  - RAIN Developer @ LTE Cplane
  - Code Reviewer
  - Code Mentor
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

## Łukasz Ziobroń

- Work at Nokia:
  - C++ software engineer @ LTE Cplane
  - C++ software engineer @ LTE OAM
  - Python developer @ LTE LOM
  - Scrum Master
  - Code Reviewer
- Trainer:
  - Practial Aspects Of Software Engineering
  - Nokia Academy
  - Coders School
  - Internal Nokia trainings
- Occassional speaker:
  - Academic Championships in Team Programming
  - code::dive community
  - code::dive conference

# Wyjątki



git clone https://github.com/LordLukin/memory_management.git

# Obsługa błędów – goto, kody błędów

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

bool isValid()
{
    return false;
}

int main()
{
    /* ... */

    if(!isValid())
    {
        goto error;
    }

    /* ... */

error:
    std::cerr << "Error occured" << std::endl;
    return 1;
}
```

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

enum ErrorCode {
    E_SUCCESS,
    E_FAIL
};

bool isValid() {
    return false;
}

ErrorCode foo() {
    if(!isValid()) {
        return E_FAIL;
    }
    /* ... */
    return E_SUCCESS;
}

int main() {
    if(foo() == E_FAIL) {
        return 1;
    }
    return 0;
}
```

# Obsługa błędów – konstruktory, operatory

```
1  struct FileWrapper
2  {
3      FileWrapper(std::string const& filePath)
4          : m_file(fopen(filePath.c_str(), "rw"))
5      {
6          /* What if file did not open? */
7      }
8
9      ~FileWrapper()
10     {
11         fclose(m_file);
12     }
13
14     FileWrapper & operator<<(std::string const& text)
15     {
16         /* What if file did not open? */
17         fputs(text.c_str(), m_file);
18         return *this;
19     }
20
21 private:
22     FILE* m_file;
23 };
```

# Obsługa błędów – konstruktory, operatory

```
1  struct FileWrapper
2  {
3      FileWrapper(std::string const& filePath)
4          : m_file(fopen(filePath.c_str(), "rw"))
5      {
6          if(!m_file)
7          {
8              throw std::runtime_error("File not opened");
9          }
10     }
11
12     ~FileWrapper()
13     {
14         fclose(m_file);
15     }
16
17     FileWrapper & operator<<(std::string const& text)
18     {
19         /* Not validation needed because invalid
20         object cannot be created */
21         fputs(text.c_str(), m_file);
22         return *this;
23     }
24
25 private:
26     FILE* m_file;
27 };
```

# try/catch

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

void foo()
{
    throw std::runtime_error("Error");
}

int main()
{
    try
    {
        foo();
    }
    catch(std::runtime_error const&)
    {
        std::cout << "std::runtime_error" << std::endl;
    }
    catch(std::exception const& ex)
    {
        std::cout << "std::exception: " << ex.what() << std::endl;
    }
    catch(...)
    {
        std::cerr << "unknown exception" << std::endl;
    }
}
```

Wynik wykonania:

```
std::runtime_error
```

# Jak działają wyjątki?

```
1  struct TalkingObject {
2      TalkingObject() {
3          cout << "Default constructor" << endl;
4      }
5      TalkingObject(TalkingObject const& src) {
6          cout << "Copy constructor" << endl;
7      }
8      ~TalkingObject() {
9          cout << "Destructor" << endl
10     }
11 };
12 void foo() {
13     throw runtime_error("Error");
14 }
15
16 int main() {
17     TalkingObject out;
18     try {
19         TalkingObject inside;
20         foo();
21     } catch(runtime_error const& ex) {
22         cout << "runtime_error: " << ex.what() << endl;
23     } catch(exception const&) {
24         cout << "exception" << endl;
25     }
26 }
```

Wynik wykonania:

Default constructor
Default constructor
Destructor
runtime_error: Error
Destructor

# Jak działają wyjątki?

```
1  struct TalkingObject {
2      TalkingObject() {
3          cout << "Default constructor" << endl;
4      }
5      TalkingObject(TalkingObject const& src) {
6          cout << "Copy constructor" << endl;
7      }
8      ~TalkingObject() {
9          cout << "Destructor" << endl
10     }
11 };
12 void foo() {
13     throw runtime_error("Error");
14 }
15
16 int main() {
17     TalkingObject out;
18     try {
19         TalkingObject inside;
20         foo();
21     } catch(runtime_error const& ex) {
22         cout << "runtime_error: " << ex.what() << endl;
23     } catch(exception const&) {
24         cout << "exception" << endl;
25     }
26 }
```

- rzucony wyjątek startuje mechanizm odwijania stosu, który działa aż do napotkania pierwszego bloku *try/catch* z pasującą klauzulą *catch*,
- wyjątek jest dopasowywany do każdej z klauzul *catch* zgodnie z kolejnością ich deklaracji,
- wyjątek jest niszczony, gdy program opuszcza blok *catch* bez ponownego rzucenia wyjątku.

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() { throw std::runtime_error("Error"); }
4
5  void bar() {
6      try {
7          TalkingObject inside;
8          foo();
9      } catch(std::exception const&) {
10         std::cout << "std::exception" << std::endl;
11         throw;
12     }
13 }
14 int main() {
15     TalkingObject outside;
16     try {
17         bar();
18     } catch(std::runtime_error const& ex) {
19         std::cout << "std::runtime_error: " << ex.what() << std::endl;
20     }
21 }
```

Wynik wykonania:

Default constructor
Default constructor
Destructor
std::exception
std::runtime_error: Error
Destructor

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() { throw std::runtime_error("Error"); }
4
5  void bar() {
6      try {
7          TalkingObject inside;
8          foo();
9      } catch(std::exception const&) {
10         std::cout << "std::exception" << std::endl;
11         throw;
12     }
13 }
14 int main() {
15     TalkingObject outside;
16     try {
17         bar();
18     } catch(std::runtime_error const& ex) {
19         std::cout << "std::runtime_error: " << ex.what() << std::endl;
20     }
21 }
```

- wyjątek rzucony ponownie znów startuje mechanizm odwijania stosu, który działa aż do napotkania kolejnego bloku *try/catch* z pasującą klauzulą *catch*,
- klauzula *catch* dla typu bazowego pozwala złapać wyjątek typu pochodnego i nie zmienia pierwotnego typu wyjątku.

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() {
4      throw std::runtime_error("Error");
5  }
6
7  void bar() {
8      try {
9          TalkingObject inside;
10         foo();
11     } catch(std::exception const&) {
12         std::cout << "std::exception" << std::endl;
13         throw;
14     }
15 }
16
17 int main() {
18     TalkingObject outside;
19     bar();
20 }
21
```

Wynik wykonania:

Default constructor
Default constructor
Destructor
std::exception
>> abort() <<

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  void foo() {
4      throw std::runtime_error("Error");
5  }
6
7  void bar() {
8      try {
9          TalkingObject inside;
10         foo();
11     } catch(std::exception const&) {
12         std::cout << "std::exception" << std::endl;
13         throw;
14     }
15 }
16
17 int main() {
18     TalkingObject outside;
19     bar();
20 }
21
```

- wyjątek, który nie został złapany przez żaden blok *try/catch* powoduje zatrzymanie programu poprzez wykonanie metody *std::terminate()*

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  struct ThrowingObject {
4      ~ThrowingObject() {
5          throw std::runtime_error("error in destructor");
6      }
7  };
8
9
10 void foo() {
11     throw std::runtime_error("Error");
12 }
13
14 int main() {
15     TalkingObject outside;
16     try {
17         ThrowingObject inside;
18         foo();
19     } catch(std::exception const&) {
20         std::cout << "std::exception" << std::endl;
21         throw;
22     }
23 }
```

Wynik wykonania:

Default constructor
Default constructor
>> abort() <<

# Jak działają wyjątki?

```
1  struct TalkingObject { /*...*/ };
2
3  struct ThrowingObject {
4      ~ThrowingObject() {
5          throw std::runtime_error("error in destructor");
6      }
7  };
8
9
10 void foo() {
11     throw std::runtime_error("Error");
12 }
13
14 int main() {
15     TalkingObject outside;
16     try {
17         ThrowingObject inside;
18         foo();
19     } catch(std::exception const&) {
20         std::cout << "std::exception" << std::endl;
21         throw;
22     }
23 }
```

- wyjątek rzucony podczas działania mechanizmu odwijania stosu powoduje zatrzymanie programu poprzez zawołanie metody *std::terminate()*

# Wyjątki - rekomendacje

- łap wyjątki przy pomocy ***const&***,
- twórz własne wyjątki jako dziedziczące po wyjątkach standardowych,
- używaj wyjątków tylko w sytuacjach wyjątkowych.

CODERS.SCHOOL

http://coders.school