Uniwersytet Wrocławski

Obsługa błędów w C++

Zbigniew Koza
Wydział Fizyki i Astronomii

Podejście tradycyjne

- Sygnalizacja błędów:
  - W zmiennych globalnych
  - W wartościach funkcji
  - Żadne z powyższych nie jest stosowane we współczesnym C++ (poza C)
  - Każde jest stosowane w bibliotekach pisanych w C

errno

```
int main()
{
  std::cout << "errno: " << errno << "\n";
  double a =  std::sqrt(-1.0);
  std::cout << "errno: " << errno << "\n";
  if (errno != 0)
  {
    perror("sqrt(-1) zakończyło się błędem");
  }
  std::cout << "errno: " << errno << "\n";
}
```

```
> ./a.out
errno: 0
errno: 33
sqrt(-1) zakończyło się błędem: Numerical argument out of domain
errno: 33
```

Obsługa błędów w C

```c
#include <stdio.h>
#include <sys/socket.h>

int main(int argc, char const *argv[])
{
  int sock = 0;
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    printf("\n Error: failded to create a socket\n");
    return -1;
  }
}
```

```c
int status;
size_t n = 37;

gsl_set_error_handler_off();

status = gsl_fft_complex_radix2_forward (data, stride, n);

if (status) {
  if (status == GSL_EINVAL) {
     fprintf (stderr, "invalid argument, n=%d\n", n);
  } else {
     fprintf (stderr, "failed, gsl_errno=%d\n", status);
  }
  exit (-1);
}
```

https://www.gnu.org/software/gsl/doc/html/err.html

asercje

- Asercji używa się do samo-diagnostyki kodu
  - Testowanie warunków, które bezwzględnie muszą być spełnione
  - Uwzględniane wyłącznie w trybie Debug

Asercje - przykład

```cpp
#include <cassert>
#include <iostream>

int dziel(int a, int b)
{
  assert (b != 0);
  return a / b;
}

int main()
{
  std::cout << dziel(5, 4) << "\n";
  std::cout << dziel(3, 0) << "\n";
  return EXIT_SUCCESS;
}
```

```
> ./a.out
1
a.out: 3.cpp:6: int dziel(int, int): Assertion `b != 0' failed.
Przerwane (zrzut pamięci)
```

asercje

- Assert to makro wyłączane poprzez zdefiniowanie makra preprocesora NDEBUG

```
> g++ -DNDEBUG 3.cpp
```

```cpp
#include <cassert>
#include <iostream>

int dziel(int a, int b)
{
  assert (b != 0);
  return a / b;
}

int main()
{
  std::cout << dziel(5, 4) << "\n";
  std::cout << dziel(3, 0) << "\n";
  return EXIT_SUCCESS;
}
```

```
1
Błąd w obliczeniach zmiennoprzecinkowych (zrzut pamięci)
```

asercje

- Używa się rzadziej niż kiedyś
  - Zastępowane przez testy automatyczne
- Assert to makro wyłączane poprzez zdefiniowanie makra prepocesora NDEBUG

Zrzut pamięci

```
g++ -DNDEBUG 3.cpp -g
```

```
#include <cassert>
#include <iostream>

int dziel(int a, int b)
{
  assert (b != 0);
  return a / b;
}

int main()
{
  std::cout << dziel(5, 4) << "\n";
  std::cout << dziel(3, 0) << "\n";
  return EXIT_SUCCESS;
}
```

```
1
Błąd w obliczeniach zmiennoprzecinkowych (zrzut pamięci)
```

```
> coredumpctl -1
TIME                            PID  UID  GID SIG    COREFILE EXE                                                    SIZE
Thu 2021-10-14 01:19:18 CEST 67933 1000 1000 SIGFPE present  /home/zkoza/Pulpit/Dydaktyka/zcpp/w2-exceptions/cpp/a.out 43.7K
```

```
coredumpctl debug 67863
```

```
Reading symbols from /home/zkoza/Pulpit/Dydaktyka/zcpp/w2-exceptions/cpp/a.out...
[New LWP 67863]
Core was generated by `./a.out'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x00005555d2cc5177 in dziel (a=3, b=0) at 3.cpp:7
7         return a / b;
```

- Schemat postępowania zależy od systemu

Wpinanie się w działający program

```
#include <unistd.h>
#include <iostream>

int main()
{
  std::cout << "pid = " << getpid() << "\n";
  int i = 0;
  while (i ==  0)
    continue;
  std::cout << "Hello!\n";
}
```

```
> ./a.out
pid = 68482
```

```
sudo gdb -p 68482
```

← np. w innej konsoli

```
Attaching to process 68482
main () at 4.cpp:9
9           continue;
```

Przykładowa sesja

```
main () at 4.cpp:9
9             continue;
(gdb) l
4         int main()
5         {
6           std::cout << "pid = " << getpid() << "\n";
7           int i = 0;
8           while (i ==  0)
9             continue;
10          std::cout << "Hello!\n";
11        }
12
13
(gdb) set var i = 1
(gdb) n
8           while (i ==  0)
(gdb) print i
$1 = 1
(gdb) n
10          std::cout << "Hello!\n";
(gdb) n
11        }
(gdb) n
0x00007ff26ce16b25 in __libc_start_main () from /usr/lib/libc.so.6
```

l = list

n = next

set var

Wyjątki

Składnia

- Zgłaszanie:

```
throw obiekt;
```

- Obsługa:

```
try{
   catch (const Obiekt & obiekt)
   { ... }
}
```

Przykład

```
int dziel(int a, int b)
{
  if (b == 0)
    throw std::invalid_argument("próba dzielenia przez 0");
  return a / b;
}

int main()
{
  try
  {
    std::cout << dziel(5, 4) << "\n";
    std::cout << dziel(3, 0) << "\n"; // wyjątek
    std::cout << dziel(1, 4) << "\n";
  }
  catch (std::invalid_argument& e)
  {
    std::cerr << e.what() << "\n";
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```
1
próba dzielenia przez 0
```

Zalety wyjątków

- Wyjątki muszą być obsłużone – nieobsłużony wyjątek powoduje „pad” programu.
- Wyjątki umożliwiają przekazanie sterowania do miejsca, w którym można obsłużyć problem (na stosie funkcji).
- Mechanizm zwijania stosu powoduje wywołanie destruktorów wszystkich obiektów lokalnych pomiędzy punktem zgłoszenia a obsługującym wyjątek blokiem `try`.
- Wyjątki umożliwiają eleganckie oddzielenie głównego kodu od kodu obsługującego ewentualne „niepowodzenia”.

Co zgłaszać?

- Teoretycznie jako wyjątek można zgłosić dowolny obiekt, zmienną, wskaźnik,…
- W praktyce używa się klas z nagłówka `<stdexcept>` lub klas z nich wyprowadzonych

<stdexcept>

- **logic_error** – błędy logiczne w kodzie
- **invalid_argument** – nieprawidłowe argumenty funkcji
- **domain_error** – wykroczenie poza „dziedzinę” funkcji
- **length_error** – przekroczenie maksymalnego rozmiaru
- **out_of_range** – argumenty poza oczekiwanym zakresem
- **runtime_error** – błąd wykrywany w czasie wykonania
- **range_error** – wykroczenie poza zakres typu danych
- **overflow_error** – przekroczenie zakresu arytmet.
- **underflow_error** – przekroczenie zakresu arytmet.

Typowy main()

```cpp
int main(int argc, const char* argv[])
{
    my_lib::utils::log::initialize_global_log_level();
    my_lib::utils::log::initialize_global_logger();
    my_lib::io::initialize_locale();
    try
    {
        return my_lib::cli::AppRunner::run(argc, argv);
    }
    catch (const my_lib::exception::IoError& e)
    {
        LOG_FATAL("Input/output error: {}", e.what());
        LOG_FATAL("Program aborted");
        return 1;
    }
    catch (const std::exception& e)
    {
        LOG_FATAL("{}", e.what());
        LOG_FATAL("Program aborted");
        return 2;
    }
    catch (...)
    {
        LOG_FATAL("Unknown error");
        LOG_FATAL("Program aborted");
        return 255;  // we should never see this
    }
}
```

Wyłapywanie wyjątków

- Pod blokiem try może być kilka bloków **`catch`**
  - Od najbardziej szczegółowych do najbardziej ogólnych
  - Argumenty catch przekazuj przez (stałą) referencję
  - Składnia: jak do funkcji

```
catch (const std::exception& e)
{
```

  - **`catch (...)`**

```
catch (...)
```

  - **`std::exception::what()`** [wirtualna]

Zwijanie stosu

- Wyjście z każdego zakresu (`{...}`) powoduje uruchomienie destruktorów obiektów lokalnych (w tym zakresie) i zwolnienie przypisanej im pamięci na stosie funkcji
- Dotyczy to też całych funkcji
- Zwijanie stosu albo kończy się na jakimś `catch`, albo powoduje awaryjne zakończenie programu

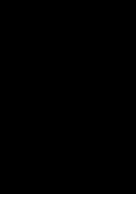
Wyjątki a RAII

- NIE

```
FILE* in = fopen("plik.txt", "r");
auto p = new int[100];
mutex.lock();

throwing_fun();
other_fun();

mutex.unlock();
delete [] p;
fclose(in);
```



Zdobywanie zasobów jest inicjalizacją

- TAK

```
std::ifstream in("plik.txt");
std::vector<int>(100);
std::lock_guard(mutex) lg;

throwing_fun();
other_fun();
```

Wyjątki: python vs. C++

Python	*C++*
Wyjątki dziedziczą z `BaseException`	Wyjątki to dowolne obiekty
`raise`	`throw`
`try:`	`try { blok }`
`execept (e):`	`catch(E & e)`
`except:`	`catch(...)`
`else:`	kod za ostatnim `catch`
`finally:`	**RAII**