Programowanie Obiektowe

C++ wyjątki i obietnice

Dariusz Brzeziński

Politechnika Poznańska, Instytut Informatyki

Motywacja

- W językach niskopoziomowych błędy zgłaszane były przez zwracanie odpowiedniego statusu (liczby)
- W C `main()` zwraca `int` żeby można było określić czy aplikacja zakończyła się sukcesem (0) czy nie (!=0)
- Problemy
  - Brak hieracrhii błędów
  - Brak propagowania błędów
  - Mieszanie logiki aplikacji z obsługą błędów

Przykład

```
int main(int argc, const char* argv[]) {
    int status = processInput();
    if (status != 0){
        return -1;
    }

    result = calculateResult();
    if (status < 0){
        return -2;
    } else {
        char* formattedResult;
        if (formatResult(result, formattedResult)) {
            return 0;
        } else {
            return -3;
        }
    }
}
```

Rozwiązanie

- W nowszych językach wprowadzono **wyjątki**
- Wyjątek to dane o błędzie, które są automatycznie propagowane przez system uruchomieniowy
- Zalety
  - Można tworzyć **hierarchie** błędów tworząc klasy
  - Można je **rzucać**, czyli propagować w górę stosu wywołań
  - Oddzielenie logiki aplikacji (**try**) od obsługi błędów (**catch**)

Rzucanie wyjątków

```
float squareRoot( float num ) {

    float result = 0;

    if( num < 0 ) {
        throw std::string(
            "Błąd: liczba nie może być ujemna!" );
    }

    result = sqrt( num );
    return result;
}
```

Czym można rzucać?

Czym można rzucać?

Praktycznie wszystkim…

Czym można rzucać? (2)

```
void doSomething() {

    throw 5;
}
```

```
void doSomething() {

    bool someVariable = false;
    throw someVariable;
}
```

```
void doSomething() {

    throw CustomException(
      "An error occurred.");
}
```

```
class CustomException {

private:
    string message;
    int errorCode;
    bool fatal;

public:
    CustomException(string message);
    inline string getMessage() {
        return message;
    };
    inline int getErrorCode() {
        return errorCode;
    };
    inline bool isFatal() {
        return fatal;
    };
};
```

Czym można rzucać? (3)

```
class FileNotFoundException : public CustomException {

    private:
    std::string filename;

    public:
    FileNotFoundException( std::string filename );
    inline std::string getFilename() { return filename; };

    // Przesłonięcie metody getMessage()
    inline std::string getMessage() {
        std::stringstream ss;
        ss << "Nie znaleziono pliku: " << filename;
        return ss.str();
    }
};
```

A jak łapać?

Aby łapać wyjątki, kod, który rzuca wyjątek musi być otoczony klauzulą try-catch. Dodatkowo w klauzuli catch musi zostać podany typ odpowiadający typowi rzucanego wyjątku.

```
try {
    std::cout << "Wywoływanie funkcji..."
        << std::endl;
    doSomething();
}
catch(CustomException e) {
    std::cout << "Błąd: " << e.getMessage();
}
```

Wyjątek po rzuceniu trafia do bloku catch i może być odczytany przy pomocy zmiennej *e* typu CustomException. Dzięki tej zmiennej można odczytać wszystkie informacje zawarte w obiekcie wyjątku i wykonać odpowiedni kod obsługi błędu.

Łapanie przez wartość/referencję

Kod z poprzedniego slajdu zadziała, ale ma pewną wadę. Za każdym razem, gdy występuje wyjątek łapiemy go *przez wartość.* Oznacza to, że przy każdym przekazaniu wyjątku w górę stosu tworzona jest jego kopia. Prowadzi to do zmniejszenia efektywności czasowej i pamięciowej kodu. By uniknąć tego problemu, lepiej łapać wyjątki przez referencję.

```
try {
    std::cout << "Wywoływanie funkcji..."
        << std::endl;
    doSomething();
}
catch(CustomException& e) {
    std::cout << "Błąd: " << e.getMessage();
}
```

Inny problem

```
try {
   // FileNotFoundException jest podklasą
CustomException
   throw FileNotFoundException("example.txt");
}
catch(CustomException e) {
   std::cout << „Błąd: " << e.getMessage();
}
```

Powyższy kod w założeniu powinien wypisać komunikat „Błąd: nie znaleziono plik example.txt”. Niestety, gdy obiekt podklasy jest kopiowany do obiektu klasy nadrzędnej to wszystkie atrybuty charakterystyczne dla klasy podrzędnej są tracone.

Kolejność łapania wyjątków

```
try{
    doSomething();
}
catch( CustomException& ce ) {
    std::cout << "Błąd: " << ce.getMessage();
}
catch(...) {
    std::cout << "Nieznany błąd!";
}
```

Aby złapać wyjątek dowolnego typu należy skorzystać z bloku postaci catch(...). W takim bloku złapany zostanie dowolny wyjątek, lecz nie ma wówczas dostępu do jego atrybutów.

Ponowne rzucanie wyjątków

```
try{
    doSomething();
}
catch(CustomException& ce) {
    std::cout << "Błąd: " << ce.getMessage();
    rethrow ce;
}
catch(...) {
    std::cout << "Nieznany typ błędu!";
    rethrow;
}
```

Obietnice...

Słowo kluczowe throw w C++ ma dwa znaczenia. Oprócz rzucania wyjątku, pozwala ograniczyć typy rzucanych przez metodę wyjątków do zadanej listy. Ograniczenie to nie jest jednak wymuszane przez kompilator!

```
void doSomething(int value) throw(int, CustomException)
{
    if(value > 5) {
        throw CustomException( "Wystąpił błąd..." );
    }
    else {
        throw 18;
    }
}
```

std::exception

Biblioteka STL zawiera już klasę, która ma służyć jako bazowa dla rzucanych wyjątków oraz kilka klas z niej dziedziczących.

```
try {
    std::cout << "Strumień wyjściowy może rzucić wyjątek...";
}
catch(std::exception& ex) {
    std::cerr << "Błąd: " << ex.what();
}
```

Wyjątek	Opis
`bad_alloc`	Rzucany przez operator new przy błędzie w trakcie alokacji pamięci
`bad_cast`	Rzucany przez dynamic_cast przy nieudanej próbie rzutowania
`bad_exception`	Rzucany, gdy wyjątek nie pasuje do żadnego bloku catch
`bad_typeid`	Rzucany przez typeid
`ios_base::failure`	Rzucany przez metody strumieni we/wy

Źródła

- http://www.ozzu.com/cpp-tutorials/tutorial-exceptions-t86515.html
- http://www.cplusplus.com/doc/tutorial/exceptions/
- http://en.wikipedia.org/wiki/Exception_guarantees
- http://en.wikipedia.org/wiki/Exception_safety
- http://www.cprogramming.com/tutorial/exceptions.html