Podstawy programowania obiektowego

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 16.05.2021

8. Wyjątki

Obsługa błędów

Dobrze zaprojektowany program powinien wykrywać i ewentualnie raportować błędne sytuacje, które mają miejsce podczas wykonania.

Źródła błędów mogą być różne:

- błędy programistów,
- sytuacja w środowisku wykonania (błąd otwarcia pliku, nieoczekiwany koniec pliku, zły format, brak pamięci)
- błędy lub specjalne akcje użytkownika (np.: użycie przycisku *cancel* podczas długotrwałej operacji)

Tradycyjna obsługa błędów

- wyjście z programu (funkcje `exit`, `abort`)
- **powrót z funkcji, zwrot kodu błędu**
- zignorowanie błędu
- wywołanie funkcji specjalnych (typu `raise` / `signal`)
- wykonanie nielokalnego `goto` (funkcje `setjmp`, `longjmp`)

Tradycyjna obsługa błędów

Problemy:

- Kody błędów zwracane przez funkcje są rzadko testowane
- Struktura kodu - podstawowym celem funkcji jest realizacja wolnego od błędów scenariusza. Reakcja na błędy podczas uruchamiania
- Poziom dyskryminacji błędów:
  - `1,true` – sukces; `0, false` – porażka
  - 0 – sukces; wartość niezerowa – szczegółowy kod błędu

Mechanizm wyjątków

- Wyjątki (ang. *exception*) są konstrukcją, który ma z założenia ułatwić proces obsługi błędów, pozwalając skupić się na **poprawnych scenariuszach wykonania**.

- Użycie wyjątków nie zwalania od testowania występowania błędów i nie zwalania od ich obsługi. Pozwala jednak na użycie mechanizmu **automatycznej propagacji informacji o błędach** od miejsca stwierdzenia ich wystąpienia do miejsca ich obsługi z pominięciem etapów pośrednich.

Generacja wyjątków

Funkcja, która w trakcie wykonania napotka błąd, którego obsługa wykracza poza zakres jej normalnego działania może

- **utworzyć obiekt** zawierający informację o błędzie i
- **przesłać** go do bliżej nieokreślonego odbiorcy,
- który będzie w stanie tę informację **przechwycić** i podjąć odpowiednie działanie.

Proces ten nazywany jest generacją (wyrzucaniem, ang. *throw*) wyjątków.

Składnia:

```
throw object;
```

Generacja wyjątków

- Instrukcja `throw` przypomina pod pewnym względem instrukcje `return`. Po jej wykonaniu funkcja kończy działanie zwracając pewną wartość.
- Zwracane wartości mogą w zasadzie być dowolnego typu: wbudowanego (np.: `int, unsigned, char*`) lub zdefiniowanego przez użytkownika.
- Najczęściej jednak definiuje się specjalne klasy przeznaczone wyłącznie do przekazywania informacji o błędach. W potocznym języku właśnie te klasy lub obiekty tych klas nazywane są wyjątkami.

```
throw -1;
throw "Zła nazwa pliku"
throw BadFilename();
throw FileNotFound("gugu.txt")
```

Przechwytywanie wyjątków

- Po wygenerowaniu wyjątku sterowanie przechodzi do bloku instrukcji odpowiedzialnych za obsługę wyjątku (ang. *exception handler*).
- Składnia przypomina nieco definicję funkcji:

```
catch(type t){...}
```

- Parametr `type` jest zazwyczaj typem podstawowym lub referencją; parametr `t` jest opcjonalny.

Blok try

- Kod obsługi wyjątku może być umieszczony wewnątrz funkcji, która wygenerowała wyjątek lub w jednej z wołających ją funkcji wyższego poziomu.
- Musi być umieszczony po bloku `try` postaci:

```
try{...}
```

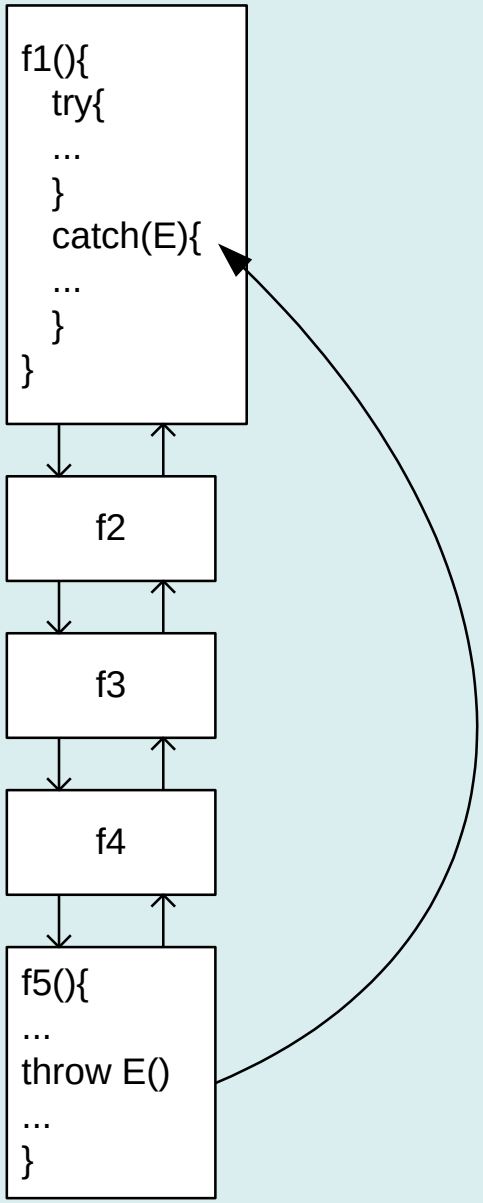
- Blok try definiuje standardowy scenariusz wykonania funkcji. Funkcja „próbuje” wykonać blok `try`. Jeżeli jedna z wołanych funkcji wyrzuci wyjątek i odpowiadający mu blok `catch` jest zdefiniowany w funkcji, wówczas sterowanie przejdzie do tego bloku:

Blok try

```
try{
        Data data = loadData(filename);
        transform(data);
        display(data);
}
catch(OpenError e){
        … // handle OpenError
}
catch(ReadError e){
        … // handle ReadError
}
```

- Po zgłoszeniu wyjątku przerywany jest normalny tryb wykonywania programu i następuje przeskok sterowania do bloku handlera wyjątku.
- Podczas przejścia zwalniany jest stos, na którym umieszczane były argumenty wywołania poszczególnych funkcji, adresy powrotu i obiekty automatyczne.

Przykład



- Funkcja `f5` generuje wyjątek `E`
- Stos wywołań funkcji jest przeglądany w poszukiwaniu pierwszego handlera zdolnego obsłużyć wyjątek `E`.

  Analogiczny handler może być umieszczony w funkcji wyższego poziomu, jednakże nie zostanie on osiągnięty.
- Następuje przeskok do bloku `catch(E)` w funkcji `f1`.
- Zwalniana jest pamięć przydzielona na stosie podczas wykonania funkcji f5,f4,f3,f2 i bloku `try` w funkcji `f1`. Przy zwalnianiu pamięci wołane są destruktory obiektów automatycznych.

Zwalnianie pamięci

Mechanizm oczyszczania stosu nie pozwala niestety na usuwanie obiektów stworzonych dynamicznie

```
class A
{
public:
    A(){cout<<"A"<<" ";}
    ~A(){cout<<"~A"<<" ";}
};
```

```
class Vector
{
    int*buf;
    int size;
public:
    class OutOfBounds{};
    Vector(int s=0):
            size(s>0?s:0), // size >= 0
            buf(s>0?new int[s]:0){
        if(!buf)size=0;
    }
    int&operator[](int i){
        if(i<0 || i>= size)
             throw OutOfBounds();
        return buf[i];
    }
};
```

Zwalnianie pamięci

Obiekt wskazywany przez wskaźnik pa nie zostanie usunięty (jego destruktor nie zostanie uruchomiony)

```
int main()
{
    Vector v(10);
    try{
        A a;
        A *pa = new A(); // nie usunięty
        int index = -1;
        cin>>index;
        cout<<v[index];
    }
    catch(Vector::OutOfBounds){
        cerr<<"Invalid index"<<endl;
    }
}
```

Wyjście:
A A -1
~A Invalid index

Zwalnianie pamięci

- Funkcja `main` odczyta wartość indeksu wektora i wypisze element na ekranie. Standardowy scenariusz działania zawarty jest w bloku `try`.
- W przypadku wartości indeksu spoza zakresu generowany jest wyjątek typu `Vector::OutOfBounds`
- Wystąpienie wyjątku powoduje przeskok z kontekstu funkcji `Vector::operator[]` do bloku obsługującego wyjątek: `catch(Vector::OutOfBounds){...}`
- Przejściu do bloku handlera towarzyszy oczyszczanie stosu – wołany jest destruktor `~A()` obiektu `a`.

Nieobsłużone wyjątki

- Jeżeli na stosie wywołań brak jest funkcji zawierającej handler danego typu wyjątków, wówczas uruchamiany jest mechanizm awaryjny: wołana jest funkcja: `terminate()`.
- Standardowa implementacja `terminate()` woła funkcję `abort()` – powoduje ona wyjście z programu.
- Funkcja `terminate()` może być zastąpiona własną funkcją, która na przykład zapisze ważne dane lub wywoła powtórną inicjalizację systemu.

Typy wyjątków

- W języku C++ można generować wyjątki dowolnego typu. Argumentem instrukcji `throw` jest obiekt, a nie typ lub klasa.
- Wyjątek jest przechwytywany na podstawie typu. W funkcji może być zdefiniowany dokładnie jeden handler dla wyjątków danego typu.
- Obiekty przesyłane do handlera wyjątków mogą nieść dodatkowe informacje:

Obiekt użyty jako wyjątek przechowuje informacje o nazwie pliku

```
class FileNotFound
{
public:
    FileNotFound(const char*_name)
    {
        strcpy(name,_name);
    }
    char name[_MAX_PATH];
};
```

Typy wyjatków

```
void read(const char*name){
    FILE*file=fopen(name,"rt");
    if(!file)throw FileNotFound(name);
    // ...
    if(should_include)read(included_file);
    // ...
    fclose(file);
}

int main(){
    try{
        read("test.txt");
    }
    catch(FileNotFound e){
        printf("File not found;%s\n",e.name);
    }
}
```

Wyjątek może zostać wygenerowany przy czytaniu włączanego pliku…

Grupowanie

Wyjątki związane z określonym typem błędów często grupuje się tak, aby móc je przetwarzać za pomocą wspólnego handlera.

```
enum Matherr {overflow, underflow, divbyzero, }

try{
}
catch(Matherr m){
    switch(m){
        case overflow:
        //…
        case underflow:
        //…
    }
}
```

Grupowanie

Grupowanie poprzez dziedziczenie -- jest to bardzo często stosowana praktyka, ponieważ umożliwia twórcom bibliotek łatwe rozszerzanie zbioru wyjątków przez dodawanie nowych klas do istniejącej hierarchii.

```
class Matherr{};
class Overflow : public Matherr{};
class Underflow : public Matherr{};
class Divbyzero : public Matherr{};

try{
}
    catch(Divbyzero){
    // handle Divbyzero only
    }
    catch(Matherr){
    // handle other Matherr
}
```

Hierarchie wyjątków

- Dziedziczenie umożliwia twórcom bibliotek łatwe rozszerzanie zbioru wyjątków przez dodawanie nowych klas do istniejącej hierarchii.
- W wielu przypadkach kod użytkownika przechwytujący wyjątki generowane w starej wersji biblioteki będzie w stanie obsłużyć nowe typy wyjątków posługując się handlerem dla klasy bazowej.

```
class Base
{
public:
    virtual const char*what()const{return "Base";}
};

class Derived : public Base
{
public:
    const char*what()const{return "Derived";}
};
```

Hierarchie wyjątków

```
void f(){   throw Derived(); }

void test1()
{
    try{
        f();
    }
    catch(Base e){
        cout<<e.what()<<endl;
    }
}
```

```
void test2()
{
    try{
        f();
    }
    catch(Base& e){
        cout<<e.what()<<endl;
    }
}
```

- Implementacja handlera wyjątków, którego parametrem jest typ bazowy może powodować zatracenie dodatkowych informacji zdefiniowanych w klasie pochodnej.
- Jeżeli hierarchia wyjątków zapewnia wirtualne funkcje o charakterze informacyjnym, np.: `what()`, wówczas, aby skorzystać z tych informacji, należy użyć handlera o parametrze typu referencyjnego.

Przechwytywanie wszystkich wyjątków

Możliwa jest definicja handlera, który będzie przechwytywał wszystkie wyjątki.

Ma on postać:

```
catch(...){
// cout<< "an exception occurred " <<endl
}
```

Powtórne wyrzucenie wyjątków

Instrukcja `throw` umieszczona wewnątrz bloku `catch` powoduje powtórne wyrzucenie przechwyconego wyjątku.

```
catch(E){
   if(can_handle){
   // handle it
   }else
   throw;
}
```

Wyjątki w konstruktorach

- Jeżeli wyjątek został zgłoszony w konstruktorze, wówczas obiekt nie został w pełni stworzony i podczas oczyszczania stosu **nie zostanie wywołany destruktor obiektu**.

- **Wywołane zostaną natomiast destruktory utworzonych obiektów**: klas bazowych i atrybutów częściowo zainicjowanego obiektu złożonego.

Wyjątki w konstruktorach

```
class A{public: ~A(){cout<<"~A ";} };
class B{public: ~B(){cout<<"~B ";} };

class Composite
{
    A a;
    B b;
public:
    Composite():a(),b()
    {
        throw 0;
    }
    ~Composite(){cout <<"~Composite ";}
};
```

- Obiekt klasy `Composite` nie zostanie utworzony
- Istnieć będą jego komponenty `A a` oraz `B b`. Dla nich zostaną wywołane destruktory.

Wyjątki w konstruktorach

```
int main()
{
    try{
        Composite c;

    }catch(int)    {
        cout<<"Exception"<<endl;
    }
}
```

Wynik

```
~B ~A Exception
```

Wyjątki w konstruktorach

- Jeżeli przed wygenerowaniem wyjątku konstruktor uzyskał dostęp do pewnych zasobów - **nie będą one możliwe do odzyskania**.
- Zasobem może być na przykład plik lub przydzielona pamięć na stercie.
- Zalecaną techniką jest opakowanie żądanych zasobów klasami i ich alokacja w postaci atomowych obiektów – atrybutów.

Przykład

- Konstruktor tej klasy zawsze wygeneruje wyjątek
- Pamięć wskazywana przez buf nigdy nie zostanie zwolniona

```
class BadDesign
{
    int*buf;
public:
    BadDesign(int size)
    {
        buf=new int[size];
        throw 0;
    }
    ~BadDesign(){
        if(buf!=0)delete []buf;
    }
};
```

Przykład

```
class IntVector
{
public:
    IntVector(int size):b(size>0?new int[size]:0){}
    ~IntVector(){
        if(b)delete b;
    }
    int*buf(){return b;}
    int*b;
};
```

`IntVector` niewiele różni się od `BadDesign`. Ale **nie generuje wyjątków w konstruktorze** i obiekt zostanie zawsze utworzony.

Przykład

```
class GoodDesign
{
    IntVector buf;
public:
    GoodDesign(int size):buf(size)
    {
        throw 0;
    }
};
```

- Obiekt klasy GoodDesign nie zostanie stworzony.
- Ale buf nie jest wskaźnikiem, ale obiektem. Wywołany zostanie jego destruktor, który zwolni pamięć

Przykład

- Zamiast własnoręcznie implementować `IntVector` można użyć bibliotecznej klasy `std::vector`
- Jest to nawet zalecane

```
class GoodDesign2
{
    std::vector<int> buf;
public:
    GoodDesign2(int size):buf(size>0?size:0)
    {
        throw 0;
    }
};
```