

Obsługa błędów

- Często pisząc jakąś operację zauważamy, że nie zawsze da się ona poprawnie wykonać.
- Nasza funkcja musi jakoś zareagować ale jak?
 - 1 Wypisać komunikat i przerwać działanie programu (bardzo brutalne).
 - 2 Przekazać wartość oznaczającą błąd.
 - 3 Przekazać jakąś poprawną wartość, natomiast ustawić zmienną sygnalizującą błąd.
 - 4 Wykonać dostarczoną przez użytkownika procedurę obsługi błędu.

Wartość oznaczająca błąd

- Dość typowe podejście w C, zwłaszcza przy wywołaniach systemowych.
- Nie zawsze możliwe:
 - ▶ może nie istnieć taka wartość,
 - ▶ funkcja może nie dawać wyniku, np. konstruktor.
- Niewygodne — wymaga sprawdzenia przy każdym wywołaniu.

Zmienna sygnalizująca błąd

- Podobnie jak poprzednio wymaga sprawdzania przy każdym wywołaniu
- Różne biblioteki (moduły) mogą mieć różne zmienne w tym celu
- Użytkownik operacji może nie być świadom takiej metody sygnalizacji błędu.

Procedura obsługi błędu

Wywołać funkcję dostarczoną przez użytkownika (np. jako parametr).

- Najlepsze, elastyczne rozwiązanie.
- Nie wymaga sprawdzania w miejscu wywołania funkcji
- Wada: każde wywołanie funkcji trzeba obciążyć dodatkowym parametrem.

Wyjątki

- Dla rozwiązania takich problemów istnieje w C++ mechanizm obsługi wyjątków.
- Pojęcie *wyjątek* oznacza błąd (nietypową, niepożądaną sytuację).
- Obsługa wyjątków oznacza reakcję programu na wykryte błędy.
- Funkcja, która napotkała problem *zgłasza* (“rzuca”) wyjątek.
- Wyjątek jest przekazywany do miejsca wywołania funkcji, gdzie może być “wyłapany” i obsłużony albo przekazany wyżej.
- Przy wychodzeniu z funkcji i bloków usuwane są obiekty automatyczne.

Składnia

Zgłoszenie wyjątku:

```
throw <wyrażenie>
```

Obsługa wyjątków:

```
try {  
    <instrukcje>  
} catch(<parametr 1>) {  
    <obsługa wyjątku 1>  
//...  
} catch(parametr n) {  
    <obsługa wyjątku n>  
}
```

Semantyka

- Gdy któraś z instrukcji w części `try` przekazała wyjątek, przerywamy wykonanie tego ciągu i szukamy `catch` z odpowiednim parametrem.
- Jeśli znajdziemy, to wykonujemy obsługę tego wyjątku, a po jej zakończeniu instrukcje po wszystkich `catch`.
- Jeśli nie znajdziemy, przechodzimy do miejsca wywołania (usuwając obiekty automatyczne bieżącej funkcji) i kontynuujemy poszukiwanie.
- Jeśli nie znajdziemy w żadnej z aktywnych funkcji, wykonanie programu zostanie przerwane.

Przykład

```
class Napis {
    char *p;  int rozmiar; // rozmiar tablicy
public:
    class Rozmiar{}; // wyjątek: zły rozmiar
    class Zakres{}; // wyjątek: wyjście poza zakres
    Napis(int r);
    ~Napis();
    char& operator[](int i)
};

Napis::Napis(int r) {
    if(r<=0) throw Rozmiar();
    rozmiar = 16*(r/16+1) // alokuj po 16 bajtów
    p = new char[rozmiar];
};
```


Przykład c.d.

```
char& Napis::operator[] (int i) {  
    if (0<=i&i<rozmiar)  
        return p[i] ;  
    else  
        throw Zakres();  
}  
  
Napis* buduj(char* s) {  
    Napis* n = new Napis(strlen(s))  
    int i;  
    for(i=0;s[i];i++)  
        napis[i] = s[i];  
    napis[i]=0;  
    return n;  
}
```

Użycie

```
try {  
    n = buduj(s) ;  
} catch(Napis::Rozmiar) {  
    // obsługa błędnego rozmiaru  
} catch(Napis::Zakres) {  
    // obsługa błędu przekroczenia zakresu  
}  
// Sterowanie dochodzi tutaj, gdy:  
// a) nie było wyjątku  
// b) był wyjątek Zakres lub Rozmiar, ale  
//    obsługa nie spowodowała wyjścia z funkcji
```

Obsługa wyjątków może być podzielona między wiele funkcji:

```
void f1() {  
    try { f2(n); }  
    catch(Napis::Rozmiar() { /* ... */ }  
}  
  
void f2(Napis& n) {  
    try { /* używanie napisu n */ }  
    catch(Napis::Zakres) { /* ... */ }  
}
```

Wyjątki przy obsłudze wyjątków

- Przy obsłudze wyjątku może się pojawić także `throw`.
- Można zgłosić nawet wyjątek, który jest właśnie obsługiwany.
- Nie powoduje to zapętlenia ani nie jest błędem.
- Wyjątek rzucony wewnątrz `catch` będzie obsługiwany tak jakby był rzucony po wszystkich blokach `catch`
- Jeśli chcemy rzucić ponownie właśnie obsługiwany wyjątek , możemy to zrobić instrukcją `throw` bez argumentu.

Wyjątki z danymi

- Instrukcja `throw` przekazuje obiekt.
- Może on posiadać składowe i przenosić informacje z miejsca zgłoszenia do miejsca obsługi.

```
class Napis {  
    char *p;  int rozmiar;  
public:  
    class Rozmiar{};  
    struct Zakres{ int indeks;  
                   Zakres(int i):indeks(i){}};  
  
    Napis(int r);  
    ~Napis();  
    char& operator[](int i)  
};
```

Użycie

```
char& Napis::operator[](int i){  
    if(0<=i&i<rozmiar) return p[i] ;  
    throw Zakres(i);  
}  
// ...  
try{ /* używanie napisu */ }  
catch(Napis::Zakres z) {  
    cerr << "Zły indeks" << z.indeks << endl;  
}
```

Hierarchie wyjątków

- Wyjątki są obiektami klas.
- Możemy tworzyć hierarchie klas wyjątków.
- Dzięki temu możemy je obsługiwać na wybrtany poziomie ogólności/szczegółowości.

Przykład

```
struct BłądMatematyczny{};
struct Nadmiar : BłądMatematyczny{};
struct Niedomiar : BłądMatematyczny{};
struct DzieleniePrzezZero : BłądMatematyczny{};
// ...
try { /* ... */ }
catch(Nadmiar) { /* Obsługa nadmiaru */ }
catch(BłądMatematyczny) { /* Obsługa pozostałych */ }
```


Obsługa dowolnego wyjątku

- Konstrukcja `catch (...)` oznacza wylapanie dowolnego wyjątku.
- Można ją wykorzystać razem z bezargumentowym `throw`:

```
void f(){  
    try { /* ... */ }  
    catch(...){  
        // Instrukcje, które zawsze muszą być wykonane  
        // na koniec f jeśli wystąpił błąd  
        throw; // Przekaż wyjątek dalej  
    }  
}
```

Zdobywanie zasobów

- Często problemem jest zwolnienie zasobów (otwarte pliki, połączenia sieciowe) przydzielonych zanim wystąpił błąd.
- Naiwne rozwiązanie tego problemu może wyglądać np. tak:

```
void f(Adres& addr) {  
    int sock = socket(PF_INET, SOCK_STREAM, 0);  
    connect(sock, addr.data, addr.len);  
    try { /* Używaj połączenia */ }  
    catch(...) {  
        close(sock); // zamknij połączenie  
        throw;  
    }  
    close(sock);  
}
```

- Powyższe rozwiązanie jest rozwlekłe i nużące.
- Jak każde rozwlekłe rozwiązanie jest podatne na błędy.
- Lepsze rozwiązanie znajdziemy myśląc obiektowo:

```
class Socket {  
    int m_sock;  
public:  
    Socket() {m_sock=socket (PF_INET, SOCK_STREAM, 0);}  
    ~Socket() {close(m_sock);}  
    void connect (Adres& a) {  
        ::connect (m_sock, a.data, a.len);  
    }  
    operator int() {return m_sock;}  
}
```

Użycie

Teraz możemy zapisać naszą funkcję krótko i elegancko:

```
void f(Adres& addr) {  
    Socket s;  
    s.connect(addr);  
    /* Używaj połączenia */  
}
```

Jeśli w trakcie używania połączenia nastąpi wyjątek, `s` jako obiekt automatyczny zostanie zwolniona, a destruktor zadba o zamknięcie połączenia.

Specyfikacja interfejsu

- Deklaracja funkcji może (ale nie musi) specyfikować wyjątki, które może zgłosić, np.
`void f() throw(x1, x2, x3);`
- Zgłoszenie wyjątku innego niż zadeklarowany powoduje błąd.
- Jeśli nie zadeklarowano żadnego wyjątku, funkcja może zgłosić każdy wyjątek.
- Jeśli funkcja ma nie zgłaszać żadnych wyjątków, to piszemy

```
void g() throw();
```