Kurs języka C++

8. Wyjątki

SPIS TREŚCI

- Ogólne spojrzenie na wyjątki
- Zgłaszanie i łapanie wyjątków
- Grupowanie wyjątków
- Dopasowywanie wyjątków
- Implementacja mechanizmu zgłaszania i łapania wyjątków
- Wyjątki w konstruktorach i w destruktorach
- Zdobywanie zasobów poprzez inicjalizację
- Specyfikacja wyjątków
- Wyjątki oraz `new` i `delete`
- Wyjątki z biblioteki standardowej
- Własne wyjątki
- Asercje

OGÓLNE SPOJRZENIE NA WYJĄTKI

- Wyjątki zaprojektowano do wspierania obsługi błędów.
- System wyjątków dotyczy zdarzeń synchronicznych – na przykład do kontroli zakresu czy błędów we/wy.
- Mechanizm obsługi wyjątków można traktować jako nielokalną strukturę sterującą, która korzysta ze zwijania stosu.

OBSŁUGA BŁĘDÓW

- Wyjątek to obiekt sygnalizujący błąd.
- Wyjątki można zgłaszać po wykryciu sytuacji krytycznej instrukcją `throw`.
- Wyjątki zgłoszone w bloku `try` można wyłapać w bloku `catch` i obsłużyć sytuację problemową.
- Wyjątki w sposób naturalny dzielą kod na część obliczeniową i część sterującą obliczeniami.

ZGŁASZANIE I ŁAPANIE WYJĄTKÓW

- Wyjątek to obiekt dowolnego typu.
- Kod wykrywający błąd zgłasza wyjątek instrukcją `throw`, na przykład:
```
throw 15;
throw "problem z dokładnością obliczeń";
throw moja_klasa(7.532);
```
- Chęć złapania wyjątku sygnalizuje się umieszczeniem kodu w instrukcji `try-catch`.
- Wynikiem działania `throw` jest zwinięcie stosu, aż do znalezienia odpowiedniego bloku `catch` (w funkcji, która bezpośrednio lub pośrednio wywołała funkcję zgłaszająca wyjątek).

ZGŁASZANIE I ŁAPANIE WYJĄTKÓW

- Przykład zgłoszenia i obsługi błędu:

```
try
{
    int x = 0;
    cerr << "integer (>0) ";
    cin >> x;
    if (! cin) throw "i/o error";
    if (x<=0) throw x;
    // …
}
catch (const char *ex)
{
    cerr << "number format error" << endl;
}
catch (int ex)
{
    cerr << "number value error" << endl;
}
```

ZGŁASZANIE I ŁAPANIE WYJĄTKÓW

- Program może obsługiwać tylko wyjątki zgłaszane w bloku `try`.
- Po zgłoszeniu wyjątku, sterowanie nie wraca już do miejsca zgłoszenia.
- Po obsłużeniu wyjątku w bloku `catch` sterowanie przenoszone jest za instrukcję `try-catch`.
- Wyjątki są rozróżniane po typie.
- Funkcje call-back'owe potrafią rozwiązać problem w miejscu jego wystąpienia.

GRUPOWANIE WYJĄTKÓW

- Wyjątki często w sposób naturalny tworzą rodziny (zastosowanie dziedziczenia w strukturalizacji wyjątków).
- Przykład hierarchii wyjątków:

```
class BladMat {};
class Nadmiar : public BladMat {};
class Niedomiar : public BladMat {};
class DzielZero : public BladMat {};
```

- Przykład organizacji rozpoznawania wyjątków:

```
try {
    // throw …;
}
// wyjątek DzielZero
catch (DzielZero) {/*…*/}
// pozostałe wyjątki dziedziczące po BladMat
catch (BladMat) {/*…*/}
```

- Kolejność bloków `catch` ma znaczenie przy rozpoznawaniu wyjątków.

DOPASOWYWANIE WYJĄTKÓW

- Rozważmy przykład:

```
try
{
    throw E;
}
catch (H)
{
    // kiedy się tutaj znajdziemy?
}
```

1) H jest tego samego typu co E;
2) H jest jednoznaczną publiczną klasą bazową dla E;
3) H i E są wskaźnikami, a dla typów na które wskazują zachodzi 1) lub 2);
4) H jest referencją, a dla typu do którego się odnosi zachodzi 1) lub 2).
5) H jest obiektem stałym, a dla typu do którego się odnosi zachodzi 1) lub 2).

ZŁAPANIE KAŻDEGO WYJĄTKU

- Można wyłapać każdy wyjątek blokiem `catch(...)`.
- Blok `catch(...)` może wystąpić tylko jako ostatni blok.
- W bloku `catch(...)` nie jest znany typ wyjątku.
- W bloku `catch` można powtórnie zgłosić ten sam wyjątek który właśnie został wyłapany instrukcją `throw` bez argumentów.
- Zgłoszenie innego wyjątku w bloku `catch` można traktować jak podmianę wyjątku (można zmienić nie tylko wartość ale i typ zgłaszanego wyjątku).

ZAGNIEŻDŻANIE INSTRUKCJI TRY-CATCH

- Instrukcję `try-catch` można umieścić w bloku `try` – wtedy niewyłapane wyjątki w wewnętrznej instrukcji `try-catch` będą zgłoszeniem wyjątku w zewnętrznym bloku `try`.
- Instrukcję `try-catch` można również umieścić w bloku `catch` – wtedy niewyłapane wyjątki w wewnętrznej instrukcji `try-catch` będą traktowane jak zgłoszenie wyjątku w zewnętrznym bloku `catch`.

IMPLEMENTACJA MECHANIZMU ZGŁASZANIA I ŁAPANIA WYJĄTKÓW

- Odwikłanie stosu – wielkie sprzątanie.
- Po zgłoszeniu instrukcją `throw` obiekt wyjątku jest umieszczany w pamięci globalnej w specjalnie do tego przeznaczonym miejscu.
- Wyjątek uznaje się za obsłużony w momencie jego wyłapania przez jakiś blok `catch`, ale dopiero przy wyjściu z tego bloku wyjątek jest likwidowany.
- Nie wolno rzucać wyjątków, gdy inny wyjątek jest w trakcie lotu!

WYJĄTKI W KONSTRUKTORACH I W DESTRUKTORACH

- Gdy wyjątek zostanie zgłoszony w konstruktorze, to obiekt nie zostanie utworzony.
- Gdy wyjątek zostanie zgłoszony w konstruktorze w trakcie inicjalizacji części odziedziczonej lub obiektu składowego, to zainicjalizowana część zostanie automatycznie zlikwidowana.
- Gdy chcemy zgłosić wyjątek w ciele konstruktora, to najpierw należy zwolnić zasoby przydzielone w ciele konstruktora.
- Nie wolno zgłaszać wyjątków w destruktorach, bo to może powodować problemy przy odwikłaniu stosu w trakcie lotu rzuconego wcześniej innego wyjątku!

ZDOBYWANIE ZASOBÓW POPRZEZ INICJALIZACJĘ

- Problem: kiedy funkcja na początku rezerwuje zasób (otwiera strumień, przydziela pamięć, ustawia klucz kontroli dostępu, itp), to może go na końcu nie zwolnić, gdy po drodze zostanie zgłoszony wyjątek.
- Rozwiązanie: zarządzanie zasobami poprzez opakowywanie ich klasami (technika RAII).
- Schemat postępowania:
  - w konstruktorze klasy opakowującej zasób zostaje zarezerwowany (gdy rezerwacja się nie powiedzie zostaje zgłoszony wyjątek);
  - klasa opakowująca udostępnia narzędzia do korzystania z zasobu;
  - w destruktorze klasy opakowującej zasób zostaje zwolniony (zadziała również w przypadku zwijania stosu przy zgłoszonym wyjątku).

ZDOBYWANIE ZASOBÓW POPRZEZ INICJALIZACJĘ

- Przykład zdobywania zasobów poprzez inicjalizację:

```
class plik
{
    FILE *wsk;
public:
    plik (const char *naz, const char *atr)
    {
        wsk = fopen(naz,atr);
        if (!wsk) throw brak_pliku;
    }
    ~plik () throw()
        { fclose(wsk); wsk = 0; }
    operator FILE* () noexcept
        { return wsk; }
};
// …
plik p(”a.txt”,”r”);
double wsp;
fscanf(p,”%lf”,&wsp);
```

SPRYTNE WSKAŹNIKI

- Wzorzec klasy `shared_ptr` wspiera technikę zdobywania zasobów poprzez inicjalizację – jego definicja znajduje się w pliku nagłówkowym `<memory>`.
- Obiekt `shared_ptr` jest inicjalizowany wskaźnikiem i można się nim posługiwać w programie jak zwykłym wskaźnikiem.
- Konstruktor i przypisanie wzorca `shared_ptr` zapewniają niejawną konwersję z `shared_ptr<P>` do `shared_ptr<B>` o ile można dokonać konwersji z `P*` do `B*`.
- Współdzielony wskaźnik `shared_ptr` jest wskaźnikiem ze zliczaniem referencji.

SPRYTNE WSKAŹNIKI

- Współdzielony wskaźnik `shared_ptr` automatycznie niszczy swoją zawartość tylko, jeśli nie ma już współdzielonych wskaźników odnoszących się do obiektu początkowo tworzonego dla współdzielonego wskaźnika – wtedy w destruktorze `shared_ptr` zapewnione jest wywołanie operatora `delete` na wskazywany obiekt.
- Wskaźnik `shared_ptr` posiada operator dereferencji do przechowywanego obiektu (`*`) i operator dostępu do pól składowych (`->`).
- Przykład:

```
struct C { int a; int b; };
shared_ptr<C> foo(new C);
foo->a = 10, foo->b = 20;
cout << "foo: " << (*foo).a
    << ' ' << (*foo).b << endl;
```

SPECYFIKACJA WYJĄTKÓW - przestarzale

- Funkcja może wyspecyfikować zbiór wyjątków, które mogą być rzucone w trakcie wykonania funkcji na liście kontrolnej.
- Lista kontrolna to fraza `throw()` na końcu nagłówka funkcji – w nawiasie umieszczamy zbiór dopuszczalnych wyjątków.
- Przykłady list kontrolnych:

```
void K::f (int) const throw(ex1, ex2, ex3);
// funkcja składowa f(int) może rzucić
// wyjątkiem typu ex1, ex2 albo ex3
double g (double) throw();
// funkcja g(double) nie zgłasza żadnych
wyjątków
void h (void);
// funkcja h() może zgłosić dowolny wyjątek
```

SPECYFIKACJA WYJĄTKÓW - przestarzale

- Nie jest możliwe w czasie kompilacji sprawdzenie każdego naruszenia specyfikacji interfejsu. Jeśli funkcja z listą kontrolną spróbuje zgłosić wyjątek spoza listy, to jest wtedy wywoływana funkcja `unexpected()`, która domyślnie wywołuje `terminate()`.
- Można jednak dostarczyć i wywołać funkcję ratującą, ustawiając uchwyt `_unexpected_handler` za pomocą `set_unexpected()` – funkcja ratująca typu `void(*)()` może na przykład zgłosić wyjątek `bad_exception` (wyjątek ten trzeba wtedy dopisać do listy kontrolnej) albo wywołać funkcję `exit()`.

SPECYFIKACJA WYJĄTKÓW - przestarzale

- Funkcję wirtualną można nadpisać tylko funkcją, której specyfikacja wyjątków na liście kontrolnej jest co najmniej tak restrykcyjna jak jej własna specyfikacja.
- Przykład nadpisywania funkcji wirtualnych:

```
class B
{
public:
     virtual void f ();
     virtual void g () throw(X,Y);
     virtual void h () throw(X);
};
class P: public B
{
 public:
     void f () override throw(X); // ok
     void g () override throw(X); // ok
     // void h () override throw(X,Y);
     // błąd – dodanie nowego wyjątku Y do listy
};
```

SPECYFIKACJA WYJĄTKÓW - współcześnie

- Lista kontrolna wyjątków jest konstrukcją przestarzałą, należy tylko podać informację, czy metoda może rzucić albo nie rzuca wyjątków.
- Deklarator `noexcept` oznacza, że metoda nie zgłasza żadnych wyjątków (jest to równoważne `throw()`).
- Deklarator `noexcept` posiada parametr boolowski: noexcept(true) – ozanacza nie
- Każdy destruktor jest domyślnie funkcją `noexcept` (wszystkie inne funkcje mogą domyślnie zgłaszać każdy wyjątek).

WYJĄTKI ORAZ NEW I DELETE

- Funkcje użyte do implementacji operatorów `new` i `delete` są zadeklarowane w `<new>`. Deklaracja tych operatorów jest następująca:
```
void* operator new (size_t) throw(bad_alloc);
void operator delete (void*) noexcept;
void* operator new[] (size_t) throw(bad_alloc);
void operator delete[] (void*) noexcept;
```
- Operator `new` (oraz `new[]`) zgłasza wyjątek `bad_alloc`, gdy nie uda się zarezerwować pamięci na obiekt (tablicę obiektów).
- Istnieje deklaracja obiektu, który powoduje, że operator `new` nie zgłasza wyjątku, tylko przekazuje wskaźnik pusty:
```
struct nothrow_t {};
extern const nothrow_t nothrow;
```
  Stworzono też specjalne wersje operatora `new` z parametrem `nothrow_t`, które zapobiegają zgłaszaniu wyjątków:
```
void* operator new (size_t, const nothrow_t &)
    noexcept;
void* operator new[] (size_t, const nothrow_t &)
    noexcept;
```

WYJĄTKI ORAZ NEW I DELETE

- Przykład użycia `nothrow` przy alokacji pamięci:

```
int *p = new int[1000000];
// może zgłosić wyjątek bad_alloc
// …
// poniższy kod nie zgłosi wyjątku
if (int *q = new(nothrow) int[1000000])
{
    // przydział się powiódł
}
else
{
    // przydział nie powiódł się
}
```

- Funkcja `uncaught_exception()` zwraca `true`, gdy wyjątek zgłoszono ale jeszcze nie wyłapano – umożliwia to specyfikowanie różnych działań w destruktorze zależnie od tego, czy obiekt jest niszczony normalnie, czy w ramach zwijania stosu.

BRAK PAMIĘCI I OPERATOR `NEW`

- Gdy operator `new` próbuje przydzielić pamięć a wolnej pamięci już nie ma, to zgłasza on wyjątek `bad_alloc`.
- Można jednak dostarczyć i wywołać funkcję ratującą, ustawiając uchwyt `_new_handler` za pomocą `set_new_handler()` – funkcja ratująca typu `void(*)()` powinna odzyskać pamięć, a jeśli się to nie uda, to powinna zgłosić wyjątek `bad_alloc`.
- Gdy nie wyłapiemy wyjątku `bad_alloc`, to program zakończy się wywołaniem funkcji `terminate()`.

BRAK PAMIĘCI I OPERATOR NEW

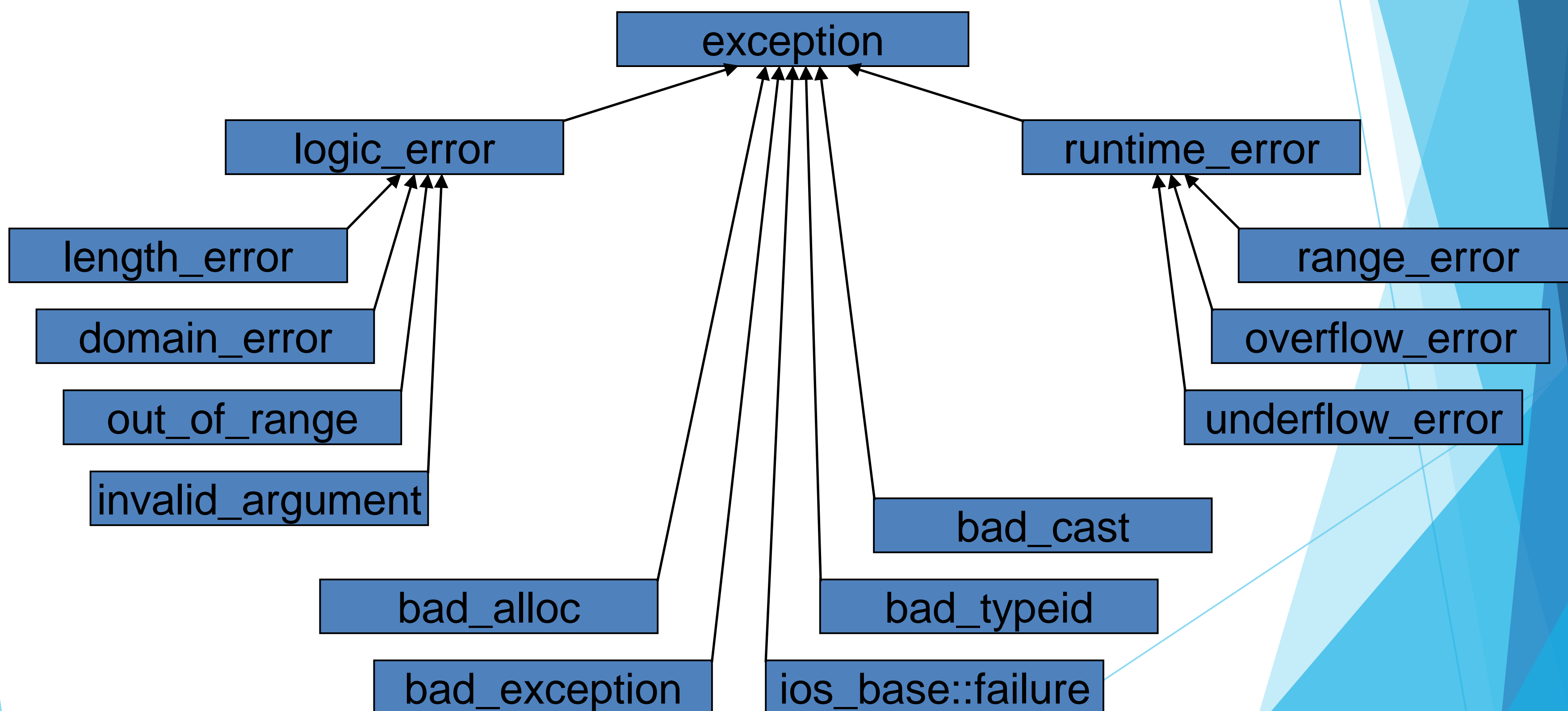
- Przykład funkcji ratunkowej w przypadku braku pamięci:
```
void new_hnd ()
{
    int bytes = find_mem();
    if (bytes<min_alloc) throw bad_alloc;
}
```
- Przykład użycia funkcji ratunkowej w przypadku braku pamięci:
```
void (*old_hnd)() = set_new_handler(new_hnd);
try {
    // …
}
catch (bad_alloc) {
    // …
}
catch (...) {
    set_new_handler(old_hnd);
    throw;
}
set_new_handler(old_hnd);
```

KOŃCZENIE PROGRAMU W KRYTYCZNYCH SYTUACJACH

- Funkcja standardowa `exit(int)` kończy program w łagodny sposób (opróżnia bufory, zamyka pliki, itp).
- Funkcja standardowa `abort()` kończy program w drastyczny sposób (bez kosmetyki dotyczącej buforów, plików, itp).
- Funkcja `terminate()`, w której jest wywoływana funkcja `abort()`, jest automatycznie wywoływana w sytuacjach krytycznych takich jak:
  - rzucenie wyjątku, którego nie złapał żaden blok `catch`;
  - rzucenie wyjątku w trakcie lotu innego wyjątku;
  - rzucenie wyjątku spoza listy kontrolnej.
- Za pomocą funkcji `set_terminate()` możemy zmienić uchwyt `_uncaught_handler` ustalając własną procedurę obsługi zdarzeń krytycznych związanych z wyjątkami. Funkcja ta podmienia uchwyt do funkcji wywoływanej w `terminate()` – standardowo jest to funkcja `abort()`.

WYJĄTKI Z BIBLIOTEKI STANDARDOWEJ

- Hierarchia klas wyjątków standardowych:



WŁASNE WYJĄTKI

- Wyjątkiem może być dowolny obiekt, ale dobrze jest projektować własną hierarchię klas wyjątków, która dziedziczy po `exception`.
- Gdy definiujesz własny wyjątek, pamiętaj aby nie zgłaszał on innych wyjątków w konstruktorach, w destruktorze i w przypisaniu kopiującym.
- Gdy definiujesz własny wyjątek dziedziczący po `exception`, pamiętaj aby zdefiniować w nim konstruktor domyślny, konstruktor kopiujący, przypisanie kopiujące, wirtualny destruktor oraz nadpisz metodę `what()`.

ASERCJE

- Za pomocą asercji możemy oznaczyć w programie niezmienniki, czyli warunki, które niezależnie od wartości zmiennych muszą pozostać prawdziwe – jeśli asercja zawiedzie, oznacza to, że popełniliśmy błąd w algorytmie albo nastąpiła sytuacja wyjątkowa.
- Makro `assert()` zdefiniowane w pliku nagłówkowym `<cassert>` służy do debuggowania programów.
- Użycie:
  `assert(`***wyrażenie***`);`
- Działanie: jeśli warunek, który testuje asercja jest prawdziwy to program nie reaguje, natomiast w przypadku gdy warunek ten jest fałszywy to na standardowe wyjście dla błędów zostanie wypisany odpowiedni komunikat o błędzie i program zostanie przerwany za pomocą funkcji `abort()`.
- Aby pozbyć się asercji, uwalniając kod od spowalniających obciążeń, wystarczy przed dołączeniem pliku nagłówkowego `<cassert>` zdefiniować makro `NDEBUG` - nie trzeba wówczas kasować żadnych wystąpień makra `assert()`.

ASERCJE STATYCZNE

- Starszy standard C++ posiada dwie metody do testowania asercji: makro `assert` i dyrektywę preprocesora `#error`, jednak żaden z nich nie jest odpowiedni do używania w szablonach (makro testuje asercje w czasie wykonywania kodu, a dyrektywa preprocesora testuje w fazie preprocesorowej, czyli przed tworzeniem instancji szablonów).
- Można testować asercje również w czasie kompilacji przy użyciu słowa kluczowego `static_assert`:
  `static_assert(`*stałe_wyrażenie*`,` *komunikat_błędu*`);`
- Kiedy *stałe_wyrażenie* jest fałszywe, wtedy kompilator zgłasza *komunikat_błędu*.
- Statyczne asercje są przydatne także poza szablonami; przykładowo, jakaś szczególna implementacja algorytmu mogłaby zależeć od tego, aby rozmiar `long` był większy niż rozmiar `int`, czyli tego, czego standard nie zapewnia.