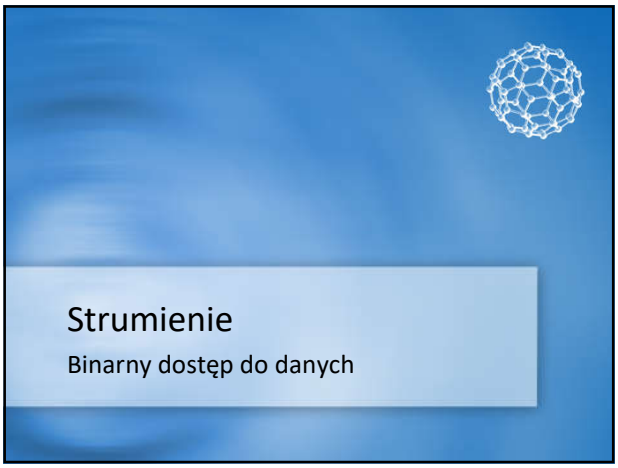


104

C++ - strumienie

Binarny dostęp do danych:

Do odczytu danych ze strumienia w taki sposób, aby można je było interpretować jako ciąg binarny[1] służy metoda klasy **istream**:

```
basic_istream& read( char_type *_Str, streamsize _Count );
```

Metoda odczytuje określoną liczbę bajtów *_Count* i zapisuje ją pod wskazany adres – wskaźnik *_Str*. Jeżeli wcześniej napotka koniec pliku, przerywa odczyt i kończy działanie ustawiając odpowiednią flagę błędu.

Ciąg binarny – ciąg bajtów (*nie bitów!*).

Interpretacja danych dowolnych typów jako ciąg binarny polega na odczytaniu tych danych z pominięciem informacji o ich strukturze, tj. bajt po bajcie (*nie bit po bicie!*).

105

C++ - strumienie

Binarny dostęp do danych:

Do zapisu danych do strumienia służy metoda klasy **ostream**:

```
basic_ostream& write( const char_type *_Str,
                      streamsize _Count );
```

Aby zastosować tę metodę do korzystania z pliku w trybie binarnym, należy obszary pamięci, do który dane mają trafić (lub skąd mają być pobrane), zrzutować na wskaźniki typu `const char_type*` (zapis do strumienia) lub `char_type*` (odczyt ze strumienia).

106

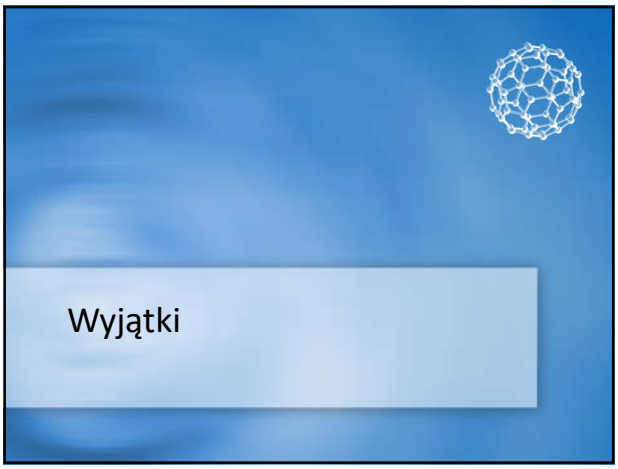
C++ - strumienie

Binarny dostęp do danych:

```
struct Bloczek {
       int a;
       double b;
       char c;
};

int main( )
{
 ofstream ofx( "test.txt", ios::binary);  ifstream ifx( "test.txt",ios::binary);
 Bloczek s={3,3.14,'a'};                  Bloczek s2;
 const char *pom = (const char*)&s;       char *pom2 = (char*)&s2;
 ofx.write((char*)&s, sizeof(s));         ifx.read((char*)&s2, sizeof(s2));
 ofx.write(pom, sizeof(s));               ifx.read(pom2, sizeof(s2));
 ofx.close();                             cout << s2.a << s2.b << s2.c;
```


107



108

Wyjątki

Wprowadzenie

- Korzystanie z wyjątków pozwala w kodzie programu oddzielić kod obsługi błędu od kodu wykonania zadania
- Przykład – otwarcie pliku:

Kod zadania

```
FILE* stream;
if((stream = fopen("foo.c", "r")) == NULL)
  printf("Pliku 'foo.c' nie otwarto\n" );
else
  printf("Plik 'foo.c' został otwarty\n");
```

Kod obsługi

109

Wyjątki

Wprowadzenie

- Zamiast
  - umieszczać w kodzie obsługi zadania sekwencje instrukcji odpowiedzialne za poprawne zachowanie programu w razie wystąpienia błędu,

  to
  - piszemy kod obsługi zadania tak, jakbyśmy zakładali, że wszystko odbędzie się bez problemów. ☺
- Kod obsługi błędu umieszczamy natomiast w innym miejscu programu i tylko **wiążemy razem te dwa bloki kodu współzależnością.**

Wyjątki

Wprowadzenie

- Wystąpienie błędu w obsłudze zadania powoduje natychmiastowe przerwanie obsługi i przekazanie sterowania do innego obszaru kodu, gdzie znajdują się instrukcje obsługujące błąd.
- Kod obsługi zadania można podzielić na kilka sekcji, które uważamy za funkcjonalnie zamknięte całości. Taka sekcja może w całości wykonać się poprawnie lub w całości zostać niewykonana

Wyjątki

Wprowadzenie

Aby mechanizm wyjątków mógł działać adekwatnie do sytuacji błędnej, musi zostać przekazana informacja o rodzaju błędu, jaki wystąpił.

Do tego celu służą obiekty, w których można zawrzeć informacje o indywidualnych okolicznościach zdarzenia.

Wyjątki

Wprowadzenie

Podsumowując, do wprowadzenia wyjątków potrzebne są:

1) sposób oznakowania sekcji kodu, o której wiadomo, że w trakcie jej wykonania może być rzucony wyjątek,
2) sposób oznakowania sekcji kodu, która nie jest zwykłym fragmentem programu, ale reprezentuje kod mający się wykonać w przypadku rzucenia wyjątku,
3) instrukcja rzucania wyjątku, która spowoduje przeniesienie sterowania do sekcji obsługi błędu,
4) klasa na podstawie której tworzone będą obiekty zawierające informacje o okolicznościach, w których wystąpił błąd.

Wyjątki

Rzucanie wyjątków

instrukcja **throw**

Powoduje:

1) przerwanie wykonywania bieżącego kodu,
2) wygenerowanie obiektu przechowującego informacje o okolicznościach błędu,
3) przeniesienie sterowania z bieżącego kodu do miejsca, które jest w stanie przyjąć obiekt danego typu i wykorzystać zawarte w nim informacje do obsługi sytuacji błędnej.

Wyjątki

Rzucanie wyjątków

Jeżeli w jakimś miejscu kodu wystąpiła sytuacja niepoprawna, to zamiast ustawiać flagę błędu, lub natychmiast kończyć działanie funkcji poleceniem **`return`**, które zwraca kod błędu (stary styl), w tym miejscu umieszczamy polecenie rzucenia wyjątku, np.:

```
if (x>0)
        y = sqrt(x);
else
        return -1; // stary sposób, typowy dla języka C

// albo:
throw MyError("nieprawidłowa próba pierwiastkowania");
        // nowy sposób, typowy dla C++.
        // MyError to nazwa klasy, której obiekt tworzymy
```

Wyjątki

Co się stanie dalej?..

1) Jeżeli jesteśmy wewnątrz funkcji, polecenie **`throw`** spowoduje przerwanie wykonania tej funkcji i przejście sterowania do miejsca, gdzie funkcja była wywołana.
2) Jeżeli była wywołana wewnątrz innej funkcji, ta również zostanie przerwana, itd. Przerwania i przejścia sterowania w górę będą następowały aż dojdą do funkcji **`main`**
3) Funkcja **`main`** również zostanie przerwana a program zakończy swoje działanie wyświetlając komunikat, że został przerwany z powodu wyrzucenia wyjątku.

To bardzo niedoskonała obsługa sytuacji błędnej ☹

Wyjątki

Aby program nie przerwał swojego działania, należy w nim zawrzeć dodatkowe instrukcje pozwalające na przechwycenie wyjątku w programie, tj.:

Należy

1) poinformować kompilator, w jakim obszarze kodu programu którakolwiek z instrukcji tego kodu może spowodować rzucenie wyjątku, oraz
2) umieścić pod tym obszarem instrukcje przechwytujące.

Wyjątki

Do oznakowania sekcji kodu, w której może wystąpić wyjątek, służy instrukcja **try**

Przykład:

```
try {
 ….  // tutaj różne instrukcje naszego programu
}
```

Jeżeli w ciele funkcji znajduje się blok **`try`**, wewnątrz którego został rzucony wyjątek, funkcja nie zostanie przerwana, tj. sterowanie nie opuści tej funkcji, ale zostanie przeniesione tylko na koniec tego bloku **`try`**, do pierwszej instrukcji za tym blokiem.

Wyjątki

Raz rzucony wyjątek musi gdzieś trafić. Jeżeli programista nie przewidział miejsca obsługi, program zostanie zatrzymany a wyjątek obsłużony przez system operacyjny.

Normalnym miejscem obsługi wyjątku jest procedura, umieszczona zaraz za blokiem **`try`**

Dlatego bezpośrednio za końcem tego bloku, jeżeli chcemy, możemy umieścić kawałek kodu, który wykona odpowiednie czynności, właściwe dla okoliczności, w których zaszła sytuacja błędna.

Taka czynność nazywana jest *obsługą wyjątku*.

Wyjątki

Obsługa wyjątku

procedura ma nazwę **`catch`** i jeden argument wywołania, którym jest obiekt określonego typu zgodnego z typem obiektu reprezentującego wyjątek:

```
try {
   …        // tutaj kod, który może generować wyjątki

} catch (typ1 id1) {
   …        // tutaj obsługa wyjątku typu typ1

} catch (typ2 id2) {
   …        // tutaj obsługa wyjątku typu typ2

}
```


Wyjątki

Zamiast:

Jest:

```
try{
    Zadanie1
    Zadanie2
}
catch( )
    Obsługa1
catch ( )
    Obsługa2
```

Wyjątki

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

void przezDziurkeOdKlucza() {
 Podgladanie P;
  for(int i = 0; i < 3; i++)
   cout <<"Nic nie widzę."<< endl;
  throw 47;
}
```

```
int main() {
 try {
  cout << "I co? widzisz coś?..."
   << endl;
  przezDziurkeOdKlucza();
  cout << "..." << endl;
 } catch(int) {
  cout << "To daj też popatrzeć."
   << endl;
 }
}
```

W którym momencie zostanie wykonany destruktor klasy **Podgladanie**?

122

Wyjątki

Obsługa wyjątku

Aby dopasować wyjątek do procedury obsługi, nie musi istnieć dokładna zgodność wyjątku i procedury go obsługującej.

Obiekt (lub referencja do obiektu) klasy pochodnej pasują do procedury dotyczącej klasy bazowej. W przypadku obiektu dochodzi jedynie do „obcięcia" obiektu do typu bazowego – składowe dodane w klasie pochodnej nie są dostępne

Z tego powodu, a także aby uniknąć kopiowania obiektów **lepiej jest używać referencji do obiektów jako argumentu ‚catch'**.

123

Wyjątki

Obsługa wyjątku

Jeżeli ..

w ciele funkcji żaden z typów zadeklarowanych we frazach **`catch`** nie pasuje do rzuconego obiektu-wyjątku

to..

sterowanie przenosi się do miejsca, gdzie funkcja została wywołana i tam szuka sekcji **`try-catch`**.

A jeżeli takiej tam nie ma, lub typy wyjątków we frazach **`catch`** nie pasują, to sterowanie dalej wędruje w górę w hierarchii wywołań funkcji w poszukiwaniu frazy **`catch`** z pasującym typem danych

Sekcje **`try-catch`** mogą być zagnieżdżone odpowiednio do hierarchii wywołań funkcji.

124

Wyjątki

```
class Brzoskwinia {
public:
  class Pestka {};
  class Mala : public Pestka {};
  class Duza : public Pestka {};
  void scisk() { throw Duza(); }
};
int main() {
  Brzoskwinia b;
  try {
    b.scisk();
  } catch(Brzoskwinia::Mala&) { // nigdy nie zostanie wywołany..
    cout << "Mala Pestka schwytana" << endl;
  } catch(Brzoskwinia::Duza&) { // .. a ten tak.
    cout << "Duza Pestka schwytana" << endl;
  } catch(Brzoskwinia::Pestka&) { // ten jest tylko pro forma.
    cout << " Pestka schwytana" << endl;
  };
```


125

Wyjątki

Obsługa wyjątku

Cytat:

An exception in C++ (Java and C# are similar) is a way to put a message in a bottle at some point in a program, abandon ship, and hope that someone is looking for your message somewhere down the call stack.

C++ Cookbook by D. Ryan Stephens; Jeff Cogswell; Jonathan Turkanis; Christopher Diggins

126

Wyjątki

Obsługa wyjątku

Przechwytywanie wyjątków dowolnego typu polega na *nie wskazywaniu* żadnego typu w argumencie frazy 'catch'

```
catch(…) {
   cout << "złapałem, co akurat leciało.." << endl;
}
```

Użycie wielokropka powoduje przechwytywanie wszelkich wyjątków, dlatego taka fraza powinna być ostatnią w sekwencji fraz **`catch`**, aby nie przechwytywać wyjątków przeznaczonych dla fraz, które znalazłyby się za nią.

127

Wyjątki

```
class Brzoskwinia {
public:
  class Pestka {};
  class Mala : public Pestka {};
  class Duza : public Pestka {};
  void scisk() { throw Duza(); }
};
int main() {
  Brzoskwinia b;
  try {
    b.scisk();
  } catch(Brzoskwinia::Mala&) { // nigdy nie zostanie wywołany..
    cout << "Mala Pestka schwytana" << endl;
  } catch(Brzoskwinia::Duza&) { // .. a ten tak.
    cout << "Duza Pestka schwytana" << endl;
  } catch(...) { // ten lapie wszystko.
    cout << "zlapalem, co akurat lecialo.." << endl;
  };
```


128

Wyjątki

Obsługa wyjątku

Może zdarzyć się, że w danej funkcji musimy:

1. obsłużyć błędną sytuację w kontekście lokalnym oraz
2. dokonać przerwania działania całej większej części kodu i
3. dokonać dodatkowej obsługi tej błędnej sytuacji na wyższym poziomie wywołań.

Np., kiedy podając w argumencie frazy 'catch' wielokropek złapaliśmy wyjątek dowolnego typu tylko po to, aby np. zamknąć pewne zasoby, które są kontrolowane przez obiekty lokalne, co jednak nie oznacza, że na tym powinna się zakończyć obsługa tego błędu.

129

Wyjątki

Obsługa wyjątku

W takiej sytuacji potrzebujemy

1. przechwycić wyjątek,
2. zrealizować stosowną obsługę,
3. puścić wyjątek dalej.

Ponowne wyrzucenie wyjątku następuje w chwili wywołania **`throw`** bez żadnych argumentów w procedurze obsługi wyjątku.

130

Wyjątki

Obsługa wyjątku

```
catch(…) {
   cout << ”złapałem, co leciało i na wszelki wypadek zwalniam co
   zająłem..” << endl;
   .. // tu następuje zwolnienie zajętych zasobów

   throw;
}
```

Wszelkie dalsze frazy **`catch`** tego samego bloku **`try`** są ignorowane.

Instrukcja **`throw;`** powoduje, że wyjątek przechodzi do procedur obsługi wyjątków następnego wyższego poziomu, a cały obiekt reprezentujący wyjątek pozostaje niezmieniony aby procedura wyższego poziomu mogła pozyskać z niego wszelkie zawarte w nim informacje

131

Wyjątki

Obsługa wyjątku

Jeżeli żadna fraza **`catch`** nie pasuje do wyjątku, przechodzi on na wyższy poziom sterowania: do funkcji lub bloku **`try`** otaczającego bieżący blok **`try`**, który nie przechwycił wyjątku.

Jeżeli wyjątek nie zostanie przechwycony na żadnym poziomie, automatycznie wywoływana jest specjalna funkcja biblioteczna **`terminate()`**. Domyślnie **`terminate()`** wywołuje funkcję **`abort()`** ze standardowej biblioteki C.

Kiedy wywoływana jest funkcja **`abort()`**, nie są wykonywane żadne działania normalnie uruchamiane przy zamykaniu programu, a więc destruktory obiektów globalnych i statycznych.

132

Wyjątki

Obsługa wyjątku

Funkcję **`terminate()`** można podmienić za pomocą funkcji **`set_terminate()`**. Zdefiniowana przez użytkownika funkcja musi być bezargumentowa i zwracać **`void`**. Nie może rzucać wyjątków:

```
#include<exception>
#include<iostream>
using namespace std;
void zakonczenie( ) {          // nowa funkcja
   cout << "Ja tu wroce.." << endl;
   abort( );
};
int main( ) {
   terminate_handler stary_uchwyt = set_terminate(zakonczenie);
   throw bad_alloc( ); // sygnalizuje niepowodzenie alokacji pamięci
};
```


133

Wyjątki

```
void nowe_zakonczenie( ) {
  cout << "Ja tu wroce.." << endl;
  exit(0);
};
void (*wf)() =
   set_terminate(nowe_zakonczenie);

class Fuszerka {
public:
  class Owoc {};
  void f() {
    cout << "Fuszerka ::f()" << endl;
    throw Owoc();
  };
  Fuszerka() { };
  ~Fuszerka() { };
};
```

```
int main() {
  try {
    Fuszerka b;
    b.f();
  } catch(...) {
    cout << "zlapalem cos.." << endl;
  }
}
```

W oknie terminala:

```
Fuszerka ::f()
zlapalem cos..
```


134

Wyjątki

```
void nowe_zakonczenie( ) {
  cout << "Ja tu wroce.." << endl;
  exit(0);
};
void (*wf)() =
   set_terminate(nowe_zakonczenie);

class Fuszerka {
public:
  class Owoc {};
  void f() {
    cout << "Fuszerka ::f()" << endl;
    throw Owoc();
  };
  Fuszerka() { };
  ~Fuszerka() {throw 'c'; };
};
```

```
int main() {
  try {
    Fuszerka b;
    b.f();
  } catch(...) {
    cout << "zlapalem cos.." << endl;
  }
}
```

W oknie terminala:

```
Fuszerka ::f()
Ja tu wroce..
```


135

Wyjątki

Obsługa wyjątku

Kiedy w trakcie wykonywania funkcji zostanie rzucony wyjątek, może dla pewnego zbioru zmiennych nastąpić wyjście sterowania z zasięgu, w którym zostały zadeklarowane (np. wyjście z funkcji, w której były zadeklarowane zmienne lokalne).

Obsługa wyjątków w C++ gwarantuje, że przy wychodzeniu z zasięgu dla wszystkich zmiennych tego zasięgu, których konstruktory zostały wywołane do końca, zostaną też wywołane destruktory.

136

Wyjątki

```
class Trace {
  static int counter;
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "konstruktor Trace #" <<
      objid << endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destruktor Trace #" <<
      objid << endl;
  }
};
```

```
int Trace::counter = 0;
int main() {
  try {
    Trace DynamicznyStos[5];
  } catch(int i) {
    cout << "przechwycono " << i <<
    endl;
  }
}
```

W oknie terminala:

```
konstruktor Trace #0
konstruktor Trace #1
konstruktor Trace #2
konstruktor Trace #3
destruktor Trace #2
destruktor Trace #1
destruktor Trace #0
przechwycono 3
```


137

Wyjątki

Podstawowe pytanie, które pojawia się przy okazji używania wyjątków:

jaka jest poprawna kolejność zwalniania zajętych zasobów?

a dokładniej – kolejność w sytuacji, kiedy wyjątek został rzucony w konstruktorze, już po zaalokowaniu przez niego pewnych zasobów, zwłaszcza jeżeli są one kontrolowane wyłącznie przez zmienne wskaźnikowe.

138

Wyjątki

RAII (Resource Acquisition Is Initialization)

Technika programowania, która pozwala na lepszą kontrolę alokowania zasobów w sytuacji ewentualnego wystąpienia wyjątku, który mógłby przerwać alokację i doprowadzić do niezwolnienia tej części zasobów, która już została zaalokowana

Jest to jeden z tzw. *wzorców projektowych*, tj. uniwersalnych, sprawdzonych rozwiązań programistycznych.

139

Wyjątki

RAII (Resource Acquisition Is Initialization)

1. polega na przypisaniu
   czynności alokowania zasobów
   do
   czynności tworzenia zmiennej lokalnej, tj. działaniu konstruktora dla tego typu zmiennej lokalnej.

2. opiera się na właściwości C++ gwarantującej, że przy wychodzeniu z określonego zasięgu kodu z powodu rzucenia wyjątku, dla wszystkich zmiennych tego zasięgu, których konstruktory zostały wywołane do końca, zostaną też wywołane destruktory.

Wyjątki

RAII (Resource Acquisition Is Initialization)

Każdą **czynność alokowania zasobów** należy zamknąć w czynności tworzenia lokalnego obiektu pewnego typu (tj.w konstruktorze tego typu),

natomiast

każdą **czynność zwalniania** – w destruktorze tego typu.

Jeżeli będziemy mieli do zaalokowania kilka zasobów, to alokując utworzymy kilka zmiennych lokalnych, z których pomocą będziemy odwoływać się do tych zasobów.

Jeżeli tworzenie jednej z tych zmiennych nie powiedzie się, te które już zostały utworzone automatycznie zostaną usunięte w poprawny sposób, tj. z uwzględnieniem wykonania ich destruktora.

Wyjątki

Jeżeli nie stosujemy **RAII**, musimy w trakcie działania programu troszczyć się o wszystkie zasoby, które ten program w trakcie swego działania kolejno alokował, np. korzystał z pliku:

```
{
   FILE* f = fopen("name", "r");
   // ...

   if (warunek_spelniony) {
     fclose(f);
     return cos_tam;
   }
   // ...

   fclose(f);
   return cos_tam_cos_tam;
}
```


Wyjątki

Zastosowanie **RAII**

```
class FILE_handle
{
  FILE* f;
  FILE_handle(const FILE_handle&);
  void operator=(const FILE_handle&);

public:
  explicit FILE_handle(FILE* file)
      : f(file) {};
  ~FILE_handle() {
    if (f)
      fclose(f);
  };
  operator FILE*() {
    return f;
  };
};
```

```
int main()
{
  FILE_handle fih(fopen("test.txt", "w"));
  fprintf(fih,"Dane testowe\n");
  return 0;
}
/*  W kodzie powyżej nie ma specjalnych
linijek kodu odpowiedzialnych za
zwalnianie zaalokowanych zasobów (!)
*/
```


Wyjątki

Zastosowanie **RAII** przy alokacji i dealokacji pamięci

```
class mem {
  char * pdata;
public:
  mem(unsigned int size=0);
  ~mem() { delete [] pdata; }
  void alloc(unsigned int size);
  char* getbuffer() const {
       return pdata;
    };
};
```

```
mem::mem(unsigned int size){
  if (size)
    pdata = new char[size];
  else
    pdata=0;
};
void mem::alloc(unsigned int size){
  if (pdata)
    return;
  pdata = new char[size];
};

int main() {
  mem bufor(100);
  strcpy(bufor.getbuffer(),
                "Taki sobie napis");
  cout << bufor.getbuffer()<< endl;
  return 0;
}
```

Visual Studio

Wyjątki

Standardowe typy wyjątków

W bibliotece standardowej **<stdexcept>** zdefiniowane są klasy reprezentujące różne typy wyjątków.

Szybciej i łatwiej jest skorzystać z gotowej klasy, a jeżeli nie do końca spełnia oczekiwania – użyć jej jako bazowej:

- **exception** – bazowa dla wszystkich wyjątków wyrzucanych z biblioteki standardowej C++
- **logic_error** – zgłasza błędy w warstwie logicznej programu, które prawdopodobnie mogą być wykryte przez uważne przejrzenie kodu
- **runtime_error** – zgłasza błędy wykonania, które mogą być wykryte właściwie tylko podczas działania programu

Wyjątki

exception

```
class exception {
public:
    exception();
    exception(const char * const &message); // [*]
    exception(const char * const &message, int); // [*]
    exception(const exception &right);
    exception& operator=(const exception &right);
    virtual ~exception();
    virtual const char *what() const;
};
```

[*] Rozszerzenia Standard C++ Library dodane przez Microsoft

146

Wyjątki

logic_error

```
class logic_error : public exception {
public:
    explicit logic_error(const string& message);
    virtual const char *what();
};
```

Przykład:

```
try
{
   throw logic_error( "Blad logiczny - walidacja kodu" );
}
catch ( exception &e )
{
   cerr << "Zlapalem: " << e.what( ) << endl;
   cerr << "Typ: " << typeid( e ).name( ) << endl;
};
```


147

Wyjątki

runtime_error

```
class runtime_error : public exception {
public:
    explicit runtime_error(const string& message);
    virtual const char *what();
};
```

Przykład:

```
try
{
   locale loc( "test" ); // dane lokalne dla kraju „test"
}
catch ( exception &e )
{
   cerr << "Zlapalem: " << e.what( ) << endl;
   cerr << "Typ: " << typeid( e ).name( ) << endl;
};
```


148

Wyjątki

Standardowe typy wyjątków

Pozostałe typy zdefiniowane w bibliotece standardowej **<stdexcept>**

- **invalid_argument** – zgłasza błędy powstałe w wyniku niezaakceptowania wartości argumentu
- **domain_error** – zgłasza błędy związane z wykroczeniem wartości na wejściu poza dozwoloną dziedzinę (zakres)
- **length_error** – zgłasza błędy związane z wykroczeniem poza dozwolony zakres założony w implementacji danego obiektu
- **out_of_range** – zgłasza błędy związane z próbą dostępu do elementów zasobu poza ich dozwolonym zakresem
- **range_error** – zgłasza błędy związane z wykroczeniem wartości poza zakres definiowany przez typ danej, mającej przechowywać ową wartość.
- **overflow_error** – zgłasza błędy typu *arithmetic overflow*
- **underflow_error** – zgłasza błędy typu *arithmetic underflow*

149

Wyjątki

Specyfikacja wyjątków

Definiując funkcję, która może powodować rzucenie wyjątku, autor powinien – ale nie musi – poinformować czy funkcja może wyjść przez rzucenie wyjątku. Tym sposobem informuje użytkownika funkcji czego może się spodziewać i jak pisać kod obsługi wyjątków.

Specyfikację wyjątków deklaruje się tuż za nagłówkiem funkcji.

Jeżeli ich nie poda, użytkownik/programista sam je sobie może znaleźć sprawdzając wystąpienia słowa **throw** w kodzie. Ale..

ponieważ użytkownikowi dostarczane są często tylko pliki nagłówkowe i skompilowane biblioteki, nieinformowanie go w takiej sytuacji o tym, że jakaś funkcja może rzucać wyjątki, świadczy o braku manier.

150

Wyjątki

Specyfikacja wyjątków

Specyfikacja rzucanych wyjątków korzysta ze słowa **throw**:

1. Funkcja nie zgłasza wyjątku:

```
void fun() throw();
void fun() noexcept;              // od C++11
void fun() noexcept(true);        // od C++11
```

2. Funkcja może zgłosić wyjątek dowolnego typu:

```
void fun() throw(...);
void fun() noexcept(false);
```

3. Funkcja może zgłosić wyjątek określonego typu (tylko w C++14 i starsze, od C++17 - zabronione):

```
void fun() throw (MyException, MyVeryBadException);
```


151

Wyjątki

Specyfikacja wyjątków

Jeżeli programista się pomyli (albo wręcz przeciwnie, jeżeli zrobi tak specjalnie) i funkcja rzuci wyjątek, którego nie zadeklarował, wywoływana jest funkcja **`unexpected()`**

152

Wyjątki

Specyfikacja wyjątków

Domyślnie funkcja **`unexpected()`** wywołuje **`terminate()`**

Możemy podmienić funkcję **`unexpected()`** za pomocą funkcji **`set_unexpected()`**:

```
#include <exception>
void nieoczekiwanie( )
{
   cout << „Ale zaskoczenie.." << endl;
   terminate( );
}

int main( )
{
   unexpected_handler stary_unex = set_unexpected( nieoczekiwanie );
   unexpected( );
}
```


153

Wyjątki

Różne rady:

Kiedy unikać wyjątków?

1. Jeżeli błąd jest przewidziany w programie i w danym miejscu kodu dysponujemy dostateczną informacją, aby obsłużyć błąd, to należy go obsłużyć, a nie rzucać wyjątek i przenosić sterowanie do innego miejsca.
2. Wyjątki to nie jest sposób na sterowanie przepływem sterowania w programie. Tak jak nie należy używać instrukcji **`goto`**, tak samo nie należy budować konstrukcji rzucania wyjątku w jednym miejscu kodu i przechwytywania w innym w sytuacjach gdzie nie ma żadnego błędu, a nam tylko zachciało się, aby wykonały się instrukcje kodu z innego fragmentu programu.
3. W prostych programach – nie ma po co wprowadzać wyjątków.
4. W starym, dobrze działającym kodzie.

154

Wyjątki

Różne rady:

Wyjątki należy używać w celu:

1. rozwiązania problemu i ponownego wywołania funkcji, która wyjątek spowodowała,
2. uporządkowania środowiska i dalszego wykonywania programu bez ponawiania wywołania funkcji, która wyjątek spowodowała,
3. zrobienia w aktualnym kontekście wszystkiego, co jest możliwe i konieczne, i ponownego wyrzucenia *tego samego* wyjątku,
4. zrobienia w aktualnym kontekście wszystkiego, co jest możliwe i konieczne, i ponownego wyrzucenia *innego* wyjątku,
5. zakończenia programu,

155

Wyjątki

Różne rady:

Wyjątki należy używać w celu (*c.d. z poprzedniego slajdu*):

6. opakowania starej funkcji w C wykorzystującej klasyczną obsługę błędów, tak aby wyrzucała wyjątki,
7. uproszczenia (*bo jeżeli użycie wyjątków prowadzi do skomplikowania kodu to jaki to ma sens..*),
8. uczynienia bibliotek i programów bezpieczniejszymi (*w krótkiej perspektywie pomaga szybciej znaleźć błędy, a w dłuższej – zwiększa niezawodność aplikacji*).

156

Wyjątki

Różne rady:

Pisząc program wykorzystujący wyjątki, lepiej jest zawsze zaczynać od sprawdzenia istniejących wyjątków standardowych – jeżeli wystarczą, nie należy tworzyć nowych.

Jeżeli nie ma dobrych wyjątków w bibliotece, należy tworzyć własne wyjątki, jednak wtedy najlepiej aby dziedziczyły one po wyjątkach standardowych (*Wszystko powinno się upraszczać o ile to możliwe, ale nie bardziej.* *Albert Einstein*)

Jeżeli tworzysz wyjątki do użycia w pewnej klasie, dobrym pomysłem jest zagnieżdżanie klas tych wyjątków albo w danej klasie albo przynajmniej w danej przestrzeni nazw. W ten sposób unikasz zaśmiecania globalnej przestrzeni nazw i wiążesz klasy wyjątków z klasami, w których mogą wystąpić.

157

Wyjątki

Różne rady:

Hierarchia wyjątków to doskonały sposób na klasyfikowanie błędów krytycznych, na jakie można natknąć się w danej klasie lub bibliotece.

Wyjątki należy przechwytywać przez referencję, a nie przez wartość z dwóch powodów:

1. aby uniknąć zbędnego kopiowania obiektu wyjątku przy przekazywaniu do procedury obsługi
2. aby uniknąć utraty informacji podczas przechwytywania wyjątku klasy pochodnej jako obiektu klasy bazowej

158

Wyjątki

Różne rady:

W konstruktorach należy szczególnie śmiało rzucać wyjątki, ponieważ to jedyny sposób aktywnego zasygnalizowania błędu.

Konstruktory nie zwracają żadnej wartości i dlatego dawniej istniały tylko dwie metody zgłoszenia błędu w trakcie tworzenia obiektu:

1. ustawienie nielokalnej flagi i liczenie na to, że użytkownika coś tknie i sprawdzi jej wartość,
2. zwrócenie częściowo utworzonego obiektu i liczenie na to, że użytkownika coś tknie i sprawdzi poprawność obiektu.

159

Wyjątki

Różne rady:

W destruktorach nie należy rzucać wyjątków, ponieważ one często mogą być wywoływane w wyniku wcześniejszego wyrzucenia innego wyjątku. Takie kaskadowe wyrzucanie może doprowadzić do wykonania metody `terminate()`, mimo iż nie było to konieczne.

160