Maciej Sobieraj

Wykład 7

Outline

1. **Wyjątki**
    1. **Błądzić jest rzeczą ludzką**
    2. Wyrażenie throw
    3. Kategoryzowanie wyjątków

Jak wpaść w kłopoty?

- Kod jest przerywany, ponieważ dzielenie przez zero nie jest możliwe w świecie liczb rzeczywistych.

```
#include <iostream>

using namespace std;

int main(void) {
	float a, b;
	cin >> a;
	cin >> b;
	cout << a / b << endl;
	return 0;
}
```

Jak wpaść w kłopoty?

- Istnieje jeden wyjątek - **jeśli w strumieniu wejściowym nie ma danych lub dane są nieprawidłowe, strumień cin zwraca wartość zerową**, która w kontekście boolowskim ma wartość false.

```
#include <iostream>

using namespace std;

int main(void) {
	float a, b;
	while(cin >> a) {
		cin >> b;
		cout << a / b << endl;
	}
	return 0;
}
```

Jak wpaść w kłopoty?

```
#include <iostream>

using namespace std;

int main(void) {
	float a, b;
	while(cin >> a) {
		cin >> b;
		if(b != 0.0)
			cout << a / b << endl;
		else
			cout << "Are you kidding me?" << endl;
	}
	return 0;
}
```

Jak wpaść w kłopoty?

- Pomysł jest prosty: **gdy funkcja odkryje, że istnieje problem** z argumentami lub wynikami pośrednimi, **natychmiast wychodzi, zwracając w wyniku fałsz**.

```
#include <iostream>
using namespace std;
bool div(float &res, float arg1, float arg2) {
	if(arg2 == 0.0)
		return false;
	res = arg1 / arg2;
	return true;
}
int main(void) {
	float r, a, b;
	while(cin >> a) {
		cin >> b;
		if(div(r,a,b))
			cout << r << endl;
		else
			cout << "Are you kidding me?" << endl;
	}
	return 0;
}
```

Jak wpaść w kłopoty?

- Spróbuj sobie wyobrazić, że nasza (bezpieczna) funkcja **jest wywoływana wiele razy przez inne funkcje**.
- Zauważ, że łańcuch wywoływanych funkcji może być bardzo długi. Jeśli za reagowanie na błędy występujące na niższych poziomach odpowiedzialne są tylko funkcje najwyższego poziomu, **może to spowodować**, **że kod „puchnie”**.
- Łańcuch zawiera kod, który nic nie robi, ale wykrywa błędy i próbuje je obsłużyć.

Jak wpaść w kłopoty?

```
bool low_level_eval(...) {
        :
        if(something_went_wrong) return false;
        :
}
bool middle_level_eval(...){
        :
        bool result = low_level_eval(...);
        if(!result) return false;
        :
}
bool top_level_eval(...){
        :
        bool result = middle_level_eval(...);
        if(!result) return false;
        :
}
int main(void){
        :
        bool result = top_level_eval(...);
        if(!result) {
                cout << "Sarcasm!" << endl;
                return 1;
        }
}
```

Jak wpaść w kłopoty?

- Wyjątkiem są dane.
- Wyobraź sobie wyjątek jako skrzydlate pudełko, zdolne do latania, które pojawia się, gdy zdarza się coś złego.
- Pudełko zawiera dane, które mogą pomóc w zidentyfikowaniu przyczyny niepowodzenia.
- Dane mogą być dowolnego typu: może to być int, float, string, obiekt dowolnej klasy.

Jak wpaść w kłopoty?

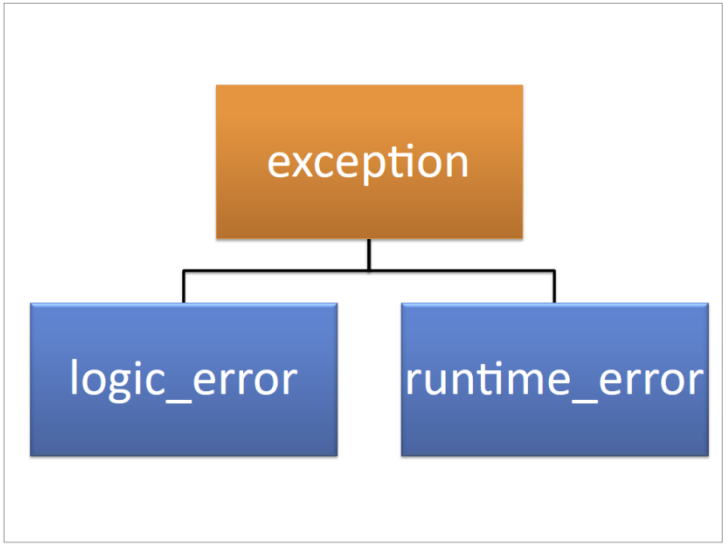
- **Część kodu, która może powodować problemy, musi być oznaczona** (faktycznie zagnieżdżona) w specjalnym rodzaju bloku. Blok ma być uważnie obserwowany podczas jego wykonywania.
- **Kiedy pojawia się wyjątek, wykonanie bloku jest przerywane**, ale sam program wciąż działa.
- **Wyjątek jest przechwytywany przez inną część kodu.**

Jak wpaść w kłopoty?

- Najwyższa klasa, o nazwie *exception*, jest podstawą wszystkich wyjątków. Klasa jest używana tylko do definiowania niektórych zachowań i właściwości wspólnych dla wszystkich wyjątków
- Dwie różne klasy pochodzą z klasy *exception*.
  - Pierwszy, *logic_error*, ma reprezentować wyjątki związane z logiką programu, tj. Algorytm, jego implementację, poprawność danych i spójność.
  - Druga klasa, o nazwie *runtime_error*, służy do identyfikowania wyjątków zgłaszanych z powodu „nieoczekiwanych” wypadków, takich jak brak pamięci.

Jak wpaść w kłopoty?

- Wszystkie te elementy są zdefiniowane w pliku nagłówkowym *exception*. Oznacza to, że linia
  - #include <exception>
- może być potrzebna w kodzie, który korzysta z którejkolwiek z poniższych klas.



Anatomia obiektu wyjątku

- Klasa wyjątków jest bardzo skromna. W rzeczywistości definiuje tylko trzy komponenty:
  - konstruktor (niezbyt przydatny dla nas, ponieważ, jak wspomnieliśmy wcześniej, obiekty tej klasy nie są tworzone
  - destruktor wirtualny, pierwotnie pusty
  - funkcja wirtualna o nazwie what, która zwraca łańcuch w stylu C (wskaźnik do tablicy znaków zakończonych znakiem pustym („0”))

```
virtual char* what()
```

Gdzie są rzucane wyjątki?

- Jeśli chcesz wykryć wyjątki, musisz zaznaczyć część kodu, w której mogą wystąpić wyjątki.
- Robisz to za pomocą instrukcji „try”.

```
try {
        :
        :
        :
}
```

Gdzie są przechwytywane wyjątki?

- Jeśli chcesz złapać „latające” wyjątki, musisz umieścić instrukcję ***catch*** bezpośrednio po instrukcji ***try***.

```
catch(what) {
        :
        :
        :
}
```

Gdzie są przechwytywane wyjątki?

- Jeśli napiszesz to w ten sposób:
  - catch(string &anyproblem) { … }
- to znaczy: *chcę przechwycić wyjątki, które zwracają ciąg znaków (string)*.
- Forma:
  - catch(exception &otherproblem) { … }
- oznacza: chcę przechwycić wyjątki przenoszące obiekty klasy *exception* lub innych klas wywodzących się z klasy *exception*.

Jak rzucane są wyjątki?

- Jeśli chcesz rzucić wyjątek, musisz użyć wyrażenia *throw*. Instrukcja *throw* wymaga danych, które zostaną „zapakowane” w wyjątek przed jego rzuceniem.

```
throw what;
```

Jak rzucane są wyjątki?

- Jeśli zamierzasz rzucić wartość *int*, napiszesz coś takiego:
  - throw 997;
- Jeśli chcesz rzucić wyjątek wyposażony w obiekt dowolnej klasy, musisz określić konstruktor, który ma zostać wywołany, aby przygotować dane:
  - throw string("Bye world!");

Jak rzucane są wyjątki?

- Funkcja zwróci wynik, jeśli argumenty są poprawne, w przeciwnym razie zostanie zgłoszony wyjątek zawierający następujący ciąg

```
float div(float a, float b) {
    if(b == 0.0)
        throw string("I can't believe - division by zero :(");
    return a / b;
}
```

Jak rzucane są wyjątki?

```
#include <iostream>
using namespace std;
float div(float a, float b) {
	if(b == 0.0)
		throw string("division by zero :(");
	return a/b;
}
int main(void) {
	float a, b;
	while(cin >> a) {
		try {
			cin >> b;
			cout << div(a, b) << endl;
		} catch (string &problem) {
			cout << "Look what you did, you bad user!" << endl;
			cout << problem << endl;
		}
	}
	return 0;
}
```

Jak rzucane są wyjątki?

- Program zostanie przerwany z komunikatem informującym, że wystąpił przypadek nieobsługiwanego wyjątku.

```
#include <iostream>
using namespace std;
float div(float a, float b) {
	if(b == 0.0)
		throw string("division by zero :(");
	return a/b;
}
int main(void) {
	float a, b;
	while(cin >> a) {
		try {
			cin >> b;
			cout << div(a, b) << endl;
		} catch (int problem) {
			cout << "Look what you did, you bad user!" << endl;
			cout << problem << endl;
		}
	}
	return 0;
}
```

Outline

1. **Wyjątki**
   1. Błądzić jest rzeczą ludzką
   2. **Wyrażenie throw**
   3. Kategoryzowanie wyjątków

throw i catch razem

```
#include <iostream>
using namespace std;
float DoCalculations(float a, float b, float c, float d) {
   try {
      float x;
      if(a == 0.0)
            throw string("Bad arg a");
      x = 1 / a;
      if(b == 0.0)
            throw string("Bad arg b");
      x /= b;
      if(c == 0.0)
            throw string("Bad arg c");
      x /= c;
      if(d == 0.0)
            throw string("Bad arg d");
      return x / d;
   } catch(string &exc) {
      cout << "Something bad happened: " << exc << endl;
      return 0;
   }
}
int main(void) {
      DoCalculations(1,2,3,0);
      return 0;
}
```

throw i catch razem

- Przykładowy program wyświetli następujący tekst na ekranie:
  - Something bad happened: Bad arg d

throw i catch razem

- Specyfikacja wyjątku umieszczona w nagłówku gałęzi catch:
  - **catch**(string &exc)
- **działa jak lokalna (automatyczna) deklaracja zmiennej**.

throw i catch razem

- Wewnątrz poniższego fragmentu kodu:

```
int main(void) {
string str;
        try {
                throw string("1");
        } catch(string &str) {
                cout << str;
        }
        return 0;
}
```

- istnieją dwie różne zmienne o nazwie str (pierwsza jest ukryta przez drugą w bloku ca

throw i catch oddzielnie

- Jak wiesz, *throw* i *catch* mogą również żyć oddzielnie. Możemy wrzucić jedną funkcję, złapać inną, a mechanizm będzie nadal działał efektywnie, ale oczywiście tylko wtedy, gdy funkcje wywołają się we właściwej kolejności.
- Oznacza to, że **obiekt wyjątku może przemieszczać się pomiędzy granicami funkcji** i może nawet pominąć więcej niż jedną funkcję, aby znaleźć swoją własną instrukcję **catch**.

throw i catch oddzielnie

```
#include <iostream>
using namespace std;
float DoCalculations(float a, float b, float c, float d) {
      float x;
      if(a == 0.0)
            throw string("Bad arg a");
      x = 1 / a;
      if(b == 0.0)
            throw string("Bad arg b");
      x /= b;
      if(c == 0.0)
            throw string("Bad arg c");
      x /= c;
      if(d == 0.0)
            throw string("Bad arg d");
      return x / d;
}
int main(void) {
  try {
      DoCalculations(1,2,3,0);
  } catch(string &exc) {
      cout << "Something bad happened: " << exc << endl;
  }
  return 0;
}
```

throw vs. epilog funkcji

- Wykonywanie funkcji składa się zasadniczo z trzech faz:
  - **prolog** (gdy wszystkie automatyczne zmienne / obiekty są tworzone),
  - **wykonanie** (gdy wykonywany jest kod funkcji) i
  - **epilog** (gdy uprzednio utworzone obiekty zostaną zniszczone).

throw vs. epilog funkcji

```
#include <iostream>
using namespace std;
class Class {
public:
      Class(void)  { cout << "Object constructed" << endl; }
      ~Class(void) { cout << "Object destructed" << endl; }
      void Hello(void) { cout << "Object says: hello" << endl; }
};

float DoCalculations(void) {
      Class object;
      object.Hello();
      return 0.0;
}

int main(void) {
  DoCalculations();
  return 0;
}
```

throw vs. epilog funkcji

- Program wygeneruje następujące dane wyjściowe:
  - Object constructed
  - Object says: hello
  - Object destructed

throw vs. epilog funkcji

- Dodaliśmy trzy instrukcje *throw* w ramach funkcji *DoCalculations*.
- Definicja klasy *Class* pozostaje bez zmian.
- Funkcja *main* function trzy razy wywoła *DoCalculations*, dzięki temu będziemy mogli obserwować zachowanie funkcji.

throw vs. epilog funkcji

```
include <iostream>
using namespace std;
class Class {
public:
      Class(void)  { cout << "Object constructed" << endl; }
      ~Class(void) { cout << "Object destructed" << endl; }
      void Hello(void) { cout << "Object says: hello" << endl; }
};
void DoCalculations(int i) {
      if(i == 0)
            throw string("fatal 1");
      Class object;
      if(i == 1)
            throw string("fatal 2");
      object.Hello();
      if(i == 2)
            throw string("fatal 3");
}
int main(void) {
   for(int i = 0; i < 3; i++) {
      try {
        cout << "-------" << endl;
        DoCalculations(i);
      } catch (string &exc) {
        cout << exc << endl;
      }
   }
   return 0;
}
```

throw vs. epilog funkcji

- Program wyświetla następujący tekst:

```
-------
fatal 1
-------
Object constructed
Object destructed
fatal 2
-------
Object constructed
Object says: hello
Object destructed
fatal 3
```

throw i obiekty, które rzuca

- Instrukcja ***throw*** jest **zobowiązana do rzucenia wartości** np. obiektu
- *throw* może rzucić **dowolny obiekt dowolnej dostępnej klasy**

throw i obiekty, które rzuca

```
#include <iostream>
using namespace std;
class Class {
public:
      string msg;
      Class(string txt):msg(txt){cout<<"Object [" << msg << "] constructed" << endl; }
      ~Class(void) { cout << "Object [" << msg << "] destructed" << endl; }
      void Hello(void) { cout << "Object [" << msg << "] says: hello" << endl; }
};
void DoCalculations(int i) {
      if(i == 0)
            throw Class("exception 1");
      Class object("object");
      if(i == 1)
            throw Class("exception 2");
      object.Hello();
      if(i == 2)
            throw Class("exception 3");
}
int main(void) {
   for(int i = 0; i < 3; i++) {
      try {
        cout << "-------" << endl;
        DoCalculations(i);
      } catch (Class &exc) {
        cout << "Caught!" << endl;
        cout << exc.msg << endl;
      }
   }
   return 0;
}
```

throw i obiekty, które rzuca

- Bądź świadomy, że wykonanie takiej linii:
  - **throw** Class("exception 1");
- spowoduje **utworzenie nowego obiektu** klasy Class.
- Oznacza to, że **odpowiedni konstruktor zostanie wywołany**, zanim funkcja zakończy swoje życie.

throw i obiekty, które rzuca

- Ten program generuje następujące dane wyjściowe:

```
-------
Object [exception 1] constructed
Caught!
exception 1
Object [exception 1] destructed
-------
Object [object] constructed
Object [exception 2] constructed
Object [object] destructed
Caught!
exception 2
Object [exception 2] destructed
-------
Object [object] constructed
Object [object] says: hello
Object [exception 3] constructed
Object [object] destructed
Caught!
exception 3
Object [exception 3] destructed
```

Jak dowiedzieć się, że wyjątek był rzucony

- Jak możemy się dowiedzieć, czy funkcja zgłasza jakieś wyjątki, czy nie?
- Warto rozważyć dwa ważne argumenty:
  - Funkcja może być bardzo długa i bardzo złożona - jej odczytanie może być czasochłonne i możesz przeoczyć niektóre instrukcje *throw*
  - Kod źródłowy funkcji może być niedostępny - może się to zdarzyć, gdy użyjesz gotowej biblioteki napisanej przez innych autorów lub gdy skompilujesz (binarne) pliki zawierające tylko kod wykonywalny i pliki nagłówkowe określające nagłówki funkcji, bez jej ciała.

Jak dowiedzieć się, że wyjątek był rzucony

```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(int i) {
        throw Class("object");
}
int main(void) {
   try {
        function(1);
   } catch(Class &exc) {
        cout << "Caught!" << endl;
   }
   return 0;
}
```

throw i jego specyfikacja

- **Funkcja, która rzuca wyjątek, może (ale nie musi) określać typy rzucanych elementów.**
- **Może być użyty nawet wtedy, gdy ciało funkcji jest niedostępne lub ukryte**.
- Pozwala programiście ogłosić wszystkie wyjątki, które mogą opuścić tę funkcję, a tym samym przygotować innych programistów na zdarzenia, które mogą się zdarzyć podczas wykonywania funkcji.

throw i jego specyfikacja

- Istnieje więcej niż jedna forma specyfikacji - najprostsza wygląda tak:
  - throw(x)
- Oznacza to, że funkcja rzuca jeden rodzaj wyjątku, na przykład typu x:
  - void function(void) throw(string);

throw i jego specyfikacja

- Bardziej złożona forma wygląda tak:
  - throw(x1,x2,..,xn)
- Oznacza to, że **funkcja rzuca na przykład n różnych wyjątków** typów x1, x2,…, xn.
  - int doit(int i) throw(int, string, Class);
- Ta funkcja może rzucać wyjątki typu *int*, *string* oraz *Class*.

throw i jego specyfikacja

- Ostatnia forma wygląda tak :
  - throw()
- i oznacza, że „**funkcja nie generuje żadnych wyjątków**”.

throw i jego specyfikacja

```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
        throw Class("object");
}
int main(void) {
    try {
        function();
    } catch(Class &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

throw i jego specyfikacja

```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
        throw string("object");
}
int main(void) {
   try {
        function();
   } catch(Class &exc) {
        cout << "Caught!" << endl;
   }
   return 0;
}
```

throw i jego specyfikacja

- Funkcja określa, że **rzuca wyjątki typu *Class***, chociaż **faktycznie rzuca typ *string***
- What'll happen then?
- Kompilacja? **OK** – brak problemów, błędów oraz ostrzeżeń.
- Wykonanie? Pojawia się problem– program został **przerwany** i pojawił się komunikat. **Nieprzechwycony wyjątek typu '*std::string*'**

throw i jego specyfikacja

- Niestety korekta w ogóle nie poprawiła naszego problemu.

```
#include <iostream>
using namespace std;
class Class {
public:
      string msg;
      Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
      throw string("object");
}
int main(void) {
   try {
      function();
   } catch(string &exc) {
      cout << "Caught!" << endl;
   }
   return 0;
}
```

throw i jego specyfikacja

```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (string) {
        throw string("object");
}
int main(void) {
    try {
        function();
    } catch(string &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

throw i jego specyfikacja

- Czy wyniki są zgodne z oczekiwaniami?

```
#include <iostream>
using namespace std;
class Class {
public:
      string msg;
      Class(string txt) : msg(txt) {}
};
void function(void) throw () {
      throw string("object");
}
int main(void) {
   try {
      function();
   } catch(string &exc) {
      cout << "Caught!" << endl;
   }
   return 0;
}
```

  - Warning: function assumed not to throw an exception but does

throw i jego specyfikacja

- **obsługa wyjątków może być dystrybuowana między różne części programu**.
- Możesz obsługiwać wyjątki w najbardziej odpowiednich miejscach i nie musisz zbierać wszystkich połowów w jednej funkcji lub module.

throw i jego specyfikacja

```
#include <iostream>
using namespace std;
class Class {
public:
      string msg;
      Class(string txt) : msg(txt) {}
};
void function(int i) throw (string,Class) {
   switch(i) {
      case 0 : throw string("string");
      case 1 : throw Class("object");
      default: cout << "OK" << endl;
   }
}
void level(int i) throw(Class) {
   try {
      function(i);
   } catch(string &exc) {
      cout << "String [" << exc << "] caught in level()" << endl;
   }
}
int main(void) {
   for(int i = 0; i < 2; i++) {
      cout << "-------" << endl;
      try {
         level(i);
      } catch(Class &exc) {
         cout << "Object [" << exc.msg << "] caught in main()" << endl;
      }
   }
   return 0;
}
```

Obsługa nieoczekiwanych wyjątków

- **Jeśli pojawi się jakiś nieoczekiwany wyjątek, wywoływana jest specjalna funkcja środowiska wykonawczego**.
- Jest to funkcja, która **kończy program i wyświetla komunikat diagnostyczny** *unexpected()*.

Obsługa nieoczekiwanych wyjątków

- Jeśli naprawdę musisz coś zrobić podczas ostatnich oddechów programu, powinieneś:
  - utwórz funkcję bez parametrów typu *void*
  - wywołaj funkcję o nazwie *set_unexpected*, przekazując jej nazwę swojej funkcji

Obsługa nieoczekiwanych wyjątków

```
#include <iostream>
#include <string>

using namespace std;
class Class {
public:
      string msg;
      Class(string txt) : msg(txt) {}
};
void function(void) throw () {
      throw string("object");
}
void lastchance(void) {
      cout << "See what you've done! You've thrown an illegal exception!" << endl;
}
int main(void) {
   set_unexpected(lastchance);
   try {
      function();
   } catch(string &exc) {
      cout << "Caught!" << endl;
   }
   return 0;
}
```

Outline

1. **Wyjątki**
    1. Błądzić jest rzeczą ludzką
    2. Wyrażenie throw
    3. **Kategoryzowanie wyjątków**

Słowo kluczowe ‘explicit’

- Słowo kluczowe *explicit* może być umieszczone przed deklaracją konstruktora klasy.
- Chroni konstruktor przed użyciem w dowolnym kontekście wymagającym użycia niejawnych konwersji.

Słowo kluczowe ‘explicit’

```
class A {
public:
  explicit A(int) {}
};

class B {
public:
  B(int) {}
};
int main(void) {
      A a = 1;  // compilation error!
      B b = 1;
      return 0;
}
```

Słowo kluczowe ‘explicit’

- Zauważ, że następująca funkcja również byłaby błędna:
  - A fun(**void**) { **return** 0; }
- podczas gdy ta nie byłaby:
  - B fun(**void**) { **return** 0; }

Klasa ‘exception’

- Klasa *exception* class jest **bazą** (lub rootem) dla wszystkich innych predefiniowanych wyjątków.
- Zawiera funkcję o nazwie *what*, która ma na celu dostarczenie wskaźnika do ciągu w tak zwanym stylu „C” (sekwencja znaków zakończona znakiem pustym) **opisującego naturę wyjątku**.

Klasa ‘exception’

```
#include <iostream>
#include <exception>
using namespace std;

class A {
public:
  virtual void f(void) {}
};

class AA : public A {
public:
  void aa(void) {};
};

int main(void) {
      A a;
      try {
            dynamic_cast<AA &>(a).aa();
      } catch (exception ex) {
            cout << "[" << ex.what() << "]" << endl;
      }
      return 0;
}
```

- Otrzymamy “[Bad dynamic_cast!]

Klasa ‘logic_error’

- exception ← logic_error
- Klasa *logic_error* pochodzi bezpośrednio z klasy *exception*.
- Ma na celu reprezentowanie wszystkich wyjątków spowodowanych **naruszeniem zasad narzuconych przez logikę algorytmu / programu**.
- Może to (ale nie zawsze) oznaczać, że wyjątkom tego rodzaju można **zapobiec**, tj. Nie zdarzą się jeśli wszystkie przetworzone dane są prawidłowe.

Klasa ‘logic_error’

- Konstruktor klasy pozwala nam „spakować” szczegółowy komunikat wewnątrz obiektu wyjątku.
- Poniższa dyrektywa jest **obowiązkowa** w kodzie, który korzysta z tych klas:
  - #include <stdexcept>

```
class logic_error : public exception {
public:
  explicit logic_error (const string& what_arg);
}
```

Klasa ‘domain_error’

- exception ← logic_error ← domain_error
- Klasa domain_error pochodzi z klasy logic_error. Ma na celu reprezentowanie wszystkich wyjątków spowodowanych przez **dane przekraczające dopuszczalny zakres**.

```
class domain_error : public logic_error {
public:
  explicit domain_error (const string& what_arg);
};
```

Klasa ‘invalid_argument’

- exception ← logic_error ← invalid_argument
- Klasa invalid_argument pochodzi z klasy logic_error. Jest zaprojektowana do reprezentowania wszystkich wyjątków spowodowanych **przekazywaniem niewłaściwych argumentów do metod lub funkcji lub konstruktorów.**

```
class invalid_argument: public logic_error {
public:
  explicit invalid_argument (const string& what_arg);
};
```

Klasa ‘length_error’

- exception ← logic_error ← length_error
- Klasa length_error pochodzi z klasy logic_error. Jest zaprojektowana do reprezentowania wszystkich wyjątków spowodowanych **użyciem niedozwolonych wartości do określenia rozmiaru / długości agregatów danych**.

```
class length_error: public logic_error {
public:
  explicit length_error(const string& what_arg);
};
```

Klasa ‘out_of_range’

- exception ← logic_error ← out_of_range
- Klasa out_of_range wywodzi się z klasy logic_error. Ma na celu reprezentowanie wyjątków związanych z **użyciem niewłaściwych indeksów / kluczy podczas uzyskiwania dostępu do kolekcji danych numerowanych / z kluczami**.

```
class out_of_range: public logic_error {
public:
  explicit out_of_range (const string& what_arg);
};
```

Klasa ‘runtime_error’

- exception ← runtime_error
- Klasa runtime_error pochodzi bezpośrednio z klasy wyjątków. Ma na celu reprezentowanie wszystkich wyjątków spowodowanych **okolicznościami, które mogą wystąpić podczas wykonywania programu**.

```
class runtime_error : public exception {
public:
  explicit runtime_error (const string& what_arg);
}
```

Klasa ‘range_error’

- exception ← runtime_error ← range_error
- Klasa range_error pochodzi z klasy runtime_error. Służy do reprezentowania wyjątków spowodowanych **uzyskaniem wyników obliczeń przekraczających dopuszczalny zakres**.

```
class range_error : public runtime_error {
public:
 explicit range_error (const string& what_arg);
};
```

Klasa ‘overflow_error’

- exception ← runtime_error ← overflow_error
- Klasa overflow_error pochodzi z klasy runtime_error. Ma na celu reprezentowanie wyjątków spowodowanych **uzyskaniem wyników zbyt dużych, aby reprezentować jakąkolwiek użyteczną wartość** (w sensie domeny).

```
class overflow_error : public runtime_error {
public:
  explicit overflow_error (const string& what_arg);
};
```

Klasa ‘underflow_error’

- exception ← runtime_error ← underflow_error
- Klasa underflow_error pochodzi z klasy runtime_error. Ma na celu reprezentowanie wyjątków spowodowanych **uzyskaniem zbyt małych wyników, aby reprezentować jakąkolwiek użyteczną wartość** (w sensie domeny).

```
class underflow_error : public runtime_error {
public:
  explicit underflow_error (const string& what_arg);
};
```

Co dalej?

- jeśli chcesz stworzyć specjalną kategorię wyjątków zaprojektowanych w celu wyróżnienia bardzo specyficznej klasy błędów (underflow), możesz to zrobić w ten sposób :

```
class underflow_speed_error : public underflow_error { };
```

bad_alloc

- Wyjątek *bad_alloc* może zostać zgłoszony jako niepożądany efekt wywołania operatorów *new* lub *new[]*, gdy środowisko wykonawcze lub system operacyjny nie mogą spełnić naszych wymagań dotyczących pamięci.

exception ← bad_alloc

bad_exception

- exception ← bad_exception
- Wyjątek *bad_exception* jest generowany, gdy funkcja próbuje zgłosić wyjątek nieokreślony w specyfikacji *throw*.
  Zauważ, że tego wyjątku nie można złapać bezpośrednio.

bad_exception

```
#include <iostream>
#include <exception>
using namespace std;
void function(void) throw(int) {
	throw 3.14;
}
int main(void) {
	try {
		function();
	} catch(double f) {
		cout << "Got double" << endl;
	} catch(bad_exception bad) {
		cout << "It's so bad..." << endl;
	}
	cout << "Done" << endl;
	return 0;
}
```

- Program nie wyświetli żadnego z komunikatów **“It's so bad...”** **lub “Done”, a nawet “Got double”**.

bad_exception

- Właściwa obsługa wyjątku *bad_exception* wymaga, aby funkcja **określiła wyjątek *bad_exception* na swojej liście rzucanych wyjątków (wygląda to na paradoks, ale to prawda), a nieoczekiwana funkcja obsługi musi zostać zdefiniowana i ustawiona**. Niespełnienie któregokolwiek z tych warunków spowoduje niepożądane zachowanie programu.

bad_exception

```
#include <iostream>
#include <exception>
using namespace std;
void unexp(void) {
	cout << "Unexpected exception arrived!" << endl;
	throw;
}
void function(void) throw(int, bad_exception) {
	throw 3.14;
}
int main(void) {
	set_unexpected(unexp);
	try {
		function();
	} catch(double f) {
		cout << "Got double" << endl;
	} catch(bad_exception bad) {
		cout << "It's so bad..." << endl;
	}
	cout << "Done" << endl;
	return 0;
}
```

bad_exception

- Program wyświetli następujące linie na ekranie:
  - Unexpected exception arrived!
  - It's so bad...
  - Done