Obsługa wyjątków

- **Pozwala zarządzać błędami wykonania w uporządkowany sposób. Umożliwia automatyczne wywołanie części kodu, funkcji, metod klas, który trzeba wykonać przy powstaniu błędów.**

```
try
{
        //blok try –tu umieszczamy kontrolowaną część kodu
}
catch(typ1 arg)
{
        //blok catch – tu będzie przekazywane sterowanie przy powstaniu wyjątków
        //typu typ1
}
catch(typ2 arg)
{
        //blok catch – tu będzie przekazywane sterowanie przy powstaniu wyjątków
        //typu typ2
}
……..
```

```
catch(typN arg)
{
        //blok catch – tu będzie przekazywane sterowanie przy powstaniu wyjątków
        //typu typN
}
```

- **W bloku *try* umieszczamy kod, który chcemy kontrolować pod kątem wystąpienia wyjątków. Przy powstaniu błędu dla generowania wyjątku używamy kluczowe słowo**

throw wyjątek

Tu wyjątek – to obiekt typu *typK*. Jeśli taki wyjątek pozostanie wygenerowany, sterowanie będzie przekazane odpowiedniemu blokowi *catch(typK arg).*

- **Blok *try* może być bardzo krótki, a może obejmować cały program. O to, który blok *catch* przechwyci wyjątek, decyduje typ wyjątku, który ustawiamy słowem kluczowym *throw*.**

- **Przechwycony może być dowolny typ danych. To mogą być klasę, stworzone przez programistę.**

- **Jeśli w bloku *try* żaden wyjątek nie pozostanie wygenerowany, żaden blok *catch* nie dostanie sterowania.**

- **Jeśli zostanie wygenerowany wyjątek, nie zadeklarowany w żadnym bloku *catch*, wykonanie programu będzie przerwane przez wywołanie funkcji z biblioteki run time C++ o nazwie *terminate()*.**

- **Przykład W17_0**

- **Zazwyczaj kod, umieszczony w bloku catch, próbuje poprawić sytuację. Przecież istnieją takie przypadki, kiedy sytuację nie da się poprawić. Wtedy wywołujemy exit(…), terminate(), abort() .**

```
#include "stdafx.h"
#include <iostream>
using namespace std;

#define MAX_CHAR  512

class MyExcept
{
public:
        char strh[MAX_CHAR];
        int ii;
public:
        MyExcept(const char *str, int e);
        MyExcept() { strh[0] = '\0'; ii = 0;}
};

MyExcept::MyExcept(const char *str, int e)
{
        strcpy_s(strh, MAX_CHAR-1, str);
        ii = e;
}

int main()
{
        try
        {
           cout << "start try\n";
           throw MyExcept("To jest wyjatek typu
                                   MyExcept", 100);
        }
        catch (MyExcept e)
        {
              cout << e.strh << "  " << e.ii << endl;
        }
        system("pause");
        return 0;
}
```

```
start try
To jest wyjatek typu MyExcept  100
Для продолжения нажмите любую клавишу . . .
```

- Wyjątek typu klasy – przyłkad W17_1

- **Wielu instrukcji catch – przykład W17_2; W16 !!!**

- **Jeśli funkcja jest włożona w blok try, można ograniczyć typy wyjątków, które funkcja może generować:**

```
zwrac_typ  nazwa_funkcji(lista_argumentów)  throw (lista_typów)
{
……
}

int fun() throw(int, double, CExcpt)
{
}
```

- **Funkcje generuje wyjątki tylko typów, oznaczonych na liście typów (na przykład, int, double, CExcpt)**

- **Microsoft Visual Studio C++ nie wspiera obsługi klauzuli throw().**

- **Przechwytywanie wyjątków wszystkich typów:**

```
catch(…) {
//…
}
```

- **Ponowne zgłoszenie wyjątku: przykład W17_3**

- **Funkcje void terminate()**
  Jest wywoływana podsystemom obsługi wyjątków, jeśli nie udało się znaleźć instrukcji catch dla zgłoszonego typu wyjątku. Domyślnie terminate() wywołuje abort(). Można zmienić to:
  - **Piszemy swoją własną funkcje**

```
void my_handler()
{
        //swój własny sposób przerywania wykonania
        //ta funkcje musi tylko przerwać wykonanie programu!
}
```

  - **Wywolujemy**

```
terminate_handler  set_terminate(terminate_handler  my_handler)  throw()
```

- **Tu: *my_handler* – wskaźnik do funkcji *my_handler*; *set_handler* zwraca wskaźnik do starej procedury obsługi *terminate()*.**

- **Nagłówek <exception>**

- **Funkcje unexpected() – jest wywoływana, gdy funkcja próbuje zwrócić wyjątek, nie zadeklarowany na liście klauzuli throw. Domyślnie unexpected() wywołuje terminate().**

- **To można zmienić:**
  - **Piszemy swój własny wariant obsługi –**

```
void my_unexpected()
{
        //zgłoszenie wyjątku, zakończenie działania programy [ terminate() ]
}
```

  - **Wywołujemy**

```
unexpected_handler  set_unexpected(unexpected_handler  my_unexpected) throw()
```

- **Tu my_unexpected – wskaźnik do funkcji, set_unexpected(…) zwraca wskaźnik do poprzedniej procedury unexpected().**

- **Typy terminate_handler i unexpected_handler pozostały zdefiniowane jako**

```
typedef void ( *terminate_handler )( );
typedef void (*unexpected_handler)( );
```

- **Procedury my_unexpected(), my_terminate() nie powinni zwracać sterowanie kodu, który ich wywołuje – powinni skończyć wykonanie programu**

Statyczne dane składowe

```
class my_class
{
        static int i;
        //……..
}
```

- **Powstaje tylko jedna kopia zmiennej, która będzie współdzielona pomiędzy wszystkimi obiektami tej klasy, a również klasami, pochodnymi od tej klasy.**

- **Każda zmienna statyczna jest faktycznie zmienną globalną, zasiąg deklaracji której jest ograniczony klasą, w którym ta zmienna pozostała zadeklarowana. To oznacza, że definicje tej zmiennej (wydzielenie pamięci) tworzy się nie w trakcie deklarowania jej w klasie, a w innym miejscu programy – poza klasą. Jest możliwe jej inicjowanie poza klasą (domyślnie – 0), a również i dostęp do jej poza klasą.**

- **Czas życia każdej zmiennej statycznej – składowej klasy – od początku wykonania programu (wcześniej od tworzenia samej klasy, do którego ta zmienna należy) i do zakończenia programy (obiekt klasy już jest zniszczony, a zmienna statyczna – nie).**

- **Główny sens użycia zmiennych statycznych – danych klasy – uniknięcie zmiennych globalnych (alternatywa namespace), komunikacje pomiędzy różnymi obiektami tej samej klasy.**

- **Przykład W18_0**

- **Składowe klasy – funkcje statyczne. Są używane dość rzadko, dla tego nie będziemy rozważali ich w obrębie tego kursu.**

Deklaracja danych const and volatile

- **Kluczowe słowo *const* oznacza że obiekt lub zmienna nie ulegają modyfikacji:**

  const declaration **;** **(const int i;)**
  member-function **const ;**

- **Kluczowe słowo *volatile* oznacza, że obiekt lub zmienna mogą być modyfikowane nie tylko instrukcjami kodu danego programu, a i systemem operacyjnym, sprzętem lub równoległe działającym wątkiem.**

- **Zmienne *volatile* nie ulegają optymalizacji przy kompilacji dla tego że ich wartość w każdej chwili może być zmienioną.**

Wskaźniki *const, volatile*:

- **Kluczowe słowo *const* ustala że wskaźnik po inicjowaniu nie może być zmieniony. Wskaźnik *const* jest chroniony przed modyfikacją.**

- **Kluczowe słowo *volatile* ustala że wartość, związana z nazwą, może być zmieniona nie tylko instrukcjami kody tego procesu, a i innego. To jest korzystne dla aplikacji z pamięcią wspólną.**

Deklarowanie obiektu, do którego wskazuje wskaźnik:

```
const char *cpch;       //elementy tablicy nie są modyfikowane
volatile char *vpch;    //elementy tablicy mogą być modyfikowane jako volatile
```

Deklarowanie wartości wskaźnika:

```
char * const pchc;      //nie możemy zmienić wartość wskaźnik  (adres)
char * volatile pchv;   //adres (wartość wskaźnika) jest zmienną typu volatile
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    double aa[] = { 1.1, 2.2, 3.3 };
    const double *p_a = const_cast<const double *>(aa);
    //*p_a = -1.0; //error C3892: 'p_a': you cannot assign to a variable that
                       is const
    p_a ++;  //OK
    cout << *p_a << endl;

    double * const pp_aa = const_cast<double * const>(aa);
    *pp_aa = -1.0;  //OK
    //pp_aa++;  // error C3892: 'pp_aa': you cannot assign to a variable that
    is const

    cout << aa[0] << "  " << aa[1] << "  " << aa[2] << endl;
    system("pause");
    return 0;
}
```

```
2.2
-1  2.2  3.3
Для продолжения нажмите любую клавишу . . .
```

Funkcje składowe const, mutable

- **Funkcje składowe mogą być zadeklarowane jako const, co sprawia, że this jest traktowany jako wskaźnik const – takie funkcje nie mogą zmieniać wartości składowych klasy.**

```
class my_cl
{
        int i;
        mutable int j;
public:
        my_cl(int ii, int jj) : i(ii), j(jj) {}
        void f() const;
};

void my_cl::f() const
{
        i = 10;  //blad - funkcje f()  jest const
        j = 20;  //OK, int  j  jest mutable
}
```

- **Funkcje składowe klasy można deklarować jako volatile – wskaźnik this jest traktowany jako volatile**

```
class A
{
public:
        void f2(int a) volatile;
};
```

- **Konstruktory, zadeklarowane jako explicit -**

```
class my_cl
{
        int i;
public:
        my_cl(int ii) { i = ii; }
};

int main()
{
        my_cl   ob(10);  //to jest standardowa instrukcja  - argument 10 będzie przekazany /
                         //konstruktorowi klasy
        ……..
        my_cl  obb = 12;  //to będzie przekonwertowane automatycznie do   my_cl obb(12);
}
```

- **Jeśli konstruktor klasy ma tylko jeden argument, można użyć dowolną z dwóch postaci inicjowania obiektu. Druga postać inicjowania polega na tym, że nadaje możliwość niejawnego konwertowania typu argumentu do typu klasy.**

- **Zastosowanie specyfikatora explicit wzbrania niejawną konwertowanie, czyli dla konstruktorów, które mają tylko jeden argument, pozostaje tylko jedna postać inicjowania**

```
class my_cl
{
        int i;
public:
        explicit my_cl(int ii) { i = ii; }
};

int main()
{
        my_cl   ob(10);  //teraz tylko taka postać inicjowania jest możliwa
}
```

- **Funkcje konwertujące. Czasami powstaje konieczność użycia obiektu klasy jednego typu w wyrażeniach, wymagających innego typu danych. Jednym ze sposobów takiego przetwarzania jest napisanie funkcji konwertujących:**

```
operator   typ()  { return wartość; }
```

***typ* – to jest typ docelowy, *wartość* – wartość klasy po konwersji. Funkcje konwertujące zwracają wartość typu *typ*.**

- **Funkcja konwertująca nie zawiera parametrów i musi być składową klasy.**

- **Przykład:**

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

class my_coord
{
	int x, y;
public:
	my_coord(int xx, int yy) : x(xx), y(yy) {}
	operator int() { return x*y; } //funkcje konwertujece
};

int _tmain(int argc, _TCHAR* argv[])
{
	my_coord a(2, 3);
	int k;
	k = a;
	cout << "k = " << k << endl;

	k = 200 + a;
	cout << "k = " << k << endl;

	system("pause");
	return 0;
}
```

```
k = 6
k = 206
Для продолжения нажмите любую клавишу . . .
```