

# Wykład 13

## Wybrane zagadnienia programowania obiektowego w C++

- Wyjątki i ich obsługa
- Konstruktor kopiujący
- Lista inicjalizacyjna konstruktora
- Jeszcze o dziedziczeniu - polimorfizm

# Wyjątki i ich obsługa

Obsługa wyjątków jest metodą reagowania na błędy, które mogą pojawić się w czasie wykonywania programu.

Trzy rodzaje błędów:

- **błędy składni** - wykrywane i usuwane w procesie kompilacji programu
- **błędy w logice działania programu** – możliwe do wykrycia dopiero po skompilowaniu programu w fazie jego testowania
- **błędy wykonania aplikacji** - mogą pojawiać się nawet po długim okresie użytkowania aplikacji – sytuacje prowadzące do ich powstania są często trudne do przewidzenia.

Ten ostatni typ błędów nosi często nazwę **wyjątku**.

Dalej pokazano przykład wystąpienia wyjątku i różne sposoby reakcji aplikacji.

Wykorzystano rekurencyjną wersję funkcji, która oblicza wartość NWD (największego wspólnego dzielnika) i przypadek wystąpienia w niej próby dzielenia przez zero.

```
// Funkcja NWD - oblicza największy wspólny dzielnik
// dwóch liczb całkowitych - I sposób reakcji na próbę
// dzielenia przez zero

int nwd(int a, int b)
{
    if (b == 0)
    {
        cout << "Błąd ! Próba dzielenia przez zero" << endl;
        return -1; // sygnalizuje wystąpienie błędu
    }
    if (a % b == 0) // b jest szukanym nwd
        return b;
    else
        return nwd(b, a % b);
}
```

Założono, że po próbie dzielenia przez 0 funkcja kończy obliczenia i zwraca wartość -1 co jest informacją dla otoczenia o wystąpieniu wyjątku.

**Sposób taki prowadzić może do powstania niejednoznaczności !**

```
// II sposób reakcji na próbę dzielenia przez zero

int nwd(int a, int b)
{
    if (b == 0)
    {
        cout << "Błąd ! Próba dzielenia przez zero" << endl;
        exit(-1); // kończy działanie całej aplikacji
                  // i sygnalizuje wystąpienie błędu
    }
    if (a % b == 0) // b jest szukanym nwd
        return b;
    else
        return nwd(b, a % b);
}
```

Funkcja standardowa `exit` kończy działanie aplikacji i przekazuje na poziom systemu operacyjnego wartość argumentu.

Wersje języka C++ zgodne ze standardem ANSI zawierają słowa kluczowe `try`, `catch`, `throw` które pomagają programowo obsługiwać wyjątki (tzw. **strukturalna obsługa wyjątków**).

Składnia instrukcji obsługi wyjątków:

```
try {  
    // polecenia, które mogą spowodować błędy  
}  
catch (typ_argumentu) {  
    // polecenia obsługi wyjątku  
}
```

Taka konstrukcja może zawierać więcej niż jeden blok `catch`.

Polecenia sekcji `try` są wykonywane zawsze, chyba że w czasie ich realizacji lub w wywoływanej przez nie funkcji zostanie zgłoszony wyjątek. Odpowiedni blok `catch` "przechwytuje" wtedy taki sygnał i odpowiednio go obsługuje.

Jeżeli typ wygenerowanego wyjątku jest taki sam jak *typ\_argumentu* to sterowanie zostaje przekazane do bloku `catch`, jeżeli nie to system sprawdza kolejno zgodność typów wyszczególnionych w kolejnych sekcjach `catch`.

Jeżeli system nie znajdzie żadnego bloku `catch` ze zgodnym typem argumentu, to przerywa działanie programu.

Przykład ilustruje sytuację instrukcji z jedną sekcją `try` i dwoma, związanymi z nią sekcjami `catch`:

```
try {
    otworz_plik();
    odczytaj_dane();
    przetworz_dane();
}
catch (int blad) {
    obsluga_bledu_1();
}
catch (double blad) {
    obsluga_bledu_2();
}
```

Typ wyjątku może być dowolny – może to być jeden z typów "wbudowanych" (`int`, `float`, `char` itp.) lub typ utworzony przez programistę za pomocą słowa kluczowego `class`.

Programista może samodzielnie zgłosić wyjątek. Do tego służy polecenie  
`throw obiekt_wyjatku;`

Wykonanie tej instrukcji powoduje zgłoszenie wyjątku, czyli przerwanie normalnego przepływu sterowania w programie. Wyjątek ten przechwytuje odpowiedni blok `catch`, a jeżeli nie, system przerwie wykonywanie całego programu.

```

// Przykład 1
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int tr;
    try {
        cout << "Cos tam, cos tam !" << endl;
        cin >> tr;
        if (tr == 0) throw 13;    // "wyrzuca" wyjątek numer 13
        cout << "Wychodze tedy" << endl; // wykona się jeżeli
                                        // wprowadzono wartość różną niż 0

        system("pause");
        return 0;
    }
    catch (int blad) {
        cout << "Wyjatek zgloszony bez powodu !" << "Numer bledu: "
             << blad << endl;
        system("pause");
        return 1;
    }
    system("pause");
    return 0;
}

```

```

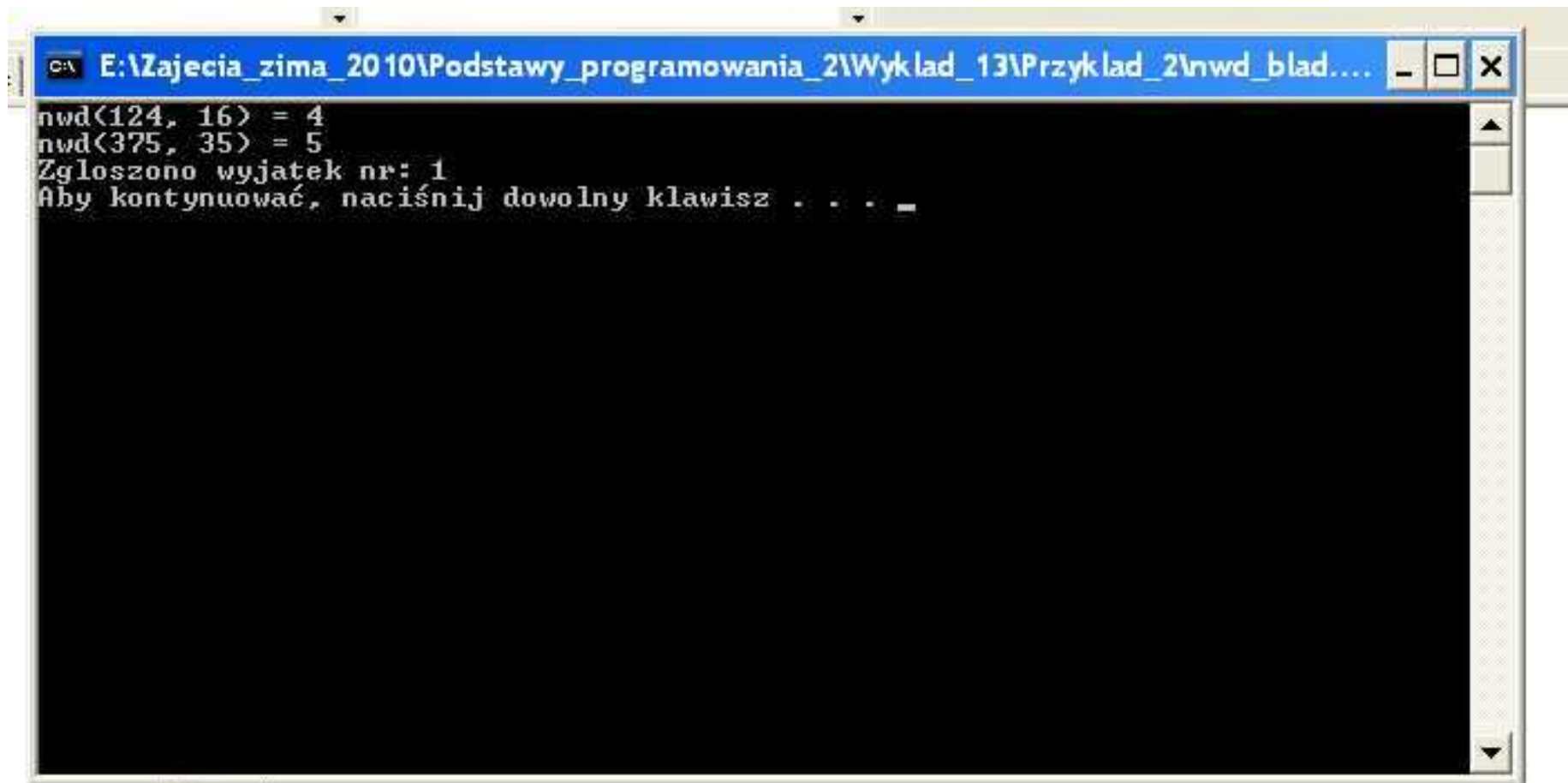
C:\ E:\Zajecia_zima_2010\Podstawy_programowania_2\Wyklad_13\Przyklad_1\main1.exe
Cos tam, cos tam !
0
Wyjatek zgloszony bez powodu !Numer bledu: 13
Aby kontynuować, naciśnij dowolny klawisz . . . _

```

```

// Przykład 2
#include <cstdlib>
#include <iostream>
using namespace std;
int nwd(int a, int b);
int main(int argc, char *argv[])
{
    try {
        cout << "nwd(124, 16) = " << nwd(124, 16) << endl;
        cout << "nwd(375, 35) = " << nwd(325, 35) << endl;
        cout << "nwd(270, 0) = " << nwd(270, 0) << endl;
        return 0;
    }
    catch (int blad) {
        cout << "Zgloszono wyjatek nr: " << blad << endl;
        system("pause");
        return 1;
    }
}
int nwd(int a, int b)
{
    if (b == 0) throw 1;
    if (a % b == 0) return b;
    else
        return nwd(b, a%b);
}

```

A screenshot of a Windows command prompt window. The title bar shows the file path: E:\Zajecia\_zima\_2010\Podstawy\_programowania\_2\Wyklad\_13\Przyklad\_2\nwd\_blad.... The window contains the following text:

```
nwd(124, 16) = 4  
nwd(375, 35) = 5  
Zgloszono wyjatek nr: 1  
Aby kontynuowac, naciśnij dowolny klawisz . . . _
```

```
return 1;
```

```
// Przykład 3 - IDE C++ Builder
#include <vcl.h>
#include <iostream>
#include <stdexcept>
using namespace std;
static void f() { throw runtime_error("blad w czasie wykonania");
}
int main ()
{
    try
    {
        f();
    }
    catch (const exception& e)
    {
        cout << "I masz wyjatek: " << e.what() << endl;
        // metoda what zwraca komunikat
        system("pause");
    }
    return 0;
}
```

```
nowvl 14 3 G > 39 0 G wolne | \ | | [-e-] | Innowvl 14,
E:\Zajecia_zima_2010\Podstawy_programowania_2\Wyklad_13\Przyklad_3\Przyklad_3...
I masz wyjątek: błąd w czasie wykonania
Aby kontynuować, naciśnij dowolny klawisz . . . _
```

Przykład zastosowania instrukcji `try .. catch` do obsługi błędu dzielenia przez zero w przykładowym programie obliczającym wartość największego wspólnego dzielnika:

```

// Przykład 4
#include <vcl.h>
#include <iostream.h>
#include <stdexcept.h>
int nwd(int a, int b);
int main(int argc, char* argv[])
{
    try {
        cout << "nwd(997, 12) = " << nwd(997, 12) << endl;
        cout << "nwd(181, 14) = " << nwd(181, 14) << endl;
        cout << "nwd(445, 0) = " << nwd(445, 0) << endl;
        return 0;
    }
    catch (const exception& e) {
        cout << "Bład ! " << e.what() << endl;
        system("pause");
    }
    return 0;
}
//
int nwd(int a, int b) {
    if (b == 0) throw runtime_error("Proba dzielenia przez zero");
    if (a % b == 0) return b;
    else return nwd(b, a%b);
}

```

```
E:\Zajecia_zima_2010\Podstawy_programowania_2\Wyklad_13\Przyklad_4\Przyklad_4...
nwd(997, 12) = 1
nwd(181, 14) = 1
Bład !Proba dzielenia przez zero
Aby kontynuować, naciśnij dowolny klawisz . . .
```

## **Konstruktor kopiujący**

**Jest to szczególny rodzaj konstruktora bo:**

- jest wywoływany bardzo często,**
- niekiedy działa w sposób niejawny,**
- jeżeli nie utworzono takiego konstruktora, to tworzy go automatycznie kompilator,**
- konstruktor kopiujący utworzony domyślnie nie "wydziedzicza" konstruktora, który utworzył programista w sposób jawny.**

**Konstruktor kopiujący jest wywoływany automatycznie:**

- przy zwracaniu przez funkcję obiektu określonej klasy. Funkcja tworzy tymczasową kopię obiektu i zwraca ją do wyrażenia, które ją wywołało,**
- gdy argument wywołania funkcji jest obiektem pewnej klasy (instancją klasy). Wtedy też system tworzy kopię argumentu, którą przekazuje do funkcji,**
- gdy wykorzystuje się obiekt do zainicjowania innego obiektu tej samej klasy.**

**Konstruktor kopiujący nie działa wtedy, gdy obiekt przekazuje się do funkcji za pomocą wskaźnika.**

**Składnia konstruktora kopiującego ma postać:**

*nazwa\_klasy(nazwa\_klasy const &źródło)*

Przykład konstruktora kopiującego pojawił się w deklaracji klasy ułamek:

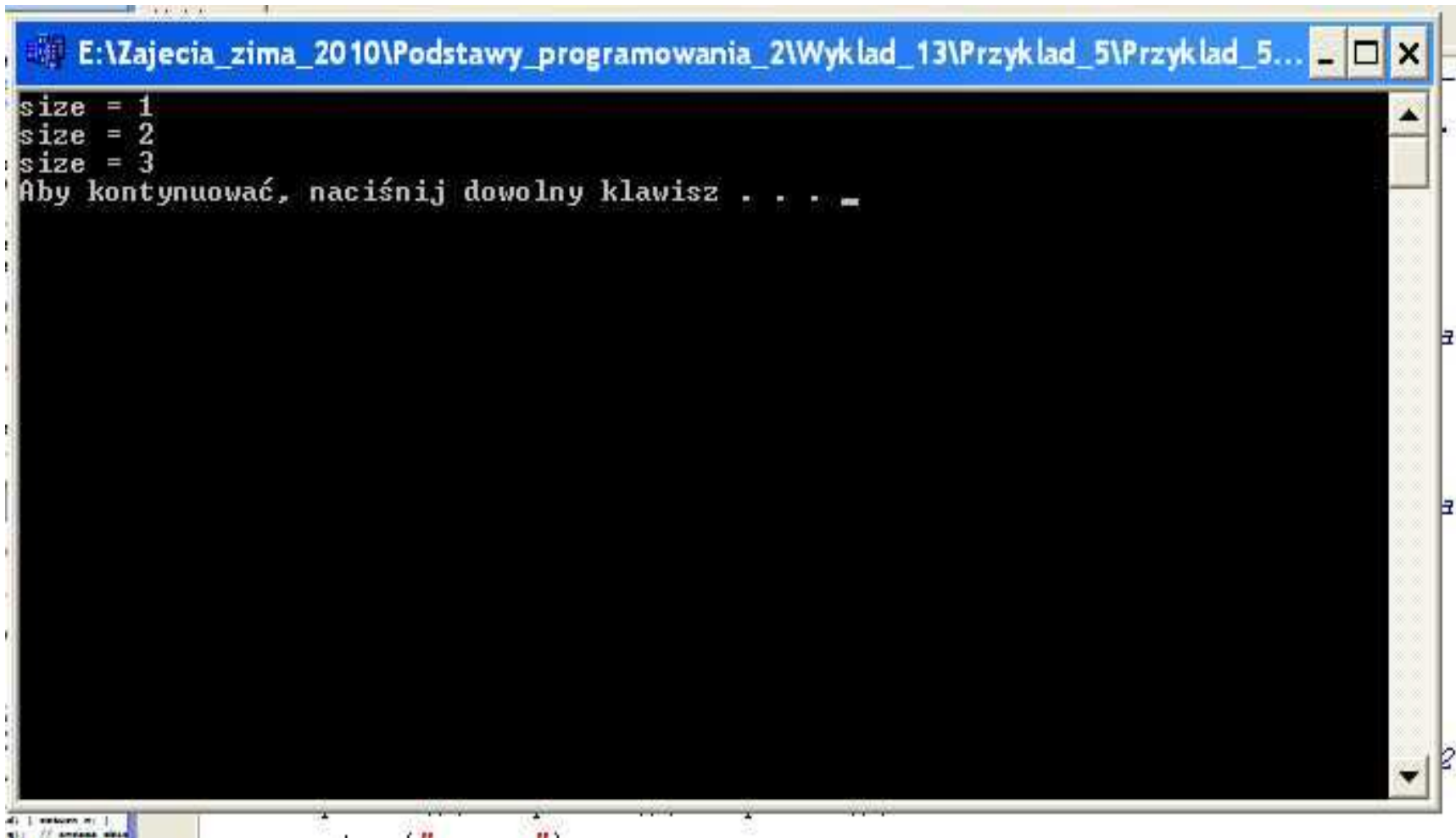
```
class ULAMEK {
    private: int l, m; // l - licznik, m - mianownik
            void normalizuj(void);
            int nwd(int a, int b); // największy wspólny dzielnik
            int nww(int a, int b); // najmniejsza wspólna wielokrotność
    public: ULAMEK() { l = 0; m = 1; }
           ULAMEK(int licz, int mian){l = licz; m = mian; normalizuj(); }
           ULAMEK(ULAMEK const &zrodlo); // Konstruktor kopiujacy;
           int pobierz_licz(void) { return l; }
           int pobierz_mian(void) { return m; }
           ULAMEK dodaj(ULAMEK q); // zwraca obiekt klasy ULAMEK
           ULAMEK pomnoz(ULAMEK q); }; // zwraca obiekt klasy ULAMEK
//////// Konstruktory //////////////////////////////////////////
ULAMEK::ULAMEK(ULAMEK const &zrodlo)
{
    cout << "Konstruktor kopiujacy" << endl;
    l = zrodlo.l;
    m = zrodlo.m;
}
```

# Lista inicjalizacyjna konstruktora

Jest to szczególne miejsce, gdzie odbywa się inicjalizacja. Przeznaczona jest przede wszystkim do inicjalizacji stałych składowych klasy.

**Przykład:**

```
// Inicjalizacja stałych w klasach - za B.Eckel, Thinking in C++
#include <iostream.h>
class Fred {
    const int size; // w tym miejscu nie wolno zainicjalizowac size
public: Fred(int sz); // konstruktor
        void print(void);
};
Fred::Fred(int sz) : size(sz) { } // konstruktor z lista
                                // inicjalizacyjna
void Fred::print(void) { cout << "size = " << size << endl; }
int main(void)
{
    Fred a(1), b(2), c(3);
    // powstaly obiekty a, b i c stalej size nadano wartosci 1, 2 i 3
    a.print(); b.print(); c.print();
    system("pause");
    return 0;
}
```



```
E:\Zajecia_zima_2010\Podstawy_programowania_2\Wyklad_13\Przyklad_5\Przyklad_5...
size = 1
size = 2
size = 3
Aby kontynuować, naciśnij dowolny klawisz . . . _
```

**Inicjalizacje znajdujące się na liście inicjalizacyjnej odbywają się jeszcze przed wykonaniem jakiegokolwiek kodu wchodzącego w skład konstruktora.**

## Jeszcze o dziedziczeniu – polimorfizm

**Polimorfizm to najważniejsza cecha programowania obiektowego. Oznacza, że implementacja pewnych funkcji (metod) może przybierać wiele różnych form.**

**Metody wirtualne – słowo kluczowe `virtual` poprzedza te metody, które w klasach bazowych mają tą samą nazwę, ale inną niż w klasie bazowej implementację. Metody takie wywołane na rzecz obiektu klasy potomnej realizują swój algorytm specyficzny dla klasy potomnej.**

```

// Przykład 6 - dziedziczenie, nie użyto metod wirtualnych
#include <vcl.h>
#include <iostream.h>
#include <iomanip.h>
class PUNKT_2D          // Klasa bazowa
{ protected: double x, y; // 1
  public:    void wyswietl();
            void przesun(double, double);
// Akcesory                2
    double GetX () const { return x; }
    double GetY () const { return y; }
    void SetX(double xx) { x = xx; }
    void SetY(double yy) { y = yy; }
};
void PUNKT_2D::wyswietl()
{
    cout << "Wspolrzedne na plaszczyznie: [" << fixed
         << setprecision(2) << setw(6) << x << ", "
         << setw(6) << y << "]" << endl;
}
void PUNKT_2D::przesun(double dx, double dy)
{
    x += dx;
    y += dy;
}
// -----

```

```

class PUNKT_3D : public PUNKT_2D          // Klasa potomna   3
{
    protected: double z;                // 4
    public: PUNKT_3D(double, double, double); // Konstruktor   5
           void wyswietl();              // 6
           void przesun(double, double, double);

// Akcesory
           double GetZ() const { return z; } // 7
           void SetZ(double zz) { z = zz; }

};
PUNKT_3D::PUNKT_3D(double _x, double _y, double _z)
{
    x = _x;
    y = _y;
    z = _z;
}
void PUNKT_3D::wyswietl()                // 8
{
    cout << "Wspolrzedne w przestrzeni: [" << fixed
         << setprecision(2) << setw(6) << x << ", " << setw(6)
         << y << ", " << setw(6) << z << "]" << endl;
}
void PUNKT_3D::przesun(double dx, double dy, double dz) // 9
{
    x += dx;
    y += dy;
    z += dz;
}

```

```

int main(int argc, char* argv[])
{
    PUNKT_3D A(1, 2, 3), B(0, 0, 0), D(0, 0, 0);    // 10
    PUNKT_2D C;    // 11
    C.SetX(0);    // 12
    C.SetY(0);
    D.SetX(12.3);
    D.SetY(-3.45);
    D.SetZ(1.14);
    cout << "Punkt A - ";
    A.PUNKT_3D::wyswietl();    // 13
    cout << "Punkt B - ";
    B.PUNKT_3D::wyswietl();
    D.PUNKT_3D::wyswietl();
    cout << "Wspolrzedne punktu D - x = " << D.GetX() << endl
         << "                - y = " << D.GetY() << endl
         << "                - z = " << D.GetZ() << endl;
    A.PUNKT_3D::przesun(2, -1, 3);    // 14
    cout << "  A po przesunieniu - ";
    A.PUNKT_3D::wyswietl();
    B.PUNKT_3D::przesun(5,-8, -2);
    cout << "  B po przesunieniu - ";
    B.PUNKT_3D::wyswietl();
    cout << "Punkt C - ";
    C.PUNKT_2D::wyswietl();    // 15
    C.PUNKT_2D::przesun(1,-2.3315);

```

```
cout << "  C po przesunięciu - ";  
    C. PUNKT_2D::wyswietl();  
    system("PAUSE");  
    return 0;  
}
```

### Uwagi i komentarze:

1. W klasie bazowej pola opatrzone modyfikatorem dostępu `protected` są dostępne z obiektów danej klasy i z klas potomnych.
2. Akcesory są niezbędne, aby można było odczytywać i modyfikować pola chronione
3. Klasa potomna "wyprowadzona" z klasy bazowej, której nazwę podaje się po znaku dwukropka
4. Pole z oznaczone jako chronione na wypadek, gdyby klasa `PUNKT_3D` była kiedyś klasą bazową dla kolejnych potomków.
5. Konstruktor klasy potomnej – uwaga nie ma dziedziczenia konstruktora domyślnego
6. Metody `wyswietl` i `przesun` w klasie potomnej mają takie same nazwy jak metody w klasie bazowej, spełniają taką samą rolę, ale ich implementacje różnią się ponieważ odnoszą się do obiektów różnych klas – o różnych cechach, a więc obiektów, których zachowanie jest różne.

7. Niezbędne akcesory do pola chronionego
8. Definicja metody `wyswietl` klasy potomnej
9. Definicja metody `przesun` klasy potomnej
10. Nie można np. zadeklarować obiektu `D` bez jawnego wywołania konstruktora w klasie potomnej, bowiem konstruktor domyślny z klasy bazowej nie podlega dziedziczeniu
11. A tutaj zadziała konstruktor domyślny obiektu klasy bazowej
12. Zmiana i odczyt wartości pól chronionych za pomocą akcesorów
13. Wywołanie metody `wyswietl` klasy potomnej
14. Wywołanie metody `przesun` klasy potomnej
15. Wywołanie metod `wyswietl` i `przesun` klasy bazowej

```

// Dziedziczeni - zastosowanie meto wirtualnych
#include <iostream.h>
#include <iomanip.h>
class PUNKT_2D      // Klasa bazowa
{ protected: double x, y;
  public:      virtual void wyswietl();           // 1
              virtual void przesun(double, double);
// Akcesory
    double GetX () const { return x; }
    double GetY () const { return y; }
    void SetX(double xx) { x = xx; }
    void SetY(double yy) { y = yy; }
};
// 1
void PUNKT_2D::wyswietl()
{   cout << "Wspolrzedne na plaszczyznie: [" << fixed
    << setprecision(2) << setw(6) << x << ", " << setw(6)
    << y << "]" << endl;
}
void PUNKT_2D::przesun(double dx, double dy)
{   x += dx;
    y += dy;
}

```

```

class PUNKT_3D : public PUNKT_2D          // 2
{
    protected: double z;                // 3
        PUNKT_3D(double, double, double); // Konstruktor
    virtual void wyswietl();             // 4
    virtual void przesun(double, double, double);

// Akcesory
        double GetZ() const { return z; }
        void SetZ(double zz) { z = zz; }
};
PUNKT_3D::PUNKT_3D(double _x, double _y, double _z)
{
    x = _x;
    y = _y;
    z = _z;
}
void PUNKT_3D::wyswietl()
{
    cout << "Wspolrzedne w przestrzeni: [" << fixed
        << setprecision(2) << setw(6) << x << ", " << setw(6)
        << y << ", " << setw(6) << z << "]" << endl;
}
void PUNKT_3D::przesun(double dx, double dy, double dz)
{
    x += dx;
    y += dy;
    z += dz;
}

```

```

int main(int argc, char* argv[])
{
    PUNKT_3D A(1, 2, 3), B(0, 0, 0), D(0, 0, 0); // 5
    PUNKT_2D C; // Konstruktor domyslny tworzy obiekt C
    C.SetX(0); // 10
    C.SetY(0);
    D.SetX(12.3);
    D.SetY(-3.45);
    D.SetZ(1.14);
    cout << "Punkt A - ";
    A.wyswietl(); // 6
    cout << "Punkt B - ";
    B.wyswietl();
    D.wyswietl();
    cout << "Wspolrzedne punktu D - x = " << D.GetX() << endl
         << " - y = " << D.GetY() << endl
         << " - z = " << D.GetZ() << endl;
    A.przesun(2, -1, 3);
    cout << " A po przesunieciu - ";
    A.wyswietl();
    B.przesun(5,-8, -2);
    cout << " B po przesunieciu - ";
    B.wyswietl();
    cout << "Punkt C - ";
}

```

```
C.wyswietl(); // 7
C.przesun(1,-2.3315);
cout << " C po przesunięciu - ";
C.wyswietl();
system("PAUSE");
return 0;
}
```

Uwagi i komentarze:

1. Słowo kluczowe `virtual` sygnalizuje, że te metody będą "przykryte" metodami właściwymi dla klasy potomnej
2. Deklaracja klasy potomnej
3. Nowa składowa, właściwa dla klasy potomnej – być może, że klasa ta stanie się kiedyś klasą bazową dla innych klas i z tego powodu jest chroniona
4. Jak wyżej – metody wirtualne
5. Klasa potomna nie dziedziczy konstruktora domyślnego
6. Metody wirtualne `wyswietl` i `przesun` wywołane na rzecz obiektów klas potomnych – "przesłoniły" metody w klasie bazowej
7. Metody wirtualne `wyswietl` i `przesun` wywołane na rzecz obiektu klasy bazowej