

Podstawy informatyki

Elektrotechnika I rok

Język C++ Klasy. Pojęcia podstawowe

Instrukcja do ćwiczenia

Tematyka i cel ćwiczenia

Celem ćwiczenia jest zapoznanie z podstawowym elementem języka C++ – klasą. Wprowadzone są następujące pojęcia: klasa, konstruktor klasy, destruktor klasy, metody klasy, funkcje zaprzyjaźnione z klasą.

Wymienione wyżej pojęcia są zilustrowane na przykładzie klasy implementującej wektor w przestrzeni trójwymiarowej.

Wprowadzenie

Celem wprowadzenia pojęcia klasy do języka C++ jest dostarczenie programiście narzędzia do tworzenia nowych typów, z których można korzystać tak samo wygodnie, jak z typów wbudowanych do języka. Projektując język C++ przyjęto, że typ zdefiniowany przez użytkownika nie powinien się różnić od typu wbudowanego sposobem używania, lecz tylko sposobem tworzenia go.

Deklaracja klasy zawiera jedno z trzech słów kluczowych `class`, `struct` lub `union` poprzedzających nazwę klasy i następnie deklarację poszczególnych pól danych, metod (funkcji składowych) klasy i funkcji zaprzyjaźnionych z klasą zawartych w nawiasach klamrowych. Przykładową deklarację klasy pokazuje przykład 1.

Przykład 1

```
struct abc {
    int x;
    double *d;
    abc();
    abc(int ax, double ad);
    ~abc();
    void fun1(void);
    int fun2(int x);
    int fun2(double x);
};
```

W przykładzie 1 zadeklarowano klasę o nazwie `abc` zawierającą dwa pola danych: jedno typu `int`, drugie typu `double*` oraz 6 metod (funkcji składowych):

- `abc()` – konstruktor 1,
- `abc(int, double)` – konstruktor 2,
- `~abc()` – destruktor,
- `void fun1(void)` – metoda `fun1`,
- `int fun2(int)` – metoda 1. `fun2`,
- `int fun2(double)` – metoda 2. `fun2`,

Metody klasy są funkcjami składowymi klasy, które powinny dostarczać użytkownikowi interfejsu do operowania na danej klasie. Każda metoda klasy ma dostęp do prywatnych (`private`) i zastrzeżonych (`protected`) pól danych i metod danej klasy. Dodatkowo do każdej metody jest przekazywany jeden argument (oprócz argumentów jawnie wyspecyfikowanych), będący wskaźnikiem do obiektu klasy, na rzecz której metoda jest wywołana. Wskaźnik ten jest dostępny wewnątrz każdej metody klasy pod nazwą `this`. Dostępność wskaźnika `this` ilustruje przykład 2.

Przykład 2

Niech dane będą następujące deklaracje:

```
class XYZ {                                // definicja klasy XYZ
    .
    .
    .
    void fun1(void);
}

XYZ x1, x2;                               // deklaracja obiektów x1 i x2
                                           // typu klasa XYZ
```

Język C++. Klasy. Pojęcia podstawowe

Jeżeli w programie pojawi się instrukcja:

```
x1.fun1();
```

która wywołuje metodę `fun1()` na rzecz obiektu `x1`, to w czasie wykonywania kodu tej metody wskaźnik `this` wskazuje na obiekt `x1` (tzn. zawiera adres obiektu `x1`).

W przypadku instrukcji:

```
x2.fun1();
```

w czasie wykonywania metody `fun1()` wskaźnik ten wskazuje na obiekt `x2`.

Szczególną rolę pełnią specjalne metody klasy: **konstruktor** i **destruktor**.

Konstruktor(ami) klasy jest metoda o nazwie identycznej z nazwą klasy. Jest ona wywoływana automatycznie przez kompilator w chwili tworzenia klas (to jest w momencie napotkania deklaracji zmiennej będącej klasą lub w momencie dynamicznego alokowania klasy w pamięci za pomocą operatora `new`) i ma na celu jej zainicjowanie. Konstruktorów może być kilka, co oznacza, że mogą być przeciążane, przy czym muszą się one między sobą różnić typami argumentów (podobnie jak funkcje przeciążone). Konstruktor nie może zwracać żadnej wartości (nawet typu `void`).

Destruktoorem klasy jest metoda o nazwie identycznej z nazwą klasy poprzedzoną znakiem tyldy (`'~'`). W danej klasie może być tylko jeden destruktor. Musi być on metodą bez żadnych argumentów i nie może zwracać żadnej wartości (nawet typu `void`). Destrukto jest wywoływany automatycznie w chwili usuwania klasy z pamięci (to jest w chwili zakończenia życia zmiennej (najczęściej przed opuszczeniem funkcji instrukcją `return`) lub w momencie usuwania z pamięci zmiennej typu klasa za pomocą operatora `delete`).

W języku C++ występuje pewna różnica w znaczeniu słów kluczowych `struct` i `union` w stosunku do języka C. W języku C słowa te służą do deklarowania obiektów przechowujących dane; w języku C++ obiekty te mogą zawierać także metody. W języku C++ można zdefiniować strukturę (`struct`) lub unię (`union`) bez żadnych metod, tworząc w ten sposób typ lub obiekt identyczny pod względem własności ze strukturą lub unią znaną z języka C. Użycie słów kluczowych `class`, `struct` lub `union` powoduje w języku C++ ustalenie domyślnych praw dostępu do poszczególnych elementów klasy.

Jeżeli w definicji klasy zostanie użyte słowo kluczowe `struct` (tak jak w przykładzie 1) lub `union`, wtedy wszystkie elementy klasy (pola danych i metody) są domyślnie dostępne z każdego miejsca programu (w zakresie widzialności danej klasy). Inaczej mówiąc elementy klasy są publiczne.

Jeżeli w definicji klasy zostanie użyte słowo kluczowe `class`, wtedy wszystkie elementy klasy (pola danych i metody) nie są domyślnie dostępne na zewnątrz klasy. Elementy klasy są dostępne tylko dla metod danej klasy. Inaczej mówiąc elementy klasy są prywatne.

W celu zmiany praw dostępu do poszczególnych elementów klasy można użyć następujących słów kluczowych:

- `public` – powodującego, że wszystkie elementy klasy występujące po tym słowie są publiczne.
- `private` – powodującego, że wszystkie elementy klasy występujące po tym słowie są prywatne.
- `protected` – powodującego, że wszystkie elementy klasy występujące po tym słowie są zastrzeżone, co oznacza, że są dostępne dla metod danej klasy i metod klas pochodnych.

Użycie wyżej wymienionych słów ilustrują przykłady 3 i 4.

Przykład 3

```
struct abc1 {  
    int x;                // public (domyślnie)  
private:  
    double *d;           // private  
public:  
    abc1();               // public
```

Język C++. Klasy. Pojęcia podstawowe

```
    abc1(int ax, double ad);    // public
    ~abc1();                    // public
    void fun1(void);            // public
private:
    int fun2(int x);            // private
public:
    int fun2(double x);         // public
};
```

Przykład 4

```
class abc2 {
    int x;                      // private (domyślnie)
    double *d;                  // private (domyślnie)
public:
    abc2();                     // public
    abc2(int ax, double ad);    // public
    ~abc2();                     // public
    void fun1(void);            // public
private:
    int fun2(int x);            // private
public:
    int fun2(double x);         // public
};
```

Funkcją zaprzyjaźnioną z daną klasą jest funkcja nie będąca metodą (funkcją składową) danej klasy lecz mająca dostęp do prywatnych (`private`) i zastrzeżonych (`protected`) elementów klasy. Funkcję zaprzyjaźnioną są funkcjami zewnętrznymi dla klasy. W związku z tym nie mają one dostępu do wskaźnika `this`, dzięki któremu mogłyby identyfikować obiekt (instancję klasy), na której dana funkcja ma operować. Z tego powodu najczęściej jednym z argumentów funkcji zaprzyjaźnionych jest wskaźnik lub referencja do klasy, na której funkcja ma operować. Funkcja zaprzyjaźniona musi być zadeklarowana wewnątrz klasy, z którą ma być zaprzyjaźniona. Jej deklaracja jest poprzedzona słowem kluczowym `friend`. Deklarację funkcji zaprzyjaźnionej pokazuje przykład 5.

Przykład 5

```
class abc4 {
    int x;
    double *d;
public :
    abc4();
    abc4(int ax, double ad);
    ~abc4();
    void fun1(void);
    friend int fun2(abc4 &A, int x);
    friend int fun2(abc4 &A, double x);
};

int fun2(abc4 &A, int x)
{
    // definicja funkcji
}

int fun2(abc4 &A, double x)
{
    // definicja funkcji
}
```

Język C++. Klasy. Pojęcia podstawowe

Zakres materiału i literatura

Do ćwiczenia należy zapoznać się z literatury z następującymi pojęciami:

- klasa, definicja klasy, konstruktor(y) klasy, destruktor klasy, metody klasy, wskaźnik `this`, dostęp do składowych klasy,
- funkcje zaprzyjaźnione,

Z literatury można polecić następujące książki:

- B. Stroustrup „Język C++”,
- S. Lipmann „Podstawy języka C++”.

Program ćwiczenia

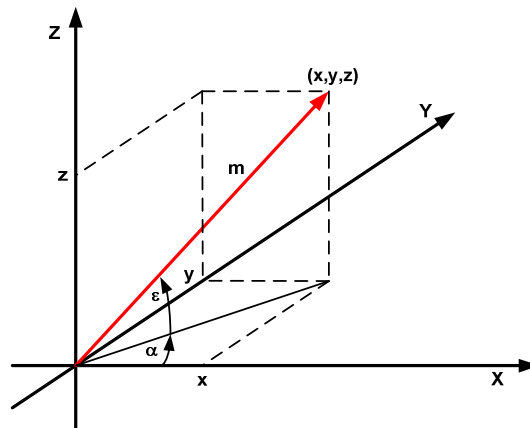
Tematem ćwiczenia jest implementacja klasy `Wektor3`, której obiekty służą do reprezentowania wektora w postaci jego współrzędnych kartezjańskich oraz wykonywania podstawowych operacji na wektorach:

- przypisywania współrzędnych wektorowi jako trzech liczb;
- przypisywania jednego wektora do drugiego (kopiowanie wektorów);
- sumowanie wektorów;
- mnożenie wektora przez liczbę;
- obliczanie współrzędnych biegunowych danego wektora (modułu, kąta azymutu i kąta elewacji);
- sprawdzanie równości i nierówności dwóch wektorów.

W klasie `Wektor3` współrzędne wektora są przechowywane jako trzy prywatne elementy `_x`, `_y` i `_z` typu `double`.

Na rysunku 1 pokazano szkic wektora w przestrzeni z oznaczeniami:

- x, y, z – współrzędne kartezjańskie wektora,
- m, α, ε – współrzędne biegunowe wektora (moduł, azymut, elewacja).



Rys. 1. Wektor w przestrzeni trójwymiarowej

Współrzędne biegunowe można policzyć na podstawie współrzędnych kartezjańskich na podstawie następujących zależności:

$$m = \sqrt{x^2 + y^2 + z^2}$$

$$\varepsilon = \arcsin \frac{z}{m}$$

$$\alpha = \begin{cases} \alpha_1 & \text{gdy } y \geq 0 \\ -\alpha_1 & \text{gdy } y < 0 \end{cases} \quad \text{gdzie} \quad \alpha_1 = \arccos \frac{x}{\sqrt{x^2 + y^2}}$$

Do instrukcji w pliku zip jest dołączony plik `PI_Cpp03.cpp`, którego listing jest przedstawiony w załączniku 1 instrukcji. Po otwarciu nowego projektu w środowisku Dev-C++ należy go skopiować do edytora i na nim wykonywać ćwiczenie.

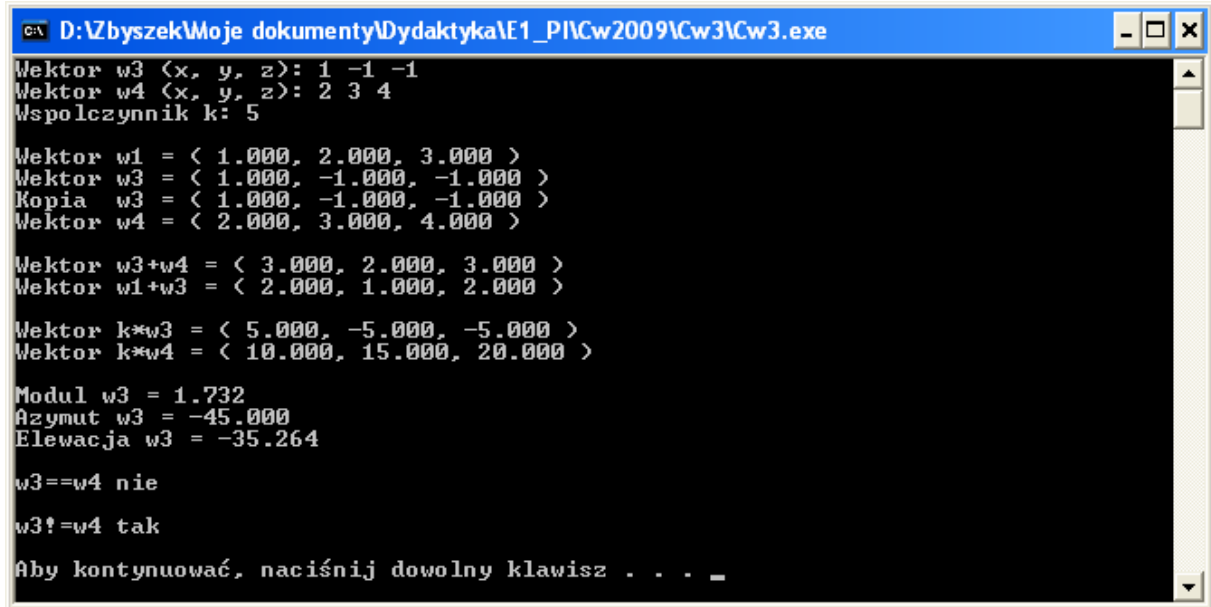
Program, którego zarys („szablon”) jest zawarty w pliku `PI_Cpp03.cpp` zawiera deklarację klasy `Wektor3`, przykładowe definicje niektórych konstruktorów i metod oraz funkcję `main()`. Po uzupełnieniu program powinien zachowywać się następująco:

- zachęca użytkownika do wprowadzenia współrzędnych wektora $w3$ i następnie $w4$;
- zachęca użytkownika do wprowadzenia współczynnika k ;
- wyświetla współrzędne poszczególnych wektorów $w1$, $w3$, $w31$ (kopii $w3$) oraz $w4$ (test konstruktorów i metody `Assign(double, double, double)`);
- oblicza i wyświetla sumę wektorów $w3+w4$ i $w1+w3$ (test metody `Plus(const Wektor3&)` i funkcji `Plus(const Wektor3&, const Wektor3&)`);

Język C++. Klasy. Pojęcia podstawowe

- oblicza i wyświetla iloczyn $k \cdot w_3$ i $k \cdot w_4$ (test metody `Mult(double)` i funkcji `Mult(double, const Wektor3&)`);
- oblicza i wyświetla moduł, azymut i elewację wektora w_3 (test metod `Mod(void)`, `Azym(void)` i `Elev(void)`);
- porównuje i wyświetla wynik porównania wektorów w_3 i w_4 (test funkcji `Eq(const Wektor3&, const Wektor3&)` i `Neq(const Wektor3&, const Wektor3&)`);

Na rysunku 2 pokazano wygląd ekranu, na którym widać przykładowe wyniki działania programu.



```
D:\Zbyszek\Moje dokumenty\Dydaktyka\E1_P\VCw2009\VCw3\VCw3.exe
Wektor w3 <x, y, z>: 1 -1 -1
Wektor w4 <x, y, z>: 2 3 4
Wspolczynnik k: 5

Wektor w1 = < 1.000, 2.000, 3.000 >
Wektor w3 = < 1.000, -1.000, -1.000 >
Kopia w3 = < 1.000, -1.000, -1.000 >
Wektor w4 = < 2.000, 3.000, 4.000 >

Wektor w3+w4 = < 3.000, 2.000, 3.000 >
Wektor w1+w3 = < 2.000, 1.000, 2.000 >

Wektor k*w3 = < 5.000, -5.000, -5.000 >
Wektor k*w4 = < 10.000, 15.000, 20.000 >

Modul w3 = 1.732
Azymut w3 = -45.000
Elewacja w3 = -35.264

w3==w4 nie
w3!=w4 tak

Aby kontynuować, naciśnij dowolny klawisz . . . _
```

Rys. 2. Przykładowy wygląd ekranu po uruchomieniu programu

W ramach ćwiczenia, bazując na pliku `PI_Cpp03.cpp`, należy uzupełnić definicję metod klasy `Wektor3` oraz funkcji z nią zaprzyjaźnionych i uruchomić program. Definicje konstruktorów `Wektor3(void)` i `Wektor3(double, double, double)` oraz metod `X(void)`, `Y(void)`, `Z(void)` oraz `Assign(double)` są pokazane przykładowo w tekście programu. Nie należy modyfikować istniejących w szablonie programu deklaracji i instrukcji (w szczególności deklaracji klasy `Wektor3` i funkcji `main()`).

Uzupełnia wymagają definicje następujących elementów klasy `Wektor3` i funkcji:

1. Konstruktor kopiujący `Wektor3::Wektor3(const Wektor3 &w)`
Konstruktor ten powinien zainicjować pola `_x`, `_y` i `_z` klasy wartościami odpowiednich pól klasy w (ma skopiować współrzędne klasy w).
2. Metoda `Wektor3::Assign(const Wektor3 &w)`
Metoda ta powinna zainicjować pola `_x`, `_y` i `_z` klasy wartościami odpowiednich pól klasy w (ma skopiować współrzędne klasy w) i zwrócić referencję do „siebie samego” tj. wektora, do którego jest przypisywany wektor w (argument). Metoda ta jest wykorzystywana do przypisania jednego wektora do drugiego. Przykładowo instrukcja
`w10.Assign(w20);`
ma działać analogicznie jak standardowy operator `=` języka C++:
`w10 = w20;`
3. Metoda `Wektor3::Mod(void)`
Metoda ta ma zwracać moduł (długość) wektora. Przykładowo instrukcja
`d = w10.Mod();`
spowoduje przypisanie zmiennej `d` modułu wektora `w10`.
4. Metoda `Wektor3::Azym(void)`
Metoda ta ma zwracać kąt azymutu α wektora wyrażony w stopniach. Kąt azymutu α ma spełniać zależność: $-180^\circ < \alpha \leq 180^\circ$. Przykładowo instrukcja

Język C++. Klasy. Pojęcia podstawowe

```
a = w10.Azym();
```

spowoduje przypisanie zmiennej a azymutu wektora w10.

5. Metoda `Wektor3::Elev(void)`

Metoda ta ma zwracać kąt elewacji ε wektora wyrażony w stopniach. Kąt elewacji ε ma spełniać zależność: $-90^\circ \leq \varepsilon \leq 90^\circ$. Przykładowo instrukcja

```
e = w10.Elev();
```

spowoduje przypisanie zmiennej e elewacji wektora w10.

6. Metoda `Wektor3& Wektor3::Plus(const Wektor3 &w)`

Metoda ta ma obliczyć sumę wektora, z którego jest wywołana i wektora w. Metoda ta powinna zwrócić referencję do wektora, z którego jest wywołana (do „siebie”). Suma ma być umieszczona w wektorze, z którego metodę wywołano. Przykładowo instrukcja

```
w10.Plus(w8);
```

spowoduje dodanie wektora w8 do wektora w10 i umieszczenie wyniku w wektorze w10. Metoda ta działa analogicznie do standardowego operatora `+=` języka C++.

7. Metoda `Wektor3& Wektor3::Mult(double k)`

Metoda ta ma obliczyć iloczyn wektora, z którego jest wywołana i liczby rzeczywistej k. Metoda ta powinna zwrócić referencję do wektora, z którego jest wywołana (do „siebie”). Iloczyn ma być umieszczony w wektorze, z którego metodę wywołano. Przykładowo instrukcja

```
w10.Mult(x);
```

spowoduje pomnożenie wektora w10 przez wartość zmiennej x i umieszczenie wyniku w wektorze w10. Metoda ta działa analogicznie do standardowego operatora `*` języka C++.

8. Funkcja `Wektor3 Plus(const Wektor3 &w1, const Wektor3 &w2)`

Metoda ta ma obliczyć sumę wektorów w1 i w2. Funkcja ta ma zwrócić wektora będący sumą wektorów składowych. Przykładowo instrukcja

```
Plus(w8, w9);
```

spowoduje dodanie wektorów w8 i w9. Funkcja ta działa analogicznie do standardowego operatora `+` języka C++. Natomiast instrukcja

```
w10.Assign(Plus(w1, w2));
```

działa analogicznie jak standardowa instrukcja C++:

```
w10 = w1 + w2;
```

9. Funkcja `Wektor3 Mult(double k, const Wektor3 &w)`

Metoda ta ma obliczyć iloczyn wektora w i liczby k. Funkcja ta ma zwrócić wektora będący iloczynem argumentów. Przykładowo instrukcja:

```
Mult(z, w9);
```

spowoduje obliczenie iloczynu wektora w9 i wartości zmiennej z. Funkcja ta działa analogicznie do standardowego operatora `*` języka C++. Dla przykładu, instrukcja:

```
w10.Assign(Mult(z, w2));
```

działa analogicznie jak standardowa instrukcja C++:

```
w10 = z * w2;
```

10. Funkcje `bool Eq(const Wektor3 &w1, const Wektor3 &w2)`

Metoda ta ma testować równość wektorów w1 i w2 i zwracać prawdę oba wektory są równe. Przykładowo instrukcja

```
Eq(w8, w9);
```

spowoduje sprawdzenie równości wektorów w8 i w9 (wynik porównania jest porzucony). Funkcja ta działa analogicznie do standardowego operatora `==` języka C++. Natomiast instrukcja:

```
b = Eq(w1, w2);
```

działa analogicznie jak standardowa instrukcja C++:

```
b = w1 == w2;
```

11. Funkcje `bool Neq(const Wektor3 &w1, const Wektor3 &w2)`

Metoda ta ma testować nierówność wektorów w1 i w2 i zwracać prawdę oba wektory nie są równe. Przykładowo instrukcja

```
Neq(w8, w9);
```

spowoduje sprawdzenie nierówności wektorów w8 i w9 (wynik porównania jest porzucony). Funkcja ta działa analogicznie do standardowego operatora `!=` języka C++. Natomiast instrukcja:

Język C++. Klasy. Pojęcia podstawowe

```
b = Neq(w1, w2);
```

działa analogicznie jak standardowa instrukcja C++:

```
b = w1 != w2;
```

Funkcje matematyczne

Do napisania kodu niektórych metod i funkcji klasy wektor potrzebne mogą być funkcje matematyczne. Poniżej wyszczególnionych jest kilka przydatnych funkcji wraz z prototypami i krótkim komentarzem:

- kwadrat liczby
Kwadrat liczby najefektywniej realizuje się poprzez mnożenie ($x * x$).
- pierwiastek kwadratowy
`double sqrt(double x);`
Argument x musi być nieujemny ($x \geq 0$).
- sinus
`double sin(double x);`
Argument x musi być wyrażony w radianach.
- kosinus
`double cos(double x);`
Argument x musi być wyrażony w radianach.
- tangens
`double tan(double x);`
Argument x musi być wyrażony w radianach.
- arcus sinus
`double asin(double x);`
Argument x musi spełniać warunek $-1 \leq x \leq 1$. Funkcja ta zwraca wartość z przedziału $[-\pi/2, \pi/2]$
- arcus cosinus
`double acos(double x);`
Argument x musi spełniać warunek $-1 \leq x \leq 1$. Funkcja ta zwraca wartość z przedziału $[0, \pi]$
- arcus tangens
`double atan(double x);`
Funkcja ta zwraca wartość z przedziału $[-\pi/2, \pi/2]$

Załącznik 1. Kod źródłowy programu do uzupełnienia (plik PI_Cpp03.cpp)

```
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <iomanip>

using namespace std;

//=====

// Deklaracja klasy Wektor3

class Wektor3
{
public:
    Wektor3( void );
    Wektor3( double x, double y, double z );
    Wektor3( const Wektor3 &w );

    double X( void ) { return _x; }
    double Y( void ) { return _y; }
    double Z( void ) { return _z; }

    Wektor3& Assign( double x, double y, double z );
    Wektor3& Assign( const Wektor3 &w );

    double Mod( void );
    double Azym( void );
    double Elev( void );

    Wektor3& Plus( const Wektor3 &w );
    Wektor3& Mult( double k );

    friend Wektor3 Plus( const Wektor3 &w1, const Wektor3 &w2 );
    friend Wektor3 Mult( double k, const Wektor3 &w );

    friend bool Eq( const Wektor3 &w1, const Wektor3 &w2 );
    friend bool Neq( const Wektor3 &w1, const Wektor3 &w2 );

private:
    double _x, _y, _z;
};

//=====

// Definicje konstruktorow, metod i funkcji zaprzyjzaznionych

Wektor3::Wektor3( void )
{
    _x = _y = _z = 0.0;
}
//-----

Wektor3::Wektor3( double x, double y, double z )
{
    _x = x; _y = y; _z = z;
}
//-----
```

Język C++. Klasy. Pojęcia podstawowe

```
Wektor3::Wektor3( const Wektor3 &w )
//Do uzupełnienia

//-----

Wektor3& Wektor3::Assign( double x, double y, double z )
{
    _x = x; _y = y; _z = z;

    return *this;
}
//-----

Wektor3& Wektor3::Assign( const Wektor3 &w )
//Do uzupełnienia

//-----

double Wektor3::Mod( void )
//Do uzupełnienia

//-----

double Wektor3::Azym( void )
//Do uzupełnienia

//-----

double Wektor3::Elev( void )
//Do uzupełnienia

//-----

Wektor3& Wektor3::Plus( const Wektor3 &w )
//Do uzupełnienia

//-----

Wektor3& Wektor3::Mult( double k )
//Do uzupełnienia

//-----

Wektor3 Plus( const Wektor3 &w1, const Wektor3 &w2 )
//Do uzupełnienia

//-----

Wektor3 Mult( double k, const Wektor3 &w )
//Do uzupełnienia

//-----

bool Eq( const Wektor3 &w1, const Wektor3 &w2 )
//Do uzupełnienia

//-----

bool Neq( const Wektor3 &w1, const Wektor3 &w2 )
//Do uzupełnienia
//=====
```

Język C++. Klasy. Pojęcia podstawowe

```
int main(int argc, char *argv[])
{
    Wektor3      w1( 1, 2, 3 );
    Wektor3      w3, w4, w5;
    double       ax, ay, az, k;

    cout << "Wektor w3 (x, y, z): ";
    cin >> ax >> ay >> az;
    w3.Assign( ax, ay, az );

    cout << "Wektor w4 (x, y, z): ";
    cin >> ax >> ay >> az;
    w4.Assign( ax, ay, az );

    cout << "Wspolczynnik k: ";
    cin >> k;

    Wektor3      w31( w3 );

    cout << fixed << setprecision( 3 ) << endl;
    cout << "Wektor w1 = ( " << w1.X( ) << ", " << w1.Y( ) << ", " <<
        w1.Z( ) << " )" << endl;
    cout << "Wektor w3 = ( " << w3.X( ) << ", " << w3.Y( ) << ", " <<
        w3.Z( ) << " )" << endl;
    cout << "Kopia w3 = ( " << w31.X( ) << ", " << w31.Y( ) << ", " <<
        w31.Z( ) << " )" << endl;
    cout << "Wektor w4 = ( " << w4.X( ) << ", " << w4.Y( ) << ", " <<
        w4.Z( ) << " )" << endl << endl;

    w5.Assign( w3 );
    w5.Plus( w4 );
    cout << "Wektor w3+w4 = ( " << w5.X( ) << ", " << w5.Y( ) << ", " <<
        w5.Z( ) << " )" << endl;

    w5.Assign( Plus( w1, w3 ) );
    cout << "Wektor w1+w3 = ( " << w5.X( ) << ", " << w5.Y( ) << ", " <<
        w5.Z( ) << " )" << endl << endl;

    w5.Assign( w3 );
    w5.Mult( k );
    cout << "Wektor k*w3 = ( " << w5.X( ) << ", " << w5.Y( ) << ", " <<
        w5.Z( ) << " )" << endl ;

    w5.Assign( Mult( k, w4 ) );
    cout << "Wektor k*w4 = ( " << w5.X( ) << ", " << w5.Y( ) << ", " <<
        w5.Z( ) << " )" << endl << endl ;

    cout << "Modul w3 = " << w3.Mod( ) << endl;
    cout << "Azymut w3 = " << w3.Azym( ) << endl;
    cout << "Elewacja w3 = " << w3.Elev( ) << endl << endl;

    cout << "w3==w4 " << ( Eq( w3, w4 ) ? "tak" : "nie" ) << endl << endl;
    cout << "w3!=w4 " << ( Neq( w3, w4 ) ? "tak" : "nie" ) << endl << endl;

    return 0;
}
```