

Język C i C++ w programach sterowania

Część 2. Język C++

Elektrotechnika IV rok
Studia dzienne

Materiały do wykładu

JĘZYK C++ – WSTĘP

Język C przyjęto jako bazę języka C++, ponieważ:

1. jest uniwersalny, zwięzły i stosunkowo niskopoziomowy,
2. sprawdza się w wielu zastosowaniach programistycznych,
3. jest dostępny na wielu platformach sprzętowych i systemowych,
4. pasuje do środowiska systemu UNIX.

CO NOWEGO W JĘZYKU C++ ?

1. typ **bool**,
2. referencje,
3. operatory **new**, **delete**,
4. klasy jako narzędzie do tworzenia abstrakcyjnych typów danych,
5. szablony funkcji i klas (ang. templates),
6. przeciążanie nazw funkcji (ang. overload functions),
7. argumenty domyślne funkcji (ang. default arguments),
8. funkcje operatorowe (ang. operator functions),
9. nowy sposób rzutowania (ang. type casting),
10. obsługa wyjątków (ang. exception handling),
11. przestrzenie nazw (ang. name spaces)

C++ – REFERENCJE

Deklaracja

Referencja – inaczej inna nazwa obiektu (alias).

Deklaracja referencji:

```
typ & NazwaRef = NazwaObiektu;
```

Przykłady

```
int    x = 5;
int    &rx = x;
int    y = rx;           // y = 5

    rx = 8;             // x =8;
    rx++;              // x++;

int    &ry;             // BŁĄD - brak inicjacji
extern int&rz;         // OK - rz inicjowane gdzie indziej
int    &rr = 6;        // BŁĄD - 6 nie jest l-wartością
```

```
int    fun1(int &x);
int    fun2(const int &x);
int    a = 5, b;

    b = fun1(a);       // OK.
    b = fun1(3);       // BŁĄD - argument jest stałą
    b = fun2(a);       // OK.
    b = fun2(4);       // OK - tworzony jest obiekt tymczas.
```

PRZECIĄŻANIE NAZW FUNKCJI

Ma miejsce wtedy, gdy istnieje kilka funkcji o tej samej nazwie różniących się listą argumentów tj. typem argumentów i/lub liczbą argumentów.

Reguły dopasowywania argumentów

1. Zgodność typów bez promowania i konwersji:

```

typ x[]      → typ*
typ f()      → typ (*) ()
typ          → const typ
    
```

2. Proste promowanie:

```

bool                → int
(unsigned) char     → (unsigned) int
(unsigned) short int → (unsigned) int
float               → double
    
```

3. Standardowe konwersje:

```

int                → double
int                → long int
int                → unsigned int
unsigned int       → int
double             → int
double             → long double
typ*              → void*
pochodna*        → podstawowa*
    
```

4. Konwersje zdefiniowane przez programistę.

5. Zgodność z zastosowanie wielokropka w deklaracji funkcji.

Przykład 1

```
void fun1(double);
void fun1(long int);
//-----
fun1(1L);           // OK - fun1(long int)
fun1(1.0);         // OK - fun1(double)
fun1(1);           // BŁĄD - niejednoznaczność
```

Przykład 2

```
void fun2(int);
void fun2(const char*);
void fun2(double);
void fun2(long int);
void fun2(char);

char c;
int i;
short s;
float f;
//-----
fun2(c);           // OK - fun2(char)
fun2(i);           // OK - fun2(int)
fun2(s);           // OK - promocja short->int - fun2(int)
fun2(f);           // OK - promocja float->double - fun2(double)
fun2(0);           // OK - fun2(int)
fun2(0L);          // OK - fun2(long int)
fun2('a');         // OK - fun2(char)
fun2("a");         // OK - fun2(const char*)
```

Przykład 3

```
void fun3(int);  
void fun3(int, int);  
void fun3(double, int);  
void fun3(int, double);  
  
//-----  
fun3(0); // OK - fun3(int)  
fun3(3, 8); // OK - fun3(int, int)  
fun3(3.0, 8); // OK - fun3(double, int)  
fun3(3.2, 5.1); // BŁĄD - niejednoznaczność
```

Przykład 4

```
int fun4(int);  
double fun4(int); // BŁĄD - taka sama lista argumentów
```

ARGUMENTY DOMYŚLNE FUNKCJI

Przykład 1

```
int    fun1(int x = 5);
//-----
fun1(6);           // fun1(6)
fun1();           // fun1(5)
```

Przykład 2

```
void   fun2(int x, double y = 3.14, char *z = "pi");
//-----
fun2(8);           // fun2(8, 3.14, "pi")
fun2(6, 2.71);    // fun2(6, 2.71, "pi")
fun2(4, 2.71, "e"); // fun2(4, 2.71, "e")
fun2();           // BŁĄD
```

Przykład 3

```
void   fun3(int x = 0, double y); // BŁĄD
```

Przykład 4

```
void   fun4(int x);
void   fun4(int x, int y = 10);
//-----
fun4(3, 8);       // fun4(int, int)
fun4(2);          // BŁĄD - niejednoznaczność
```

KLASY

Ideę klasy przedstawiono na przykładzie obiektu do przechowywania daty i zestawu funkcji do operowania na obiekcie 'data'.

Implementacja „w duchu” języka C

Definiujemy strukturę `Data` do przechowywania danych i zestaw funkcji do operowania na strukturze `Data`. W przykładzie wykorzystano możliwość przeciążenia nazwy funkcji (funkcja `DataSet()`)

```
struct Data {
    int    d, m, r;
};
//-----
void    DataSet(Data &d, int dd, int mm, int rr);
void    DataSet(Data &d, int dd, int mm);
void    DataSet(Data &d, int dd);
int     DataDzien(Data &d);
int     DataMiesiac(Data &d);
int     DataRok(Data &d);
```

Implementacja „bardziej w duchu” języka C++

Funkcje do operowania na strukturze `Data` deklarujemy jako metody struktury. Można do nich odwoływać stosując operator wyboru elementu struktury (`.` lub `->`), co powoduje skojarzenie metody z danymi, na rzecz których jest ona wywołana. Należy zwrócić uwagę na brak argumentu typu `Data&` w tych metodach. Argument ten jest niejawnie przekazywany do metody i jest w niej dostępny jako wskaźnik `this`. W przykładzie wykorzystano możliwość przeciążenia nazwy funkcji (metoda `Set()`)

```
struct Data {
    int    d, m, r;

    void    Set(int dd, int mm, int rr);
    void    Set(int dd, int mm);
    void    Set(int dd);
    int     Dzien(void);
    int     Miesiac(void);
    int     Rok(void);
};
```

Implementacja „jeszcze bardziej w duchu” języka C++

Użycie słowa kluczowego `class` zamiast `struct` powoduje domyślne „uprywatnienie” pól klasy. Są one wtedy dostępne tylko dla metod klasy; nie można się do nich odwoływać z zewnątrz. W przykładzie zastosowano funkcję `Set()` z domyślnymi argumentami zamiast przeciążania jej nazwy.

```
class Data {
    int    d, m, r;

public:
    void   Set(int dd, int mm = 0, int rr = 0);
    int    Dzien(void);
    int    Miesiac(void);
    int    Rok(void);
};
```

Implementacja 1. w duchu języka C++

Klasę `Data` uzupełniono o konstruktor.

Konstruktor jest specjalną metodą mającą nazwę identyczną jak nazwa klasy i jest wywoływany automatycznie przez kompilator w momencie tworzenia klasy (deklarowania lub tworzenia za pomocą operatora `new`). Konstruktory mogą być przeciążane.

Klasa może zawierać również destruktor (tylko jeden). Ma on nazwę identyczną jak nazwa klasy poprzedzoną znakiem tyldy (`~Data()`).

Destruktor jest wywoływany automatycznie przez kompilator w momencie niszczenia klasy (wyjścia z zasięgu zmiennej lub usuwania za pomocą operatora `delete`).

Konstruktor ten pozwala inicjować obiekt typu `Data` w momencie deklaracji. Ponieważ wymaga on co najmniej jednego argumentu, nie jest możliwe zadeklarowanie obiektu typu `Data` bez inicjowania. Aby taka deklaracja była możliwa trzeba zadeklarować drugi konstruktor bez argumentu (`Data(void)`).

```
class Data {
    int    d, m, r;

public:
    Data(int dd, int mm = 0, int rr = 0);
    void   Set(int dd, int mm = 0, int rr = 0);
    int    Dzien(void);
    int    Miesiac(void);
    int    Rok(void);
};
```

Implementacja 2. w duchu języka C++

Deklaracja uzupełniona o drugi konstruktor. Część dostępna dla użytkownika (publiczna) jest przesunięta na początek deklaracji. Część zawierająca dane, która nie powinna być bezpośrednio dostępna dla użytkownika jest przesunięta na koniec. W celu uniemożliwienia bezpośredniego dostępu do niej konieczne jest użycie klauzuli `private:`.

```
class Data {  
public:  
    Data(int dd, int mm = 0, int rr = 0);  
    Data(void);  
    void Set(int dd, int mm = 0, int rr = 0);  
    int Dzien(void);  
    int Miesiac(void);  
    int Rok(void);  
private:  
    int    d, m, r;  
};
```

Metody inline i metody stałe

Metody (i ogólnie funkcje) `inline` to takie, których kod jest wstawiany bezpośrednio w miejscu wywołania. Pozwala to na przyspieszenie wykonywania programu; zwiększa jednak wielkość kodu.

Metody stałe to takie mogą być wywoływane na rzecz obiektu stałego (zadeklarowanego z atrybutem `const`).

Przykład deklaracji

```
class Data {
    int    d, m, r;
public:
    Data(int dd, int mm = 0, int rr = 0);
    void Set(int dd, int mm = 0, int rr = 0);
    int  Dzień(void) const;           // stała
    int  Miesiąc(void) { return m; } // inline
    int  Rok(void) const;           // stała
};
//-----
int Data::Dzień(void) const          // nie inline ale const
{
    return d;
}
//-----
inline int Data::Rok(void) const     // inline i const
{
    return r;
}
```

Przykład

```
Data          d1(16);
const Data    d2(1, 9, 1939);
int           x;

d1.Set(1, 1, 2000);           // Ok.
x = d1.Dzień();              // Ok.

d2.Set(1, 2, 2002);           // Błąd !
x = d2.Dzień();              // Ok.
```

Składowe statyczne

Pola danych i metody statyczne występują w jednym egzemplarzu dla wszystkich instancji klasy danego typu. Metody statyczne nie mają z tego powodu dostępu do wskaźnika `this`. Statyczne pola danych muszą deklarowane niezależnie poza klasą.

Przykład

```
class Data {
    int    d, m, r;
    static Data  dataDft;

public:
    Data(int dd = 0, int mm = 0, int rr = 0);
    void Set(int dd, int mm = 0, int rr = 0);
    int Dzien(void) const { return d; }
    int Miesiac(void) const { return m; }
    int Rok(void) const { return r; }
    static void UstawDft(int dd, int mm, int rr);
};

//----- deklaracja / definicje składowych statycznych
Data Data::dataDft = Data(1, 1, 1970);

void Data::UstawDft(int dd, int mm, int rr)
{
    Data::dataDft = Data(dd, mm, rr);
}

//----- definicja konstruktora
Data::Data(int dd, int mm, int rr)
{
    d = dd > 0 ? dd : dataDft.d;
    m = mm > 0 ? mm : dataDft.m;
    r = rr > 0 ? rr : dataDft.r;
}
```

Przykład 1 – definicja klasy wektor

Przykład klasy Wektor służącej do przechowywania wartości typu int. Klasa ta posiada dwa konstruktory i destruktor.

```

class Wektor {
    int    size;
    int    *arr;

public:
    Wektor(int ASize = 10);           // konstruktor 1
    Wektor(Wektor &Wekt);           // konstruktor 2
    ~Wektor();                       // destruktor
    int Get(int i) const { return arr[i]; } // metoda
    void Set(int i, int x) { arr[i] = x; } // metoda
};

//-----

Wektor::Wektor(int ASize)
{
    size = ASize;
    arr = new int[size];
}

//-----

Wektor::Wektor(Wektor &Wekt)
{
    size = Wekt.size;
    arr = new int[size];
    for(int i = 0; i < size; i++) arr[i] = Wekt.arr[i];
}

//-----

Wektor::~~Wektor()
{
    delete[] arr;
}

```

Przykład 2 – konstrukcja i destrukcja klasy

```

Wektor w0(100); // Wektor::Wektor(int)

// .....

void fun( ... )
{
    Wektor w1; // Wektor::Wektor(int)
    Wektor w2 = w1; // Wektor::Wektor(int) !!
    Wektor w3(w1); // Wektor::Wektor(Wektor&)
// ...
    Wektor *pw1 = new Wektor(25); // Wektor::Wektor(int)
    Wektor *pw2 = new Wektor(w1); // Wektor::Wektor(Wektor&)

// ...

    delete pw1; // Wektor::~~Wektor()
    delete pw2; // Wektor::~~Wektor()
// ...

} // Wektor::~~Wektor() (3 razy)

```

FUNKCJE OPERATOROWE

Wprowadzenie

- Można definiować znaczenie następujących operatorów:

+	-	*	/	%	
^	&		~	>>	<<
++	--	=			
!	&&				
<	>	<=	>=	==	!=
+=	-=	*=	/=	%=	
^=	&=	=	<<=	>>=	
->*	,	->	[]	()	
new	new []	delete	delete []		

- Nie można definiować znaczenia następujących operatorów:
:: . (kropka) .*
- Jednym z argumentów funkcji operatorowej musi być klasa lub wyliczenie.
- Nie można zmieniać priorytetu i wiązania operatorów.
- Nie można definiować nowych symboli operatorów.
- Nie można zmieniać liczby argumentów operatorów.

Deklaracja

Poza klasą:

```
typ operator @ (typ1 arg1);           // jednoargumentowy
typ operator @ (typ1 arg1, typ2 arg2); // dwuargumentowy
```

Wewnątrz klasy:

```
class KLASA {
// ...
    typ operator @ (void);           // jednoargumentowy
    typ operator @ (typ1 arg1);     // dwuargumentowy
};
```

Przykład 1 – definicja 1 operatorów + i *

W klasie `Wektor` definiujemy funkcje operatorowe `+` i `*` jako metody klasy.

```
class Wektor {
// ...
    void operator + (Wektor &w);
    void operator * (int k);
}

//-----

void Wektor::operator + (Wektor &w)
{
    int max = size < w.size ? size : w.size;
    for(int i = 0; i < max; i++) arr[i] += w.arr[i];
}

//-----

void Wektor::operator * (int k)
{
    for(int i = 0; i < size; i++) arr[i] *= k;
}
```

```
Wektor W1(30), W2(15);
```

```
W1 + W2;           // W1.operator +(W2)
W2 + W1;           // W2.operator +(W1)
W1 * 10;           // W1.operator *(10)
10 * W1;           // BŁĄD. Nie ma funkcji
                    // int.operator * (Wektor&)
```

Przykład 2 – definicja 2 operatora *

Definiujemy funkcję operatorową `*` dla argumentów `int` i `Wektor&` na zewnątrz klasy `Wektor`.

```
void operator * (int k, Wektor &w)
{
    for(int i = 0; i < w.size; i++) w.arr[i] *= k;
}
```

Pojawią się błędy w czasie kompilacji z powodu żądania dostępu do elementów prywatnych klasy.

Aby umożliwić zewnętrznej funkcji dostęp do prywatnych elementów klasy należy ją **zaprzyjaźnić** z klasą.

```
class Wektor {
// ...
    void operator + (Wektor &w);
    void operator * (int k);
    friend void operator * (int k, Wektor &w);
}
```

Przykład 3 – definicja operatora =

W wyrażeniu poniżej instrukcja przypisania nie zadziała zgodnie z oczekiwaniami. Nie zostaną przekopiuwane zawartości tablic `arr`.

```
Wektor w1(20), w2(20);
// ...
w1 = w2;           // zadziała „niezbyt” poprawnie.
```

Można zdefiniować operator `=` dla klasy **Wektor**.

```
class Wektor {
// ...
    Wektor & operator = (Wektor &w);
    Wektor & operator = (int c);
}

//-----

Wektor & Wektor::operator = (Wektor &w)
{
    int max = size < w.size ? size : w.size;
    for(int i = 0; i < size; i++) arr[i] = w.arr[i];
    return *this;
}

//-----

Wektor & Wektor::operator = (int c);
{
    for(int i = 0; i < size; i++) arr[i] = c;
    return *this;
}
```

```
Wektor x1(15), x2(15);

x1 = 7;           // Wektor::operator = (int)
x2 = x1;         // Wektor::operator = (Wektor&)
x1 = x2 = 11;    // oba wymienione powyżej
```

Przykład 4 – definicja operatora []

W klasie `Wektor` można zdefiniować operator `[]` w celu „naturalnego” dostępu do elementów wektora.

```
class Wektor {
// ...
    int & operator [] (int i);
}

//-----

int & Wektor::operator [] (int i)
{
    return arr[i];
}
```

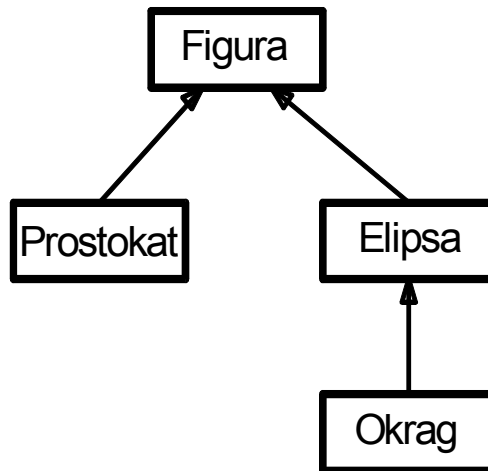
```
Wektor x1(15), x2(15);
int c1, c2;
// ...
c1 = x1.Get(5);
c2 = x2[5];

x1.Set(c1, c2);
x1[c1] = c2;           // operator musi zwracać referencję
```

KLASY POCHODNE

Budujemy klasy `Figura`, `Prostokat`, `Elipsa` i `Okrąg` reprezentujące figury geometryczne: prostokąt, elipsę i okrąg. Klasa `Figura` jest klasą podstawową dla pozostałych klas. Kolejne klasy w hierarchii dziedziczą cechy klasy bazowej. Cechy, którymi różnią się poszczególne figury, są modyfikowane w danej klasie.

Hierarchia



Klasa `Figura`

```

class Figura {
    int FillMode;
public:
    Figura(void) : FillMode(0) { };
    void FillOn(void) { FillMode = 1; }
    void FillOff(void) { FillMode = 0; }
    int Rysuj(void) {return 0; }
    void Przesun(int dx, int dy) { }
    double Pole(void) { return 0.0; }
};
  
```

Klasa `Prostokat`

```

class Prostokat : public Figura {
    int x1, y1, x2, y2;
public:
    Prostokat(int ax1, int ay1, int ax2, int ay2);
    int Rysuj(void);
    void Przesun(int dx, int dy);
    double Pole(void);
};
  
```

```
Prostokat::Prostokat(int ax1, int ay1, int ax2, int ay2) :
    Figura(), x1(ax1), y1(ay1), x2(ax2), y2(ay2)
{
}
```

```
double Prostokat::Pole(void)
{
    return fabs((x2 - x1) * (y2 - y1));
}
```

Klasa Elipsa

```
class Elipsa : public Figura {
    int x0, y0, rx, ry;
public:
    Elipsa(int ax0, int ay0, int arx, int ary);
    int Rysuj(void);
    void Przesun(int dx, int dy);
    double Pole(void);
};
```

```
Elipsa::Elipsa(int ax0, int ay0, int arx, int ary) :
    Figura(), x0(ax0), y0(ay0), rx(arx), ry(ary)
{
}
```

```
double Elipsa::Pole(void)
{
    return 3.1415 * rx * ry;
}
```

Klasa Okrag

```
class Okrag : public Elipsa {
public:
    Okrag(int ax0, int ay0, int ar);
};
```

```
Okrag::Okrag(int ax0, int ay0, int ar) :
    Elipsa(ax0, ay0, ar, ar)
{
}
```

FUNKCJE WIRTUALNE

Przykład 1

Na bazie klas zdefiniowanych w poprzednim rozdziale tworzymy kolekcję figur.

```
Figura      *F[10];
double      S[10];

F[0] = new Prostokat(0, 0, 10, 20);
F[1] = new Prostokat(2, 5, 20, 25);
F[2] = new Elipsa(1, 2, 10, 7);
F[3] = new Okrag(3, 5, 10);

// Rysujemy figury
for(int i = 0; i < 4; i++)
    F[i]->Rysuj();                // źle działa !

// Obliczamy pola
for(int i = 0; i < 4; i++)
    S[i] = F[i]->Pole();          // źle działa !
```

W liniach zawierających komentarz „źle działa” wywoływane są metody `Rysuj()` i `Pole()` klasy `Figura`, a nie odpowiednich klas `Prostokat`, `Elipsa` i `Okrag`.

Przykład 2

Przykład pokazuje jak wywoływane są funkcje wirtualne i niewirtualne przy odwoływaniu się do nich poprzez wskaźniki.

```
class K1 {
    int    x;

public:
    void f1(void);
    void f2(void);
    virtual void vf1(void);
    virtual void vf2(void);
};

//-----
```

```

class K2 : public K1{
    int    y;
public:
    void f1(void);
    void f2(void);
    virtual void vf1(void);
    virtual void vf2(void);
};

//-----

K1 *c1 = new K1;
K2 *c2 = new K2;
K1 *cptr;

c1->f1();           // K1::f1();
c1->f2();           // K1::f2();
c1->vf1();          // K1::vf1();
c1->vf2();          // K1::vf2();

c2->f1();           // K2::f1();
c2->f2();           // K2::f2();
c2->vf1();          // K2::vf1();
c2->vf2();          // K2::vf2();

cptr = c1;
cptr->f1();          // K1::f1();
cptr->f2();          // K1::f2();
cptr->vf1();         // K1::vf1();
cptr->vf2();         // K1::vf2();

cptr = c2;
cptr->f1();          // K1::f1();
cptr->f2();          // K1::f2();
cptr->vf1();         // K2::vf1();
cptr->vf2();         // K2::vf2();

```

Przykład 3

Modyfikujemy klasy `Figura`, `Prostokat`, `Elipsa` i `Okrag` zdefiniowane w poprzednim rozdziale tak, aby wyeliminować nieprawidłowe zachowanie pokazane w przykładzie 1.

Klasa `Figura`

```
class Figura {
    int    FillMode;

public:
    Figura(void) : FillMode(0) { };
    void FillOn(void) { FillMode = 1; }
    void FillOff(void) { FillMode = 0; }
    virtual int Rysuj(void) = 0;           // !!
    virtual void Przesun(int dx, int dy) = 0; // !!
    virtual double Pole(void) = 0;       // !!
};
```

Metody `Rysuj()`, `Przesun()` i `Pole()` są czystymi funkcjami wirtualnymi. Powoduje to, że nie można stworzyć instancji klasy, w której znajdują się takie metody. Klasa taka stanowić może być klasą bazową dla innych klas, w których czyste funkcje wirtualne muszą być zdefiniowane.

Klasa `Prostokat`

```
class Prostokat : public Figura {
    int    x1, y1, x2, y2;

public:
    Prostokat(int ax1, int ay1, int ax2, int ay2);
    virtual int Rysuj(void);
    virtual void Przesun(int dx, int dy);
    virtual double Pole(void);
};

Prostokat::Prostokat(int ax1, int ay1, int ax2, int ay2) :
    Figura(), x1(ax1), y1(ay1), x2(ax2), y2(ay2)
{
}

double Prostokat::Pole(void)
{
    return fabs((x2 - x1) * (y2 - y1));
}
```

Klasa Elipsa

```

class Elipsa : public Figura {
    int    x0, y0, rx, ry;
public:
    Elipsa(int ax0, int ay0, int arx, int ary);
    virtual int Rysuj(void);
    virtual void Przesun(int dx, int dy);
    virtual double Pole(void);
};

Elipsa::Elipsa(int ax0, int ay0, int arx, int ary) :
    Figura(), x0(ax0), y0(ay0), rx(arx), ry(ary)
{
}

double Elipsa::Pole(void)
{
    return 3.1415 * rx * ry;
}

Klasa Okrag

class Okrag : public Elipsa {
public:
    Okrag(int ax0, int ay0, int ar);
};

Okrag::Okrag(int ax0, int ay0, int ar) :
    Elipsa(ax0, ay0, ar, ar)
{
}

```

Implementacja

```
class K1 {
    int    x;
public:
    void f2(void);
    virtual void vf1(void);
    virtual void vf2(void);
};
//-----
```

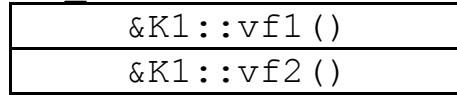
```
class K2 : public K1{
    int    y;
public:
    void f1(void);
    virtual void vf1(void);
    virtual void vf2(void);
};
//-----
```

```
K2 *c2 = new K2;
```

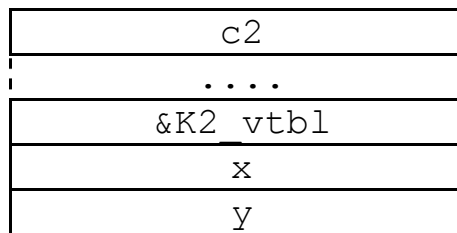
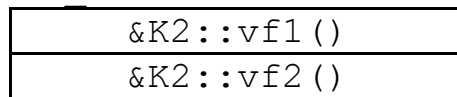
```
c2->f1 ();
```

```
c2->vf2 ();
```

K1 vtbl



K2 vtbl



```
call K2::f1
```

```
mov edx, c2
mov eax, [edx]
call dword ptr +4H[eax]
```

WZORCE FUNKCJI

Przykład deklaracji funkcji `cmp()` o sparametryzowanym typie argumentów i drugiej, która posiada inny algorytm dla argumentów typu `char*`.

```
template <class T>
int cmp(T x1, T x2)
{
    if(x1 > x2) return 1;
    if(x1 < x2) return -1;
    return 0;
}

int cmp(char *x1, char *x2)
{
    return strcmp(x1, x2);
}

//-----
cmp(1, 6);           // cmp<int>(1, 6)
cmp(4.23, 5.12);    // cmp<double>(4.23, 5.12)
cmp("abc", "xyz");  // cmp(char *, char *)
cmp(2, 4.23);       // BŁĄD - niejednoznaczność.
```

Sposoby rozstrzygnięcia niejednoznaczności:

- jawne wywołanie konkretnej funkcji:

```
cmp<int>(2, 4.23);    // cmp<int>(2, (int)4.23);
cmp<double>(2, 4.23);
// cmp<double>((double)2, 4.23);
```

- dodatkowe deklaracje (wyszczególnienie konkretnych przypadków):

```
inline double cmp(int x1, double x2)
{
    return cmp<double>(x1, x2);
}

inline double cmp(double x1, int x2)
{
    return cmp<double>(x1, x2);
}
```

WZORCE KLAS

Przykład deklaracji klasy Wektor o sparametryzowanym typie T.

```

template <class T>
class Wektor {
    int    size;
    T      *arr;

public:
    Wektor(int ASize = 10);
    Wektor(Wektor &Wekt);
    ~Wektor();
    T Get(int i) const { return arr[i]; }
    void Set(int i, T x) { arr[i] = x; }
};

//-----
template <class T> Wektor::Wektor(int ASize)
{
    size = ASize;
    arr = new T[size];
}

//-----
template <class T> Wektor::Wektor(Wektor &Wekt)
{
    size = Wekt.size;
    arr = new T[size];
    for(int i = 0; i < size; i++) arr[i] = Wekt.arr[i];
}

//-----
template <class T> Wektor::~~Wektor()
{
    delete [] arr;
}

```