Konstruktory kopiującen, metody wirtualne

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Wydział Elektroniki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Copyright©2018 Bogdan Kreczmer

Niniejszy dokument zawiera materiały do wykładu dotyczącego programowania obiektowego. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych prywatnych potrzeb i może on być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Niniejsza prezentacja została wykonana przy użyciu systemu składu LaTeX oraz stylu `beamer`*, którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Plan prezentacji

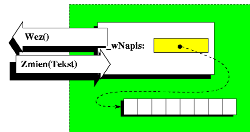
1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Definicja klasy Napis

```
class KopiaNapisu { // ..............................................
    char *_wNapis;
  public :



}; // ..............................................................
```

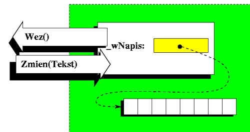
Definicja klasy Napis

```
class KopiaNapisu { //.................................................................
    char *_wNapis;
  public :



};  //...................................................................................
```



Definicja klasy Napis

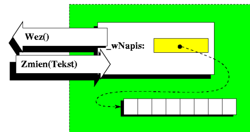
```
class KopiaNapisu { // ..............................................
    char *_wNapis;
  public :

    const char* Wez( ) const { return _wNapis; }
    void Zmien( const char* wTekst );
}; // ...............................................................
```



Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

Definicja klasy Napis

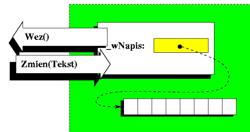
```
class KopiaNapisu { // .........................................
    char  *_wNapis;
  public :
    KopiaNapisu( )  { _wNapis = nullptr; }


    const  char* Wez( ) const  { return  _wNapis; }
    void  Zmien( const  char* wTekst );
}; // ..........................................................
```



Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

Definicja klasy Napis

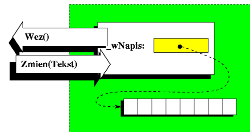
```
class KopiaNapisu { // .............................................
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }


    const char* Wez( ) const { return _wNapis; }
    void Zmien( const char* wTekst );
}; // ..............................................................


void KopiaNapisu::Zmien( const char* wTekst )
{


}
```



Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

Definicja klasy Napis

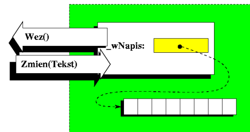
```
class KopiaNapisu { //.............................................
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }

    const char* Wez( ) const { return _wNapis; }
    void Zmien( const char* wTekst );
}; //..............................................................

void KopiaNapisu::Zmien( const char* wTekst )
 {
  delete [ ] _wNapis;  _wNapis = nullptr;

 }
```



Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

Definicja klasy Napis

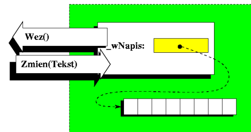
```
class KopiaNapisu { //.................................................
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }

    const char* Wez( ) const { return _wNapis; }
    void Zmien( const char* wTekst );
}; //..................................................................

void KopiaNapisu::Zmien( const char* wTekst )
 {
  delete [ ] _wNapis; _wNapis = nullptr;
  if (wTekst && (_wNapis = new char[strlen(wTekst)+1])) strcpy(_wNapis, wTekst);
 }
```



Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

Definicja klasy Napis

```
class KopiaNapisu { // ...............................................
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }
    ~KopiaNapisu( ) { delete [ ] _wNapis; }

    const char* Wez( ) const { return _wNapis; }
    void Zmien( const char* wTekst );
}; // .................................................................
```

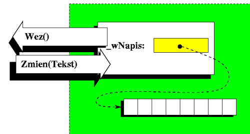



```
void KopiaNapisu::Zmien( const char* wTekst )
{
  delete [ ] _wNapis; _wNapis = nullptr;
  if (wTekst && (_wNapis = new char[strlen(wTekst)+1])) strcpy(_wNapis, wTekst);
}
```

Klasa ***KopiaNapisu*** ma dobrze zdefiniowany interfejs, który zabezpiecza ją przed przypadkowymi zmianami pola `_wNapis` oraz stowarzyszonego z nią napisu.

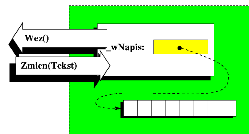
Klasa Napis w akcji

```
class KopiaNapisu { //..................................
    char *_wNapis;
  public :
      ...
}; //...................................................
```



Klasa Napis w akcji

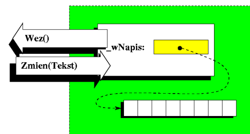
```
class KopiaNapisu { // ...................................
      char  *_wNapis;
   public :
        ...
}; // ......................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }
```

Klasa Napis w akcji

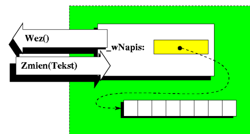
```
class KopiaNapisu { // ...................................
     char *_wNapis;
   public :
       ...
}; // ...................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{



}
```

Klasa Napis w akcji

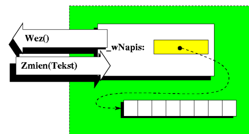
```
class KopiaNapisu  { // ...................................
      char  *_wNapis;
   public :
         ...
}; // ...................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int  main( )
{
   KopiaNapisu   Ob;



}
```

Klasa Napis w akcji

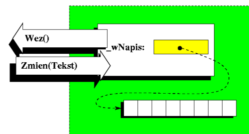
```
class KopiaNapisu { // ...................................
      char  *_wNapis;
   public :
        ...
}; // ..............................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int  main( )
{
   KopiaNapisu   Ob;
   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );



}
```

Klasa Napis w akcji

```
class KopiaNapisu { // ..................................
    char *_wNapis;
  public :
      ...
}; // ...................................................
```

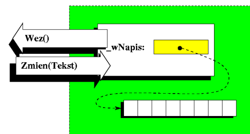



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu Ob;
  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );



}
```

Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi – (780-850) arabski matematyk urodzony w Bagdadzie, jednym z jego najważniejszych dokonań jest napisanie w 825 podręcznika pt. "*Kitab al jabr w'al-muqabala*". Słowo *algebra* pochodzi z tytułu tego podręcznika (*al jabr*). Natomiast słowo *algorytm* pochodzi od jego nazwiska. Uważał, że każdy problem matematyczny można rozwiązać w serii pojedynczych kroków.

Klasa Napis w akcji

```
class KopiaNapisu { // ...................................
      char  *_wNapis;
   public :
         ...
}; // ......................................................
```

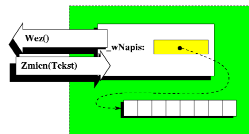



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int  main( )
{
   KopiaNapisu   Ob;

   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
   cout << "main 1: " << Ob.Wez( ) << endl;


}
```

Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi – (780-850) arabski matematyk urodzony w Bagdadzie, jednym z jego najważniejszych dokonań jest napisanie w 825 podręcznika pt. "*Kitab al jabr w'al-muqabala*". Słowo *algebra* pochodzi z tytułu tego podręcznika (*al jabr*). Natomiast słowo *algorytm* pochodzi od jego nazwiska. Uważał, że każdy problem matematyczny można rozwiązać w serii pojedynczych kroków.

Klasa Napis w akcji

```
class KopiaNapisu { // ...................................
    char *_wNapis;
  public :
      ...
}; // .......................................................
```

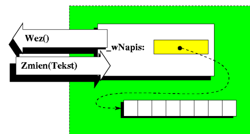



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu  Ob;
  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
  cout << "main 1: " << Ob.Wez( ) << endl;
  Wyswietl(Ob);

}
```

Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi – (780-850) arabski matematyk urodzony w Bagdadzie, jednym z jego najważniejszych dokonań jest napisanie w 825 podręcznika pt. "*Kitab al jabr w'al-muqabala*". Słowo *algebra* pochodzi z tytułu tego podręcznika (*al jabr*). Natomiast słowo *algorytm* pochodzi od jego nazwiska. Uważał, że każdy problem matematyczny można rozwiązać w serii pojedynczych kroków.

Klasa Napis w akcji

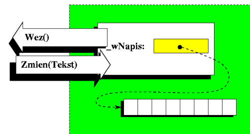
```
class KopiaNapisu { //..................................
      char *_wNapis;
   public :
      ...
}; //...................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
   KopiaNapisu Ob;
   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
   cout << "main 1: " << Ob.Wez( ) << endl;
   Wyswietl(Ob);
   cout << "main 2: " << Ob.Wez( ) << endl;
}
```

Klasa Napis w akcji

```
class KopiaNapisu { // ..................................
    char *_wNapis;
  public :
      ...
}; // ....................................................
```

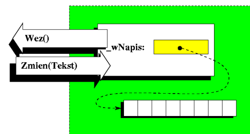



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu  Ob;

  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
  cout << "main 1: " << Ob.Wez( ) << endl;
  Wyswietl(Ob);
  cout << "main 2: " << Ob.Wez( ) << endl;
}
```

Co się wyświetli?

Klasa Napis w akcji

```
class KopiaNapisu { // ...................................
      char  *_wNapis;
   public :
        ...
}; // ...................................................
```

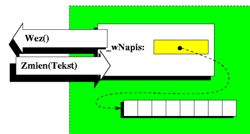



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int  main( )
{
   KopiaNapisu   Ob;
   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
   cout << "main 1: " << Ob.Wez( ) << endl;
   Wyswietl(Ob);
   cout << "main 2: " << Ob.Wez( ) << endl;
}
```

```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
```

Klasa Napis w akcji

```
class KopiaNapisu { //.................................
     char *_wNapis;
  public :
       ...
}; //..................................................
```

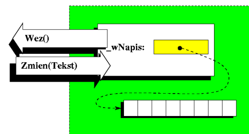



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu Ob;
  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
  cout << "main 1: " << Ob.Wez( ) << endl;
  Wyswietl(Ob);
  cout << "main 2: " << Ob.Wez( ) << endl;
}
```

```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
main 2:
```

Klasa Napis w akcji

```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ...................................................
```




```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
   KopiaNapisu  Ob;
   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
   cout << "main 1: " << Ob.Wez( ) << endl;
   Wyswietl(Ob);
   cout << "main 2: " << Ob.Wez( ) << endl;
}
```

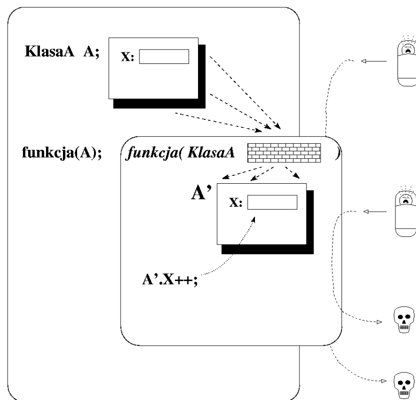
```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
main 2:
```

Co stało się z napisem?

Plan prezentacji

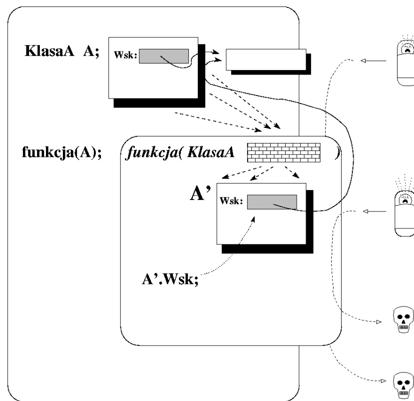
1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Jak to działa



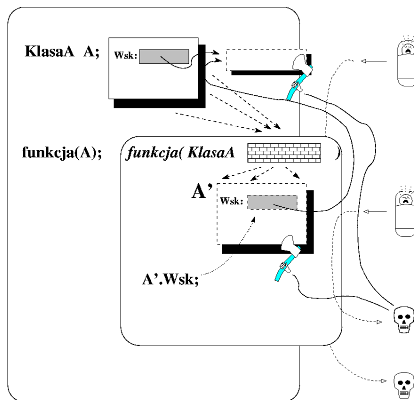
Przekazaniu parametru wywołania funkcji/metody przez wartość towarzyszy utworzenie jego kopii na poziomie tej funkcji/metody. Istnienie kopii kończy się wraz z zakończeniem działania funkcji/metody.

Jak to działa



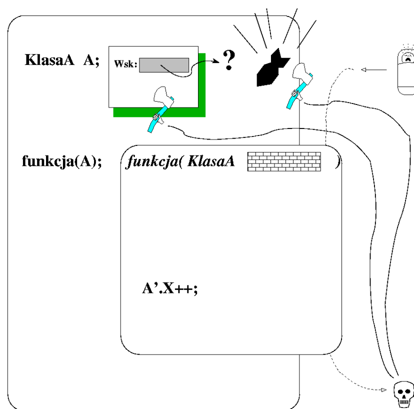
Przy tworzeniu kopii przepisana zostaje cała zawartość obiektu łącznie z zawartością pól wskaźnikowych. Powoduje to, że dwa obiekty połączone są poprzez wskaźniki z tymi samymi strukturami.

Jak to działa



Destrukcja kopii obiektu pociąga za sobą destrukcję dynamicznych struktur z nim stowarzyszonych. Usunięte zostają w ten sposób struktury, które pierwotnie należały do oryginału.

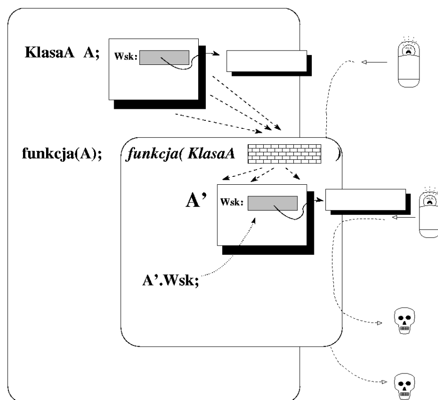
Jak to działa



Destrukcja oryginału, po wcześniejszym niekontrolowanym usunięciu struktur stowarzyszonym z danym obiektem, prowadzi do nieprzewidzianych skutków

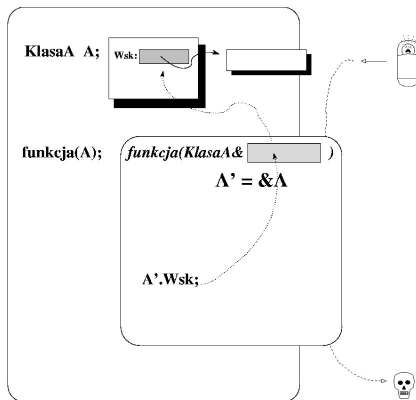
(zwykle jest to: `segmentation fault` :-) .

Jak to działa



Implementacja konstruktora kopiującego pozwala na poprawne powielenie struktur stowarzyszonych z oryginalnym obiektem. Tym samym zapewnia prawidłową późniejszą destrukcję takiego obiektu.

Przekazywanie obiektu przez referencję



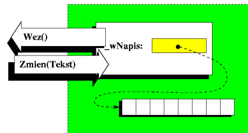
Przekazywanie obiektu przez referencję pozwala uniknąć tworzenia kopii obiektu, która czasem może być zbędna.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Kopiowanie zawartości

```
class KopiaNapisu { // ..........................................
    char *_wNapis;
  public :
      ...
      ...
      ...
}; // ...........................................
```



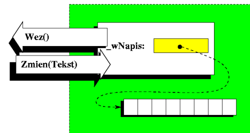

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu Ob;
  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
  cout << "main 1: " << Ob.Wez( ) << endl;
  Wyswietl(Ob);
  cout << "main 2: " << Ob.Wez( ) << endl;
}
```

```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
main 2:
```

Co zrobić aby było dobrze?

Kopiowanie zawartości

```
class KopiaNapisu { // ...............................................
      char *_wNapis;
   public :
       ...
      KopiaNapisu( const KopiaNapisu& Ob )  ...
       ...
}; // ...............................................
```



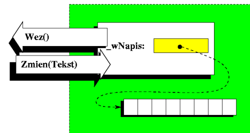

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
   KopiaNapisu  Ob;

   Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
   cout << "main 1: " << Ob.Wez( ) << endl;
   Wyswietl(Ob);
   cout << "main 2: " << Ob.Wez( ) << endl;
}
```

```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
main 2:
```

Implementacja konstruktora kopiującego zapewnia właściwe przekazywanie parametru i nie tylko.

Kopiowanie zawartości

```
class KopiaNapisu { //.............................................
    char *_wNapis;
  public :
      ...
     KopiaNapisu( const KopiaNapisu& Ob )
         { _wNapis = nullptr; Zmien(Ob._wNapis); }
}; //...............................................
```



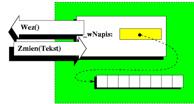

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

int main( )
{
  KopiaNapisu Ob;
  Ob.Zmien( "Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmi" );
  cout << "main 1: " << Ob.Wez( ) << endl;
  Wyswietl(Ob);
  cout << "main 2: " << Ob.Wez( ) << endl;
}
```

```
main 1: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
Wyswietl: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
main 2: Abu Ja'far Mohammed ibn Musa al-Khowarizmi
```

Implementacja konstruktora kopiującego zapewnia właściwe przekazywanie parametru i nie tylko.

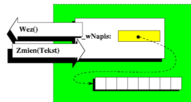
Operacja podstawienia

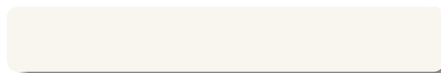
```
class KopiaNapisu { // ....................................
     char  *_wNapis;
  public :
      ...
}; // ..........................................................


int  main( )
{
  KopiaNapisu   Ob1;




}
```



Operacja podstawienia

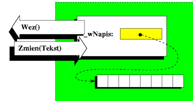
```
class KopiaNapisu  { // ....................................
      char  *_wNapis;
   public :
         ...
}; // ......................................................................


int  main( )
{
   KopiaNapisu   Ob1;

   Ob1.Zmien( "Hippasus z Metapontum"  );

}
```

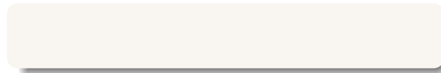




Operacja podstawienia

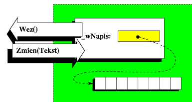
```
class KopiaNapisu { // ....................................
    char *_wNapis;
  public :
      ...
}; // ...................................................................


int main( )
{
  KopiaNapisu  Ob1;

  Ob1.Zmien( "Hippasus z Metapontum" );
  {



  }

}
```

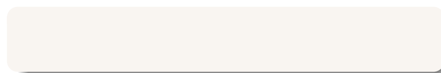




Operacja podstawienia

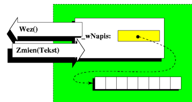
```
class KopiaNapisu { // ....................................
    char *_wNapis;
  public :
      ...
}; // ...........................................................................


int main( )
{
  KopiaNapisu Ob1;
  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu Ob2;



  }

}
```

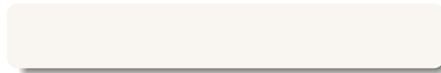




Operacja podstawienia

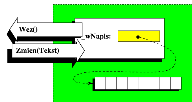
```
class KopiaNapisu { // ....................................
    char *_wNapis;
  public :
      ...
}; // ..........................................................................

int main( )
{
  KopiaNapisu  Ob1;
  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu  Ob2;
    Ob2 = Ob1
  }
}
```





Operacja podstawienia

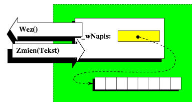
```
class KopiaNapisu { // ...............................
    char *_wNapis;
  public :
      ...
}; // ...........................................................
```




```
int main( )
{
  KopiaNapisu Ob1;
  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu Ob2;
    Ob2 = Ob1
    cout << "Ob1: " << Ob1.Wez( ) << endl;

  }

}
```

Operacja podstawienia

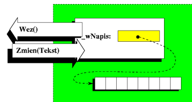
```
class KopiaNapisu { // ....................................
    char *_wNapis;
  public :
      ...
}; // ..........................................................................


int main( )
{
  KopiaNapisu  Ob1;

  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu  Ob2;
    Ob2 = Ob1
    cout << "Ob1: " << Ob1.Wez( ) << endl;

  }

}
```



Ob1: Hippasus z Metapontum

Operacja podstawienia

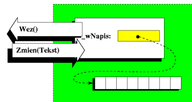
```
class KopiaNapisu  { // ................................
      char  *_wNapis;
   public :
         ...
}; // ..............................................................


int  main( )
{
   KopiaNapisu   Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
      KopiaNapisu   Ob2;
      Ob2 = Ob1
      cout << "Ob1: " << Ob1.Wez( ) << endl;
      cout << "Ob2: " << Ob2.Wez( ) << endl;
   }

}
```



Ob1: Hippasus z Metapontum

Operacja podstawienia

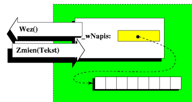
```
class KopiaNapisu { // ....................................
    char *_wNapis;
  public :
      ...
}; // ....................................................................


int main( )
{
  KopiaNapisu Ob1;

  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu Ob2;
    Ob2 = Ob1
    cout << "Ob1: " << Ob1.Wez( ) << endl;
    cout << "Ob2: " << Ob2.Wez( ) << endl;
  }

}
```




```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
```

Operacja podstawienia

```
class KopiaNapisu { // ...............................
    char *_wNapis;
  public :
      ...
}; // ..................................................
```

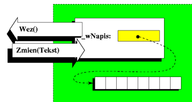



```
int main( )
{
  KopiaNapisu Ob1;
  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu Ob2;
    Ob2 = Ob1
    cout << "Ob1: " << Ob1.Wez( ) << endl;
    cout << "Ob2: " << Ob2.Wez( ) << endl;
  }
  cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
```

Operacja podstawienia

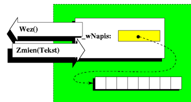
```
class KopiaNapisu { // ..................................
    char *_wNapis;
  public :
      ...
}; // ...................................................................


int main( )
{
  KopiaNapisu  Ob1;
  Ob1.Zmien( "Hippasus z Metapontum" );
  {
    KopiaNapisu  Ob2;
    Ob2 = Ob1
    cout << "Ob1: " << Ob1.Wez( ) << endl;
    cout << "Ob2: " << Ob2.Wez( ) << endl;
  }
  cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```




```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1:
```

Operacja podstawienia

```
class KopiaNapisu { // ...............................
     char *_wNapis;
   public :
        ...
}; // .................................................................


int main( )
{
   KopiaNapisu  Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
     KopiaNapisu  Ob2;
     Ob2 = Ob1
     cout << "Ob1: " << Ob1.Wez( ) << endl;
     cout << "Ob2: " << Ob2.Wez( ) << endl;
   }
   cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

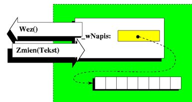



```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1:
```

Co stało się z napisem?

Operacja podstawienia

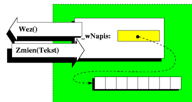
```
class KopiaNapisu { // .................................
     char *_wNapis;
  public :
       ...
    KopiaNapisu &  operator = (const  KopiaNapisu &Ob) ...
}; // ..................................................................

int main( )
{
   KopiaNapisu  Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
     KopiaNapisu  Ob2;
     Ob2 = Ob1
     cout << "Ob1: " << Ob1.Wez( ) << endl;
     cout << "Ob2: " << Ob2.Wez( ) << endl;
   }
   cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```




```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1:
```

Stosowanie pól wskaźnikowych wymaga prawie zawsze implementacji konstruktora kopiującego i przeciążenia operatora przypisania.

Operacja podstawienia



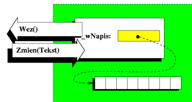

```
class KopiaNapisu { //..............................
   ...
 public :
   ...
  KopiaNapisu &  operator = (const  KopiaNapisu &Ob)
                        { Zmien(Ob._wNapis);  return  *this; }
}; // ..........................................................

int  main( )
{
 KopiaNapisu  Ob1;

 Ob1.Zmien( "Hippasus z Metapontum" );
 {
   KopiaNapisu  Ob2;
   Ob2 = Ob1
   cout << "Ob1: " << Ob1.Wez( ) << endl;
   cout << "Ob2: " << Ob2.Wez( ) << endl;
 }
 cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1: Hippasus z Metapontum

Stosowanie pól wskaźnikowych wymaga prawie zawsze implementacji konstruktora kopiującego i przeciążenia operatora przypisania.

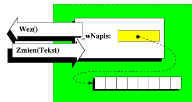
Inicjalizacja obiektu

```
class KopiaNapisu { //.................................
    char *_wNapis;
  public :
      ...
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                            { Zmien(Ob._wNapis); return *this; }
}; //..........................................................
```

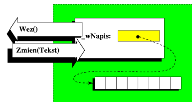


Inicjalizacja obiektu

```
class KopiaNapisu { //...........................
    char *_wNapis;
  public :
      ...
    KopiaNapisu( const KopiaNapisu & Ob )
                          { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                          { Zmien(Ob._wNapis); return *this; }
}; // ..............................................................
```

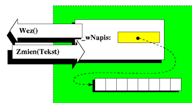


Inicjalizacja obiektu



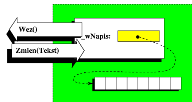

```
class KopiaNapisu { //.....................................
    char *_wNapis;
  public :
    ...
    KopiaNapisu( const KopiaNapisu & Ob )
                        { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                        { Zmien(Ob._wNapis); return *this; }
}; //.........................................................


int main( )
{
  KopiaNapisu Ob1;



}
```

Inicjalizacja obiektu



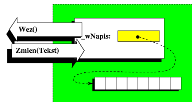

```
class KopiaNapisu { //..................................
    char *_wNapis;
  public :
    ...
    KopiaNapisu( const KopiaNapisu & Ob )
                        { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                        { Zmien(Ob._wNapis); return *this; }
}; //..........................................................................


int main( )
{
  KopiaNapisu Ob1;

  Ob1.Zmien("Ten tekst nic nie znaczy ;-)");


}
```

Inicjalizacja obiektu



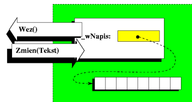

```
class KopiaNapisu { //...............................
    char *_wNapis;
  public :
    ...
    KopiaNapisu( const KopiaNapisu & Ob )
                     { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                     { Zmien(Ob._wNapis); return *this; }
}; // ..............................................................


int main( )
{
  KopiaNapisu Ob1;

  Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

  KopiaNapisu Ob2(Ob1);

}
```

Inicjalizacja obiektu



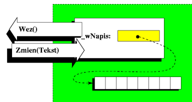

```
class KopiaNapisu { //.................................
    char *_wNapis;
  public :
      ...
    KopiaNapisu( const KopiaNapisu & Ob )
                        { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                        { Zmien(Ob._wNapis); return *this; }
}; // ...............................................................

int main( )
{
  KopiaNapisu Ob1;

  Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

  KopiaNapisu Ob2(Ob1);
  KopiaNapisu Ob3 = Ob1;
}
```

Inicjalizacja obiektu




```
class KopiaNapisu { //...............................
    char *_wNapis;
  public :
     ...
    KopiaNapisu( const KopiaNapisu & Ob )
                            { _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                            { Zmien(Ob._wNapis); return *this; }
}; //..................................................................


int main( )
{
  KopiaNapisu Ob1;

  Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

  KopiaNapisu Ob2(Ob1);
  KopiaNapisu Ob3 = Ob1;
}
```

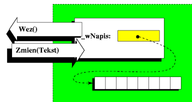
Inicjalizacja obiektu




```
class KopiaNapisu { //....................................
    char *_wNapis;
  public :
      ...
    KopiaNapisu( const KopiaNapisu & Ob )
       { cout << "K. kopiujacy" << endl; _wNapis = nullptr; Zmien(Ob._wNapis); }
    KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                              { Zmien(Ob._wNapis); return *this; }
}; //..........................................................................


int main( )
{
  KopiaNapisu  Ob1;

  Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

  KopiaNapisu  Ob2(Ob1);
  KopiaNapisu  Ob3 = Ob1;
}
```

Inicjalizacja obiektu

```
class  KopiaNapisu  { //.................................
      char  *_wNapis;
   public :
        ...
      KopiaNapisu( const  KopiaNapisu & Ob )
         { cout << "K. kopiujacy" << endl; _wNapis = nullptr;  Zmien(Ob._wNapis); }
      KopiaNapisu &  operator = ( const  KopiaNapisu &Ob )
                              { Zmien(Ob._wNapis);  return  *this; }
}; // .........................................................................
```



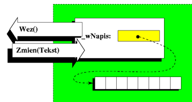

```
int  main( )
{
   KopiaNapisu   Ob1;

   Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

   KopiaNapisu   Ob2(Ob1);
   KopiaNapisu   Ob3 = Ob1;
}
```

Wynik działania

K. kopiujacy
K. kopiujacy

Inicjalizacja obiektu

```
class KopiaNapisu { //.................................
      char *_wNapis;
   public :
        ...
      KopiaNapisu( const KopiaNapisu & Ob )
         { cout << "K. kopiujacy" << endl; _wNapis = nullptr; Zmien(Ob._wNapis); }
      KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                              { Zmien(Ob._wNapis); return *this; }
}; // ..............................................................
```




```
int main( )
{
   KopiaNapisu Ob1;

   Ob1.Zmien("Ten tekst nic nie znaczy ;-)");

   KopiaNapisu Ob2(Ob1);
   KopiaNapisu Ob3 = Ob1;
}
```

Wynik działania

K. kopiujacy
K. kopiujacy

Przy inicjalizacji zawsze i wyłącznie wywoływane są konstruktory. W szczególności zapis operacji przypisania odpowiada uruchomieniu operatora kopiującego.

Klasy z polami referencyjnymi

```
class Wektor2 {   //.....................................................................
    public :
      int     &x,  &y;
      int     Tab[2];


};  //...........................................................................
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //.....................................
    public :
      int     &x,  &y;
      int     Tab[2];


};  //...................................................




int  main( )
{
    Wektor2    W1;



}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //.......................................................
    public :
      int      &x,  &y;
      int      Tab[2];


};   //...........................................................


int  main( )
{
    Wektor2  W1;


}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //..............................................................
     public :
       int     &x,  &y;
       int     Tab[2];

       Wektor2( ): x(Tab[0]), y(Tab[1]) { }
};   //...................................................................................




int  main( )
{
     Wektor2    W1;



}
```

Klasy z polami referencyjnymi

```
class Wektor2 {  //..............................................................
    public :
      int    &x,  &y;
      int    Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
};  //...........................................................................




int  main( )
{
     Wektor2    W1;
     Wektor2    W2(W1);


}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //.........................................................
    public :
      int     &x,  &y;
      int     Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
};   //...........................................................................
```

```
int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);
}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   // ...........................................
    public :
      int     &x,  &y;
      int     Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
      Wektor2(const  Wektor2 &W) ...
};  // ..............................................................



int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);


}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //..........................................................
    public :
      int    &x,  &y;
      int    Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
      Wektor2(const  Wektor2 &W): x(Tab[0]), y(Tab[1]) { x = W.x; y = W.y; }
 };   //..........................................................



int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);


}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //..............................................
    public :
      int     &x,  &y;
      int     Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
      Wektor2(const  Wektor2 &W): x(Tab[0]), y(Tab[1]) { x = W.x; y = W.y; }
};  //................................................................


int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);

    W1 = W2;
}
```

Klasy z polami referencyjnymi

```
class Wektor2 {  //..............................................................
   public :
     int     &x,  &y;
     int     Tab[2];

     Wektor2( ): x(Tab[0]), y(Tab[1]) { }
     Wektor2(const  Wektor2 &W): x(Tab[0]), y(Tab[1]) { x = W.x; y = W.y; }
 };  //..............................................................

int  main( )
{
   Wektor2    W1;
   Wektor2    W2(W1);

   W1 = W2;
}
```

(Linia `W1 = W2;` jest na slajdzie przekreślona.)

Klasy z polami referencyjnymi

```
class Wektor2 {   //.................................................................
    public :
      int     &x,  &y;
      int     Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
      Wektor2(const  Wektor2 &W): x(Tab[0]), y(Tab[1]) { x = W.x; y = W.y; }
      Wektor2& operator = ( const Wektor2& W ) ...
};   //..............................................................................


int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);

    W1 = W2;
}
```

Klasy z polami referencyjnymi

```
class Wektor2 {   //.........................................................
    public :
      int     &x,  &y;
      int     Tab[2];

      Wektor2( ): x(Tab[0]), y(Tab[1]) { }
      Wektor2(const  Wektor2 &W): x(Tab[0]), y(Tab[1]) { x = W.x; y = W.y; }
      Wektor2& operator = ( const Wektor2& W ) { x = W.x; y = W.y; }
};  //.......................................................................


int  main( )
{
    Wektor2    W1;
    Wektor2    W2(W1);

    W1 = W2;
}
```

Podsumowanie (1)

- W przypadku klas, w których zdefiniowane są pola wskaźnikowe, może koniecznym okazać się zdefiniowanie konstruktora kopiującego oraz przeciążenie operatora przypisania. Jest to niezbędne wtedy, gdy obiekty tej klasy stowarzyszone są ze strukturami tworzonymi dynamicznie i usuwanymi w destruktorze.
- Jeżeli stowarzyszone z danym obiektem struktury danych nie są usuwane w destruktorze, to na ogół można ograniczyć się do domyślnej implementacji konstruktora kopiującego i operatora przypisania.

Podsumowanie (1)

- W przypadku klas, w których zdefiniowane są pola wskaźnikowe, może koniecznym okazać się zdefiniowanie konstruktora kopiującego oraz przeciążenie operatora przypisania. Jest to niezbędne wtedy, gdy obiekty tej klasy stowarzyszone są ze strukturami tworzonymi dynamicznie i usuwanymi w destruktorze.
- Jeżeli stowarzyszone z danym obiektem struktury danych nie są usuwane w destruktorze, to na ogół można ograniczyć się do domyślnej implementacji konstruktora kopiującego i operatora przypisania.

Podsumowanie (1)

- W przypadku klas, w których zdefiniowane są pola wskaźnikowe, może koniecznym okazać się zdefiniowanie konstruktora kopiującego oraz przeciążenie operatora przypisania. Jest to niezbędne wtedy, gdy obiekty tej klasy stowarzyszone są ze strukturami tworzonymi dynamicznie i usuwanymi w destruktorze.
- Jeżeli stowarzyszone z danym obiektem struktury danych nie są usuwane w destruktorze, to na ogół można ograniczyć się do domyślnej implementacji konstruktora kopiującego i operatora przypisania.

Podsumowanie (2)

- W klasach, w których definiowane są pola referencyjne nie istnieje domyślna implementacja konstruktora kopiującego i operatora przypisania. Wynika to z fakty, że zwykłe przepisanie bajt po bajcie zawartości obiektów zmieniałoby referencje. Z definicji zaś referencji wynika, że w trakcie swojego istnienia nie może ona ulegać zmianom.
- Jeżeli obiekty klasy zawierającej pola referencyjne mają być przekazywane jako parametr wywołania funkcji/metody lub przez nie zwracane lub też w sposób jawny ma być wywoływany konstruktor kopiujący, to jego zdefiniowanie jest bezwzględnie konieczne.
- W przypadku, gdy ma być wykonywana operacja przypisania, konieczne jest wówczas zdefiniowanie operatora przypisania.
- Jeżeli wyżej wymienione operacje nie będą wykonywane, to nie ma potrzeby definiowania zarówno konstruktora kopiującego, jak też operatora przypisania.

Podsumowanie (2)

- W klasach, w których definiowane są pola referencyjne nie istnieje domyślna implementacja konstruktora kopiującego i operatora przypisania. Wynika to z fakty, że zwykłe przepisanie bajt po bajcie zawartości obiektów zmieniałoby referencje. Z definicji zaś referencji wynika, że w trakcie swojego istnienia nie może ona ulegać zmianom.
- Jeżeli obiekty klasy zawierającej pola referencyjne mają być przekazywane jako parametr wywołania funkcji/metody lub przez nie zwracane lub też w sposób jawny ma być wywoływany konstruktor kopiujący, to jego zdefiniowanie jest bezwzględnie konieczne.
- W przypadku, gdy ma być wykonywana operacja przypisania, konieczne jest wówczas zdefiniowanie operatora przypisania.
- Jeżeli wyżej wymienione operacje nie będą wykonywane, to nie ma potrzeby definiowania zarówno konstruktora kopiującego, jak też operatora przypisania.

Podsumowanie (2)

- W klasach, w których definiowane są pola referencyjne nie istnieje domyślna implementacja konstruktora kopiującego i operatora przypisania. Wynika to z fakty, że zwykłe przepisanie bajt po bajcie zawartości obiektów zmieniałoby referencje. Z definicji zaś referencji wynika, że w trakcie swojego istnienia nie może ona ulegać zmianom.
- Jeżeli obiekty klasy zawierającej pola referencyjne mają być przekazywane jako parametr wywołania funkcji/metody lub przez nie zwracane lub też w sposób jawny ma być wywoływany konstruktor kopiujący, to jego zdefiniowanie jest bezwzględnie konieczne.
- W przypadku, gdy ma być wykonywana operacja przypisania, konieczne jest wówczas zdefiniowanie operatora przypisania.
- Jeżeli wyżej wymienione operacje nie będą wykonywane, to nie ma potrzeby definiowania zarówno konstruktora kopiującego, jak też operatora przypisania.

Podsumowanie (2)

- W klasach, w których definiowane są pola referencyjne nie istnieje domyślna implementacja konstruktora kopiującego i operatora przypisania. Wynika to z fakty, że zwykłe przepisanie bajt po bajcie zawartości obiektów zmieniałoby referencje. Z definicji zaś referencji wynika, że w trakcie swojego istnienia nie może ona ulegać zmianom.
- Jeżeli obiekty klasy zawierającej pola referencyjne mają być przekazywane jako parametr wywołania funkcji/metody lub przez nie zwracane lub też w sposób jawny ma być wywoływany konstruktor kopiujący, to jego zdefiniowanie jest bezwzględnie konieczne.
- W przypadku, gdy ma być wykonywana operacja przypisania, konieczne jest wówczas zdefiniowanie operatora przypisania.
- Jeżeli wyżej wymienione operacje nie będą wykonywane, to nie ma potrzeby definiowania zarówno konstruktora kopiującego, jak też operatora przypisania.

Podsumowanie (2)

- W klasach, w których definiowane są pola referencyjne nie istnieje domyślna implementacja konstruktora kopiującego i operatora przypisania. Wynika to z fakty, że zwykłe przepisanie bajt po bajcie zawartości obiektów zmieniałoby referencje. Z definicji zaś referencji wynika, że w trakcie swojego istnienia nie może ona ulegać zmianom.
- Jeżeli obiekty klasy zawierającej pola referencyjne mają być przekazywane jako parametr wywołania funkcji/metody lub przez nie zwracane lub też w sposób jawny ma być wywoływany konstruktor kopiujący, to jego zdefiniowanie jest bezwzględnie konieczne.
- W przypadku, gdy ma być wykonywana operacja przypisania, konieczne jest wówczas zdefiniowanie operatora przypisania.
- Jeżeli wyżej wymienione operacje nie będą wykonywane, to nie ma potrzeby definiowania zarówno konstruktora kopiującego, jak też operatora przypisania.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Klasa std::string

- Klasa łańcuchów napisowych została zaprojektowana, tak aby można ją było wykorzystywać jak normalny typ wbudowany. Pozwala to na ułatwienie przetwarzania tekstów
- Jedną z najważniejszych cech typu `std::string` jest to że jest zdefiniowana dla niego operacja kopiowania z wykorzystaniem zarówno konstruktora kopiującego jak też operatora podstawienie =. Rozwiązuje to problem dynamicznego przydziału i zwalniania pamięci, co na poziomie języka C jest zawsze kłopotliwe.
- Zdefiniowane są operacje porównywaniałańcuchów (operatory: `==`, `<`, `>`, `<=`, `>=`, `!=`), oraz operacja konkatenacji (operatory: `+`, `+=`).
- Dostępnych jest wiele dodatkowych udogodnień pozwalających na wyszukiwaniu znaków lub podciągów, wstawianie sekwencji znaków, zamiany itp.
- Nie są zdefiniowane metody wyszukiwania w oparciu o wyrażenia regularne.

Przykład prostych operacji

```
int main( )
{
    std::string Zyczenia;

}
```

Przykład prostych operacji

```
int main( )
{
    std::string Zyczenia;
    std::string Naglowek = "Z okazji spotkania Marsjan\n";




}
```

Przykład prostych operacji

```
int main( )
{
    std::string Zyczenia;
    std::string Naglowek = "Z okazji spotkania Marsjan\n";

    Zyczenia = Naglowek + "wszystkiego najlepszego zyczy\n";


}
```

Przykład prostych operacji

```
int main( )
{
    std::string Zyczenia;
    std::string Naglowek = "Z okazji spotkania Marsjan\n";

    Zyczenia = Naglowek + "wszystkiego najlepszego zyczy\n";
    Zyczenia += "Ziemianin";

}
```

Przykład prostych operacji

```
int  main( )
{
    std::string  Zyczenia;
    std::string  Naglowek = "Z okazji spotkania Marsjan\n";

    Zyczenia = Naglowek + "wszystkiego najlepszego zyczy\n";
    Zyczenia += "Ziemianin";
    cout <<  Zyczenia <<  endl;
}
```

Przykład prostych operacji

```
string UniwersalneZyczenia( const char *Naglowek, const char *Zakonczenie )
{
    string Zyczenia = Naglowek;

    Zyczenia += "wszystkiego najlepszego zyczy\n";
    Zyczenia += Zakonczenie;
    return Zyczenia;
}


int main( )
{




}
```

Przykład prostych operacji

```
string  UniwersalneZyczenia( const  char *Naglowek, const  char *Zakonczenie )
{
    string  Zyczenia = Naglowek;

    Zyczenia += "wszystkiego najlepszego zyczy\n";
    Zyczenia += Zakonczenie;
    return  Zyczenia;
}


int  main( )
{
    string  Zyczenia = UniwersalneZyczenia("Z okazji spotkania Marsjan ","Ziemianin");
    cout << Zyczenia << endl;



}
```

Przykład prostych operacji

```
string UniwersalneZyczenia( const char *Naglowek, const char *Zakonczenie )
{
    string Zyczenia = Naglowek;

    Zyczenia += "wszystkiego najlepszego zyczy\n";
    Zyczenia += Zakonczenie;
    return Zyczenia;
}


int main( )
{
    string Zyczenia = UniwersalneZyczenia("Z okazji spotkania Marsjan ","Ziemianin");
    cout << Zyczenia << endl;
    const char *Zycz_C = Zyczenia.c_str( );



}
```

Przykład prostych operacji

```
string UniwersalneZyczenia( const char *Naglowek, const char *Zakonczenie )
{
    string Zyczenia = Naglowek;

    Zyczenia += "wszystkiego najlepszego zyczy\n";
    Zyczenia += Zakonczenie;
    return Zyczenia;
}


int main( )
{
    string Zyczenia = UniwersalneZyczenia("Z okazji spotkania Marsjan ","Ziemianin");
    cout << Zyczenia << endl;
    const char *Zycz_C = Zyczenia.c_str( );

    Zyczenia = "Z okazji pierwszej swiatowej inwazji Marsjan na supermarkety ...";

}
```

Przykład prostych operacji

```
string UniwersalneZyczenia( const char *Naglowek, const char *Zakonczenie )
{
    string Zyczenia = Naglowek;

    Zyczenia += "wszystkiego najlepszego zyczy\n";
    Zyczenia += Zakonczenie;
    return Zyczenia;
}


int main( )
{
    string Zyczenia = UniwersalneZyczenia("Z okazji spotkania Marsjan ","Ziemianin");
    cout << Zyczenia << endl;
    const char *Zycz_C = Zyczenia.c_str( );

    Zyczenia = "Z okazji pierwszej swiatowej inwazji Marsjan na supermarkety ...";
    cout << Zycz_C << endl;
}
```

Przeciążone operatory

Przeciążenia w klasie **string**:

`< > <= >= == !=`	– porównania,
`=`	– przypisanie,
`+, +=`	– konkatenacja,
`[ ]`	– bezpośredni dostęp do znaku bez kontroli zakresu,

Przeciążenia zewnętrzne dla klasy **string**:

`<<`	– zapis do strumień klasy **ostream**,
`>>`	– cztanie ze strumienia klasy **istream**.

Przykład metod

Tworzenie tablic i łańcuchów znakowych w sensie języka C:

`c_str`	– zwraca łańcuch w sensie języka C,
`data`	– zwraca łańcuch w postaci tablicy znakowej,
`copy`	– przekopiowuje do tablicy znakowej zadana ilość znaków.

Ekstrahowanie podłańcuchów:

`substr`	– zwraca podciąg (obiekt klasy **string**),

Dostęp do poszczególnych elementów łańcucha:

`at`	– kontrolowany dostęp do danego znaku.

Przykład metod

Rozmiar i wielkość łańcucha:

`length`	– podaje długość łańcucha w sensie języka C,
`size`	– podaje rozmiar łańcucha,
`max_size`	– podaje maksymalny mo¿liwy rozmiar łańcucha,
`empty`	– informuje czy dany łańcuch jest pusty.

Pojemność łańcucha i jej zmiana:

`capacity`	– określenie pojemności,
`reserve`	– rezerwacja pamięci.

Szukanie i porównywanie:

`find`	– szukania znaków i ciągów znaków,
`compare`	– porównuje dwa ciągi.

Przykład metod

```
string  RdzenNazwyPliku( const  string  &NazwaPliku )
{
     size_t  Ind = NazwaPliku.find('.');

     if  ( Ind == string::npos ) return NazwaPliku;
     return   NazwaPliku.substr( 0, Ind );
}


int  main( )
{
     cout <<  RdzenNazwyPliku("rownanie_liniowe.dane") <<  endl;
}
```

Przykład metod

```
int main()
{
    cout << CzyNalezyDoGrupy("jk","cdrom:x:24:jk,installer,mythtv") << endl;
}
```

Przykład metod

```
bool CzyJestWLiscie( string LoginUzyt, string ListaUzyt )
{
    LoginUzyt = "," + LoginUzyt + ",";
    ListaUzyt = "," + ListaUzyt + ",";

    return ListaUzyt.find(LoginUzyt) != string ::npos;
}
```

```
int main()
{
    cout << CzyNalezyDoGrupy("jk","cdrom:x:24:jk,installer,mythtv") << endl;
}
```

Przykład metod

```
bool CzyJestWLiscie( string LoginUzyt, string ListaUzyt )
{
    LoginUzyt = "," + LoginUzyt + ",";
    ListaUzyt = "," + ListaUzyt + ",";

    return ListaUzyt.find(LoginUzyt) != string ::npos;
}

bool CzyNalezyDoGrupy( const char * sLoginUzyt, const char * sListaGrupy )
{
    istringstream StrmWe(sListaGrupy);
    string ListaUzyt;

    for (int i = 0; i < 3; ++i) StrmWe.ignore(1000,':');
    return CzyJestWLiscie(sLoginUzyt,ListaUzyt);
}

int main()
{
    cout << CzyNalezyDoGrupy("jk","cdrom:x:24:jk,installer,mythtv") << endl;
}
```

Plan prezentacji

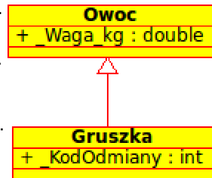
1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Rzutowanie wskaźnika

```
struct Owoc { // ..................................................................
        double  _Waga_kg;
}; // ..............................................................................

struct Gruszka: public  Owoc { // ..............................................
        int      _KodOdmiany;
}; // ..............................................................................

int  main( )
{

}
```

Rzutowanie wskaźnika

```
struct Owoc { // .........................................
        double  _Waga_kg;
}; // ....................................................

struct Gruszka: public  Owoc { // ........................
        int       _KodOdmiany;
}; // ....................................................

int  main( )
{

}
```



Rzutowanie wskaźnika

```
struct Owoc { // ....................................
        double  _Waga_kg;
}; // ...............................................

struct Gruszka: public  Owoc { // ...................
        int      _KodOdmiany;
}; // ...............................................

int  main( )
{

}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie wskaźnika

```
struct Owoc { // ..........................................
        double  _Waga_kg;
}; // .....................................................

struct Gruszka: public  Owoc { // .........................
        int      _KodOdmiany;
}; // .....................................................

int  main( )
{
   Gruszka    *wGr1 = new Gruszka();



}
```

Rzutowanie wskaźnika

```
struct Owoc { // ...............................
        double  _Waga_kg;
}; // ..........................................

struct Gruszka: public  Owoc { // ..............
        int      _KodOdmiany;
}; // ..........................................

int  main( )
{
    Gruszka    *wGr1 = new Gruszka();
    Owoc       *wOwocX = wGr1;



}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie wskaźnika

```
struct Owoc { // ..........................................
        double  _Waga_kg;
}; // .....................................................

struct Gruszka: public  Owoc { // .........................
        int      _KodOdmiany;
}; // .....................................................

int  main( )
{
    Gruszka    *wGr1 = new Gruszka();
    Owoc       *wOwocX = wGr1;

}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie na nadklasę (klasę bazową) może być wykonane niejawnie.

Rzutowanie wskaźnika

```
struct Owoc { // ..........................................
        double  _Waga_kg;
}; // .....................................................

struct Gruszka: public  Owoc { // .........................
        int     _KodOdmiany;
}; // .....................................................

int  main( )
{
   Gruszka     *wGr1 = new Gruszka();
   Owoc        *wOwocX = static_cast<Owoc*>(wGr1);

}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Można to również zrobić w sposób jawny.

Rzutowanie wskaźnika

```
struct Owoc { // ...........................................
        double  _Waga_kg;
}; // ......................................................

struct Gruszka: public  Owoc { // ..........................
        int       _KodOdmiany;
}; // ......................................................

int  main( )
{
    Gruszka    *wGr1 = new Gruszka();
    Owoc       *wOwocX = wGr1;

    cout << "Addr wGr: " << wGr1 << endl;
    cout << "Addr wOw: " << wOwocX << endl;
}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie wskaźnika

```
struct Owoc { // ..................................
        double  _Waga_kg;
}; // ..............................................

struct Gruszka: public  Owoc { // ..................
        int       _KodOdmiany;
}; // ..............................................

int  main( )
{
   Gruszka    *wGr1 = new Gruszka();
   Owoc       *wOwocX = wGr1;

   cout << "Addr wGr: " << wGr1 << endl;
   cout << "Addr wOw: " << wOwocX << endl;
}
```


Wynik działania:

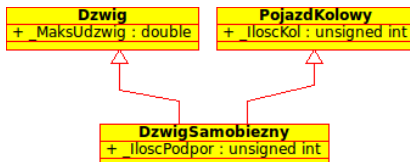
```
Addr wGr: 0x804a008
Addr wOw: 0x804a008
```

Brak różnicy w adresach wynika z tego, że obiekt klasy bazowej znajduje się “na początku” obiektu klasy pochodnej.

```
struct Dzwig { double _MaksUdzwig; }; // ........................................

struct PojazdKolowy { unsigned int _IloscKol; }; // ..............................

struct DzwigKolowy: public Dzwig, public PojazdKolowy, { // .......................
      unsigned int   _IloscPodpor;
}; // ..........................................................................

int main( )
{



}
```

```
struct Dzwig {   double  _MaksUdzwig;   }; // .......................................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ...........................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // .........................
      unsigned int      _IloscPodpor;
}; // ................................................................................

int  main( )
{

}
```

```
struct Dzwig {   double  _MaksUdzwig;   }; //......................   _MaksUdzwig

struct PojazdKolowy {   unsigned int  _IloscKol;   }; //................   _IloscKol

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........   _MaksUdzwig
      unsigned int      _IloscPodpor;                                     _IloscKol
}; //................................................................   _IloscPodpor

int  main( )
{
}
```



```
struct Dzwig { double _MaksUdzwig; }; //.........................

struct PojazdKolowy { unsigned int _IloscKol; }; // .................

struct DzwigKolowy: public Dzwig, public PojazdKolowy, { //........
      unsigned int    _IloscPodpor;
}; //..............................................................

int main( )
{
   DzwigKolowy   *wDzwKol = new DzwigKolowy();

}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

```
struct Dzwig {   double  _MaksUdzwig;   }; // ........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
      unsigned int     _IloscPodpor;
}; // ............................................................

int  main( )
{
   DzwigKolowy    *wDzwKol = new DzwigKolowy();
   PojazdKolowy   *wPojazdK = wDzwKol;



}
```

```
struct Dzwig {   double  _MaksUdzwig;   }; // ........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // .......
      unsigned int      _IloscPodpor;
}; // ...................................................................

int  main( )
{
   DzwigKolowy    *wDzwKol = new DzwigKolowy();
   PojazdKolowy   *wPojazdK = wDzwKol;



}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

Tak jak to miało miejsce wcześnij, rzutowanie na nadklasę (klasę bazową) może być wykonane niejawnie.

```
struct Dzwig {   double  _MaksUdzwig;   }; // ..........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // .................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
      unsigned int      _IloscPodpor;
}; // ....................................................................

int  main( )
{
   DzwigKolowy    *wDzwKol = new DzwigKolowy();
   PojazdKolowy   *wPojazdK = static_cast<PojazdKolowy*>(wDzwKol);

}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

I tym przypadku można to również zrobić w sposób jawny.

```
struct Dzwig { double _MaksUdzwig; }; // ........................

struct PojazdKolowy { unsigned int _IloscKol; }; // ..................

struct DzwigKolowy: public Dzwig, public PojazdKolowy, { // ........
      unsigned int _IloscPodpor;
}; // ..............................................................

int main( )
{
   DzwigKolowy   *wDzwKol = new DzwigKolowy();
   PojazdKolowy  *wPojazdK = wDzwKol;

   cout << "Addr wDzK: " << wDzwKol << endl;
   cout << "Addr wPoj: " << wPojazdK << endl;
}
```

```
struct Dzwig {   double  _MaksUdzwig;   }; //........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // .................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int      _IloscPodpor;
}; // .................................................................

int  main( )
{
   DzwigKolowy    *wDzwKol = new DzwigKolowy();
   PojazdKolowy   *wPojazdK = wDzwKol;

   cout << "Addr wDzK: " << wDzwKol << endl;
   cout << "Addr wPoj: " << wPojazdK << endl;
}
```


Wynik działania:

```
Addr wDzK: 0x916c000
Addr wPoj: 0x906c004
```

W tym przykładzie rzutowanie wiąże się z fizyczną zmianą adresu. Różnica adresów związana jest z tym, że obiekt nadklasy ***PojazdKolowy*** jest drugi w kolejności w strukturze obiektu ***DzwigKolowy***.

Rzutowanie na refencję

```
struct Owoc { // ........................................
        double  _Waga_kg;
}; // ...................................................

struct Gruszka: public  Owoc { // .......................
        int     _KodOdmiany;
}; // ...................................................

int  main( )
{
   Gruszka   Gr1;



}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie na refencję

```
struct Owoc { // ........................................
        double  _Waga_kg;
}; // ...................................................

struct Gruszka: public  Owoc { // .......................
        int      _KodOdmiany;
}; // ...................................................

int  main( )
{
   Gruszka    Gr1;
   Owoc       &OwocX = Gr;

}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie na referencję daje analogiczne efekty. Jest ono domyślnie realizowane niejawnie.

Rzutowanie na refencję

```
struct Owoc { // ..........................................
        double  _Waga_kg;
}; // .....................................................

struct Gruszka: public  Owoc { // .........................
        int       _KodOdmiany;
}; // .....................................................

int  main( )
{
   Gruszka    Gr1;
   Owoc       &OwocX = static_cast<Owoc&>(Gr1);


}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Zawsze można wykonać je jawnie wykorzystując operator rzutowania `static_cast`.

Rzutowanie na refencję

```
struct Owoc { // ..............................................
        double  _Waga_kg;
}; // .........................................................

struct Gruszka: public  Owoc { // .............................
        int      _KodOdmiany;
}; // .........................................................

int  main( )
{
    Gruszka    Gr1;
    Owoc       &OwocX = Gr;

    cout << "Addr Gr: " << &Gr1 << endl;
    cout << "Addr Ow: " << &OwocX << endl;
}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Wynik działania:

```
Addr Gr: 0x7239c0a
Addr Ow: 0x7239c0a
```

Z tych samych przyczyn co poprzednio, oba wyświetlane adresy są takie same.

Rzutowanie na referencję

```
struct Dzwig {   double  _MaksUdzwig;   }; // ........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
      unsigned int      _IloscPodpor;
}; // ...............................................................

int  main( )
{
   DzwigKolowy    DzwKol;

}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

Rzutowanie na referencję

```
struct Dzwig {   double  _MaksUdzwig;   }; // ..........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // .................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
       unsigned int      _IloscPodpor;
}; // ..................................................................

int  main( )
{
   DzwigKolowy    DzwKol;
   PojazdKolowy   &PojazdK = DzwKol;



}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

Tutaj również rzutowanie jest niejawne i przebiega ono analogicznie jak w tym samym przykładzie dla wskaźników.

Rzutowanie na referencję

```
struct Dzwig {   double  _MaksUdzwig;   }; // ...........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ..................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
       unsigned int      _IloscPodpor;
}; // ..................................................................

int  main( )
{
    DzwigKolowy     DzwKol;
    PojazdKolowy    &PojazdK = static_cast<PojazdKolowy&>(DzwKol);



}
```


Oczywiście można je zrealizować jawnie.

Rzutowanie na referencję

```
struct Dzwig {   double  _MaksUdzwig;   }; //..........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ...............

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //.......
      unsigned int     _IloscPodpor;
}; //..............................................................

int  main( )
{
   DzwigKolowy    DzwKol;
   PojazdKolowy   &PojazdK = DzwKol;

   cout << "Addr DzK: " << DzwKol << endl;
   cout << "Addr Poj: " << PojazdK << endl;
}
```

_MaksUdzwig

_IloscKol

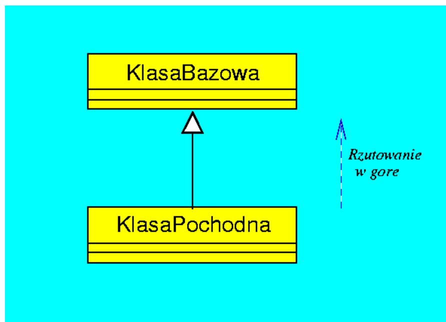
_MaksUdzwig
_IloscKol
_IloscPodpor

Wynik działania:

```
Addr wDzK: 0x7181d00
Addr wPoj: 0x7181d04
```

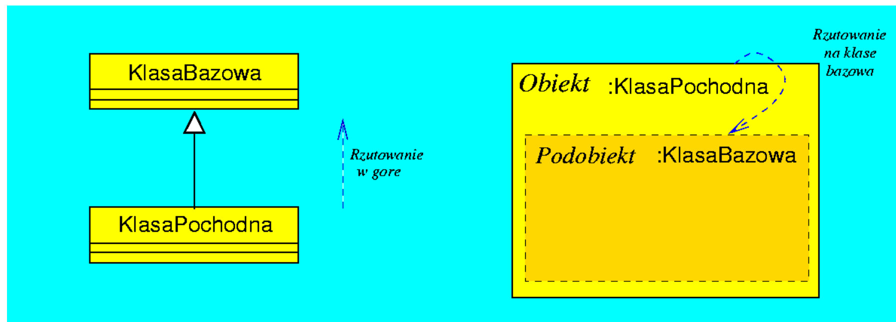
Z tych samym przyczyn, co w analogicznym przykładzie dla wskaźników, rzutowanie wiąże się z fizyczną zmianą adresu.

Rzutowanie w górę



Rzutowanie "w górę" jest rzutowaniem na klasę bazową. Tego typu rzutowanie zawsze się powiedzie, gdyż obiekt klasy pochodnej musi zawierać podobiekt klasy bazowej. Z tego powodu rzutowanie to może być realizowane niejawnie.

Rzutowanie w górę



Rzutowanie "w górę" jest rzutowaniem na klasę bazową. Tego typu rzutowanie zawsze się powiedzie, gdyż obiekt klasy pochodnej musi zawierać podobiekt klasy bazowej. Z tego powodu rzutowanie to może być realizowane niejawnie.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; //.........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // .................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int      _IloscPodpor;
}; //.............................................................

int  main( )
{
   DzwigKolowy    DzwKol;



}
```

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig { double _MaksUdzwig; }; //..........................

struct PojazdKolowy { unsigned int _IloscKol; }; //.................

struct DzwigKolowy: public Dzwig, public PojazdKolowy, { //........
      unsigned int    _IloscPodpor;
}; //..............................................................

int main( )
{
   DzwigKolowy    DzwKol;
   PojazdKolowy   *wPojazdK = &DzwKol;

}
```


Rzutowanie w górę przebiega w sposób niejawny. Tu jest OK.

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; // ..........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // .................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
       unsigned int      _IloscPodpor;
}; // ................................................................

int  main( )
{
   DzwigKolowy     DzwKol;
   PojazdKolowy    *wPojazdK = &DzwKol;
   DzwigKolowy     *wDzwigKo = wPojazdK;


}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

A co z tym przypadkiem?

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; // ........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
       unsigned int      _IloscPodpor;
}; // ...................................................................

int  main( )
{
    DzwigKolowy     DzwKol;
    PojazdKolowy    *wPojazdK = &DzwKol;
    DzwigKolowy     *wDzwigKo = wPojazdK;


}
```


Ten typ rzutowania nie może być niejawny.

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; //..........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; //.................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int     _IloscPodpor;
}; //...................................................................

int  main( )
{
   DzwigKolowy     DzwKol;
   PojazdKolowy    *wPojazdK = &DzwKol;
   DzwigKolowy     *wDzwigKo = static_cast<DzwigKolowy*>(wPojazdK);



}
```


Można jednak wymusić rzutowanie w sposób jawny wykorzystując operator rzutowania `static_cast`.

Rzutowanie na wskaźnik do nadobiektu

```
struct Dzwig { double _MaksUdzwig; }; // .......................... _MaksUdzwig

struct PojazdKolowy { unsigned int _IloscKol; }; // ................ _IloscKol

struct DzwigKolowy: public Dzwig, public PojazdKolowy, { // ........ _MaksUdzwig
      unsigned int    _IloscPodpor;                                  _IloscKol
}; // ............................................................. _IloscPodpor

int main( )
{
   DzwigKolowy    DzwKol;
   PojazdKolowy   *wPojazdK = &DzwKol;
   DzwigKolowy    *wDzwigKo = static_cast<DzwigKolowy*>(wPojazdK);

   cout << "Addr DzK1: " << &DzwKol << endl;
   cout << "Addr DzK2: " << wDzwigKo << endl;
}
```

Wynik działania:

```
Addr wDzK1: 0x7918c08
Addr wDzK2: 0x7918c08
```

Dzięki tej operacji dysponując częścią obiektu możemy *odzyskać* adres całego obiektu.

Rzutowanie na referencję do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; //.........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; //.................
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int      _IloscPodpor;
}; //..............................................................

int  main( )
{
   DzwigKolowy     DzwKol;
   PojazdKolowy    &PojazdK = DzwKol;


}
```

Rzutowanie na referencję do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; //........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int      _IloscPodpor;
}; // ...................................................................

int  main( )
{
    DzwigKolowy     DzwKol;
    PojazdKolowy    &PojazdK = DzwKol;
    DzwigKolowy     &DzwigKo = static_cast<DzwigKolowy&>(PojazdK);


}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

Możemy wymusić rzutowanie w górę na referencję do obiektu

Rzutowanie na referencję do nadobiektu

```
struct Dzwig {   double  _MaksUdzwig;   }; //........................
struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ...............
struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
      unsigned int      _IloscPodpor;
}; //..................................................................

int  main( )
{
    DzwigKolowy     DzwKol;
    PojazdKolowy    &PojazdK = DzwKol;
    DzwigKolowy     &DzwigKo = static_cast<DzwigKolowy&>(PojazdK);

    cout << "Addr DzK1: " << &DzwKol << endl;
    cout << "Addr DzK2: " << wDzwigKo << endl;
}
```

_MaksUdzwig

_IloscKol

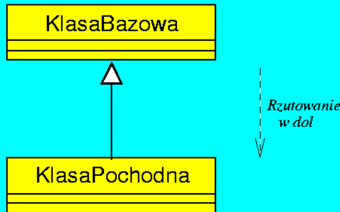
_MaksUdzwig
_IloscKol
_IloscPodpor

Wynik działania:

```
Addr wDzK1: 0x7918c08
Addr wDzK2: 0x7918c08
```

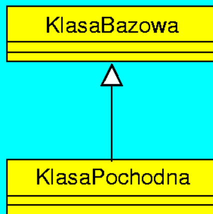
Dzięki tej operacji dysponując częścią obiektu możemy *odzyskać* adres całego obiektu.

Rzutowanie w dół

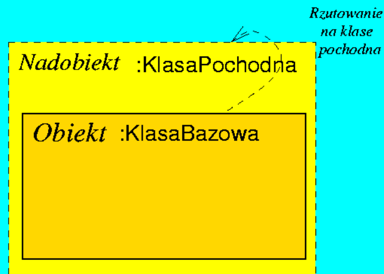


Rzutowanie "w dół" jest rzutowaniem na klasę pochodną. Ze względu na to, że obiekt klasy bazowej nie zawsze musi być składnikiem obiektu klasy pochodnej (np. może występować samodzielnie), rzutowanie to może nie powieść się.

Rzutowanie w dół



Rzutowanie
w dol



Rzutowanie "w dół" jest rzutowaniem na klasę pochodną. Ze względu na to, że obiekt klasy bazowej nie zawsze musi być składnikiem obiektu klasy pochodnej (np. może występować samodzielnie), rzutowanie to może nie powieść się.

Błędne rzutowanie w górę

```
struct Dzwig {   double  _MaksUdzwig;   }; // ..........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; // ..................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { // ........
      unsigned int      _IloscPodpor;
}; // ...................................................................

int  main( )
{
   PojazdKolowy    PojK;



}
```


Ten typ rzutowania może być niebezpieczny. Rozważmy następującą sytuację . . .

Błędne rzutowanie w górę

```
struct Dzwig {   double  _MaksUdzwig;   }; //........................

struct PojazdKolowy {   unsigned int  _IloscKol;   }; //.................

struct DzwigKolowy: public  Dzwig, public  PojazdKolowy, { //........
       unsigned int      _IloscPodpor;
}; //.........................................................

int  main( )
{
    PojazdKolowy    PojK;
    DzwigKolowy    *wDzwigK = static_cast<DzwigKolowy*>(&PojK);




}
```

_MaksUdzwig

_IloscKol

_MaksUdzwig
_IloscKol
_IloscPodpor

_MaksUdzwig
_IloscKol
_IloscPodpor

Rzutując otrzymujemy wskaźnik na nieistniejący obiekt. Odwołując się do niego prawie na pewno w którymś momencie zobaczymy komunikat `segmentation fault` :(

Błędne rzutowanie w górę

```
struct Dzwig { double _MaksUdzwig; }; //..........................
struct PojazdKolowy { unsigned int _IloscKol; }; // ................
struct DzwigKolowy: public Dzwig, public PojazdKolowy, { // ........
      unsigned int    _IloscPodpor;
}; // ..............................................................

int main( )
{
   PojazdKolowy   PojK;
   DzwigKolowy   *wDzwigK = static_cast<DzwigKolowy*>(&PojK);

   cout << "Addr PojK: " << &PojK << endl;
   cout << "Addr DzwK: " << wDzwigK << endl;
}
```


Wynik działania:

```
Addr PojK: 0x6198e04
Addr DzwK: 0x6198e00
```

Adres dostępny poprzez `wDzwigK`, nie jest adresem rzeczywistego obiektu.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Niejawne rzutowanie w liście parametrów funkcji

```
struct Owoc { // .........................................
        double  _Waga_kg;
}; // ....................................................

struct Gruszka: public  Owoc { // ........................
        int     _KodOdmiany;
}; // ....................................................

void Wyswietl(const Owoc  *wOwoc)
{
   cout << "Waga: " << wOwoc->_Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;


}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Niejawne rzutowanie w liście parametrów funkcji

```
struct Owoc { // ....................................
        double  _Waga_kg;
}; // ...............................................

struct Gruszka: public  Owoc { // ...................
        int       _KodOdmiany;
}; // ...............................................

void Wyswietl(const Owoc  *wOwoc)
{
   cout << "Waga: " << wOwoc->_Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;


}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Niejawne rzutowanie w liście parametrów funkcji

```
struct Owoc { // ..........................
        double  _Waga_kg;
}; // ..........................

struct Gruszka: public  Owoc { // ..........................
        int      _KodOdmiany;
}; // ..........................

void Wyswietl(const Owoc  *wOwoc)
{
   cout << "Waga: " << wOwoc->_Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;

   Wyswietl(&Gr1);
}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Niejawne rzutowanie pozwala wykorzystać funkcjonalności stworzone dla klasy bazowej na potrzeby obiektów klasy pochodnej.

Niejawne rzutowanie w liście parametrów funkcji

```
struct Owoc { // ..........................................
       double  _Waga_kg;
}; // .....................................................

struct Gruszka: public  Owoc { // ..........................
       int      _KodOdmiany;
}; // .....................................................

inline  void Wyswietl(const Owoc  *wOwoc)
{
   cout << "Waga: " << wOwoc->_Waga_kg << endl;
}

int  main( )
{
   Gruszka   Gr1;
   { const Owoc  *wOwoc = &Gr1;
     cout << "Waga: " << wOwoc->_Waga_kg << endl;
   }
}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Niejawne rzutowanie w liście parametrów jest niczym innym jak rzutowaniem w przypadku operacji podstawienia.

Rzutowanie na referencję w liście parametrów

```
struct Owoc { // ..............................
        double  _Waga_kg;
}; // .........................................

struct Gruszka: public  Owoc { // .............
        int        _KodOdmiany;
}; // .........................................

void Wyswietl(const Owoc  &Ow)
{
   cout << "Waga: " << Ow._Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;

}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie na referencję w liście parametrów

```
struct Owoc { // ....................................
        double  _Waga_kg;
}; // ...............................................

struct Gruszka: public  Owoc { // ...................
        int       _KodOdmiany;
}; // ...............................................

void Wyswietl(const Owoc  &Ow)
{
   cout << "Waga: " << Ow._Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;


}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Rzutowanie na referencję w liście parametrów

```
struct Owoc { // ....................
      double  _Waga_kg;
}; // ...............................

struct Gruszka: public  Owoc { // ...
      int     _KodOdmiany;
}; // ...............................

void Wyswietl(const Owoc  &Ow)
{
   cout << "Waga: " << Ow._Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;

   Wyswietl(Gr1);
}
```


Możliwość niejawnego rzutowania na referencje do obiektów klasy bazowej, jeszcze bardziej rozszerza użyteczność funkcji tworzonych dla tych obiektów.

Niejawne rzutowanie na referencję w liście parametrów

```
struct Owoc { // .............................................
        double  _Waga_kg;
}; // .........................................................

struct Gruszka: public  Owoc { // ............................
        int       _KodOdmiany;
}; // .........................................................

inline  void Wyswietl(const Owoc  &Ow)
{
   cout << "Waga: " << Ow._Waga_kg << endl;
}

int  main( )
{
   Gruszka    Gr1;
   { const Owoc  &Ow = Gr1;
     cout << "Waga: " << Ow._Waga_kg << endl;
   }
}
```

_Waga_kg

_Waga_kg
_KodOdmiany

Niejawne rzutowanie w liście parametrów jest niczym innym jak rzutowaniem w przypadku operacji podstawienia.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Jak rozumieć *polimorfizm*

Polimorfizm — (*gr. wielopostaciowość*) w przypadku podejścia obiektowo zorientowanego przez pojęcie *polimorfizmu* rozumie się możliwość stworzenia obiektu, który może mieć więcej niż jedną formę. W praktyce oznacza to, że może on mieć metodę, która dla tego samego typu obiektów mają różne definicje.

Wykorzystywanie funkcjonalności klasy bazowej

```
struct  KlasaBazowa { // ..........................................
  int Wartosc() const  { return  1; }
}; // ..............................................................

struct  KlasaPochodna:  public KlasaBazowa { // .....................
  int Wartosc() const  { return  9; }
}; // ..............................................................

int main( )
{
  KlasaPochodna  ObP;
  KlasaBazowa   *wskB = &ObP;

  cout << ObP.Wartosc() << endl;
  cout << ObP.KlasaBazowa::Wartosc() << endl;
  cout << wskB->Wartosc() << endl;
   . . .
```

Wykorzystywanie funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.........................................
        float  Pole( ) const  { return   0; }
}; //..................................................................

struct Kwadrat: public  FiguraGeometryczna { //........................
      float  _a;
      float  Pole( ) const  { return   _a*_a; }
}; //..................................................................


void  WyswietlPole( const  FiguraGeometryczna   & Fig )
{
      cout << "Pole: " << Fig.Pole( ) << endl;
}

int  main( )
{
      Kwadrat    Kw;

      Kw._a = 2;
      WyswietlPole( Kw );
}
```

Wykorzystywanie funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //..........................................
        float  Pole( ) const  { return   0; }
}; //...................................................................

struct Kwadrat: public  FiguraGeometryczna { //.........................
     float  _a;
     float  Pole( ) const  { return   _a*_a; }
}; //...................................................................


void  WyswietlPole( const  FiguraGeometryczna   & Fig )
{
     cout << "Pole: " << Fig.Pole( ) << endl;
}

int  main( )
{
     Kwadrat    Kw;

     Kw._a = 2;
     WyswietlPole( Kw );
}
```

Wykorzystywanie funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.........................................
        float  Pole( ) const  { return   0; }
}; //..................................................................

struct Kwadrat: public  FiguraGeometryczna { //........................
      float  _a;
      float  Pole( ) const  { return   _a*_a; }
}; //..................................................................


void  WyswietlPole( const  FiguraGeometryczna   & Fig )
{
      cout << "Pole: " << Fig.Pole( ) << endl;
}

int  main( )
{
      Kwadrat    Kw;

      Kw._a = 2;
      WyswietlPole( Kw );
}
```

Wykorzystywanie funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //.......................................
        float  Pole( ) const  { return   0; }
}; //...............................................................

struct Kwadrat: public  FiguraGeometryczna { //.......................
     float  _a;
     float  Pole( ) const  { return   _a*_a; }
}; //...............................................................


void  WyswietlPole( const  FiguraGeometryczna   & Fig )
{
     cout << "Pole: " << Fig.Pole( ) << endl;
}

int  main( )
{
     Kwadrat    Kw;

     Kw._a = 2;
     WyswietlPole( Kw );
}
```

Wynik działania:

Pole: 0

Wykorzystywanie funkcjonalności klasy bazowej

```
struct FiguraGeometryczna { //...................................
        float  Pole( ) const  { return   0; }
}; //............................................................

struct Kwadrat: public  FiguraGeometryczna { //..................
      float  _a;
      float  Pole( ) const  { return   _a*_a; }
}; //............................................................

void  WyswietlPole( const  FiguraGeometryczna   & Fig )
{
      cout << "Pole: " << Fig.Pole( ) << endl;
}

int  main( )
{
      Kwadrat    Kw;

      Kw._a = 2;
      WyswietlPole( Kw );
}
```

Dysponując klasą pochodną możemy skorzystać z funkcji stworzonych dla klasy bazowej. Operują one jednak tylko i wyłącznie na podobiektach klasy bazowej.

Wynik działania:

Pole: 0

Wykorzystywanie funkcjonalności klasy bazowej

```
void  WyswietlPole( const  FiguraGeometryczna  & Fig )
{
     float  Pole;

     switch  ( ??? ) {
          case  ? : Pole = Fig.Pole( );   break ;
          case  ? : Pole = static_cast <Kwadrat&>(Fig).Pole( );   break ;
     }

     cout << "Pole: " << Pole << endl;
}
```

Jeżeli w jakiś sposób odzyskamy informację o nadtypie obiektu, to możemy odpowiednio zmodyfikować funkcję, tak aby można było ją również stosować dla obiektów klasy pochodnej.

Rozpoznawanie typu poprzez identyfikator

```
struct FiguraGeometryczna { //.............................................
    FiguraGeometryczna( ): _TypID( ) { }

    int  ID( ) const  { return   _TypID; }
    float  Pole( ) const  { return   0; }

  protected :
    int  _TypID;

    FiguraGeometryczna( int  ID ): _TypID( ID )  { }
}; // ......................................................................


struct Kwadrat: public  FiguraGeometryczna { //.......
    float  _a;

    Kwadrat( ):  FiguraGeometryczna( 1 )  { }
    float  Pole( ) const  { return   _a*_a; }
}; // ..............................................
```

Aby móc rozpoznać typ nadobiektu można wprowadzić dodatkowe pole.

Rozpoznawanie typu poprzez identyfikator

```
struct FiguraGeometryczna { //.........................................
    FiguraGeometryczna( ): _TypID( ) { }

    int  ID( ) const  { return  _TypID; }
    float  Pole( ) const  { return  0; }

  protected :
    int  _TypID;

    FiguraGeometryczna( int  ID ): _TypID( ID )  { }
}; //..................................................................


struct Kwadrat: public  FiguraGeometryczna { //........
    float  _a;

    Kwadrat( ):  FiguraGeometryczna( 1 )  { }
    float  Pole( ) const  { return  _a*_a; }
}; //..................................................
```

Aby móc rozpoznać typ nadobiektu można wprowadzić dodatkowe pole. Musi ono zawierać wartość stanowiącą identyfikator danej klasy lub nadklasy. Identyfikator ten powinien być przypisany polu w momencie tworzenia obiektu, tzn. w konstruktorze.

Wykorzystanie identyfikatora typu

```
void WyswietlPole( const FiguraGeometryczna & Fig )
{
    float Pole;

    switch ( Fig.ID( ) ) {
        case 0 : Pole = Fig.Pole( );   break ;
        case 1 : Pole = static_cast <Kwadrat&>(Fig).Pole( );   break ;
    }

    cout << "Pole: " << Pole << endl;
}
```

Wprowadzone pole identyfikatora pozwala rozpoznać klasę, której częścią jest dany obiekt.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Definiowanie metod wirtualnych w klasie

```
struct FiguraGeometryczna { //..............................................
    float Pole( ) const { return 0; }
}; //.......................................................................

struct Kwadrat: public FiguraGeometryczna { //..............................
    float _DlugoscBoku;
    float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //.......................................................................


void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Definiowanie metod wirtualnych w klasie

```
struct FiguraGeometryczna { //..........................................
      virtual float Pole( ) const { return 0; }
}; //...................................................................

struct Kwadrat: public FiguraGeometryczna { //..........................
      float _DlugoscBoku;
      virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //...................................................................


void WyswietlPole( const FiguraGeometryczna & Fig )
{
      cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
      Kwadrat Kw;

      Kw._DlugoscBoku = 2;
      WyswietlPole( Kw );
}
```

Definiowanie metod wirtualnych w klasie

```
struct FiguraGeometryczna { //.........................................
    virtual float Pole( ) const { return 0; }
}; //..................................................................

struct Kwadrat: public FiguraGeometryczna { //.........................
    float _DlugoscBoku;
    virtual float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //..................................................................


void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Wprowadzenie metody wirtualnej w połączeniu z dziedziczeniem sprawia, że podobiekt *wie*, że jest składnikiem większej całości.

Wynik działania:

Pole: 4

Występowanie słowa kluczowego **virtual**

```
struct FiguraGeometryczna { //......................................
    virtual float Pole( ) const { return 0; }
}; //..............................................................

struct Kwadrat: public FiguraGeometryczna { //......................
    float _DlugoscBoku;
    float Pole( ) const { return _DlugoscBoku*_DlugoscBoku; }
}; //..............................................................

void WyswietlPole( const FiguraGeometryczna & Fig )
{
    cout << "Pole: " << Fig.Pole( ) << endl;
}

int main( )
{
    Kwadrat Kw;

    Kw._DlugoscBoku = 2;
    WyswietlPole( Kw );
}
```

Wynik działania:

Pole: 4

W klasie pochodnej nie jest konieczne używanie modyfikatora **virtual** do zdefiniowania metody wirtualnej. Wystarczy, że metoda ta z modyfikatorem **virtual** została zdefiniowana w klasie bazowej. Każda następna definicja w klasach pochodnych automatycznie uznawana jest jako definicja metody wirtualnej.

Definicja metody poza ciałem definicji klasy

```
struct FiguraGeometryczna { //..........................................
      virtual float  Pole( ) const;
}; //...................................................................

float FiguraGeometryczna::Pole( ) const
 {
   return  0;
 }

struct Kwadrat: public  FiguraGeometryczna { //.........................
      float  _DlugoscBoku;
      virtual  float  Pole( ) const;
}; //...................................................................

float Kwadrat::Pole( ) const
 {
   return  _DlugoscBoku*_DlugoscBoku;
 }
```

Metody wirtualne definiowane poza ciałem definicji klasy definiowane są tak samo jak zwykłe metody.
Modyfikator **virtual** występuje *tylko i wyłącznie w definicji klasy*.

Plan prezentacji

1. Kopiowanie obiektów
   - Klasy z polami wskaźnikowymi
   - Mechanizm przekazywania obiektów przez wartość
   - Konstruktor kopiujący
   - Klasa std::string
2. Rzutowanie
   - Rzutowanie w górę
   - Rzutowaie w dół
   - Rzutowanie w liście parametrów wywołania funkcji/metody
   - Utrata informacji przy rzutowaniu *w górę*
3. Metody wirtualne
   - Definiowanie metod wirtualnych
   - Destruktory wirtualne

Destruktory

```
struct FiguraGeometryczna { //......................................
    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //...............................................................

struct Kwadrat: FiguraGeometryczna { //.............................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
    ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //...............................................................


int  main( )
{
    FiguraGeometryczna   *wObFig = new   Kwadrat( );

    delete   wObFig;
}
```

Destruktory

```
struct FiguraGeometryczna { //.............................................
    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //......................................................................

struct Kwadrat: FiguraGeometryczna { //....................................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
    ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //......................................................................

int main( )
{
    FiguraGeometryczna *wObFig = new Kwadrat( );

    delete wObFig;
}
```

Wynik działania:

```
Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: FiguraGeometryczna
```

Destruktory

```
struct FiguraGeometryczna { //.............................................
    FiguraGeometryczna( )   { cout << "Konstruktor: FiguraGeometryczna" << endl; }
   ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //.......................................................................

struct Kwadrat: FiguraGeometryczna { //....................................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
   ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //.......................................................................


int main( )
{
    FiguraGeometryczna  *wObFig = new  Kwadrat( );

    delete  wObFig;
}
```

Wynik działania:

Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: FiguraGeometryczna

Destruktory

```
struct FiguraGeometryczna { //............................................
    FiguraGeometryczna( )   { cout << "Konstruktor: FiguraGeometryczna" << endl; }
   ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //......................................................................

struct Kwadrat: FiguraGeometryczna { //...................................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
   ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //......................................................................


int main( )
{
    FiguraGeometryczna  *wObFig = new  Kwadrat( );

    delete  wObFig;
}
```

Wynik działania:

```
Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: FiguraGeometryczna
```

Jeżeli w klasie bazowej zdefiniowany jest zwykły destruktor, to operacja destrukcji dokonywana za pośrednictwem wskaźnika na podobiekt inicjalizowana jest tylko dla klasy bazowej. Własność ta nie zależy od tego czy klasa bazowa ma metody wirtualne, czy też nie.

Definiowanie destruktorów wirtualnych

```
struct FiguraGeometryczna { // ..............................................

    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    virtual ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; // .......................................................................

struct Kwadrat: FiguraGeometryczna { // .....................................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
    virtual ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; // .......................................................................


int  main( )
{
    FiguraGeometryczna  *wObFig = new  Kwadrat( );

    delete  wObFig;
}
```

Definiowanie destruktorów wirtualnych

```
struct FiguraGeometryczna { // ..........................................

    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    virtual ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; // ...................................................................

struct Kwadrat: FiguraGeometryczna { // .................................
    ...
    Kwadrat( )    { cout << "Konstruktor: Kwadrat" << endl; }
    virtual ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; // ...................................................................


int  main( )
{
    FiguraGeometryczna   *wObFig = new   Kwadrat( );

    delete   wObFig;
}
```

Wynik działania:

```
Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: Kwadrat
Destruktor: FiguraGeometryczna
```

Definiowanie destruktorów wirtualnych

```
struct FiguraGeometryczna { //.........................................
    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    virtual ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //.................................................................

struct Kwadrat: FiguraGeometryczna { //...............................
    ...
    Kwadrat( )   { cout << "Konstruktor: Kwadrat" << endl; }
    virtual ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //.................................................................


int  main( )
{
    FiguraGeometryczna  *wObFig = new  Kwadrat( );

    delete  wObFig;
}
```

Wynik działania:

```
Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: Kwadrat
Destruktor: FiguraGeometryczna
```

Definiowanie destruktorów wirtualnych

```
struct FiguraGeometryczna { //.............................................
    FiguraGeometryczna( )    { cout << "Konstruktor: FiguraGeometryczna" << endl; }
    virtual ~FiguraGeometryczna( ) { cout << "Destruktor: FiguraGeometryczna" << endl; }
}; //......................................................................

struct Kwadrat: FiguraGeometryczna { //....................................
    ...
    Kwadrat( )    { cout << "Konstruktor: Kwadrat" << endl; }
    virtual ~Kwadrat( ) { cout << "Destruktor: Kwadrat" << endl; }
}; //......................................................................


int main( )
{
    FiguraGeometryczna  *wObFig = new  Kwadrat( );

    delete  wObFig;
}
```

Wynik działania:

Konstruktor: FiguraGeometryczna
Konstruktor: Kwadrat
Destruktor: Kwadrat
Destruktor: FiguraGeometryczna

Wprowadzenie destruktora wirtualnego umożliwia właściwą destrukcję obiektu. Obiekt w momencie destrukcji *wie*, że jest częścią większej całości.

Podsumowanie

- Polimorfizm (wielopostaciowość) jest jedną z ważniejszych cech programowania obiektowego. Wraz z dziedziczeniem jest on podstawowym mechanizmem wykorzystywanym przy tworzeniu bibliotek obiektowo zorientowanych.
- Polimorfizm wraz z możliwością niejawnego rzutowaniem *w górę* pozwala tworzyć uniwersalne narzędzia dla wszystkich obiektów należących do klas dziedziczących daną klasę bazową.
- Destruktory wirtualne wspierają operacje na obiektach tworzonych dynamicznie. Dzięki niejawnemu rzutowaniu *w górę* w narzędziach stworzonych dla klasy bazowej, obiekty te można bezpiecznie poddawać destrukcji.
- Metody wirtualne udostępniają wygodny mechanizm tworzenia własnego systemu identyfikacji typu w trakcie działania programu.

Koniec prezentacji
Dziękuję za uwagę