Wartości domyślne, konstruktory kopiujące, std::string, szablony

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Wydziału Elektroniki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

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

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

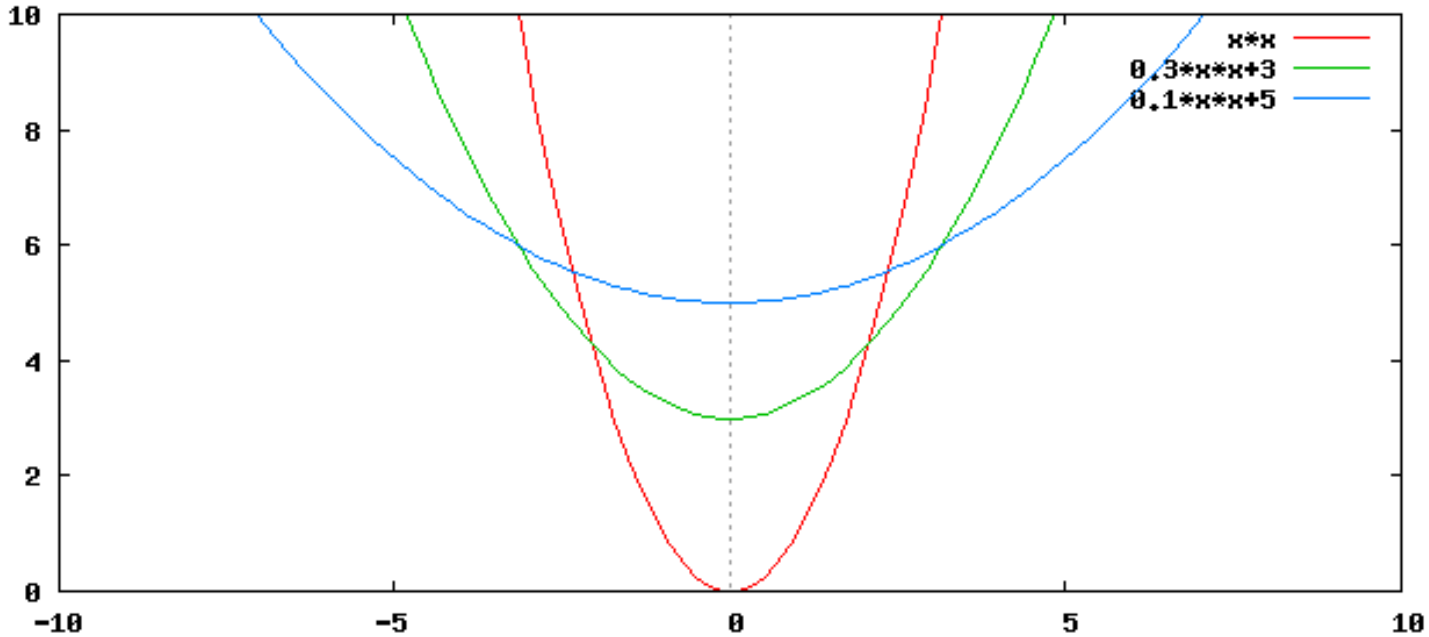
Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Plan prezentacji

Plan prezentacji

Cel

Zbudujmy program, który umożliwi wyliczanie współrzędnych punktów dla rodziny parabol:



Ogólna postać równania:

$$y = ax^2 + b$$

Implementacja

```
float Parabola(float x, float a, float b)
{
    return a*x*x+b;
}
```

Implementacja

```
float Parabola(float x, float a, float b)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,1,0);
}
```

Definiowanie wartości domyślnych

```
float Parabola(float x, float a, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,1,0);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,1);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,1);   // ⇒ Parabola(x,1,0)
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x,1);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x);   // ⇒ Parabola(x,1,0)
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x);
}
```

Co zrobić jeśli chcemy aby $a = 2$?

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,2);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x,2);   // ⇒ Parabola(x,2,0)
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x,2);
}
```

A jeśli ma być $b = 5$?

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,2,5);
}
```

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x,2,5);
}
```

A jeśli chcemy aby $a = 1$ i $b = 5$?

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x,5);
}
```

A jeśli chcemy aby $a = 1$ i $b = 5$? Czy można tak?

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}

int main()
{
    float y, x=5;
    y = Parabola(x, ,5);
}
```

A jeśli chcemy aby $a = 1$ i $b = 5$? A może tak?

Korzystanie z wartości domyślnych

```
float Parabola(float x, float a = 1, float b = 0)
{
    return a*x*x+b;
}


int main()
{
    float y, x=5;
    y = Parabola(x,1,5);
}
```

Jeśli chcemy zmodyfikować wartość jednego z parametrów domyślnych, to musimy podać wartości wszystkich parametrów, aż do miejsca, na którym występuje zmodyfikowana wartość.

Korzystanie z wartości domyślnych

void PokazOdleglosc(**float** t, **float** a = 9.81, **const char*** Jedn = "m")

Korzystanie z wartości domyślnych

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Korzystanie z wartości domyślnych

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}

int main( )
{
   PokazOdleglosc(1, 9.81, "cm");
}
```

Korzystanie z wartości domyślnych

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}

int main( )
{
   PokazOdleglosc(1, "cm");
}
```

Czy można to skrócić korzystając z tego, że zadeklarowane wartości domyślne dla *a* i *Jedn* są różne od *Jedn*?

Korzystanie z wartości domyślnych

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}

int main( )
{
   PokazOdleglosc(1, "cm");
}
```

Tego typu wywołanie jest niepoprawne. To że parametry są różnego typu nie ma żadnego znaczenia.

Korzystanie z wartości domyślnych

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}

int main( )
{
   PokazOdleglosc(1, 9.81, "cm");
}
```

Jedyną poprawną formą jest wypisanie wszystkich wartości parametrów pośrednich, niezależnie od tego czy są one zgodne z wartościami domyślnymi, czy też nie.

Zapowiedź definicji funkcji

```
int main( )
{
   PokazOdleglosc(1);
}

void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Zapowiedź definicji funkcji

```
int main( )
{
   PokazOdleglosc(1);
}

void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Wywołanie funkcji przed jej definicją jest możliwe wtedy i tylko wtedy, gdy wcześniej wystąpi nagłówek zapowiedzi definicji.

Zapowiedź definicji funkcji

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}


void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

To też nie jest dobrze.

Zapowiedź definicji funkcji

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}


void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Jeżeli w jednostce translacyjnej występuje zapowiedź definicji z parametrami domyślnymi oraz sama definicja, to wartości tych parametrów mogą być zadeklarowane tylko raz w nagłówku jej zapowiedzi.

Zapowiedź definicji funkcji

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}


void PokazOdleglosc( float t, float a, const char* Jedn )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Teraz jest już dobrze.

Zapowiedź definicji funkcji – moduły

prog.cpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
void PokazOdleglosc( float t, float a, const char* Jedn )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Jeżeli występuje więcej jednostek translacyjnych, to ta sama funkcja może być różnie traktowana w każdej z nich.

Zapowiedź definicji funkcji – moduły

prog.cpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
void PokazOdleglosc( float t, float a = 981, const char* Jedn = "cm" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Wartości parametrów domyślnych mogą być różne w każdej z nich lub w ogóle mogą nie występować. To nie jest jednak dobre rozwiązanie.

Zapowiedź definicji funkcji – moduły

prog.cpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );


int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
void PokazOdleglosc( float t, float a = 981, const char* Jedn = "cm" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Wartości parametrów domyślnych mogą być różne w każdej z nich lub w ogóle mogą nie występować. To nie jest jednak dobre rozwiązanie.

Zapowiedź definicji funkcji – moduły

modul.hpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );
```

prog.cpp

```
int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
void PokazOdleglosc( float t, float a = 981, const char* Jedn = "cm" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Lepszym rozwiązaniem jest wydzielenie pliku nagłówkowe.

Zapowiedź definicji funkcji – moduły

modul.hpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );
```

prog.cpp

```
#include "modul.hpp"

int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
void PokazOdleglosc( float t, float a = 981, const char* Jedn = "cm" )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Zapowiedź definicji funkcji – moduły

modul.hpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );
```

prog.cpp

```
#include "modul.hpp"

int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
#include "modul.hpp"
void PokazOdleglosc( float t, float a, const char* Jedn )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Dodanie pliku nagłówkowego zapobiega niejednoznacznemu traktowania parametrów domyślnych.

Zapowiedź definicji funkcji – moduły

modul.hpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );
```

prog.cpp

```
#include "modul.hpp"

int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
#include "modul.hpp"
void PokazOdleglosc( float t, float a, const char* Jedn )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Nie jest to jednak jedyny niezbędny plik nagłówkowy.

Zapowiedź definicji funkcji – moduły

modul.hpp

```
void PokazOdleglosc( float t, float a = 9.81, const char* Jedn = "m" );
```

prog.cpp

```
#include "modul.hpp"

int main( )
{
   PokazOdleglosc(1);
}
```

modul.cpp

```
#include "modul.hpp"
#include <cmath>
void PokazOdleglosc( float t, float a, const char* Jedn )
{
   cout << a*pow(t,2)/2 << Jedn << endl;
}
```

Plan prezentacji

Domyślne wartości parametrów dla metod

```
class LZespolona {
    public :
        float  re, im;
        void  Zmien(float  r, float  i){ re = r;  im = i; }
};


int  main( )
{
     LZespolona  Z;
     Z.re = 2.1;    Z.im = 0;
}
```

Domyślne wartości parametrów dla metod

```
class LZespolona {
    public :
        float  re, im;
        void  Zmien(float  r, float  i){ re = r;  im = i; }
};


int  main( )
{
     LZespolona   Z;
     Z.Zmien(2.1, 0);
}
```

Domyślne wartości parametrów dla metod

```
class LZespolona {
    public :
        float re, im;
        void Zmien(float r, float i = 0){ re = r; im = i; }
};


int main( )
{
    LZespolona Z;
    Z.Zmien(2.1);
}
```

Definiowanie metody poza ciałem klasy

```
class LZespolona {
    public :
        float re, im;

        void Zmien(float r, float i = 0);
};

void LZespolona::Zmien(float r, float i)
{
    re = r;   im = i;
}

int main( )
{
    LZespolona Z;

    Z.Zmien(2.1);
}
```

Wartości domyślne versus przeciążenie

```
class LZespolona {
    public :
        float  re, im;
        void  Zmien(float  r, float  i = 0) { re = r;  im = i; }
};


int  main( )
{
     LZespolona  Z;
     Z.Zmien(2.1);
}
```

Wartości domyślne versus przeciążenie

```
class LZespolona {
    public :
        float  re, im;

        void  Zmien(float  r, float  i) { re = r;  im = i; }
        void  Zmien(float  r) { re = r;  im = 0; }
};


int  main( )
{
     LZespolona  Z;

     Z.Zmien(2.1);
}
```

Plan prezentacji

Konstruktor bezparametryczny

```
class LZespolona {
    public :
        float  re, im;



};


int  main( )
{
     LZespolona   Z1;


}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;


        LZespolona( ) { re = im = 0; }
};


int  main( )
{
     LZespolona   Z1;


}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;


        LZespolona( ) { re = im = 0; }
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);

}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;

        LZespolona(float  r ,float  i) { re = r;  im = i;}
        LZespolona( ) { re = im = 0; }
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);

}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;


        LZespolona(float  r = 0, float  i = 0) { re = r;  im = i;}
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);

}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;


        LZespolona(float  r = 0, float  i = 0) { re = r;  im = i;}
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);
     LZespolona   Z3(4);
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;

        LZespolona( ) { re = im = 0; }
        LZespolona(float  r) { re = r;  im = 0;}
        LZespolona(float  r, float  i) { re = r;  im = i;}
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);
     LZespolona   Z3(4);
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;

        LZespolona( ) { re = im = 0; }
        LZespolona(float  r) { re = r;  im = 0;}
        LZespolona(float  r, float  i) { re = r;  im = i;}
};


int  main( )
{
     LZespolona   Z1;
     LZespolona   Z2(4, 6);
     LZespolona   Z3(4);
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float  re, im;

        LZespolona( ) { re = im = 0; }
        LZespolona(float  r) { re = r;  im = 0;}
        LZespolona(float  r, float  i) { re = r;  im = i;}
};


int  main( )
{
     LZespolona   Z;

     Z = 9;
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
  public :
    float re, im;

    bool operator == (const LZespolona Z2) const { ... }
    LZespolona( ) { re = im = 0; }
    LZespolona(float r) { re = r; im = 0;}
    LZespolona(float r, float i) { re = r; im = i;}
};


int main( )
{
    LZespolona Z;
    if ( Z == 7 ) { ... }
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        LZespolona( ) { re = im = 0; }
        LZespolona(float r) { re = r;  im = 0;}
        LZespolona(float r, float i) { re = r;  im = i;}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LiczbaD);
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        LZespolona( ) { re = im = 0; }
        explicit LZespolona(float r) { re = r; im = 0;}
        LZespolona(float r, float i) { re = r; im = i;}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LiczbaD);
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        LZespolona(float r = 0, float i = 0) { re = r; im = i;}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LZespolona(LiczbaD));
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        explicit LZespolona(float r = 0, float i = 0) {...}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LZespolona(LiczbaD));
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        explicit LZespolona( ) { re = im = 0; }
        explicit LZespolona(float r) { re = r; im = 0;}
        explicit LZespolona(float r, float i) { re = r; im = i;}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LZespolona(LiczbaD));
}
```

Konsekwencje wartości domyślnych

```
class LZespolona {
    public :
        float re, im;

        explicit LZespolona(float r = 0, float i = 0) {...}
};

LZespolona FunkcjaZ(LZespolona Z) { ... }

int main( )
{
    double LiczbaD;
    FunkcjaZ(LZespolona(LiczbaD));
}
```

Problemy . . .

```
class Wektor {
    public :
        float  x, y;
        bool  operator  == (const  Wektor  W2) const  { . . . }
        Wektor(float  x_nowa = 0, float  y_nowa = 0) { . . . }
};


int  main( )
{
     Wektor  W;
     if  ( W == 7 ) { . . . }
}
```

Problemy . . .

```
class Wektor {
    public :
        float  x, y;
        bool  operator  == (const  Wektor  W2) const  { . . . }
        explicit  Wektor(float  x_nowa = 0, float  y_nowa = 0) { . . . }
};


int  main( )
{
     Wektor  W;
     if  ( W == 7 ) { . . . }
}
```

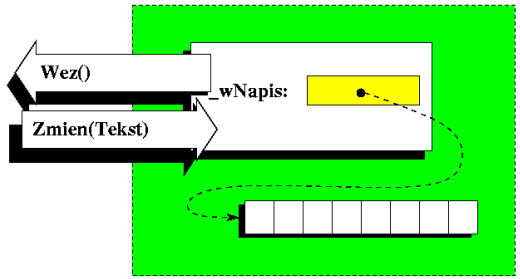
Plan prezentacji

Definicja klasy `Napis`

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




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

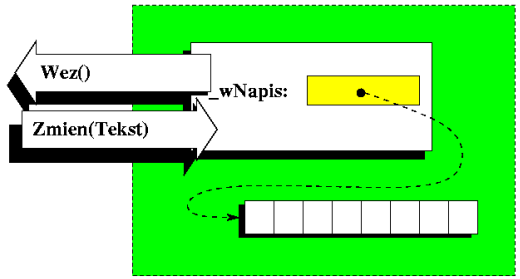
Definicja klasy `Napis`

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



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



Definicja klasy `Napis`

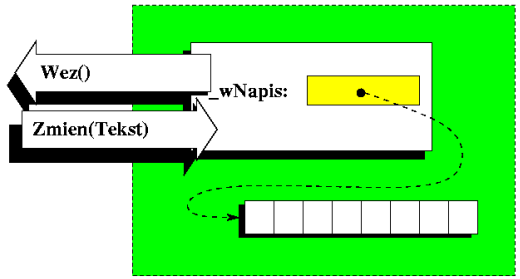
```
class KopiaNapisu { //.....
    char *_wNapis;
  public :

    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`

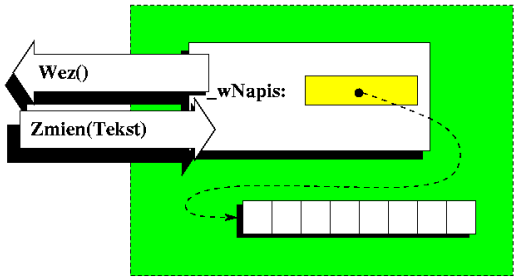
```
class KopiaNapisu { //.....
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }

    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`

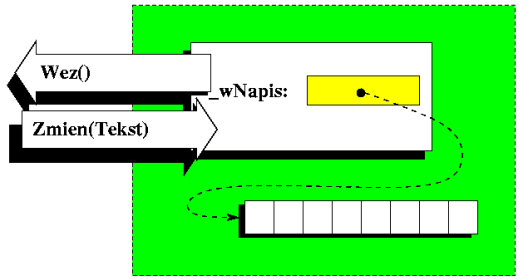
```
class KopiaNapisu { //.....
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }

    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`

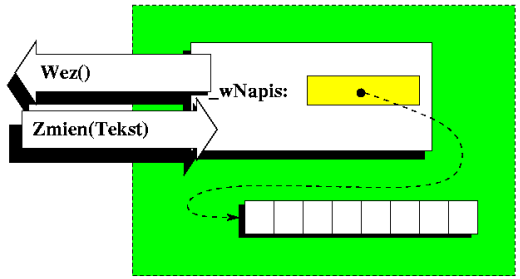
```
class KopiaNapisu { //.....
    char *_wNapis;
  public :
    KopiaNapisu( ) { _wNapis = nullptr; }

    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`

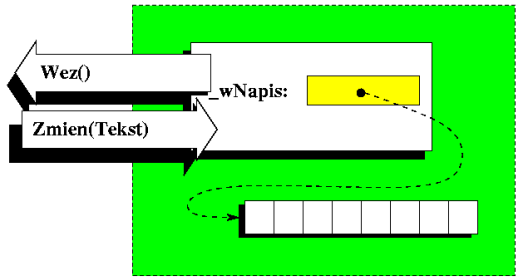
```
class KopiaNapisu { //.....
    char *_wNapis;
  public:
    KopiaNnapisu( ) { _wNapis = nullptr; }

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

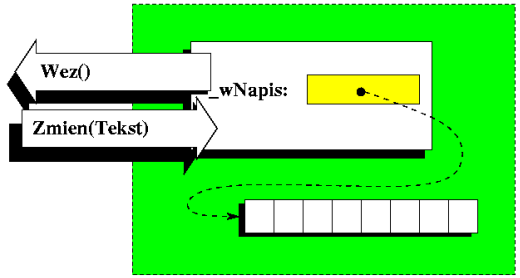



```
void KopiaNapisu::Zmien( const char* wTekst )
 {
   delete [ ] _wNapis;  _wNapis = nullptr;
   if (wTekst && (_wNapis = new char[strlen(wTekst)+1])) strcpy(_wNapis, wTekst);
 }
```

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

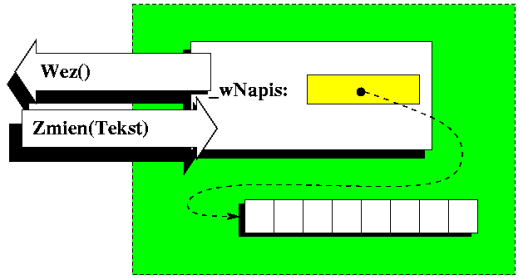
Klasa Napis w akcji

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



Klasa Napis w akcji

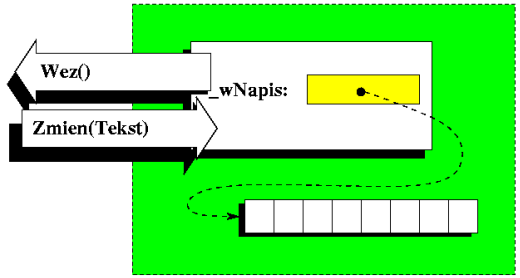
```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ..................................................
```




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

Klasa Napis w akcji

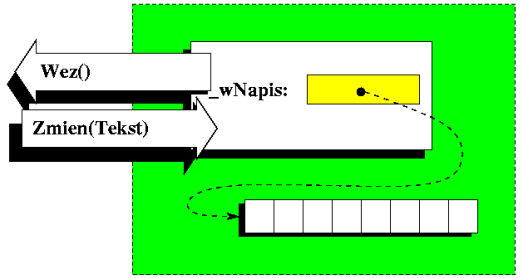
```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ..................................................
```




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

int main( )
{




}
```

Klasa Napis w akcji

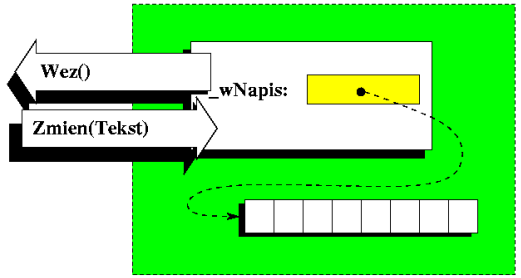
```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ..................................................
```




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

int main( )
{
   KopiaNapisu Ob;




}
```

Klasa Napis w akcji

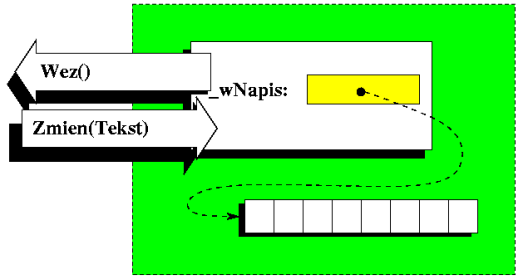
```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ..................................................
```




```
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 *_wNnapis;
   public :
         ...
}; // ..................................................
```

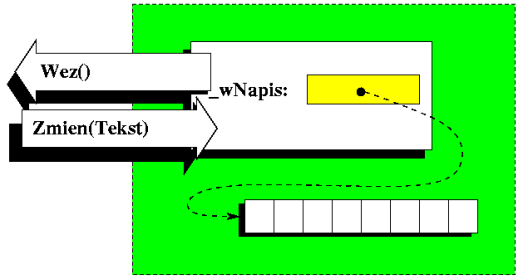



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

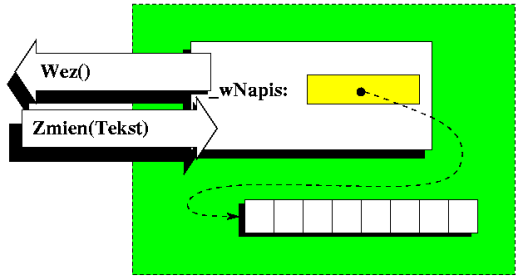



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

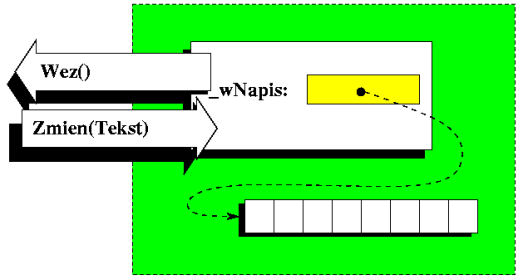



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

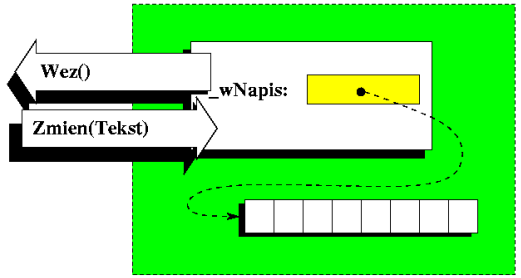
```
class KopiaNapisu { // ..................................
      char *_wNapis;
   public :
         ...
}; // ..................................................
```




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

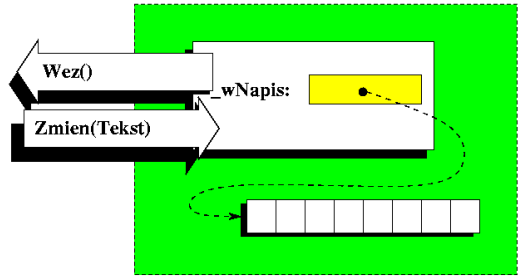



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

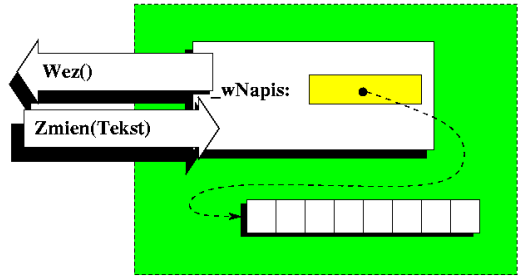



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

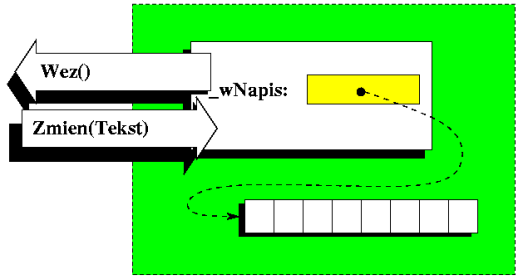



```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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



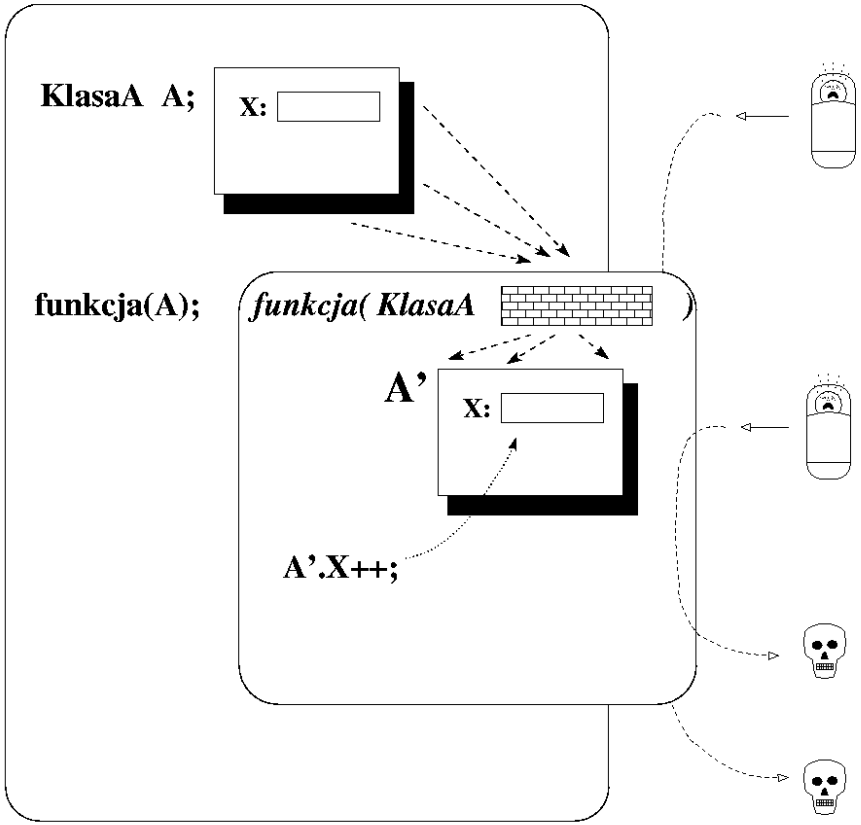

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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?

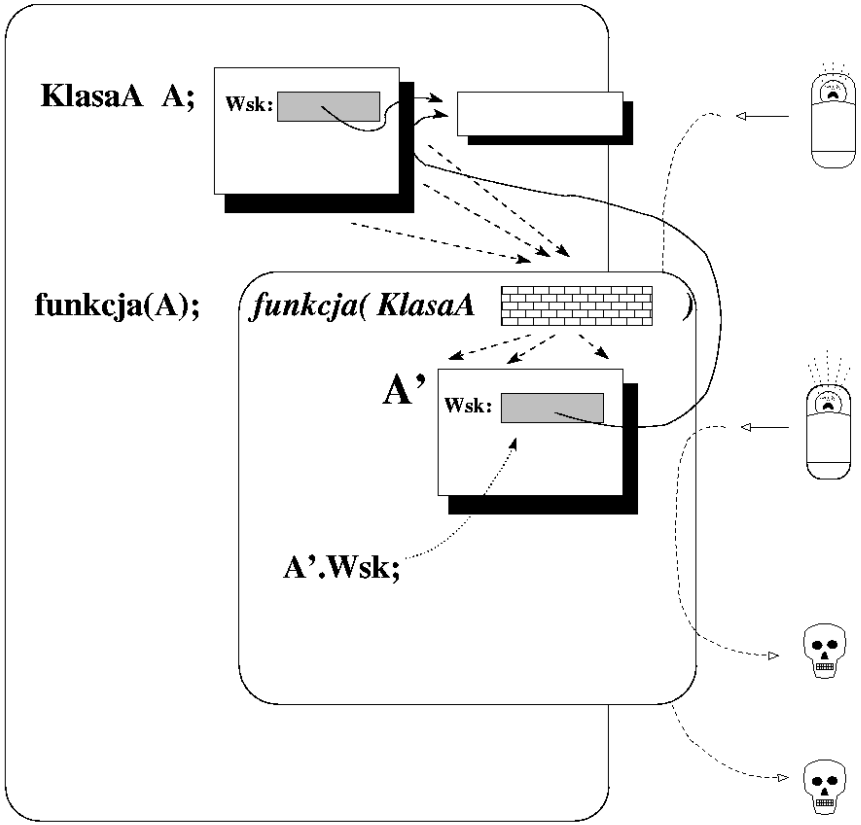
Plan prezentacji

Jak to działa



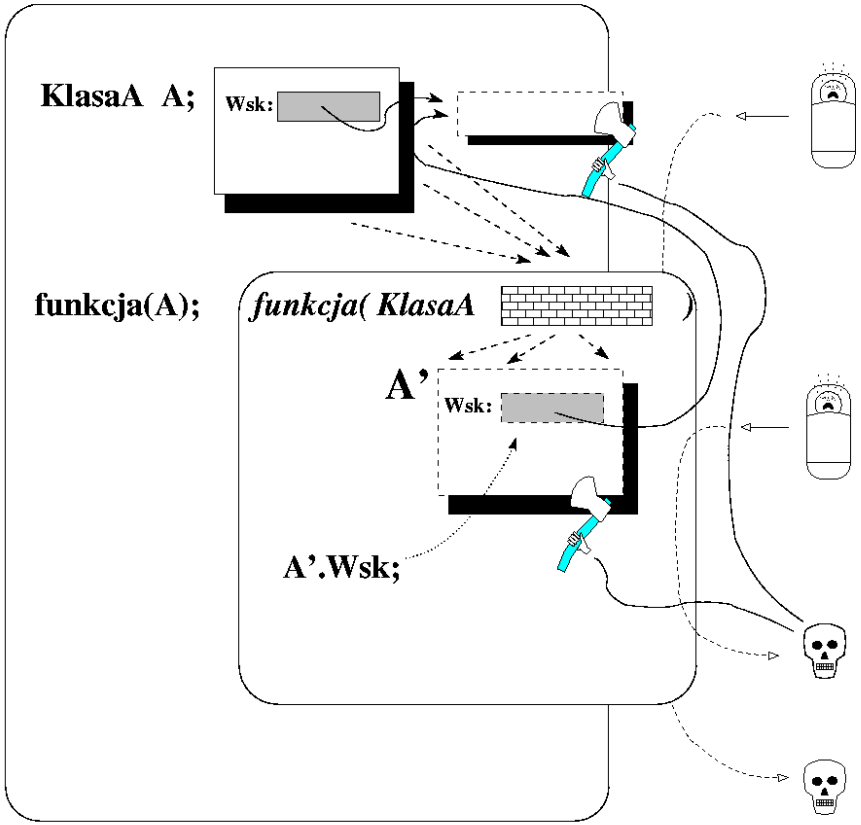
Przekazaniu parametru wywołania funkcji/metody przez wartość towarzyszy utworzenie jego kopii na poziomie tej funkcji/metody. Istnienie kopii kończy się wraz z zakończeniem działania funkcji/metody.

Jak to działa



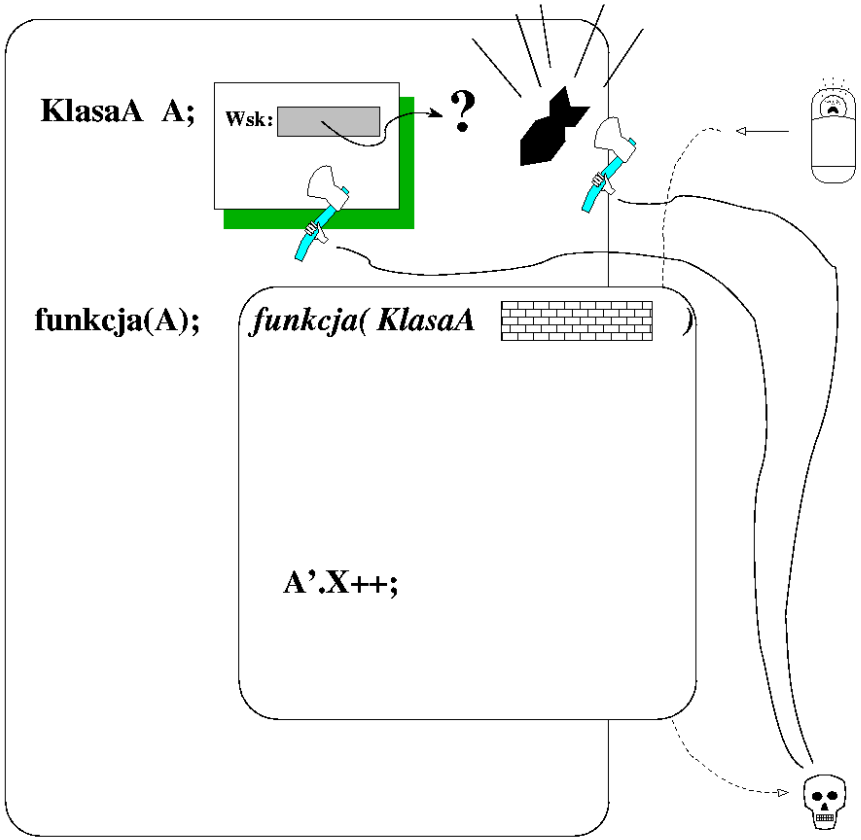
Przy tworzeniu kopii przepisana zostaje cała zawartość obiektu łącznie z zawartością pól wskaźnikowych. Powoduje to, że dwa obiekty połączone są poprzez wskaźniki z tymi samymi strukturami.

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.

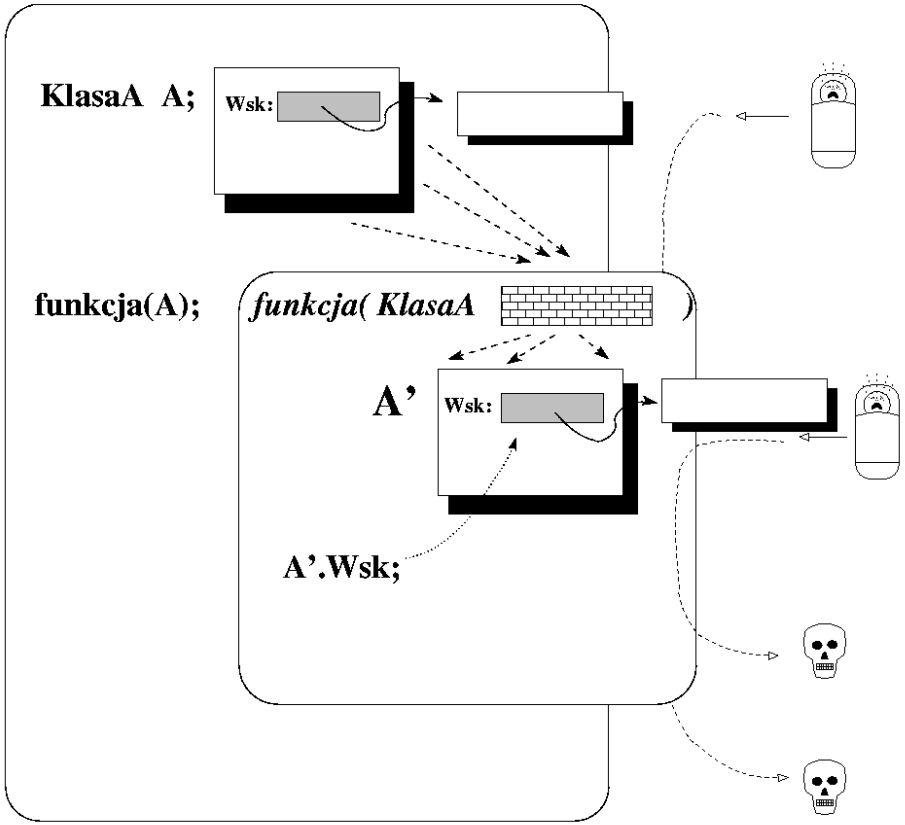
Jak to działa



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

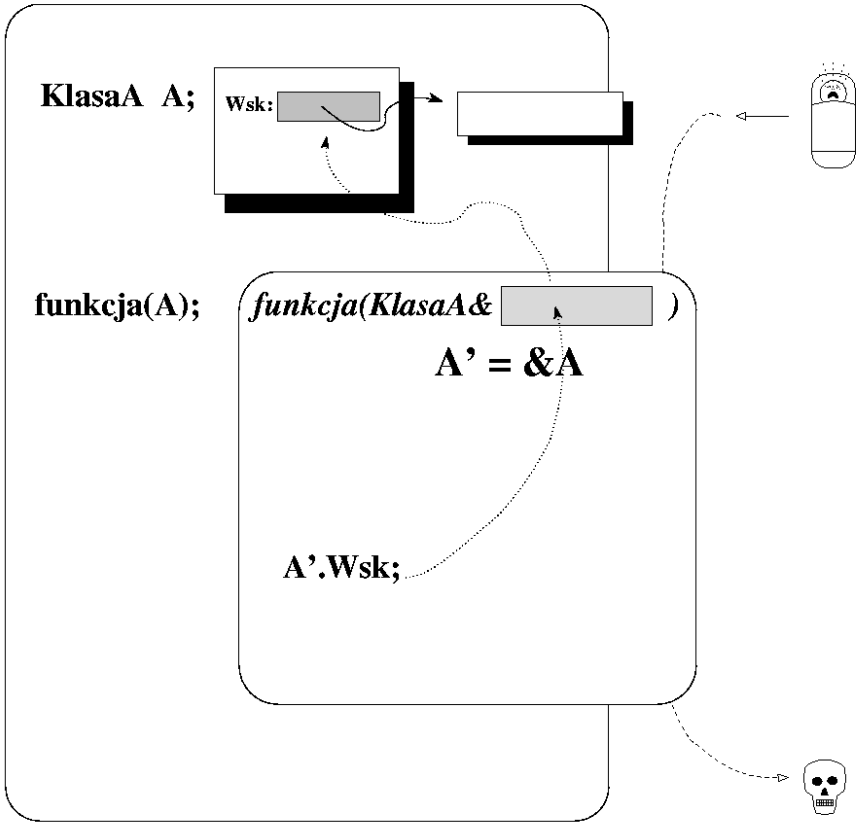
(zwykle jest to: `segmentation fault` :-) .

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

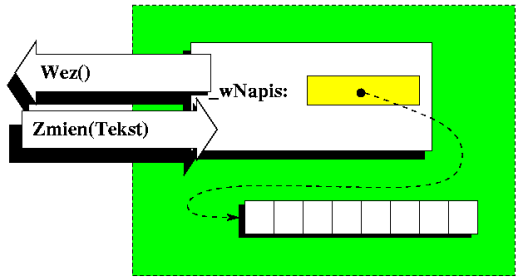


Przekazywanie obiektu przez referencję pozwala uniknąć tworzenia kopii obiektu, która czasem może być zbędna.

Plan prezentacji

Kopiowanie zawartości

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



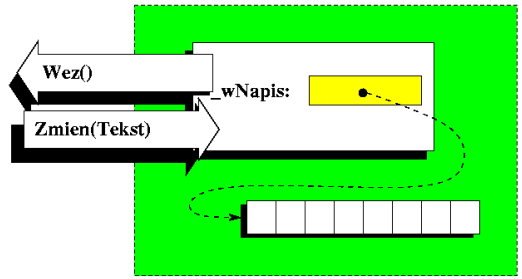

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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



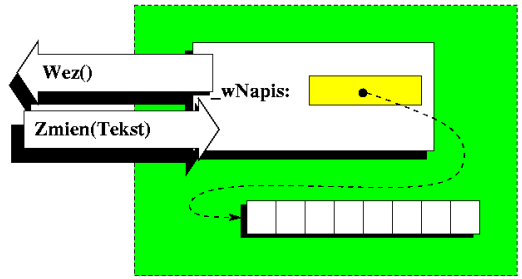

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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



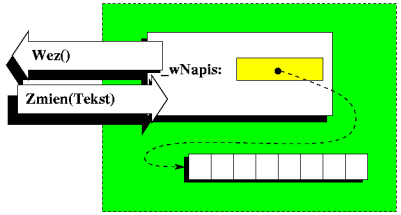

```
void Wyswietl(KopiaNapisu Ob) { cout << "Wyswietl: " << Ob.Wez( ) << endl; }

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

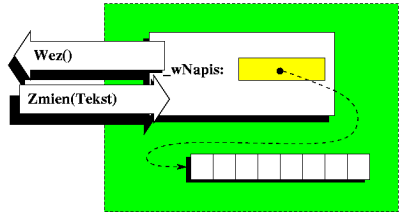
```
class KopiaNapisu { // ..........................................
      char *_wNapis;
   public :
        ...
}; // ..........................................................
```




```
int main( )
{
   KopiaNapisu Ob1;




}
```

Operacja podstawienia

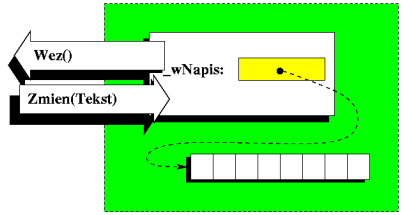
```
class KopiaNapisu { // ..........................................
      char *_wNapis;
   public :
         ...
}; // ..........................................................
```




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




}
```

Operacja podstawienia

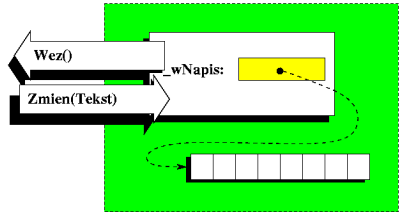
```
class KopiaNapisu { // ..........................................
      char *_wNnapis;
   public :
         ...
}; // ..........................................................
```




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




   }

}
```

Operacja podstawienia

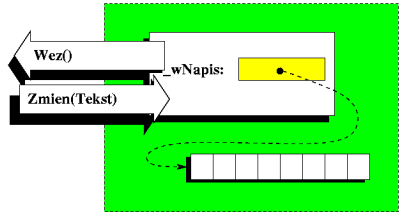
```
class KopiaNapisu { // ..........................................
      char *_wNapis;
   public :
        ...
}; // ..........................................................
```




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



   }

}
```

Operacja podstawienia

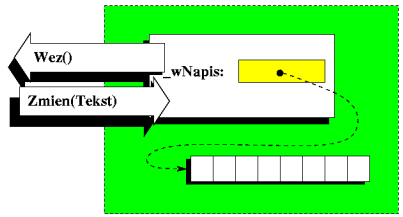
```
class KopiaNapisu { // ..........................................
      char *_wNapis;
   public :
         ...
}; // ..........................................................
```




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


   }

}
```

Operacja podstawienia

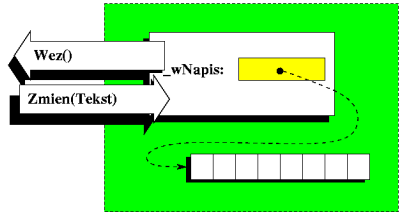
```
class KopiaNapisu { // ..........................................
      char *_wNapis;
   public :
         ...
}; // ..........................................................
```




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

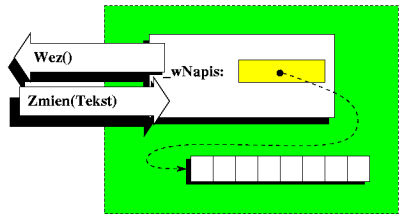



```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
     KopiaNapisu Ob2;
     Ob2 = Ob1
     cout << "Ob1: " << Ob1.Wez( ) << endl;

   }

}
```

Ob1: Hippasus z Metapontum

Operacja podstawienia

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

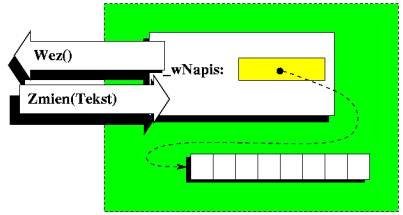



```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
      KopiaNapisu Ob2;
      Ob2 = Ob1
      cout << "Ob1: " << Ob1.Wez( ) << endl;
      cout << "Ob2: " << Ob2.Wez( ) << endl;
   }

}
```

Ob1: Hippasus z Metapontum

Operacja podstawienia

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

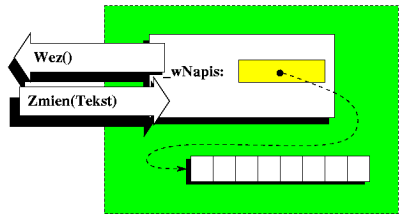



```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
      KopiaNapisu Ob2;
      Ob2 = Ob1
      cout << "Ob1: " << Ob1.Wez( ) << endl;
      cout << "Ob2: " << Ob2.Wez( ) << endl;
   }

}
```

Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum

Operacja podstawienia

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

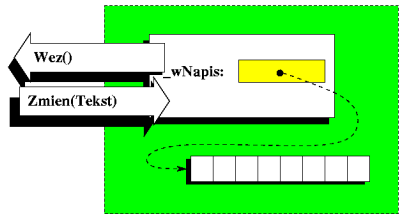



```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
     KopiaNapisu Ob2;
     Ob2 = Ob1
     cout << "Ob1: " << Ob1.Wez( ) << endl;
     cout << "Ob2: " << Ob2.Wez( ) << endl;
   }
   cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

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

Operacja podstawienia

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

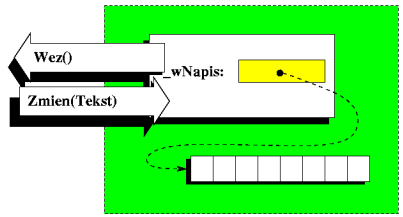



```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
      KopiaNapisu Ob2;
      Ob2 = Ob1
      cout << "Ob1: " << Ob1.Wez( ) << endl;
      cout << "Ob2: " << Ob2.Wez( ) << endl;
   }
   cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

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

Operacja podstawienia

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



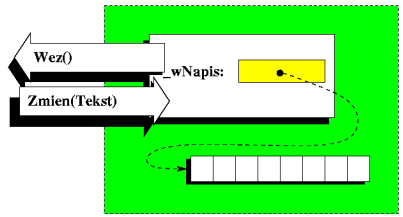

```
int main( )
{
   KopiaNapisu Ob1;
   Ob1.Zmien( "Hippasus z Metapontum" );
   {
      KopiaNapisu Ob2;
      Ob2 = Ob1
      cout << "Ob1: " << Ob1.Wez( ) << endl;
      cout << "Ob2: " << Ob2.Wez( ) << endl;
   }
   cout << "Ob1: " << Ob1.Wez( ) << endl;
}
```

```
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;
}
```

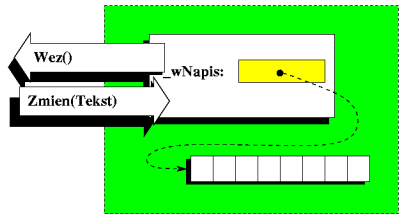



```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1:
```

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;
}
```

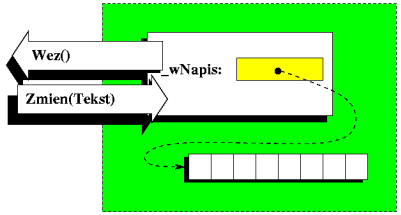



```
Ob1: Hippasus z Metapontum
Ob2: Hippasus z Metapontum
Ob1: Hippasus z Metapontum
```

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

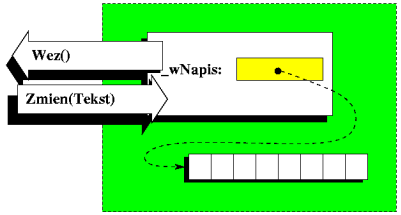
Inicjalizacja obiektu

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



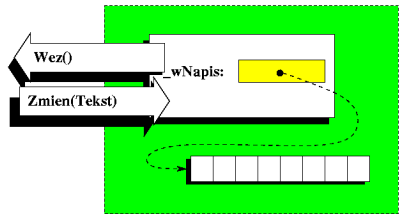
Inicjalizacja obiektu

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



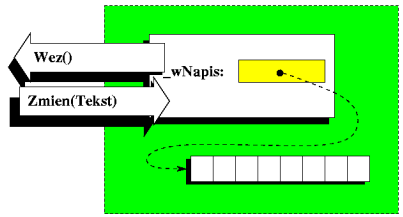
Inicjalizacja obiektu

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


int main( )
{
   KopiaNapisu Ob1;



 }
```



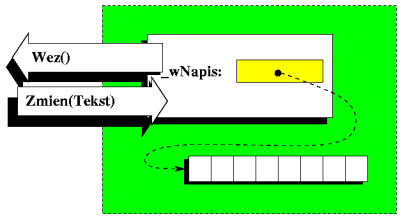
Inicjalizacja obiektu

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

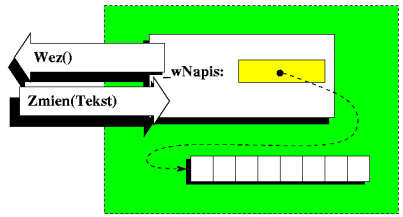
```
class KopiaNapisu { //.....................................
      char *_wNapis;
   public :
        ...
      KopiaNapisu( const KopiaNapisu & Ob )
                                { _wNapis = nullptr; Zmien(Ob._wNapis); }
      KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                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




```
int main( )
{
   KopiaNapisu Ob1;

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

   KopiaNapisu Ob2(Ob1);

 }
```

Inicjalizacja obiektu

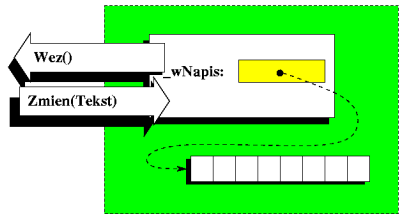
```
class KopiaNapisu { //.....................................
      char *_wNapis;
   public :
        ...
      KopiaNapisu( const KopiaNapisu & Ob )
                                { _wNapis = nullptr; Zmien(Ob._wNapis); }
      KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                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




```
int main( )
{
   KopiaNapisu Ob1;

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

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

Inicjalizacja obiektu

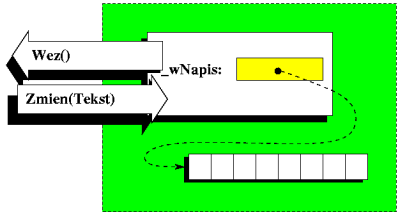
```
class KopiaNapisu { //.....................................
      char *_wNapis;
   public :
        ...
      KopiaNapisu( const KopiaNapisu & Ob )
                                { _wNapis = nullptr; Zmien(Ob._wNapis); }
      KopiaNapisu & operator = ( const KopiaNapisu &Ob )
                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




```
int main( )
{
   KopiaNapisu Ob1;

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

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

Inicjalizacja obiektu

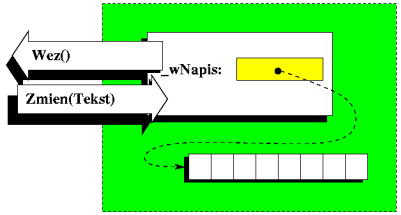
```
class KopiaNapisu { //.....................................
      char *_wNapis;
   public :
        ...
      KopiaNapisu( const KopiaNapisu & Ob )
          { cout << "K. kopiujacy" << endl; _wNapis = nullptr; Zmien(Ob._wNapis); }
      KopiaNapisu & operator = ( const KopiaNapisu &Ob )
      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



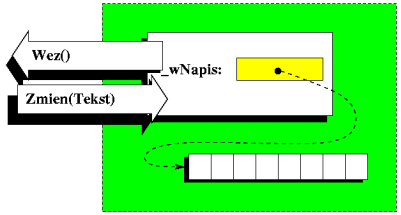
Wynik działania

K. kopiujacy
K. kopiujacy

```
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

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

Plan prezentacji

Co się zmienia gdy pojawią się pola referencyjne

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


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

Co się zmienia gdy pojawią się pola referencyjne

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


};   //.....


int  main( )
{
      Wektor2    W1;


}
```

Co się zmienia gdy pojawią się pola referencyjne

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


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



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



}
```

Co się zmienia gdy pojawią się pola referencyjne

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

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

int  main( )
{
      Wektor2    W1;


}
```

Co się zmienia gdy pojawią się pola referencyjne

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

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

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


}
```

Co się zmienia gdy pojawią się pola referencyjne

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

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

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


}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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);


}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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);


}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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;
}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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;
}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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;
}
```

Co się zmienia gdy pojawią się pola referencyjne

```
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;
}
```

Ważny wyjątek

W klasach zawierających pola referencyjne nie ma niejawnego zdefiniowanego:

- **konstruktora bezparametrycznego,**
- **konstruktora kopiującego,**
- **operatora podstawienia.**

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.

Plan prezentacji

Dlaczego szablony?

W językach takich jak *C* i *Pascal* mamy do czynienia z separacją kodu i typu parametrów. Wartości z jakimi wywoływane są funkcje i procedury mogą parametryzować ich działanie. Jednak ich typy zostają ustalone raz na zawsze w momencie ich definicji.

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Dlaczego szablony?

W językach takich jak *C* i *Pascal* mamy do czynienia z separacją kodu i typu parametrów. Wartości z jakimi wywoływane są funkcje i procedury mogą parametryzować ich działanie. Jednak ich typy zostają ustalone raz na zawsze w momencie ich definicji.

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Dlaczego szablony?

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****∗ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****∗ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Najlepszym rozwiązaniem dla postawionego problemu jest koncepcja szablonów.

Dlaczego szablony?

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Najlepszym rozwiązaniem dla postawionego problemu jest koncepcja szablonów.

Podstawowe cechy

- Szablony pozwalają na definiowanie funkcji, których typy parametrów są także parametrami tych funkcji.
- Możliwe jest definiowanie klas, które parametryzowane mogą być typami pól występujących w tych klasach i/lub też typami parametrów metod.
- Programista definiuje tylko raz dany szablon. Kompilator dokonuje dedukcji typów parametrów danego szablonu i konkretyzuje go tworząc kod dla użytych typów w wywołaniu funkcji lub definicji obiektu danej klasy.
- Programista może też jawnie określić “wartości” parametrów szablonu.

Wady i zalety

Zalety:

- ⋆ Szablony dają możliwość tworzenia uniwersalnych algorytmów i uniwersalnych struktur danych.
- ⋆ W odróżnieniu od makr możliwe jest zachowanie przejrzystości kodu.
- ⋆ W odróżnieniu od wykorzystywania typów bazowych pozwalają zachować ścisłą kontrolę typów w trakcie kompilacji.

Wady:

- ◦ Brak możliwości tworzenia oddzielnych jednostek kompilacji (modułów) w postaci "czystych" szablonów.

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

enum Symbole { a=1, b, c };

int main( )
{
    cout << Max(1,2) << endl;
    cout << Max(1.1, 2.2) << endl;
    cout << Max('A','B') << endl;
    cout << Max( a , b ) << endl;
}
```

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ  Max( Typ  w1, Typ  w2 )
{
    return   w1 < w2 ? w2 : w1;
}

enum   Symbole { a=1, b, c };

int  main( )
{
    cout <<  Max(1,2) <<  endl;
    cout <<  Max(1.1, 2.2) <<  endl;
    cout <<  Max('A','B') <<  endl;
    cout <<  Max( a , b ) <<  endl;
}
```

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ  Max( Typ  w1, Typ  w2 )
{
    return  w1 < w2 ? w2 : w1;
}
```

W tym przykładzie kompilator generuje kod funkcji *Max* dla czterech przypadków. Słowo kluczowe **class** może zostać zastąpione przez **typename**.

```
enum  Symbole { a=1, b, c };

int  main( )
{
    cout <<  Max(1,2) <<  endl;
    cout <<  Max(1.1, 2.2) <<  endl;
    cout <<  Max('A','B') <<  endl;
    cout <<  Max( a , b ) <<  endl;
}
```

Szablony funkcji – własna klasa

```
template <class Typ>
Typ  Max( Typ w1, Typ w2 )
{
    return  w1 < w2 ? w2 : w1;
}
```

Czy szablon można stosować również dla własnych klas?

```
int  main( )
{
    Wektor  W1(1,1), W2(4,5), W3;

    W3 = Max(W1,W2);
}
```

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};


template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};
```

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

W takiej postaci na pewno nie będzie dobrze. Dlaczego?

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};
```

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

Problemem jest operacja porównania dwóch wektorów. Dlaczego?

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
    bool operator < ( const Wektor& W ) const
            { return x*x+y*y < W.x*W.x+W.y*W.y; }
};

template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

Aby było dobrze, należy zdefiniować przeciążenie operatora porównania.

Plan prezentacji

Szablony klas

- Szablony klas pozwalają na przedstawienie ogólnych pojęć i wzajemnych ich związków.
- Pozwalają na abstrahowanie od typu poszczególnych atrybutów związanych z danym pojęciem.
- Pozwalają programiście skoncentrować na ogólnych zależnościach i mechanizmach.
- Szablony pozwalają na generowanie i optymalizowania już na etapie kompilacji poprzez użycie specyficznych konstrukcji programistycznych.
- Umożliwiają realizację idei programowania uogólnionego.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Stos

```
class Stos {
      int                 _Tab[ROZ_STOSU];
      int unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(int& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  int& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Słowo kluczowe *typename* sygnalizuje, że parametr szablonu jest typem.

Przykład szablonu stosu

```
template < class   TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<double[100]>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::string>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>  St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream*>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream&>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream&>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos< Stos< Stos< char[20]  > > >   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float, 100>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar= 100 >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float, 100>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar= 100 >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float>   St;
}
```

Plan prezentacji

Klasa wektor – plik nagłówkowy: wektor.hh

```
#ifndef WEKTOR_HH
#define WEKTOR_HH
#include <iostream>

#define ROZMIAR 3
#define TYP double

class Wektor {
        TYP _Wsp[ ROZMIAR ];
      public :
        Wektor( );
        TYP operator [ ] (int Ind) const { return _Wsp[Ind]; }
        TYP& operator [ ] (int Ind) { return _Wsp[Ind]; }
};

std::ostream & operator << (std::ostream & StrmWy, const Wektor & Wek);
std::istream & operator >> (std::istream & StrmWe, Wektor & Wek);
#endif
```

Klasa wektor – plik modułu: wektor.cpp

```
#include "wektor.hh"
using namespace std;

Wektor::Wektor( )
{
   for (int Ind = 0; Ind < ROZMIAR; ++Ind) _Tab[Ind] = 0;
}

ostream & operator << (ostream & StrmWy, const Wektor & Wek)
{
      ...
}

istream & operator >> (istream & StrmWe, Wektor & Wek)
{
      ...
}
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
#ifndef WEKTOR_HH
#define WEKTOR_HH
#include <iostream>


template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};



#endif
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
...
template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};

template <typename Typ, int Rozmiar>
std::ostream & operator << (std::ostream & StrmWy, Wektor<Typ,Rozmiar>& Wek)
{
   ...
}

template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
   ...
}
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
...
template <typename Typ, int Rozmiar>
class Wektor {
        ...
        public :
           Wektor();
           ...
};

template <typename Typ, int Rozmiar>
Wektor<Typ,Rozmiar>::Wektor()
{
    for (int Ind = 0; Ind < Rozmiar; ++Ind) _Wsp[Ind] = 0;
}

 ...
template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
    ...
}
```

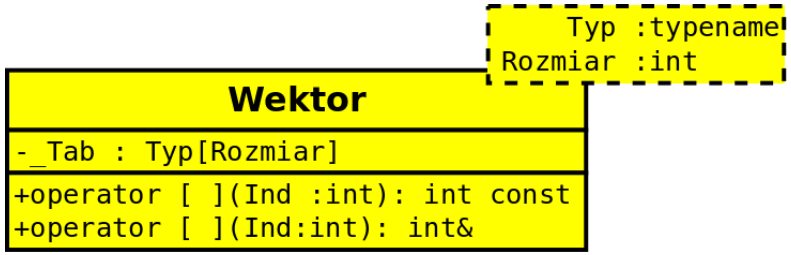
Klasa wektor – plik nagłówkowy: wektor.hh

```
...
inline
bool Wczytaj_OkreslonyZnak(std::istream &StrmWe, const char Wzorzec)
{
    ...
}

template <typename Typ>
bool Wczytaj_Liczbe_OkreslonyZnak(std::istream & StrmWe, Typ & Liczba, const char Wzorzec)
{
    ...
}


template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
  ...
}
...
```

Plan prezentacji

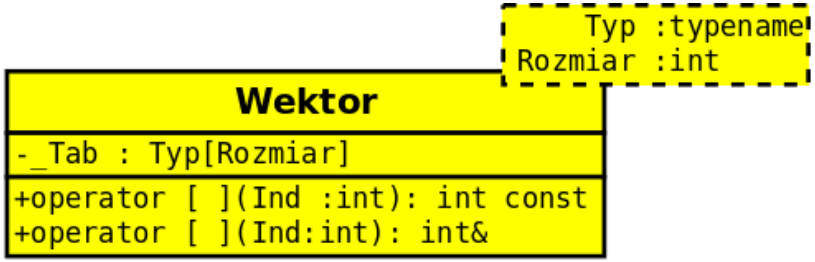
Szablon klasy Wektor w UML

```
template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};
```



Klasa będąca instancją szablonu

```
template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};
```




```
Wektor<double, 3>   W;
```

Koniec prezentacji
Dziękuję za uwagę