Wykład obiektowe

Paweł majdzik

Wskaźniki

Deklaracja zmiennych wskaźnikowych (wskaźników)

int * wsk_i; float * wsk_f; char * wsk_c; void * wsk_v;
int unsigned * wsk_iu;

Operacje na zmiennych wskaźnikowych

Operacje na wartościach zmiennych wskaźnikowych

Wskaźnik służący do pokazywania na obiekty jednego typu nie nadaje się do pokazywania na obiekty innych typów[1].

wsk_iu = wsk_i; // poprawne
wsk_f = wsk_i // **błąd!!!**
wsk_f = (float *) wsk_i; // poprawne (rzutowanie)
wsk_v = wsk_i; // poprawne
wsk_v = wsk_f; // poprawne

wsk_f = wsk_v; // **błąd!!!**

[1] **Uwaga**: wyjątek: w C++ w przypadku dziedziczenia ta reguła nie będzie miała zastosowania.

Wskaźnik każdego (niestałego) typu można przypisać wskaźnikowi typu **void**.

void < ---------- dowolny wskaźnik (nie const)

Odwrotne przypisanie nie możliwe;

dowolny wskaźnik < ---------- void // **błąd**

rzutowanie :

wsk_i = **(int *)** wsk_v; wsk_f = **(float *)** wsk_v; wsk_c = **(char*)** wsk_v;

//wszystkie powyższe instrukcje poprzez zastosowanie rzutowania //są poprawne

Odczytywanie wartości zmiennych, na które pokazują zmienne wskaźnikowe, tzn. których adresy są przechowywane przez zmienne wskaźnikowe.

```
int a=32;                 float b=4.4;

int *wsk_i=&a;            float *wsk_f=&b ;   // & - operator wyłuskiwania
                          adresu
                                                //zmiennej
// wartością zmiennej wsk_i  jest adres zmiennej a
// wartością zmiennej wsk_f  jest adres zmiennej b

cout<<*wsk_f<<endl;    // wypisze wartość zmiennej b
                       // tzn. wypisze wartość zmiennej, której adres jest
                       // wartością zmiennej wskaźnikowej wsk_f
                       // * operator wyłuskania wartości zmiennej, na którą
                       // pokazuje wskaźnik wsk_f
cout<<*wsk_i<<endl;    // wypisze wartość zmiennej a
cout<<wsk_i<<endl;     //  wypisze adres zmiennej a
cout<<wsk_f<<endl;     // wypisze adres zmiennej b

*wsk_i = 344;          // zmiana wartości zmiennej a = 344
*wsk_f = 4.5;
*wsk_i =*wsk_f          // zmiana wartości zmiennej a = 4 (trywialna
konwersja
                       // float-→ int
```

Przypomnienie: nie istnieje niejawna trywialna konwersja (np. float* -→ int*)
przy przepisywaniu wartości wskaźników (zmiennych wskaźnikowych). Wskaźnik służący do pokazywania na obiekty jednego typu nie nadaje się do pokazywania na obiekty innych typów.

wskaźnik typu void

```
void *wsk_v;
wsk_v=wsk_i;
//cout <<*wsk_v; // niepoprawne
cout <<*(float*)wsk_v; // wartość nieokreślona
cout <<*(int*)wsk_v; // wypisze 34 czyli wartość zmiennej a
//wsk_i=wsk_v;  // błąd
wsk_i=(int*)wsk_v; // poprawne

#include<iostream.h>
#include<conio.h>

int a=32;  float b=4.4;

//////////////////////
float *wsk_f=&b ;
int *wsk_i=&a;
//////////////////////
void main()
{
clrscr();
wsk_f=&b;
wsk_i=&a;
cout<<*wsk_f<<endl;                 // wypisze wartość zmiennej b
cout<<*wsk_i<<endl;                 // wypisze wartość zmiennej a
cout<<wsk_i<<endl;                  //  wypisze adres zmiennej a
cout<<wsk_f<<endl;          // wypisze adres zmiennej b
*wsk_i=*wsk_f;                      //poprawne-przepisywanie
wartości(konwersja
                                                        //trywialna
                                                    )
// wsk_i=wsk_f;                     //niepoprawne
// wsk_f=wsk_i;                   //niepoprawne
wsk_i = (int*)wsk_f;            // poprawne (rzutowanie)
//wsk_i =(int*)&b;              // równoważne instrukcji wsk_i = (int*)wsk_f;
```

```
cout<<wsk_i<<endl;          // teraz wypisze adres zmiennej b czyli wartość
                            //zmiennej wsk_f
cout<<(float*)*wsk_i<<endl; // wypisze adres nieokreślony
cout<<*(float*)wsk_i<<endl; // wypisze 4.4 czyli wartość zmiennej b
cout<<(float)*wsk_i<<endl;  // wypisze wartość nieokreśloną
cout<<*wsk_i<<endl;         // wypisze wartość nieokreśloną

// to samo w odwrotną stronę z float * --> int *
a=34;
wsk_i=&a;
wsk_f =(float*)wsk_i;
cout<<*(int*)wsk_f<<endl; // wypisze 34 czyli wartość zmiennej a
}
```

Wskaźniki stałe i do stałych

```
int const stala = 22;
int const *wsk_ic=&stala;
wsk_v = wsk_ic;          //  błąd nie można wskaźnikowi typu void * przypisać
                         //  wskaźnika do obiektu typu const
int const *wsk_ic=&stala;
int const *wsk_v = wsk_ic;   //  ok.
wsk_v = &stala ;   //  ok.

int * const wsk_ci=&wsk_i;
wsk_v = wsk_ci;   //ok
```

Zastosowania wskaźników:

- ✓ *ulepszenie pracy z tablicami*,
- ✓ *przesyłanie do funkcji argumentów przez adres* (również argumentów będących wskaźnikami do funkcji) co pozwala na

zmianę wartości przesyłanych argumentów, lub w przypadku funkcji możliwość wywoływania różnych funkcji w ciele innej funkcji,

- ✓ *dostęp do specjalnych komórek pamięci*
- ✓ *rezerwację obszarów pamięci* (zmienne dynamiczne)

Zastosowanie wskaźników w pracy z tablicami

int * wsk_i; int tab [4];

wsk_i = &tab[0] ; == wsk=tab;

przesuwanie wskaźnika: wsk_i++; wsk=wsk+n;

*wsk++ ; **(*wsk)++;**

```
for (int i =0; i<4; i++)
    *(wsk++) = i+10;                //   tab = [10,11,....,13]
```

Nazwa tablicy, a wskaźnik

wsk_i=tab;

tab[3] równoważne *(tab+3)

wsk_i[3]; *(wsk_i+3)

wsk_i++; //ok. tab++; // błąd

Nazwa tablicy jest jakby stałym wskaźnikiem do jej zerowego elementu.

Dlaczego jakby? Ponieważ *nazwa tablicy nie jest obiektem*. Stąd:

int * wsk_i; // wskaźnik to jakaś zmienna, obiekt

&wsk // to jest jego adres

&tab // błąd – nazwa nie jest obiektem. Tablica tak, nazwa nie.

wskaźnik do wskaźnika

```
int ** wsk_wi;      int * wsk_i;        int a;
        wsk_wi = &wsk_i;
```

Arytmetyka wskaźników TABLICE

Wskaźnik do składników klasy

int a;

int * wsk = &z1.re;

```
        class l_zes {
        public:
                int re;
                int im;
           public:
            //.....konstruktory
        };
wsk
l_zes z1;
int * wsk_z = &z1.re   //można bo publiczny składnik
int l_zes::*wsk_do_kl;
```

Zwykły wskaźnik pokazuje na konkretne miejsce w pamięci, określony adres.

Wskaźnik do składnika klasy nie pokazuje na żadne konkretne miejsce w pamięci, ale określa przesunięcie w pamięci od początku obiektu do składnika klasy.

```
int l_zes::*wsk_do_kl = &l_zes::re;
l_zes z1, z2;       z2.*wsk_do_kl;
z1.*wsk_do_kl;
```

l_zes * wsk=&z1;

wsk->*wsk_do_kl;

Uwaga: nie można pokazywać na składniki nie publiczne.

nie można pokazac na coś co nie posiada nazwy. Na tablice można na jej np. piąty element nie, bo nie posiada nazwy.

```
class string  {
  public:
       char *wsk;
          int roz;
  public:
           //...konstruktory
};
```

char *string:: *wsk_do_sk; string s1(“jakis lancuch”);

wsk_do_sk = &string::wsk;

*(s1.*wsk_do_sk) =’s’;

cout << *((s1.*wsk_do_sk)+2);

Wskaźnik do funkcji składowych

```
class l_zes {
        int re;
          int im;
  public:
           //...konstruktory
     void wypisz();
  int ini(int a, int b);
};
```

int (l_zes::*wsk_fun_sk) (int, int); void (l_zes::*wsk_fun_sk1) ()

wsk_fun_sk = &l_zes:: ini;

l_zes z1; (z1.*wsk_fun_sk) (2,11);

Składnik statyczny klasy

Na składnik statyczny klasy należy pokazywać zwykłym wskaźnikiem, ponieważ ten składnik nie jest elementem konkretnego obiektu.

```
class klasa {
static char st;
int cos;
public:
klasa (int ini):cos(ini){}
static char daj() {st=999;
      return st;}
int funkcja() {cout << cos; return cos;}
};
////////////////////
char klasa::st;
klasa k1(12), k2(33);
int  (klasa::*fun_zwy)()=&klasa::funkcja;
int  (klasa::*fun_zwy1)()=klasa::funkcja;
//char (klasa::*fun_sta2)()=&klasa::daj; // blad
//char (klasa::*fun_sta2)()= klasa::daj; // blad
char (*fun_sta)()=&klasa::daj; // ok
char (*fun_sta1)()=klasa::daj; // ok
//////////////
void main()
{
clrscr();
fun_zwy=&klasa::funkcja; //obydwa prawidlowe dlaczego
(k1.*fun_zwy)();
fun_zwy1=klasa::funkcja;  //
```

```
(k2.*fun_zwy1)();
getchar();
}
```

Przykład 2

```
class klasa {
static int st;
//int cos;
public:
int cos;
klasa (int ini):cos(ini){}
static void daj() {st=999; return st;}

//////////
static int st1;
//.............
};
int klasa::st1;
int klasa::st;
klasa k1(12), k2(33);
klasa *wsk1 = &k1;
void main()
{
klasa::st1=90;
clrscr();
cout<<&((klasa*)0)->cos;
cout<<wsk1->cos<<endl;
((klasa*)0)->daj();
((klasa*)&k1)->daj();
```

```
cout<<((float *)0)<<'\n';
int g;
cout<<((klasa*)g)->cos;
((klasa)g).cos=77;
cout<<endl;
cout<<((klasa*)g)->cos;  // okreslone nie wypisuje 77
cout<<((klasa*)wsk1)->cos;  // okreslone bo wsk1->k1
((klasa*)&k1)->daj();     // okreslone bo k1
cout<<((float *)0)<<'\n';
getchar();
}
```

Klasa - podstawy

- **Klasa = typ definiowany przez użyktownika**

```
class nasz_typ {
        // składniki klasy
};
```

nasz_typ* wsk;	wsk = &obiekt; wsk->skladnik;
nasz_typ obiekt;	obiekt.składnik;
nasz_typ & ref=obiekt;	ref.składnik;

- **Enkapsulacja**

- **Ukrywanie informacji**

private: public: protected:

Dlaczego nie public?

Oddzielenie implementacji od interfejsu – zmiana implementacji klasy nie wpływa na zmianę interfejsu.

Brak potrzeby ponownej kompilacji

- **Klasa, a obiekt**

Klasa to typ obiektu (typ abstrakcyjny), stąd

```
class nasz_typ {
        char nazwisko [40];
        int wiek = 21;  //błąd
};
```

- **Funkcje składowe**

```
class l_zes {
        int re;
            int im;
   public:
    void wypisz();
    void ini(int a, int b);
};
```

obiekt.funkcja(parametry aktualne);

l_zes z1; l_zes *wsk; wsk=&z1;

wsk ->wypisz(); wsk ->ini(4,5);

- **Definiowanie funkcji składowych (w klasie i poza)**

funkcja zdefiniowana w klasie = inline

poza klasą

```
void l_zes::ini(int a, int b)
            {//ciało fun       }

inline void l_zes::ini(int a, int b)
            {//ciało fun       }
```

- **Wskaźnik this**

```
class l_zes {
        int re;
            int im;
           public:
     void wypisz();
    void ini(int a, int b);
```

```
};

l_zes z1, z2, z3;
void l_zes::ini(int a, int b)
        {
            this->re=a;  this->im=b;
        }
```

typ wskaźnika this **X const *** // stały wskaźnik

- **Zasłanianie nazw**

```
char b;             //zmienna i funkcja globalne
 void fun2 ( );

class jakas {
public:
        int a;
        char b;
    void fun1 ( );
   void fun2 ( );
};

Jakas j1;

  void jakas::fun1 ()
    {
     cout << b;                    //składnik klasy;
     cout << ::b;                  //zmienna globalna
     int b;
     cout << b;                    // zmienna lokalna
     cout << jakas::b;             // składnik klasy;
     cout << ::b;                  //zmienna globalna
     int fun2;
     fun2();                       //błąd
     jakas::fun2();
    }
```

- **Przesyłanie do funkcji argumentów będących obiektami**

```
class osoba  {
            char nazwisko [40];
            int wiek;
        public:
            void ini(char * st, int lata);
            void wypisz() {
                cout<< nazwisko<< "   , lat : " << wiek <<  endl;
              }
    };

void fun(osoba ktos); //przez wartość
void fun(osoba & ktos); //przez referencje
```

- Składnik statyczny, statyczna funkcja składowa

```
    class l_zes {
            int re;
               int im;
          static int cos;
      public:
          static int ile;
        static fun ( ) {cout << ile; cout << im; //błąd
          void wypisz();
        void ini(int a, int b);
    };

int l_zes::cos = 29;
int l_zes::ile; // 0

l_zes::ile;   z1.ile;              l_zes *wsk; wsk=&z1; wsk -> ile;

l_zes::fun();       z1.fun();           l_zes *wsk; wsk=&z1; wsk -> fun();
```

Wewnątrz statycznej funkcji:

- ✓ nie jest możliwe jawne odwołanie się do wskaźnika **this**, (nie zawiera
  składnika this)
- ✓ nie jest możliwe odwołanie się do składnika zwykłego,
- ✓ jest możliwe wywołanie funkcji nawet, gdy nie ma żadnego wskaźnika.

Ale istnieje jawna możliwość odwołania się do składnika klasy np. ob1.re;

Funkcje składowe typu const

```
class wspolrzedne {
    int  x , y;
public :
    wspolrzedne (int a, int b ) {x = a ; y = b ; }
    void wypis (void) const ;                    //
    void przesun(int a, int b);
} ;

void wspolrzedne::wypis() const                  //
{
    cout << x << ", " << y << endl ;
//     x=22; blad
}

void wspolrzedne::przesun(int a , int b)
{
    x = a;    y = b ;                    //
}
/*******************************************************/
main()
{
clrscr();
    wspolrzedne w1(1, 1),w2(2, 2) ;
    const wspolrzedne w3(5, 5) ;
    w1.wypis() ;
    w2.wypis() ;
    w3.wypis() ;

    w1.przesun(4,10) ;
    w2.przesun(50, 50) ;
    w3.przesun(3,3) ;
```

```
    w3.wypis() ; }
```

Funkcje zaprzyjaźnione

```
class kwadrat ;

class punkt {
     int  x, y ;
     char nazwa[30] ;
public :
     punkt(int a, int b) ;
    firend ostream & operator << (ostream & st, const punkt &p);
     friend int spr(punkt & p, kwadrat & k) ;   //
} ;
----------------
class kwadrat  {
     int  x, y,
          bok ;
     char nazwa[30] ;
public :
     kwadrat(int a, int b, int dd) ;
     friend int punkt::spr (kwadrat & k) ;
} ;
-----------
punkt::punkt(int a, int b,char *opis) {
//detale}
--------
kwadrat::kwadrat(int a, int b, int dd)
{ //detale
;}
------------
int spr (punkt & pt, kwadrat & kw)                    //

{
     if( (pt.x >= kw.x)  &&  (pt.x <= (kw.x + kw.bok) )
          &&
           (pt.y >= kw.y)  &&  (pt.y <= (kw.y + kw.bok) ))
{
//pozostale detale}
}
/*****************************************************/
main()
{
     kwadrat k1(10,10, 40) ;
```

```
    punkt     p1( 20, 20);
    spr(p1, k1 );                              //
}

class punkt ;


class kwadrat  {
    int  x, y, bok ;
public :
    kwadrat(int a, int b, int dd) ;
    int spr (punkt & p);                       //
} ;
----------
class punkt {
    int  x, y ;
public :
    punkt(int a, int b) ;
    friend int kwadrat::spr(punkt & p) ;            //
} ;
------
punkt::punkt(int a, int b)     // konstruktor
{
}
/*******************************************************/
kwadrat::kwadrat(int a, int b, int dd)
{  // detale
}
/*******************************************************/
int kwadrat::spr (punkt & pt)                    //

{
    if( (pt.x >= x)  && (pt.x <= (x + this->bok) )       //
        &&
         (pt.y >= y)  && (pt.y <= (y + bok) ))
  {
//detale }
}
/*******************************************************/
main()
{
    kwadrat     k1(2,3,4) ;
    punkt     p1( 20, 20);
```

```
    k1.spr(p1);
}
```

Zagnieżdżona definicja klasy

```
class A
{

.....

   class B
     {
        // detale
        void funkcja ();
     };
.....
};
```

*void A::*B::funkcja();

Obiekty klasy B można tworzyć tylko w obrębie klasy A
Składniki prywatne obu klas są niedostępne.

Konstruktory
Standardowy, Domniemany, Kopiujący

Argumenty domniemane , a przeładowanie

Kiedy konstruktor kopiujący:

- ✓ istnieją składniki powiązane z klasą jednak nie należące bezpośrednio do klasy(zmienne dynamiczne)
- ✓ chcemy znać liczbę obiektów danej klasy

```
class string {
        int roz;
        char *wsk;
    public:
        string(char tab[]);
        string(int i =255);
        string(const string & s);
string () {delete [] wsk; cout << " to ja destructor "}
};
string::string(char tab[]): roz(strlen(tab)), wsk(new char [roz+1])
            { // roz = strlen(tab);
              strcpy(wsk,s.wsk);
             }

string::string(const string & s): roz(s.roz), wsk(new char [roz+1])
            { // roz = strlen(tab);
              strcpy(wsk,s.wsk);
             }

class B  {
              B( …  );
              };
        class A   {
              int  cos;
              B   obiekt_skl;
              public:
              A(B &, int);
};
```

A a1;

A::A (**B & n**, int i) **:** cos(i) { obiekt_skl = n; }

1. A::A(**B & n**, int i) { **obiekt_skl=n**; cos = i;} - powolny
2. A::A (**B & n**, int i) **:** **obiekt_skl(n)**, cos(i) {} - szybki konstruktor

Przykład 1
A a1;

Gdy jest tworzony obiekt klasy A, element **obiekt_skł** najpierw zainicjalizowany jest przy użyciu konstruktora standardowego klasy B, a następnie jego wartość zmieni w ciele konstruktora klasy A::A(**B & n**, int i) poprzez przypisanie. W przypadku konstruktora 2 jest wykonywana jedna operacja – inicjalizacja poprzez wywołanie konstruktora B::B(n) - (zysk czasu około 25%).
Ponieważ typy wbudowane nie mają konstruktorów nie ma znaczenia czy element **cos** klasy A zostanie zainicjalizowany na liście czy też poprzez przypisanie.

```
class string {
char *wsk;
    public:
string(char [] = "")  ;
string(int a);
string(const string & );

string & operator = (char t[] ){
{  if (roz != strlen (t))
```

```
                    {roz= strlen(t);
                     delete [] wsk;
                     wsk = new char [roz+1];
                     }
strcpy(wsk, t);
}


string & operator = (const string & s);
{  if (roz != s.roz) {roz= s.roz;
                     delete [] wsk;
                     wsk = new char [roz+1];
                     }
strcpy(wsk, s.wsk);
return *this;
}
};
string s1 (“ala”);
s1=s1;

//////////////////////
class osoba {
string nazwisko;
int wiek;
    public:
osoba (int a, char t[]);
osoba (int a, const string&);

//.....
```

```
};
//////////////////////
osoba:: osoba (int a, char t[]): wiek(a), nazwisko(t) {}
osoba:: osoba (int a, const  string & s): wiek(a), nazwisko(s) {}
//osoba:: osoba (int a, const  string & s): wiek(a)
//{nazwisko= s; }
main()
{int a=22;
string s("ABCD");
osoba os1(a,s);}
////////////////////////////////
string::string(char t[]) {
}
string::string(int a)
{
}
string::string(const string & s)
{
}
string&string:: operator=(string & s)
{
if (&s !=this) {
delete [] wsk;
wsk= new char [strlen(s.wsk)+1];
strcpy(wsk,s.wsk);
}
```

Kolejność inicjalizownia składowych

Składnik const tylko na liscie
Składnik odniesienie (ref) tylko na liscie

Skladnik statyczny nigdzie

Konstruktor syntezowany

Przykład 1

```
#include<iostream.h>
#include<conio.h>

class klasa1 {
  int k;
   //... detale
public:
   klasa1(){k=33;}            //  brak tego konstruktora błąd – domyślny lub
brak
                              //  konstruktorów
   klasa1(int i) {k=i;}
};

class klasa2 {
klasa1 obiekt1;
char * wsk;
//... detale
public:
klasa2() { wsk=0;}
};


void main()
{

clrscr();
klasa1 k1;
klasa2 k2;
}
```

Przykład 2

```
#include<iostream.h>
#include<conio.h>

class klasa1 {
  int k;
   //... detale
public:
   klasa1(){k=33;}
   klasa1(int i) {k=i;}
```

```
};
class klasa2 {
klasa1 obiekt1;
int cos;
char * wsk;
//... detale
public:
klasa2(int a ): obiekt1(a) { cos=a;wsk=0;}
// klasa2(int a ) { cos=a;wsk=0;} – także poprawnie

};


void main()
{  klasa1 k1;
   klasa2 k2(3);
```

Uchwyt

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class string{
private:
    class string_rep{
    public:
      int licznik;
      char * wsk;
      string_rep(const char *);
      string_rep(const string_rep&);
      ~string_rep();
      void zwieksz();
      void zmniejsz();
      const string_rep& operator =(const string_rep&);
      int daj_licznik() {return licznik;}
    };

   string_rep * rep;
public:
   string(const char* = "");
```

```
   ~string();
   string(const string&);
   const string& operator=(const string&);
   int daj() {return rep->daj_licznik();}

};

string s1("ala");
string s2("ala ma kota");
string s3=s2;
string s4("abc");
main()
{
clrscr();
cout << "obiekt 1 = " << s1.daj()<< endl;
cout << "obiekt 2 = " << s2.daj()<< endl;
cout << "obiekt 3 = " << s3.daj()<< endl;
cout << "obiekt 4 = " << s4.daj()<< endl;
s4=s3;
cout << endl<< endl<< endl<< endl;

cout << "obiekt 1 = " << s1.daj()<< endl;
cout << "obiekt 2 = " << s2.daj()<< endl;
cout << "obiekt 3 = " << s3.daj()<< endl;
cout << "obiekt 4 = " << s4.daj()<< endl;

}


//////////////Implementacja string_rep///////////////
string::string_rep::string_rep(const char * t):licznik(0),
                          wsk(new char[strlen(t)+1])
{
strcpy(wsk,t);
}
string::string_rep::~string_rep()
{
delete [] wsk;
}
string::string_rep::string_rep(const string_rep& w): licznik(0),
                                  wsk(new char[strlen(w.wsk)+1])
```

```
{
strcpy(wsk,w.wsk);
}

void string::string_rep:: zwieksz(){
        ++licznik;
}

void string::string_rep:: zmniejsz(){
        if (--licznik==0)
        delete this;
}

const string_rep& string::string_rep::operator =(const string_rep& w)
{
if (this !=&wsk){
     delete [] wsk;
     wsk= new char[strlen(w.wsk)+1];
     strcpy(wsk,w.wsk);
     }
     return *this;
}

//////////////Implementacja string///////////////

string::string(const char* w): rep (new string_rep(w))
{
rep->zwieksz();
}

string:: ~string(){
rep->zmniejsz();
}

string::string(const string& s):rep(s.rep)
{
rep->zwieksz();
}

const string& string::operator=(const string& s){
if (rep !=s.rep){
```

```
     rep->zmniejsz();
     rep=s.rep;
     rep->zwieksz();
      }
 return *this;
}
```

Optymalizacja wartości zwracanej

```
class l_zes {
int re, im ;

public:
  friend void dodaj (l_zes &a , l_zes &b , l_zes &rezult)
  friend l_zes operator+ (l_zes &a , l_zes &b);
  l_zes(int i=0, int j=0):re(i), im (j) {cout << "kon-zwykly"<<'\n';}
  l_zes(const l_zes & z): re(z.re), im (z.im) {cout <<"kon-kopia"<<'\n';}
  l_zes & operator = (const l_zes & z){cout << "="<< '\n';}
 ~l_zes() {cout << "destruktor"<<'\n';}
};


void main()
{
clrscr();
          l_zes z1, z2, z3;
          z3=z1+z2;
}
//////////////////////////
l_zes  operator+ (l_zes &a , l_zes&b)
{   l_zes lok;
            lok.re = a.re + b.re;
            lok.im = a.im + b.im;
    return lok;
}

l_zes  operator+ (l_zes &a , l_zes&b)
{   return l_zes (a.re + b.re, a.im + b.im);
}
```

z3=z1+z2;

Pseudokod C++

```
void l_zes_Add (const l_zes & _tempResult, const l_zes & a, const l_zes
& b)
{
        struct l_zes lok;
        lok.l_zes::l_zes();  // wygenertowane przez kompilator wywołanie
                                  //konstruktora domyślnego
           lok.re = a.re + b.re;
           lok.im = a.im + b.im;
        _tempResult.l_zes::l_zes(lok)   // wygenertowane przez kompilator
                                          //wywołanie  konstruktora kopjującego
           lok.l_zes:: ~l_zes();      // likwidacja obiektu lok
            return;
//

}
```

Pseudokod C++ - po zastosowaniu optymalizacji OWZ (NRV)(RVO)

```
void l_zes_Add (const l_zes & _tempResult, const l_zes & z2, const l_zes
& z3)
{
        _tempResult.l_zes::l_zes();  // wygenertowane przez kompilator
wywołanie
                                          //konstruktora domyślnego
            _tempResult.re = a.re + b.re;
           _tempResult.im = a.im + b.im;
            return;
}
```

Optymalizacja może nie być realizowana np.

- ✓ funkcja posiada wiele instrukcji return zwracających obiekty o różnych nazwach
- ✓ gdy powrót z funkcji nie jest na najwyższym poziomie zagnieżdżenia

Zastosowanie OWZ zależy od konkretnej implementacji kompliatora. Należy zapoznać się dokumnatacją kompilatora i wykonać próby, aby móc stwierdzić, czy i kiedy OWZ jest stosowana.

Alternatwą jest zastosowanie konstruktora obliczeniowego.

Konstruktory obliczeniowe

```
class l_zes {
          int re, im ;
l_zes(const l_zes & a, const l_zes & b ):
       re(a.re + b.re), im (a.im + b.im) {}

l_zes(const l_zes & a, const l_zes & b, char c):
       re(a.re * b.re), im (a.im * b.im) {}

l_zes(const l_zes & a, const l_zes & b, int c):
       re(a.re - b.re), im (a.im - b.im) {}
public:
  friend l_zes operator* (l_zes &a , l_zes&b)
{return l_zes(a,b,'f')};
  l_zes(int i=0, int j=0):re(i), im (j) {}
  l_zes(const l_zes & z): re(z.re), im (z.im) {}
  l_zes & operator = (const l_zes & z){ }
 ~l_zes() {}
};


void main()
{
clrscr();
l_zes z1, z2, z3;
for(long i=0;i<1000000; i++) z3=z1+z2;
}

l_zes operator+ (l_zes &a , l_zes&b)
{
  return l_zes(a,b);
}

l_zes operator- (l_zes &a , l_zes&b)
{
  return l_zes(a,b,0);
}
```

Z3 = z1 –z2;
lub zamiast konstruktora obliczeniowego

```
l_zes operator+ (l_zes &a , l_zes&b)
{
        return l_zes (a.re + b.re, a.im + b.im);
         // tak samo szybko jak obliczeniowy
}
```

o 10 % wolniejszy od konstruktora oblicznniowego

Obiekty tymczasowe

Niezgodnośc typów

Tworzenie obiektów tymczasowych przy niezgodności typów.

```
l_zes z1;
z1=200;

Pseudokod C++
        l_zes _temp;
        _temp.l_zes:: l_zes(200, 0);
        z1.l_zes::operator = (_temp);
        temp::l_zes:: ~l_zes();

class l_zes {
            int re, im ;

public:
  friend l_zes operator+ (l_zes &a , l_zes&b);
explicit    l_zes(int i=0, int j=0):re(i), im (j) {}  // zapobiega realizacji
konwersji
                                                      //niejawnych przy użyciu
tego
                                                      //konstruktora
l_zes(const l_zes & z): re(z.re), im (z.im) {}
  l_zes & operator = (const l_zes & z){ }
 ~l_zes() {}
};
```

Można wyeliminować obiekt całkowity przeciążając funkcje

```
l_zes :: operator =(int a) {re = a; im = 1; return *this; }
```

Tworzenie obiektów tymczasowych przy reprezantacji stałych

```
.....
l_zes z1, z2;
int temp = 5;
for (int i; i<100; i++)
 z1 = i*z2 + temp;



//wyganerowanie obiektu tymczasowego, aby przedstawić
                                //wartość 5
  Tymczasowy obiekt jest generowany w każdej iteracji pętli
l_zes z1, z2;
l_zes tym(5);

for (int i; i<100; i++)
        z1 = i*z2 + tym;   // 30% zysku
```

Przekazywanie przez wartość

Jeśli to możliwe przekazuj przez referencje - np.:

```
...fun (const klasa & ref,  ....)
```

Powrót przez wartość

```
class string{
      char * wsk;
public:
      string(const char * = "");
      string(const string&);
      ~string();
      string& operator =(const string&);
      friend string operator +(const string& a, const string& b);
    };

string operator +(const string& a, const string& b)
{
```

```
        cout << "operator +"<<endl;
        char *bufor = new char[strlen(a.wsk)+strlen(b.wsk)+1];
         strcpy(bufor,a.wsk);
        strcat(bufor,b.wsk);
        string lok(bufor);
   return lok;
}


main()
{
clrscr();
string s1("ala");
string s2(" ma kota");
string s3;
s3 = s1+s2;
}
```

Wywoływane funkcje i tworzone obiekty
operator + - dla s1 i s2
string:: string (const char *) - dla bufora
string:: string (const string &) - do utworzenia obiektu tymczasowego do
przechowywania wartości zwracanej przez operator +
string:: ~string () - likwidowanie obiektu lok
string:: operator =(const string &) - przypisanie obeiktu tymczasowego do s3
string:: ~string () - likwidowanie obiektu tymczasowego
Można zastosować OWZ weliminuje obiekt lok, czyli zmienjszy się o wywołanie 1 konstruktora i 1 destruktora
Dlaczego obiekt tymczasowy dla

```
s3 = s1+s2;
```

Musimy starą wartość s3 zastąpić nową wartościa, stąd kompilator nie może pominąc funkcji operator =() i dlatego obiekt tymczasowy jest konieczny.

Eliminacja obiektu tymczasowego:

```
string s4("ala");
```

```
string s5(" ma kota");
string s6 = s4+s5;
```

Eliminowanie obiektów tymczasowych przez op=()

```
class string{
      char * wsk;
public:
      string(const char * = "");
      string(const string&);
      ~string();
      string& operator =(const string&);
      friend string operator +(const string& a, const string& b);
      string& operator +=(const string& a);
      friend ostream& operator <<(ostream&, string &);
    };
//////////////Implementacja string///////////////
string& string::operator +=(const string& a)
{
char * bufor = new char[strlen(wsk)+strlen(a.wsk)+1];
strcpy(bufor,wsk);
strcat(bufor,a.wsk);
wsk = bufor;
return *this;}
///////////////
string operator +(const string& a, const string& b)
{
char *bufor = new char[strlen(a.wsk)+strlen(b.wsk)+1];
strcpy(bufor,a.wsk);
strcat(bufor,b.wsk);
string lok(bufor);
return lok;
}

main()
{
clrscr();
string s1("ala");
string s2(" ma kota");
string s3;
```

```
for(long i=0;i<10000000; i++) {
s3=s1+s2;
for(long i=0;i<10000000; i++) {
s3=s1;
s3+=s2;        // brak obiektu tymczasowego zysk – ponad 50%
}
```

Eliminowanie funkcji zaprzyjaźnionych

```
string operator +(const string& a, const string& b)                    // teraz nie musi być

//zaprzyjaźniona
{
string lok =a;
lok += b;
return lok;
}
```

Podsumowanie

- ✓ Obiekt tymczasowy może podwójnie naruszyć wydajność poprzez przetwarzanie konstruktora i destruktora
- ✓ Zadeklarowanie konstruktora **explicit** zapobiegnie wykorzystaniu przez kompilator konwersji typów
- ✓ Obiekty tymczasowe często są tworzone przez kompilator w celu naprawienia niezgodności typów. Można tego uniknąć poprzez przeciążenie funkcji
- ✓ Jeśli można, należy unikać kopoiwania obiektów oraz przekazywać i zwracać obiekty poprzez odwołanie
- ✓ Obiekty tymczasowe można wyeliminować korzystając z operatoróe op=(), gdzie op jest jednym z operatoróe : +, - , *, /

Funkcje operatorowe

operator =
operator + , - ,   *, /

```
operator += ,  -= , =*,  /
operator ++p , p++
operator <<
operator >>

z3 = operator + (z1,33);
z3 = z1. operator + (z2);

z3 = z1+ z2;
z3 = z1+23;
z3 = a + z2;

class l_zes {
int re, im ;
public:
friend l_zes operator+ (int b, const l_zes &a);
friend l_zes operator+ (l_zes &a , l_zes&b);
friend l_zes operator+ (l_zes &a , int b);

l_zes  operator ++();
l_zes  operator ++(int);
l_zes(int i=0, int j=0):re(i), im (j) {}
l_zes(const l_zes & z): re(z.re), im (z.im) {}
  l_zes & operator = (const l_zes & z){ }
 ~l_zes() {}
  friend ostream& operator<< (ostream&, l_zes&);
  friend istream& operator >> (istream& kl, l_zes& z);

};

//////////////////////////

ostream & operator << (ostream& ek, l_zes& z)
{

ek  << ”re = ” << z.re << ” im =  ” << z.im;
return ek;
}
//////////////////////

l_zes l_zes:: operator+ ( l_zes&b)
{
return l_zes(re+b.re, im+ b.im);
```

```
}
////////////////////////////////////
l_zes operator+ (l_zes &a , l_zes&b)
{
//l_zes lok;
//lok.im = a.im + b.im;
//lok.re = a.re + b.re;
// return lok;
return l_zes(a.re+b.re, a.im+ b.im);
}
```

```
l zes &  l_zes:: operator ++()
  {
re++; im ++;
return *this;
  }

l_zes z1(1,1), z2(2,2);

z2 = ++z1;

z2 = z1++;


  l_zes_b l_zes::  operator ++(int)
  {  l_zes lok = *this;
     re++;
     im++;
   return lok;
  }
////////////////////////////
void main()
{
l_zes z1, z2, z3;
z1++;
cout << z1;
++z2;
cout << z2;
```

```
cout << z1++;
cout << z1;
cout << ++z1;
cout << z1;
z1+=z2;
l_zes & l_zes::operator += (const l_zes & z)
{
re += z.re;
im = im +z.im;
return * this;
}
```

Tablice obiektów

```
class osoba {
        public:
         char nazwisko[30];
         char imie [15];
           int wiek;
  };
```

osoba tab [20]; osoba *wsk; wsk=&tab[0];

tab[3].wiek wsk -> wiek

tablica obiektów dynamicznych

```
                          cin >> r;
osoba  *wsk;              wsk = new osoba[r];
                        int * wsk = new int[r];
        *(wsk+4)     //notacja wskaźnikowa
           wsk[4]    //notacja tablicowa
(*(wsk+8).wiek;   // obiekt.składnik
(wsk+8)->wiek;   // wskaźnik->składnik
wsk[8].wiek       // wwkaźnik[8].składnik
```

Lepszy wskaźnik osoba * const wsk_s;

wsk = new osoba[15];

wsk_s[1].wiek=22;

(*(wsk_s+1)).wiek=22; // poprawne 3 odwołani

(wsk_s+1)->wiek=22;

*(wsk_s++).wiek=33; // błąd

delete [] wsk_s; delete [] wsk;

Inicjalizacja tablic obiektów

Inicjalizacja tablic obiektów – **agregatów**

Agregat – to klasa, która:

nie ma składników typu private lub protected
nie ma konstruktorów
nie ma klas podstawowych
nie ma funkcji wirtualnych

osoba s1 = {”kowalski”, ”jan”, 33} // inicjalizacja

osoba s1;
strcpy(s1.nazwisko,”kowalski”)
..... // przypisanie

s1.wiek = 33}

niech w klasie osoba dodano składnik int const wzrost;

konieczna jest inicjalizacja czyli nadania wartości w momencie tworzenia obiektu

osoba s1 = {”kowalski”, ”jan”, 33, 1.84}

osoba s1[2] = {”kowalski”, ”jan”, 33, ”nowak”, ”jan”, 22};

Inicjalizacja tablic, które nie są **agregatami**

```
class osoba {
        char nazwisko[30];
        char imie [15];
         int wiek;
   public:
        osoba(char* n, char* i, int w);
        osoba();
          };
```

Jeżeli klasa nie jest agregatem, to należy posłużyć się konstruktorem, ponieważ składnik **prywatny** nie jest dostępny poza klasą, a lista inicjalizatorów nie leży w zakresie ważności klasy.

```
osoba s = osoba(”kowalski”, ”jan”, 33);
osoba tab[14] = {osoba(”kowalski”, ”jan”, 33),
                 osoba(),
                 osoba(),
                 osoba(”nowak”, ”tomasz”, 13)   };
```

konstruktor jest wywoływany jawnie, to oznacza, że:

- ✓ tworzony jest obiekt chwilowy-służy do inicjalizacji elementu tablicy
- ✓ inicjalizacja odbywa się za pomocą konstruktora kopiującego

```
class osoba {
        char nazwisko[30];
        char imie [15];
         int wiek;
```

```
  public:
      osoba(char* n, char* i, int w)
       osoba ()            // konstruktor konieczny dla poniższego przykładu
(kiedy
                          //  nie  inicjalizujemy  wszytkich  elementów
                      konstruktorem
                          //osoba(char* n, char* i, int w)  lub w
                        // klasie nie może być żadnego konstruktora
        };

osoba tab[6] = {  osoba(”kowalski”, ”jan”, 11),
                      osoba(”kowal”, ”paweł”, 87),
                      osoba(”nowak”, ”jan”, 31)   };
```

elementy tab[4] i tab [5] są inicjalizowane konstruktorem domniemanym

```
class osoba {
      char *wsk_n   // tablica dynamiczna;
      char *wsk_i;   //// tablica dynamiczna;
       int wiek;
  public:
      osoba(char* n, char* i, int w)
       osoba ()
       osoba(osoba & k)  // konstruktor kopiujący konieczny
        };

osoba tab[6] = {  osoba(”kowalski”, ”jan”, 11),
                      osoba(”kowal”, ”paweł”, 87),
                      osoba(”nowak”, ”jan”, 31)   };
```

konstruktor jest wywoływany jawnie, to oznacza, że:

- ✓ tworzony jest obiekt chwilowy-służy do inicjalizacji elementu tablicy
- ✓ inicjalizacja odbywa się za pomocą konstruktora kopiującego, czyli należy zdefiniować konstruktor kopiujący.

Inicjalizacja tablic tworzonych **dynamicznie**

Nie można tablic tworzonych dynamicznie inicjalizować jawnie (w oparciu o listę)

```
class osoba {
char nazwisko[30];
char imie [15];
 int wiek;
public:
osoba(char* n, char* i, int w);
osoba();
 };
```

```
Cin >> ROZ
osoba * wsk;              wsk = new osoba [ROZ] =")};
```

aby utworzyć tablicę dynamiczną w klasie musi być:

- ✓ albo zdefiniowny konstruktor domniemany,
- ✓ albo nie może być zdefiniowany żaden konstruktor.

Przeładowanie nazw funkcji

Prawidłowe

```
void fun(int);
void fun(unsigned int);

void fun(float);
void fun(char *);
void fun(int, char = '0');
void fun(int );          //błąd
```

Dodaje do nazwy nazwy argumentów np. fun_Ff, fun_Fic

Linkowanie z modułami napisanymi w C lub innych jezykach

deklaracja w module C++ **extern void funkcja (int, int)**; zostanie rozszerzona do **funkcja_Fii(int, float)**;

prawidlowa deklaracja w module C++

extern ”C” void funkcja (int, int)

Identyczność typów

typedef int cal;

void fun(int);
void fun(cal); // **błąd**
//

void fun(int * wsk);
void fun(int tab[]); //**błąd**

int t[6];
fun (t)

//////////////////////////////////

void fun(int t1[2][3]);
void fun(int t2[2][4]); //**OK**

//////////////////////////////////

void fun(int t1[5][9]);
void fun(int t2[2][9]); //**błąd**

void fun(int t1[5][9][10]);
void fun(int t2[2][9][10]); //**błąd**

void fun(int t1[5]);
void fun(int t2[2]); //**błąd**

Do obliczania pozycji elementu tablicy w pamięci pierwszy wymiar nie jest potrzebny, stąd jeśli deklaracja tablicy jako parametru formalnego funkcji różni się tylko jednym wymiarem to przeładowanie funkcji jest nieprawidłowe.

//////////////////////////
void fun(int &a);
void fun(int k); //**błąd**

typy **T** i **T&** mają te same inicjalizatory wywołania funkcji

//////////////////////////

```
void fun(int a);
void fun(const int k);  //błąd
```

typy **T** i **const T** mają te same inicjalizatory wywołania funkcji

//

```
void fun(int *wsk);
void fun(const int *wsk_s);  //OK
int a;          const int b;
```

typy **T *** i **const T*** wymagają odmienny inicjalizatorów wywołania funkcji

Uwaga: można wywołać void fun(const int *wsk_s) z parametrem typu int
(int a) , fun(&a), ale dokonywana jest konwersja **const int * → *int**
(tak jak np. float→int) .
Natomiast nie jest możliwe wywołanie funkcji void fun(int *wsk) z
parametrem typu const int (const int stala), fun(&stała).

//

```
void fun(int &wsk);
void fun(const int &wsk_s);  //OK
```

jak wyżej

Konwersje

Przykład

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
```

```
class l_zespol {
  public:
     float re, im;
l_zespol (float r = 0 , float i = 0) : re(r), im (i) {}
};
////////////////////////////////
l_zespol operator  (l_zespol a, l_zespol b)
{
     l_zespol l (0,0);
     l.re = a.re + b.re;
     l.im = a.im + b.im;
     return l;
}
////////////////////////////////
l_zespol dodaj (l_zespol a, float b)
{
     l_zespol l (0,0);
     l.re = a.re + b;
     l.im = a.im;
     return l;
}
////////////////////////////////
l_zespol operator+ (float b, l_zespol a)
{
     l_zespol l (0,0);
     l.re = a.re + b;
     l.im = a.im;
     return l;
}
int main(void)
{
   l_zespol z1(1,1), z2(5,5), z3(0,0);
   z3 = dodaj (z1,z2);
   z3 = dodaj (z1,6);
return 0;
}

z3 = z1+z2;
z3 = z1+2.2;   z3 = z1 + l_zespol(2.2,0);
z3 = 11.5+z2;
```

Konwersje obiektu typu A na typ Z mogą być zdefiniowane przez użytkownika

Służą do tego:

- albo konstruktor klasy Z przyjmujący jako jedyny argument obiekt typu A
- albo specjalna funkcja składowa klasy A zwana funkcją konwertującą (inaczej - operatorem konwersji)

konwersje są przeprowadzane w sposób niejawny

1. Konstruktor jako konwenter

Konstruktor, przyjmujący jeden argument, określa konwersję do typu tego argumentu do typu klasy, do której sam należy.

```
class l_zespol {
  public:

     float re, im;
     l_zespol (float r = 0, float i = 0) : re(r), im (i) {}
     //l_zespol ();
};
```

////////////////////////////////////// określamy konwersje typu **float** na typ **l_zespol**

```
 l_zespol :: l_zespol (float r)
 {
        re = r;
 }

 //// lub l_zespol (float r, float i = 0);


int main(void)
{
   l_zespol z1(1,1), z2(5,5), z3(0,0);
   z3 = dodaj (z1,z2);
   z3 = dodaj (z1,6);                    //  z3 = dodaj(z1,
l_zespol(6));
   l_zespol z4 = l_zespol(3.2)           // jawne wywołanie
konstruktora
```

```
return 0;
 }
```

Funkcja konwertująca - operator konwersji

```
void fun (float x);

l_zespol z;
      fun(z);  // konstruktora nie można zastosować, ponieważ nie można
napisać                  // konstruktora w klasie float
```

Funkcja konwertująca jest funkcją składową klasy K, która się nazywa

K :: operator **T**()

gdzie **T** jest nazwą typu, na który chcemy dokonać konwersji.
(**T** może być nawet nazwą typu wbudowanego**)**
Czyli funkcja konwertująca dokonuje konwersji z typu **K** na typ **T**

```
class l_zespol {
    float re, im;
public:
    //..
     operator int()
        {
        return (int) re;                  // return re
        }
};
```

od tej pory możemy obiekt klasy l_zes wysłac do wszystkich funkcji, które jako argument spodziewają się typu int.

```
void fun1(int i);
l_zespol z(5.2, 4.6);
fun1(z);
```

Wywołanie jawne funkcji konwertującej

```
int i;
i = ( int ) z;
i =  int ( z );
```

- funkcja konwertująca musi być **funkcją składową klasy.**
- funkcja konwertująca nie ma **określenia typu rezultatu zwracanego**. Funkcja ta zwraca taki typ, jak się sama nazywa.
- funkcja ta ma **pustą liczbę argumentów**. Nie ma przeładowania.
- funkcja konwertująca **jest dziedziczona**. Czyli klasy pochodne mają ją automatycznie - chyba, że ją zasłonią definiując swoją własną wersję.
- funkcja konwertująca może być **funkją wirtualną**.

Przykład

Definiujemy funkcję konwertującą z typu **l_zespol** na typ **numer**, czyli w odwrotną niż w poprzednią stronę.

```
operator numer();

l_zespol :: operator numer()
{
   numer n (re, ”powstal z zespolonej”);
   return n;
}
```

Funkcja operatorowa

```
class jakas {
```

```
    //.............
    public:
       //.......
     jakas (int, int =0);
    operator double( );
};

double pierwiastek (double);

main( ) {

jakas r(4,1);

 double sq = pierwiastek ( r ); // przekształcenie niejawne z jakas na
double
```

Równoważne jawne

```
    double tym = r.operator double( );
    double sq = pierwiastek ( tym );
```

Konstruktor o jednym argumencie

```
class string {
//.............
public:
       string(const char*  = "");

};

void drukuj (const string&);


drukuj("jakis napis");  // niejawna konwersja z char* na string


class liczba {

//.........
public:
       liczba(long licznik = 0, long mianownik =1);

};
```

```
int operator == (const liczba &, const liczba &);

int bez_zera(const liczba &)  {
        return r==0;  // 0 jest niejawnie przekształcone na liczba (drugi
                      // argument  konstruktora liczba to standardowo 1)
}
```

Wersja równoważna (jawna)

```
int bez_zera(const liczba &)  {
        return r==liczba(0,1);
}
```

Lepsza wersja (optymalizacja kodu)

```
int bez_zera(const liczba &)  {
        static const liczba zero(0,1) // definicja zmiennej zero typu liczba
        return r== zero;
}
```

Ta wersja funkcji pozwala uniknąć kosztu tworzenia i usuwania nowego obiektu liczba przy każdym wywołaniu funkcji bez_zera.

Klasy zawierające więcej niż jeden operator konwersji

```
class string {
        char * wsk;
public:
        string(const char*  = "");
```

```
        operator const char*( ) const {return wsk}  // funkcja typu const[2]
//.................
};
main()    {
string s (”abcd”);
cout << s<<endl;   //string zostanie przekształcony na const char*
}
```

Dodanie nowego operatora konwersji do klasy string

```
class string {
        char * wsk;
public:
        string(const char*  = ””);
        operator const char*( ) const {return wsk}  // funkcja typu const
        operator int ( ) const {return atoi(wsk);}
//.................
};
main()    {
string s (”abcd”);
cout << s<<endl;    // błąd kompilacji dwuznaczna konwersja bo w tej
instrukcji
                    // można przekształcić string na const  char * lub int
}
```

Rozwiązania problemu:

1) jawna konwersja typu

```
 cout << (const char *) s<<endl;
```

lub

2) jawna konwersja w oparciu w zwykła funkcje składową w klasie string

```
class string {
        char * wsk;
public:
        string(const char*  = ””);
```

[2] Funkcja typu const to funkcja, która gwarantuje, że jeśli zostanie wywołana na rzecz jakiegoś obiektu, to nie będzie modyfikować jego danych składowych.

```
        operator const char*( ) const {return wsk}  // funkcja typu const
        int zmiana_ int ( ) {return atoi(wsk);}
//.................
};
```

teraz konieczne jest jawne wywołanie funkcji zmiana_int

```
void fun (const string & a) {
//.............
      int k = a.zmiana_int();
//.............
}
```

Jak kompilator dopasowuje najpierw szuka funkcji dokładnie pasującej później konwersja itd

Przykład 1

funkcja (int); funkcja (X); X::operator int ();

X obiekt;
funkcja (obiekt); // wywołana zostanie funkcja funkcja (X) , pasuje doklanie
// do wywolania

Przykład 2

funkcja (float); X::operator int ();

X obiekt;
funkcja (obiekt); // dwie konwersje X-> int oraz int ->float

Przykład 3

Dane sa klasy X, Y, Z.
Konwersja zdefiniowana przez programste może być zastosowana w danym wyrazenie jednokrotnie.

Przykład 4

funkcja (float); funkcja (X); int ------→X

funkcja (1); // wywołana zostanie funkcja (int) bo wystarczyła stand.
//konwersja

Przykład 5

funkcja (float); funkcja (int); X ------→int X ------→float

X obiekt;
funkcja (obiekt); // zadna funkcja - błąd

Przykład 6

funkcja (float); funkcja (long); X ------→int X ------→float

X obiekt;
funkcja (obiekt); // zostanie wywołana funkcja (float)

Przykład 7
Gdy jedna funkcja konwertująca jest prywatna a druga publiczna/./ błąd

Operator (), New , delete

```
template <class T>
class macierz
{
   T *wsk;
   int m, n;
public:
   macierz(int i, int j, int v = 1);
   macierz(const macierz &r);
   macierz(T k);
```

```
    ~macierz();
    macierz & operator =(const macierz &r);
    class rzad
      {
       rzad(T *wsk, int roz) : w(wsk), r(roz){  }
    public:
       T & operator[] (int x)
        {
          if (x >= r)
            {  cout << "Zbyt duzy indeks wiersza" << endl;
               return *w; }
           return w[x];
       }
    private:
       float *w;
       int r;
       friend class macierz<T>;
    };
    rzad operator [] (int x)
    {
       if (x >= n)
       {
          cout << "Zbyt duzy indeks kolumny" << endl;
          return rzad(wsk, m);
       }
       return rzad(wsk + (x * m), m);
    }
//////////////
    macierz &operator ~();

    friend macierz operator +(const macierz &a, const macierz &b);
    friend macierz operator -(const macierz &a, const macierz &b);
    friend macierz operator *(const macierz &a, const macierz &b);

    friend ostream &operator <<(ostream &s, const macierz &r);
    friend istream &operator >>(istream &s, macierz &r);
};

template <class T>
macierz<T>::macierz(int i, int j, int v) : wsk(new T[i*j]), m(i), n(j)
// detale
```

```
}

template <class T>
macierz<T>::macierz(const macierz<T> &r) : wsk(new T[r.m*r.n]),
m(r.m), n(r.n)
{
// detale
}


int main()
{
    clrscr();
    macierz<float> a(3, 3), b(2, 2), c(2, 2);
    cout << "Wprowadz macierz a" << endl;
    cin >> a;
    cin  >> a[2][1];
    cout << a;
    return 0;
}
```

Szablony Funkcji

```
#include <iostream.h>
template <class jakis_typ>
jakis_typ wieksza(jakis_typ a, jakis_typ b)
{ return (a > b) ? a : b ;}


int wieksza(float a, float b)
{       return (a > b) ? a : b ;}


main()
{
 int    a = 4,              b = 11;
double x = 12.6 ,          y = 67.8;
```

```
        cout <<  wieksza(a, b) << endl;
        cout <<  wieksza(x, y) << endl;
        cout <<  wieksza('A', 'Z' ) << endl;
                 wieksza(2,2);
}
```

Własności szablonu funkcji:

✓ Parametrem szablonu funkcji może być jedynie nazwa typu.

✓ Typ rezultatu nie ma znaczenia

```
int a; float b;
a = wieksza(1,3);
b = wieksza (1,4);
template <class jakis_typ>
jakis_typ wieksza(jakis_typ a, jakis_typ b)
```

kompilator wygeneruje tylko funkcje

int wieksza(int , int)

```
template <class jakis_typ>
jakis_typ wieksza(jakis_typ a, jakis_typ b)
{
        return (a > b) ? a : b ;
}
```

Rozważmy tą funkcje dla klasy

```
class osoba {
      public:
        char imie[10];
        int wiek;//..detale
};
```

```
osoba s1, s2;
większa (s1, s2);
```

funkcja, która powstałaby z szablonu nie byłaby poprawna

aby funkcja była poprawna należy zdefiniować w klasie operator

```
int osoba::operator > (osoba b);    lub int operator > (osoba a, osoba b);
```

Przykład:

```
template <class typ>
void uniwers(typ obj, int wartosc)
{
        typ obiekt1(wartosc);           // inicjalizacja konstruktorem
        typ obiekt2 = typ(wartosc);     // jawne wywołanie konstruktora
        obiekt2.typ::~typ();            // jawne wywołanie destruktora
}
```

```
class liczba
{       int w ;
        char opis[30];
public:
                liczba(int wart): w(wart) {}
                ~liczba(){ cout << "to jest destruktor"<< endl; };
};

main()
{
        liczba a(7);
        uniwers(a, 17);          // standard bo przesyłamy obiekt klasy
```

```
        int n = 3 ;
        uniwers(n, 13);
        cout << "Koniec programu\n" ;
}
```

Kompilator dla parametru funkcji szablonowej typu int akceptuje instrukcje konstruktora, destruktora tzn:

int obiekt1(wartość) // równoważne int obiekt1=wartość

int obiekt2 = int (wartość) // równoważne rzutowaniu

obiekt2.int::~int();

Deklaracja funkcji szablonowej

```
template <class typ>
typ funkcja (typ a, typ b);
```

Definicja funkcji szablonowej

```
template <class typ>
typ funkcja (typ a, typ b);
{
//.. ciało funkcji
}
```

Szablon o dwóch typach parametrów

```
template <class typ1, class typ2>
int funkcja (typ1 a, typ2 b,int a, char c, typ1 aa)
{
}
```

```
template <class typ1, class typ2>
int funkcja (typ1 a, typ2 b, typ1 c)
{
}
```

Przeładowanie nazwa szablonów

```
template <class typ1, class typ2>
int funkcja (typ1 a, typ2 b)
{
}
template <class typ>
int funkcja (typ a, typ b)
{
}
funkcja (1, 2);     //bląd
funkcja (1,'c');
```

Parametr szablonu użyty w ciele szablonu

```
template <class typ>
typ funkcja (typ a, typ b)
{
typ  zmienna;      // zakres ważności nazwy parametru obejmuje całą
definicje
                   // szablonu
//... detale
return zmienna;
}
```

Przykłady

```
#include <iostream.h>
template <class typ1, class typ2>
```

```
typ2 wieksza(typ1 a, typ2 b)
{
        return (a > b) ? a : b ;
}

main()
{
int i = 25L ;
double d = 44.123 ;

cout << wieksza(2 , 2) << endl;
cout << wieksza(d, i) << endl ;
int (*wsk)(char, int) = wieksza;
}

//////////////////////

#include <iostream.h>
void fun(int k);
template <class fun>
fun funkcja (fun a, int b)
{
        ::fun(b) ;              // wywołanie fun globalnej
        return (a+b);
}
main()
{
        double x = 5.5;
        funkcja (x, 7);
}
/*************************************************************/
```

Wskaźnik do funkcji szablonowej

```
template <class typ1, class typ2>
typ2 wieksza(typ1 a, typ2 b)
{
        return (a > b) ? a : b ;
}

int (*wsk) (int, int);
wsk = wieksza;   // kompilator wygeneruje funkcje int wieksza (int, int) i
                 // podstawi jej adres wskaźnik wsk
```

Przykłady

```
#include <iostream.h>
void zamiana(int & lewy, int & prawy)
{int pomocniczy ;
        pomocniczy = lewy ;
        lewy = prawy ;
        prawy = pomocniczy ;
}
/*************************************************************/
void zamiana(int* & lewy, int* & prawy)
{
int *pomocniczy ;
        pomocniczy = lewy ;
        lewy = prawy ;
        prawy = pomocniczy ;
}
#include <iostream.h>
#include <string.h>
template <class lancuch>
```

```
void zamiana(lancuch & lewy, lancuch & prawy)
{
lancuch pomocniczy ;
                pomocniczy = lewy ;
                lewy = prawy ;
                prawy = pomocniczy ;
}
main()
{
float  f = 3.3, g = 2.2;
        zamiana(f,g);
char  c1 = 'a', c2 = 'z' ;
        zamiana(c1, c2);

char * wsk1 = new char[70] ;
char * wsk2 = new char[70] ;
strcpy(wsk1, "abc");              // ((6))
strcpy(wsk2, "defg");
zamiana(wskA, wskB);
        cout << "Po zamianie adresow zapisanych we wskaznikach\n"
                << "wsk1 pokazuje na: " << wsk1 << endl
                << "wsk2 pokazuje na: " << wsk2 << endl;
// zamiana adresów nie wartości wsk1 -> ”defg”  wsk2->”abc”

zamiana(c1, 'm');// po zamianie c1=’m’ ale obiekt chwilowy
                        //char chwilowy= ‘m’ który przyjmie wartość c1
                        // przestaje istnieć        po wykonaniu funkcji.
}
```

Parametr a typ rezultatu funkcji

template <**class** typ_r, **class** typ>

typ_r fun (typ a);

//błąd bo dla wywołania fun(2) kompilator nie wie jaką funkcje

wygenerować

// int fun (int) czy float fun (int).. itd

Ponadto kompilator, gdyby dopuszczał taki szablon to mógłby

wygenerować następujące funkcje:

int fun (int);
char fun (int);
float fun (int);
float* fun (int); // nieprawidłowe przeładowanie

Wskaźnik do funkcji jako parametr szablonu

```
#include <iostream.h>
#include <conio.h>
int funkcja(int, int);
template <class typ>
void fun(typ wsk_do_f)
{
        cout<<wsk_do_f(11,11);
}

void fun1(int (*wsk)(int, int))   // dla porównania zwykła globalna funkcja
{
        cout<<wsk(4,4);
}

main()
{
clrscr();
fun(funkcja);  // tutaj tworzymy funkcje szablonową void fun(int (*wsk)(int, int))
```

```
fun1(funkcja); // wywołanie funkcji globalnej
}
```

Obiekty statyczne w szablonie funkcji

```
template <class T>
void fun(T a)                                                    // ((1))
{
static int licznik_pracy=0 ;
        licznik_pracy++;

}
main()
{
        fun(1);
        fun(1);
        fun(3.14);   // powstaje nowa funkcja szablonowa i nowa zmienna
                     // statyczna

        fun(0);
        fiolek('x');   // powstaje nowa funkcja szablonowa i nowa zmienna
                     // statyczna

}
```

Funkcje specjalizowane

```
template <class typ>
typ wieksza(typ a, typ b)
{return (a > b) ? a : b ;}
```

```
// funkcja specjalizowana
char* wieksza(char *a, char *b)
 {      if(strlen(a) > strlen(b)) return a ;
        else    return b ;}

main()
{int    i = 5, j = 1;
char  z1 = 'a', z2 = 'b' ;
        cout <<  wieksza(i, j)  << endl;
        cout << wieksza(z1, z2) << endl;
        char s1[10] = { "abc" };
        char s2[10] = { "xyzzzz" };
        cout << wieksza(s1, s2) << endl;   // funkcja specjalizowana

}
```

Alokacja funkcji specjalizowanej w programie

Definicja szablonu

Deklaracja (definicja) funkcji specjalizowanej

Wywołania

Błędne umieszczenie funkcji specjalizowanej

Definicja szablonu

Wywołania

Deklaracja (definicja) funkcji specjalizowanej

Przeładowanie nazw szablonów funkcji

Dwa szablony funkcji o identycznej nazwie mogą istnieć wtedy, gdy nie będą produkowały funkcji o takich samych argumentach.

```
#include <iostream.h>
template <class typ>                    // szablon
```

```
typ funkcja(typ obiekt)
{
        // detale
        return obiekt;
}
/*************************************************************/
template <class typ1>                       // szablon
typ1 funkcja (typ1 obiekt, int a)
{
        // detale
        return obiekt;
}
/*************************************************************/
void funkcja (char* wsk)                    // funkcja specjalizowana
{
        // ...detale
}
/*************************************************************/
void funkcja (char * s1, char * s2)  // zwykla funkcja   ((4))
{
        // ...detale
}
main()
{
        wywolanie(5);        // funkcja z szablonu pierwszego

        wywolanie(3.14);     // funkcja z szablonu pierwszego

        wywolanie(5, 2);         // funkcja z szablonu drugiego
        wywolanie(3.14, 2);      // funkcja z szablonu drugiego

        wywolanie("abcd");       // funkcja specjalizownana
        wywolanie("abc", ”cos”); // zwykla funkcja
}
```

Dopasowanie dla funkcji szablonowych

Etapy dopasowania

1. dopasowanie dokładne - szukanie funkcji o takiej samej nazwie i pasujących `````parametrach wywołania
2. dopasowanie do szablonu – szukanie szablonu, który może wyprodukować funkcje o argumentach identycznych, jak występujące w wywołaniu. Uwaga!. Kompilator nie realizuje żadnych konwersji (w tym konwersji trywialnych)
3. dopasowanie do funkcji nieszablonowych:
    - ✓ z trywialną konwersją
    - ✓ z konwersją zdefiniowaną przez uzytkownika
    - ✓ dopasowanie do funkcji z wielokropkiem

Dopasowanie funkcji specjalizowanej

Etap 1. – próba dopasowania dokładnego
Etap 3. – próba dopasowania z zastosowaniem konwersji

Szablon klasy jako parametr funkcji globalnej

```
template <class param>
class sz_klasa
{
        char *nazwa  ;
        param dana ;
//... skladniki
public:
        sz_klasa (char* na, param d)
        {
                nazwa = na;
                dana = d ;
//.skaldniki
        }

friend ostream & operator<<(ostream & ekran, sz_klasa<param> & obiekt);
            //  deklaracja przyjaźni z szablonem funkcji
};

// Szablon funkcji

template <class typ>                                                    //
ostream & operator<<(ostream & ekran, sz_klasa<param> & obiekt)
{
        ekran  <<  obiekt.nazwa
        << ", dana= "
        << obiekt.dana << endl
// << skaldniki
        return ekran ;
}
/*************************************************************/
main()
{        lzes z1 (1,2);
        sz_klasa<int> s1("ABC", 6);
        sz_klasa<int> s2("DDD", 2);
 s1.wstaw();
        // obiekty innej klasy
        sz_klasa<char> s3("AAA", ‘A');
        sz_klasa<char> s4("BBB", 'B');
```

```
        cout << s1 ;
        cout << s2 ;
        cout << s3 ;


}
```

Zawieranie obiektów klas szablonowych

```
template <class param>
class sz_klasaA
{
//... skladniki
};
////////////////////////////
template <class parametr>
class sz_klasaB
{
sz_klasaA<int> o1;
sz_klasaA<char> o2;
sz_klasaA<parametr> o3;

//.....
sz_klasaB b1(double);
};

main()
{
sz_klasaB b3;
sz_klasaB <double>   b1;
sz_klasaB <int* >   b2;

}
```

Uwaga szablon klas musi być definiowany w zakresie globalnym, sta nie można zagnieżdżać definicji szablonu w innej klasie i w innym szablonie klasy.

Składniki statyczne w szablonie klas

```
template <class typ>
class klasa
{
public:
        // skladnik statyczne:
        //    typu bedacego parametrem
        static typ t;
        static int i =22;

// konstruktory itd.....

static void funkcja_stat()
        {
// tylko składniki statyczne
        }
};

// Definicje składnikow statycznych dla szablonu klasy
template<class typ>
int klasa<typ>::i;
// skladnik, ktorego typ jest parametrem szablonu
template <class typ>
typ klasa <typ>::t;


main()
{  klasa<int> k1;
   k1.i = 1 ;
 klasa <int>::t = 77 ;
 klasa<int>::funstat();                          // ------------------------------

// inna klasa szablonowa: klasa<char> ======================

        klasa <char> c1;
        klasa <char> c2;

        klasa <char>::i = 16 ;
        klasa <char>::t = 'a' ;
        klasa <char>::funstat();
return 0 ;
}
```

Parametry szablonu

Parametrem szablonu klasy może być:

- ✓ nazwa typu,
    - ✓ adresem funkcji globalnej
    - ✓ adresem składnika statycznego klasy
- ✓ stałe wyrażenie będące:
    - ✓ wartością całkowita

Parametr szablonu - nazwa typu

```
template <class param>
class sz_klasaA
{
//... skladniki
};
////////////////////////////
template <class parametr>
class sz_klasaB

sz_klasaA<int> i;
sz_klasaA<char> i;
sz_klasaA<parametr> i
```

Parametr szablonu - stałe wyrażenie będące wartością całkowita

```
template < int roz>
class  string1
{
char tablica [roz];
//... skladniki
};
////////////////////////////

string1<23> a1;
string1<23> a2;
string1<22> a3;
string1<11> a4;
string1<14> a5;




tab<int, (a>b)> a2;  // błąd
tab<int, (a>b)> a2;  //

template <int roz>
class tab
{
char tablica [roz]
//... skladniki
};

tab<7>  a1;
tab<7+4>  a2;
tab<4+3>  a3;
tab<2> a4;
```

Parametr szablonu – adres obiektu

```
struct osoba
{
        char * nazwa;
                // detale
        obiekt (char* txt) { nazwa = txt ; }
        void funkcja ()
        { // cout << .....
```

```
        }
};


////////////////////////////////////////////////////////////
template <class osoba * adres>
class szablon
{
public:
        void fun1(){ /*...*/ };
        void fun2()
        {
                //.....
                adres->funkcja() ;
        };
};
////////////////////////////////////////////////////////////
// definicje globalnych obiektów

osoba a1("Kowalski') ;
osoba a2("Nowak') ;
/************************************************************/
main()
{
        szablon <&a1> s1, s2 ;
        s1.fun2();
        s2.fun1();

        szablon <&a2> s3 ;
        s3.fun2();
        s3.fun1();
        return 0 ;
}
```

Parametr szablonu – adres funkcji

```
template <void (*wskfun)() >
class samochod
{ // składniki np. tablica nazw
public:
        // skladniki // ...
        void uszereguj()
```

```
        {
                        wskfun() ;
        }
};
/*************************************************************/
void alfabetycznie()                                    // ((4))
{
        // sortowanie
}
/*************************************************************/
void wg_ceny()
{
        // sortowanie
}
/*************************************************************/
main()
{
        samochod <wg_cen>salon1;
        salon1.uszereguj();

         samochod <alfabetycznie> salon2;
        salon2.uszereguj();
        return 0 ;
}
```

Specjalizacja - szablon klas

```
template <class jakis_typ>
class szablon{
        jakis_typ skladnik;
public:
        szablon () ;
        void fun_sz1 (jakis_typ co)
        {
                składnik = 33+składnik;
        }
        jakis_typ fun_s2() ;
};

template <class jakis_typ>
szablon<jakis_typ>:: szablon() : wsk(NULL)    { }

/*************************************************************/
template <class jakis_typ>
jakis_typ szablon<jakis_typ>::fun_s2()
{
        //..detale
}
/*************************************************************/

//  szablon <char*>  s1 ;                       // blad!


//     specjalizowana klasa szablonowa

class szablon<char*>{
        char * wsk;
public:
        szablon();

        //-------------------------------------------------------------

        void fun_sz1(char *nowy);
        char * fun_sz2()
        {       }
        //--- destruktor-----------------------------------------------
        ~szablon
        {       delete wsk;         }
```

```
        };


szablon<char *>:: szablon() : wsk(NULL)        { }

void szablon<char*>::fun_sz1(char *nowy)
{
wsk =  new char …..
}
/*************************************************************/
main()
{
        szablon<float> k ;

        k.fun_sz1(11.23);

        szablon<char*> s1 ;
        s1. fun_sz1(”jakis lanchuch ”);
        return 0 ;
}
```

Specjalizacja funkcji składowej szablonu klas

```
template <class jakis_typ>
class szablon{
        jakis_typ zloze;
public:
        szablon: zloze(0) { } ;

       void fun_sz1 (jakis_typ co);
        jakis_typ fun_sz2();
 } ;
/*************************************************************/
template <class jakis_typ>
jakis_typ szablon<jakis_typ>::fun_sz2()
{
}
/*************************************************************/
template <class jakis_typ>
void szablon<jakis_typ>::fun_sz1(jakis_typ co)
{
```

```
//detale
}

/*********    specjalizowana funkcja skladowa    */

void szablon<char*>::fun_sz1(char * w)
{
//detale
}
main()
{
        szalon<float> k;
        k.fun_sz1 (44.2);
        szablon <char *> s1 ;
        s1.f_sz1 (”abc”);          // wywolanie funkcji specjalizowanej

        return 0 ;
}
```

Klasa zwykła dziedziczy klasę szablonową

```
template <class typ>
class szablon{
///.........
szablon(typ i);
};

class zwykla : public szablon <float>{
///.........

public:
zwykla(int k):szablon<float>(1,1),.......
};

class zwykla : public szablon <int >{
///.........
};
```

Szablon klas dziedziczy klasę szablonową

```
template <class typ>
class szablon {
///.....
public:
szablon(typ k)
        {
        //.....
        }
};

template <class typ1>
class szablon1: public szablon <float>{
typ1 cos;
public:
szablon1(typ1 a, float k):szablon<float>(k), cos(a)
///.....
};
```

Szablon klas dziedziczy inny szablon

```
template <class typ>
class szablon {
///.....
public:
szablon(typ k)
        {
        //.....
        }
};

template <class typ1>
class szablon1: public szablon <typ1>{
typ1 cos;
int x;
public:
szablon1(typ1 a, int i ): szablon<typ1>(a), cos(a), x(i)
///.....
};

template <class typ1, class typ2>
```

```
class szablon1: public szablon <typ2>{
typ1 cos;
int x;
public:
szablon1(typ1 a, typ2 b, int i ):szablon<typ2>(b), cos(a), x(i)
///.....
};
```

Klasa zwykla i klasa specjalizowana nie może dziedziczyć szablonu klas

```
template <class T>
class macierz
{
    T *wsk;
    int m, n;
public:
    macierz(int i, int j, int v = 1);
    macierz(const macierz &r);
    macierz(T k);
    ~macierz();
    macierz & operator =(const macierz &r);
    class rzad
       {
        rzad(T *wsk, int roz) : w(wsk), r(roz){  }
    public:
        T & operator[]    (int x)
         {
           if (x >= r)
             {  cout << "Zbyt duzy indeks wiersza" << endl;
                return *w; }
            return w[x];
        }
    private:
        float *w;
        int r;
```

```
        friend class macierz<T>;
    };
    rzad operator [] (int x)
    {
        if (x >= n)
        {
            cout << "Zbyt duzy indeks kolumny" << endl;
            return rzad(wsk, m);
        }
        return rzad(wsk + (x * m), m);
    }
//////////////
    macierz &operator ~();

    friend macierz operator +(const macierz &a, const macierz &b);
    friend macierz operator -(const macierz &a, const macierz &b);
    friend macierz operator *(const macierz &a, const macierz &b);

    friend ostream &operator <<(ostream &s, const macierz &r);
    friend istream &operator >>(istream &s, macierz &r);
};


macierz<double> A;

cout<< A[2][3];
template <class T>
macierz<T>::macierz(int i, int j, int v) : wsk(new T[i*j]), m(i), n(j)
// detale
}

template <class T>
macierz<T>::macierz(const macierz<T> &r) : wsk(new T[r.m*r.n]),
m(r.m), n(r.n)
{
// detale
}
```

Zarządzanie pamięcią

```
class liczba {
   int licz, mian
public:
   liczba (int a=0, int b=1): licz(a), mian(b) {}


};
```

Test – globalne operatory new i delete

liczba * tab[1000]

```
for (int j=0; j< 250;j++) {
        for (int i=0; i< 1000;i++) {
tab [i] = new liczba (i);
delete tab[i]
}
```

Czas wykonywania 700

Przeładowanie operatorów new i delete

```
#include <conio.h>
#include <iostream.h>

class lista {
public:
lista * nast;
} ;
////////////
class liczba {

int licz, mian;

static lista * lista_wolna;
static void rozszerz();
enum {rozmiar = 32};
public:
    liczba (int a=0, int b=1): licz(a), mian(b) {}
    void * operator new (size_t size);
    void operator delete(void * s, size_t size);
    static void newpamiec() {rozszerz();}
    static void deletepamiec();
};
```

```
//////////////
inline void * liczba::operator new (size_t size)
{
if (0==lista_wolna) rozszerz();

lista * pocz = lista_wolna;
lista_wolna = pocz->nast;
return pocz;
}

////////
inline void  liczba::operator delete(void * s, size_t size)
{
lista * pocz = (lista*) s;
pocz->nast= lista_wolna;
lista_wolna=pocz;
}
/////////////////
void liczba::rozszerz()
{
size_t size = (sizeof(liczba) > sizeof(lista)) ?
sizeof(liczba) :sizeof(lista);
lista * wsk = (lista*) new char [size];
lista_wolna = wsk;
for (int i=0; i<rozmiar;i++)
  {
   wsk->nast = (lista*) new char [size];
   wsk =wsk->nast;
  }
 wsk->nast = 0;
}
////////////
void liczba::deletepamiec()
{
lista * nast_wsk;
for (nast_wsk=lista_wolna;nast_wsk!=NULL; nast_wsk=lista_wolna)
   {
      lista_wolna=lista_wolna->nast;
      delete [] nast_wsk;
   }

}
```

```
///////////////
lista * liczba::lista_wolna=0;

/////////////////
void main()
{
liczba* tab[1000];
liczba ::newpamiec();
for (int j=0; j<250; j++)
  {
    for (int i=0; i<1000; i++)
    tab[i]=new liczba(i);
    for (int ii=0; ii<1000; ii++)
    delete tab[ii];
  }
liczba ::deletepamiec();
}
```

Czas wykonywania 21