PRZECIĄŻANIE (*overloading*) NAZW FUNKCJI

W języku „**C++**” (w odróżnieniu od „C”) można zdefiniować dowolną ilość funkcji o tej samej nazwie, ale o różnej ilości parametrów lub o różnym typie parametrów. Kompilator sam wybierze odpowiednią funkcję na podstawie ilości i typu parametrów przy wywołaniu.

np.

```
int MAX( int a, int b )                  // Funkcja maksimum dla liczb całkowitych
  { return( a > b ? a : b ); }           // if(a>b) return(a); else return(b);

double MAX( double a, double b )         // Maksimum z liczb rzeczywistych
  { return( a > b ? a : b ); }

struct Zespolona                         // Struktura opisująca liczby zespolone
  { double rzecz,uroj; } ;
#include <math.h>
Zespolona MAX( Zespolona a, Zespolona b )
  {                                      //  Maksimum z liczb zespolonych
   if( pow(a.rzecz,2) + pow(a.uroj,2) > pow(b.rzecz,2) + pow(b.uroj,2) )
      return a;
   else
      return b;
  }

int MAX( int a, int b, int c)            // maksimum z trzech liczb całkowitych
  {
    int max=a;
    if( max < b ) max = b ;
    if( max < c ) max = c ;
    return( max );
  }

void main( void )
  {
    int  i=3,  j=7,  k,  l ;
    double  x=12.5,  y=-3.25,  z ;
    Zespolona  z1={2,-3},  z2={-1,2.5},  w ;
    k = MAX( i, j );            // maksimum dla  int
    z = MAX( x, y );            // maksimum dla double
    w = MAX( z1, z2 );                          // maksimum dla Zespolona
     l = MAX( i, j, k );        // maksimum z trzech liczb int
  }
```

C++ – NARZĘDZIE ABSTRAKCJI DANYCH

Język C++ umożliwia definiowanie nowych typów danych,
które mogą się zachowywać „prawie” tak samo jak typy wbudowane !!!

→ zadecyduj jakie chcesz mieć typy (co reprezentują),

→ dla każdego typu zdefiniuj pełny zbiór operacji.

```
#include <iostream.h>
class Wsp_XY                    // wektor dwu współrzędnych x i y na płaszczyźnie
  {
      double x, y;                          // składowe wektora współrzędnych
  public:
      Wsp_XY( void ) : x( 0 ), y( 0 )  { }                    // konstruktory
      Wsp_XY( double xx, double yy ) : x( xx ), y( yy )  { }
      //  .  .  .
      void Drukuj( void )                   // metoda wydruku wartości na ekran
        { cout << '[' << x << ',' << y << ']' ; }
      //  .  .  .                                    //przykładowe „operatory”
      friend  Wsp_XY operator + ( Wsp_XY, Wsp_XY );
      friend  ostream& operator<<( ostream& strumien, Wsp_XY );
  };

Wsp_XY operator + ( Wsp_XY a, Wsp_XY b)
{  // przykładowa implementacja operatora dodawania  ‘ + ’
   return( Wsp_XY( a.x + b.x, a.y + b.y ) );
}

ostream& operator<<( ostream& strumien, Wsp_XY z)
{  // przykładowa implementacja operatora przesłania ‘ <<’ danych wektora do strumienia
   strumien << '[' << z.x << ',' << z.y << ']' ;
   return strumien;
}

void main( void )
{
   Wsp_XY a(1,2);
   Wsp_XY b = Wsp_XY(3,4);
   Wsp_XY c;
   c = a + b;
   c.Drukuj();
   cout << c ;                                  //   operator<<( cout , c );
}
```

PRZECIĄŻANIE OPERATORÓW

W języku „**C++**” można zdefiniować nowe znaczenia standardowych operatorów dla obiektów nowych klas (podobnie jak jedna nazwa funkcji może posłużyć do zdefiniowania tej samej operacji dla kilku różnych typów).

Definiowanie nowego znaczenia operatorów pozwala zaprojektować bardziej typową i wygodną notację dla nowych obiektów.

Programista może napisać funkcje definiujące nowe znaczenia dla następujących operatorów :

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

Uwaga: Nie można zmienić składni ani pierwszeństwa w/w operatorów

Definicja nowego znaczenia dowolnego operatora @ wygląda następująco:

1) Dla operatorów jednoargumentowych: @ **x** np. **–x** lub **++x**

```
<typ_danych>  operator @ ( <typ_danych> x )
  {
    // definicja sposobu działania operatora dla danej typu <typ_danych>
  }
```

np. negacja liczby zespolonej:

```
Zespolona  operator – ( Zespolona x )
  {
    x.rzecz = – x.rzecz ;
    x.uroj = – x.uroj ;
    return( x );
  }

// przykład wykorzystania
Zespolona a, b ;                    // utworzenie dwóch zmiennych zespolonych
b = operator – ( a ) ;              // jawne wywołanie funkcji operatorowej
b = –a ;                            // użycie samego operatora jest skrótem wywołania w/w funkcji
```

2) Dla operatorów dwuargumentowych: **x @ y** np. **x+y** lub **x+=y**

```
<typ_danych>  operator @ ( <typ_danych> x , <typ_danych> y )
  {
    // definicja działania operatora dla dwóch danych typu <typ_danych>
  }
```

np. sumowanie liczb zespolonych:

```
Zespolona  operator + ( Zespolona x, Zespolona y )
  {
    Zespolona suma;
    suma.rzecz = x.rzecz + y.rzecz ;
    suma.uroj = x.uroj + y.uroj ;
    return( suma );
  }

// przykład wykorzystania
Zespolona a, b, c ;                  // utworzenie trzech zmiennych zespolonych
c = operator + ( a, b ) ;            // jawne wywołanie funkcji operatorowej
c = a + b ;          // użycie samego operatora jest skrótem wywołania w/w  funkcji
```

np. operator przypisania z sumowaniem:

```
// UWAGA !!!  Pierwszy argument musi być przekazywany przez referencję.
Zespolona  operator += ( Zespolona &x, Zespolona y )
  {
    x.rzecz += y.rzecz ;
    x.uroj += y.uroj ;
    return( x );
  }

// przykład wykorzystania
Zespolona a, b ;                     // utworzenie dwóch zmiennych zespolonych
operator += ( a, b ) ;               // jawne wywołanie funkcji operatorowej
a += b ;             // użycie samego operatora jest skrótem wywołania w/w  funkcji
```