

# Об'єктно-орієнтоване програмування

Лекція №3. Перевантаження операторів та  
функцій в C++

# Перевантаження функцій

- Перевантажені функції – це функції з одним й тим же ім'ям, що мають різні списки параметрів. Параметри можуть вирізнятися типами та/або кількістю. Тип повертаємого функцією значення до уваги не береться

# Приклад

```
void f(int);
```

```
void f(char);
```

```
void f(long);
```

```
void f(float, int);
```

```
void f(int, int, int);
```



# Важливо!

- Функції, що перевантажуються, не повинні мати співпадаючі списки параметрів (в тому числі і при використанні параметрів за замовчуванням).

# Вибір перевантаженої функції

Якщо визначені декілька функцій з однаковим ім'ям та різними списками параметрів (перевантажені функції) і в програмі зустрічається виклик функції, компілятор повинен вибрати одну з перевантажених функцій. Існує певний алгоритм вибору функцій, у відповідності з яким вибирається функція, яка найкращим чином відповідає виклику. Якщо не буде встановлена відповідність жодній з перевантажених функцій чи буде встановлена неоднозначна відповідність, на етапі компіляції генерується повідомлення про помилку.



# Правила порівняння

- Точні збіги
- Розширення
- Стандартні перетворення
- Перетворення, що потребують тимчасові змінні  
параметр визначений як посилання, а аргумент  
потребує перетворення (наприклад, перетворення з  
float в int&) чи заданий виразом, значення якого не  
може бути змінено.
- Перетворення, визначені користувачем

# Приклади

```
void print(int);  
void print(const char *);  
void print(double);  
void print(long);  
void print(char);  
char c; int i; short s; float f;  
print(c);           // правило 1; викликається print(char)  
print(i);           // правило 1; викликається print(int)  
print(s);           // правило 2; викликається print(int)  
print(f);           // правило 2; викликається print(double)  
print("a7");        // правило 1; викликається print(char)  
print(49);          // правило 1; викликається print(int)
```



# Приклад з помилкою

```
void f(int, float);
```

```
void f(float, int);
```

Виклик, який приведе к генерації повідомлення про помилку (неоднозначний вибір):

```
f(1.5, 1.5);
```



# Перевантаження операторів

**Перевантаження оператора** полягає в зміні сенсу оператора (наприклад, оператора плюс (+), який звичайно в C++ використовується для додавання) при використанні його з певним класом.

# Оператори, дозволені до перевантаження

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

%	^	&
<	>	+=
^=	&=	=
==	!=	<=
--	->*	,
new[]	delete	delete[]



# Оператори, заборонені до перевантаження

Існують також оператори, заборонені до перевантаження. Зміна їх змісту зруйнувало би логіку програми. До таких операторів належать

- :: (оператор дозволу області видимості),
- . (“точка” — оператор доступу до члена класу),
- ?: (тернарний оператор),
- .\* (доступ до розіменованого вказівника-члена класу),
- sizeof, typeid, static\_cast, dynamic\_cast, const\_cast і reinterpret\_cast.**

Крім того, не рекомендується перевантажувати логічні оператори **&&** і **||**, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів.



# Синтаксис операторних функцій

```
тип_значення_що_повертається operator  
    символ_операції (параметри)  
{  
    ...  
}
```

Наприклад, операторна функція, що перевантажує операцію +, називається `operator+()`.

Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями.

# Обмеження, що супроводжують застосування перевантажених операторів

1. Перевантажені функції не можуть змінити пріоритет операторів.
2. Кількість операндів фіксована: жодного, один чи два.
3. Значення операндів не можна задавати за замовчуванням.



# ПЕРЕВАНТАЖЕННЯ УНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ФУНКЦІЙ- ЧЛЕНІВ

Оператори можуть бути унарними і бінарними.

**Унарний оператор** має один операнд, а бінарний — два.

Нагадаємо, що до унарних операторів, що перевантажуються, належать такі оператори, як `+`, `-`, `++`, `--`, `&`, `~` і `!`.

**До бінарних операторів**, що перевантажуються, належать всі інші оператори, перераховані в приведеній вище таблиці.

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника `this`. Отже, така функція-член класу не має явних параметрів.



# Унарні оператори “плюс” і “мінус”

```
class TComplex
{
    double Re;  double Im;
public:
    TComplex(double x, double y){Re=x;Im=y;}
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    TComplex operator-() {Re = -Re; Im = -Im; return *this;}
};

int main()
{
    TComplex z(1,1),u(0,0);
    z.print();
    u=-z;
    u.print();
    getch();  return 0;
}

void TComplex::print()
{ cout<<"("<<Re<<" "<<Im<<")\n"; }
```

# Оператори інкремента і декремента

```
TComplex& operator++()  
{  
    ++Re; ++Im;  
    printf("Префіксна форма ++ \n");  
    return *this;  
}  
const TComplex operator++(int i)  
{  
    ++Re; ++Im;  
    printf("Постфіксна форма ++ %d\n",i);  
    return *this;  
}  
TComplex& operator--()  
{  
    --Re; --Im;  
    printf("Префіксна форма -- \n");  
    return *this;  
}  
const TComplex operator--(int i)  
{  
    --Re; --Im;  
    printf("Постфіксна форма -- %d\n",i);  
    return *this;  
}
```



```
int main()
{
    TComplex z(1,1);
    ++z;
    z.print();
    z++;
    z.print();
    --z;
    z.print();
    z--;
    z.print();
    return 0;
}
```

Якщо символ операції ++ стоїть перед операндом, викликається операторна функція `operator++()`, якщо після — операторна функція `operator++(int i)`. Змінна `i` відіграє роль прапора, що повідомляє компілятору, що дана функція перевантажує постфіксну форму оператора інкремента і декремента.



# Унарні оператори !, & і ~

Оператори заперечення (!), взяття адреси (&) і побітового заперечення (~) допускають перевантаження, але не мають універсальних альтернатив, що варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора ! можна позначати операцію звертання матриці, а за допомогою символу ~ — її транспонування. Щоправда, застосування тильди закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанину. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.

# Перевантаження оператора ->

```
class TClass
{
    int n; int counter;
public:
    TClass(int x){n=x;counter=0;}
    TClass* operator->();
    int get(void) { return n;}
    int ref(void) { return counter; }
};
TClass* TClass::operator ->()
{
    counter++;
    return this;
}
int main()
{ TClass a(1), b(2);
  printf("n = %d \n",a->get());
  printf("n = %d \n",b->get());
  printf("n = %d \n",a->get());
  printf("counter = %d \n",a->ref());
  printf("counter = %d \n",b->ref());
  getch();
  return 0;
}
```



# ПЕРЕВАНТАЖЕННЯ БІНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ФУНКЦІЙ-ЧЛЕНІВ

**Бінарний оператор** має два операнди. Його виклик виконується об'єктом, розташованим у лівій частині оператора. Отже, бінарний оператор

$a+b$

еквівалентний такому оператору.

$a.operator+(b)$

Таким чином, бінарна операторна функція-член повинна має тільки один параметр, що задає другий операнд

Вказівник *this* на перший операнд вона одержує неявно.

Для того щоб бінарну операторну функцію можна було застосовувати усередині виразів, необхідно, щоб вона повертала об'єкт свого класу.



# Перевантаження бінарного оператора +

```
class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y){Re=x;Im=y;}
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    TComplex operator+(TComplex z)
    {
        TComplex w(0,0);
        w.Re = Re+z.Re;
        w.Im = Im+z.Im;
        return w;
    }
};

int main()
{
    TComplex u(1,1),v(2,2),z(0,0);
    z=u+v;
    z.print();
    return 0;
}
```

# Перевантаження оператора присвоювання

```
class TArray
{
    int *p;
    int size;
public:
    TArray(long n, int x);
    TArray(TArray&);
    TArray& operator=(TArray& X);
    void view();
};
int main()
{
    TArray x(10,1),y(10,0);
    x.view();
    y = x;
    y.view();
    getch();
    return 0;
}
TArray::TArray(long n, int x)
{
    size = n;
    p = new int[size];
    for (long i=0; i<size; i++)p[i] = x;
}
```



# Перевантаження оператора присвоювання

```
TArray::TArray(TArray& X)
{
    size=X.size;
    p = new int[size];    // Глибоке копіювання
    for (long i=0; i<X.size; i++) p[i] = X.p[i];

}

void TArray::view()
{
    for(long i=0; i<size; i++) printf("%d ",p[i]);
}

TArray& TArray::operator=(TArray& X)
{
    if(this == &X) return *this; // Перевірка самоприсвоювання.
    if(size==X.size)            // Глибоке копіювання
                                // (вказівник не копіюється!)
    for (long i=0; i<X.size; i++) p[i] = X.p[i];
    else printf(" Size ! \n");
    printf("\n");
    return *this;
}
```

# Перевантаження скорочених операторів присвоювання

```
class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    const TComplex operator+=(const TComplex& z) // Додавання
    {
        Re = Re+z.Re;
        Im = Im+z.Im;
        printf("Operator += \n");
        return *this;
    }
    const TComplex operator+(const TComplex& z) // Додавання
    {
        TComplex w=*this;
w+=z;
        printf("Operator + \n");
        return w;
    }
};
```



```
int main()
{
    TComplex u(1,1),v(2,2),z(3,3);
    u+=v;
    u.print();
    z=u+v;
    z.print();
    getch();
    return 0;
}
```

# ПЕРЕВАНТАЖЕННЯ БІНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ДРУЖНІХ ФУНКЦІЙ

```
class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    friend void print(TComplex z);
    friend TComplex operator+(TComplex x, TComplex y);
};

int main()
{
    TComplex u(1,1),v(2,2),z(0,0);
    z=u+v;
    print(z);
    getch();
    return 0;
}
```



```
TComplex operator+(TComplex x, TComplex y)
{
    TComplex w(o,o);
    w.Re = x.Re+y.Re;
    w.Im = x.Im+y.Im;
    return w;
}

void print(TComplex z)
{
    cout<<"("<<z.Re<<", "<<z.Im<<")\n";
}
```