

# Лекція 11.

## Дружні класи та функції.

## Перевантаження операторів.



# План на сьогодні

1

Дружні класи

2

Дружні функції

3

Перевантаження операторів



# Дружні класи



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Дружні класи

Клас-друг може отримати доступ до приватних і захищених членів інших класів, у яких він оголошений як друг

В C++ оголошується, використовуючи ключове слово `friend`.

# Синтаксис

```
friend class class_name; // declared in the base class
```

```
class Geeks {  
    // GFG is a friend class of Geeks  
    friend class GFG;  
}
```

Base Class

Syntax

```
class GFG {  
    Statements;  
}
```

Friend Class

# Приклад

```
class GFG {  
private:  
    int private_variable;  
  
protected:  
    int protected_variable;  
  
public:  
    GFG()  
    {  
        private_variable = 10;  
        protected_variable = 99;  
    }  
  
    // friend class declaration  
    friend class F;  
};
```

```
class F {  
public:  
    void display(GFG& t)  
    {  
        cout << "The value of Private Variable = "  
              << t.private_variable << endl;  
        cout << "The value of Protected Variable = "  
              << t.protected_variable;  
    }  
};  
  
int main()  
{  
    GFG g;  
    F fri;  
    fri.display(g);  
    return 0;  
}
```

```
The value of Private Variable = 10  
The value of Protected Variable = 99
```

Примітка: Ми можемо оголосити клас-друг або функцію-друга в будь-якому місці тіла базового класу, незалежно від того, чи це приватний, захищений або публічний блок.

# Дружні функції



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Дружні функції

Як і клас-друг, функція-друг може отримати спеціальний доступ до приватних і захищених членів класу в C++.

Вони не є членами класу, але можуть отримувати доступ і маніпулювати приватними та захищеними членами цього класу, оскільки оголошені як друзі.

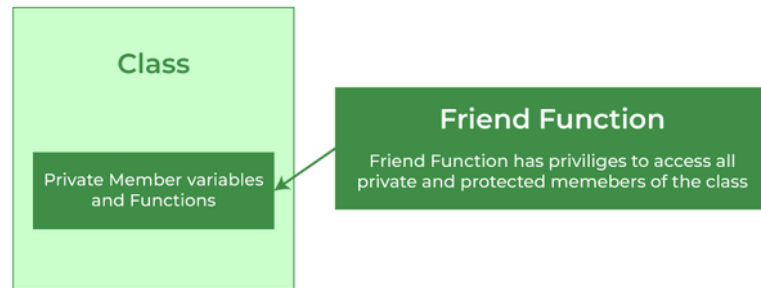


# Види дружніх функцій

Глобальні функції

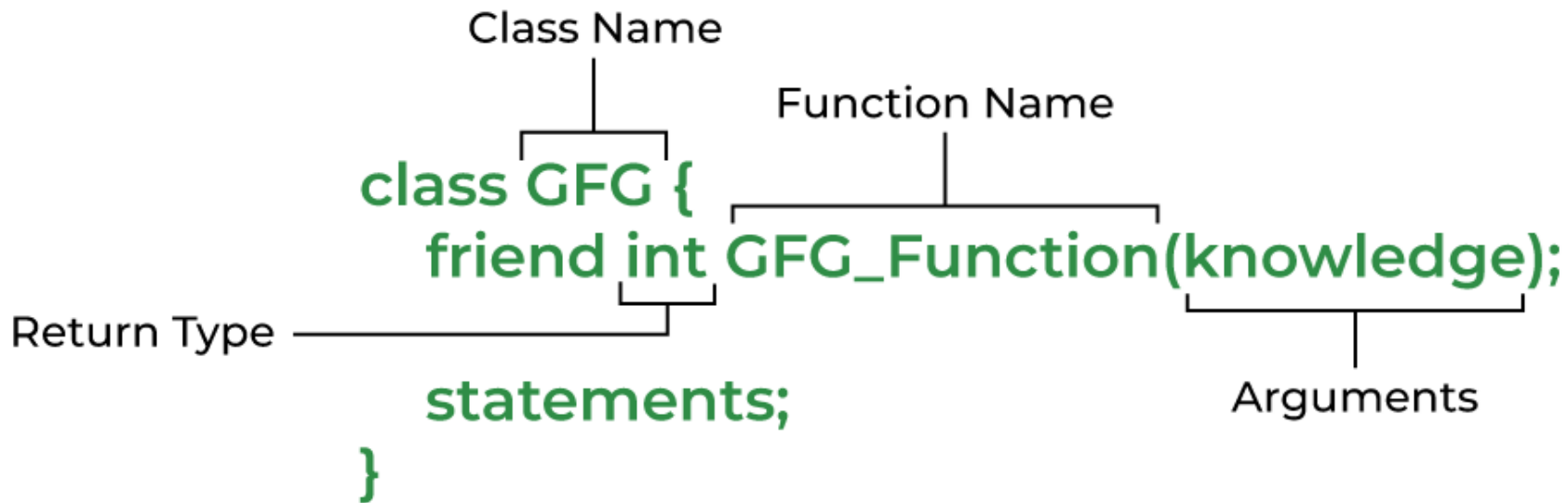
Методи інших класів

## Friend Function



```
friend return_type function_name (arguments);    // for a global function  
  
friend return_type class_name::function_name (arguments);    // for a member  
function of another class
```

# Синтаксис



# Приклад глобальної функції

```
class Base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    Base()
    {
        private_variable = 10;
        protected_variable = 99;
    }

    // friend function declaration
    friend void friendFunction(Base& obj);
};

// friend function definition
void friendFunction(Base& obj)
{
    cout << "Private Variable: " << obj.private_variable
        << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
    Base object1;
    friendFunction(object1);

    return 0;
}
```

Private Variable: 10  
Protected Variable: 99

# Метод іншого класу як функція-друг

```
class Base;

class AnotherClass {
public:
    void memberFunction(base& obj);
};

class Base {
private:
    int private_variable;
protected:
    int protected_variable;
public:
    Base( )
    {
        private_variable = 10;
        protected_variable = 99;
    }
    // friend function declaration
    friend void AnotherClass::memberFunction(Base&);
};
```

```
// friend function definition
void anotherClass::memberFunction(Base& obj)
{
    cout << "Private Variable: " <<
obj.private_variable
    << endl;
    cout << "Protected Variable: " <<
obj.protected_variable;
}

// driver code
int main()
{
    Base object1;
    AnotherClass object2;
    object2.memberFunction(object1);

    return 0;
}
```

Private Variable: 10  
Protected Variable: 99

Примітка: Порядок, у якому ми визначаємо функцію-друга іншого класу, має значення і потребує уваги. Ми завжди повинні визначити обидва класи перед визначенням функції.

# Особливості дружніх функцій

Привілей доступу до приватних і захищених даних класу

Оголошується як друг за допомогою ключового слова **friend** у межах класу

Ключове слово **friend** вказується лише в **оголошенні** функції-друга, але не в її визначенні або виклику.

Викликається як звичайна функція. Її не можна викликати за допомогою імені об'єкта та оператора крапки.

Можна оголосити в будь-якому розділі класу, тобто public, private або protected.

# Дружня функція до декількох класів

```
// Forward declaration
class ABC;

class XYZ {
    int x;

public:
    void set_data(int a)
    {
        x = a;
    }

    friend void max(XYZ, ABC);
};

class ABC {
    int y;

public:
    void set_data(int a)
    {
        y = a;
    }

    friend void max(XYZ, ABC);
};
```

```
void max(XYZ t1, ABC t2)
{
    if (t1.x > t2.y)
        cout << t1.x;
    else
        cout << t2.y;
}

// Driver code
int main()
{
    ABC _abc;
    XYZ _xyz;
    _xyz.set_data(20);
    _abc.set_data(35);

    // calling friend function
    max(_xyz, _abc);
    return 0;
}
```

# Переваги дружніх функцій

Доступ до членів класу без необхідності наслідування.

“Міст” між двома класами, отримуючи доступ до їхніх приватних даних.

Підвищення універсальності перевантажених операторів.

Можна оголосити в будь-якій частині класу: public, private або protected.

# Недоліки дружніх функцій

Доступ до приватних членів класу з зовні порушує принцип приховування даних.

Не можуть виконувати поліморфізм під час виконання.



# Важливо відмітити

Слід використовувати лише для обмежених цілей.

*(Занадто багато функцій або зовнішніх класів, оголошених друзями класу з доступом до захищених або приватних даних, знижує цінність інкапсуляції окремих класів в об'єктно-орієнтованому програмуванні.)*

“Дружба” не є взаємною. Якщо клас А є другом класу В, то В автоматично не стає другом класу А.

“Дружба” не успадковується.

# Перевантаження операторів

# Перевантаження операторів

Оператор + не визначений

```
class A {  
    statements;  
};  
  
int main()  
{  
    A a1, a2, a3;  
  
    a3 = a1 + a2;  
  
    return 0;  
}
```

Для того, щоб оператор “+” додавав два об’єкти класу, потрібно перевизначити значення оператора “+” так, щоб він складав два об’єкти класу. Це досягається за допомогою концепції “перевантаження операторів”.

# Перевантаження операторів

В C++ перевантаження операторів є поліморфізмом на етапі компіляції.

Це концепція надання особливого значення існуючому оператору в C++ без зміни його початкового значення.

# Перевантаження операторів

```
returned_type operator@(list of arguments) ;
```

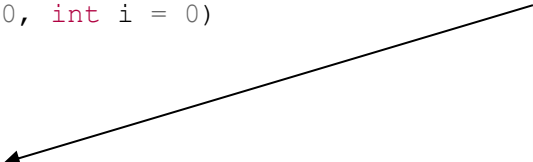


Позначення операції (наприклад “+”)

Оператори повинні мати прямий доступ до членів класу. Отже, вони мають бути або методами класу, або дружніми глобальними функціями.

# Приклад

```
class Complex {  
private:  
    int real, imag;  
public:  
    Complex(int r = 0, int i = 0)  
    {  
        real = r;  
        imag = i;  
    }  
    Complex operator+(Complex const& obj)  
    {  
        Complex res;  
        res.real = real + obj.real;  
        res.imag = imag + obj.imag;  
        return res;  
    }  
    void print() { cout << real << " + i" << imag << '\n'; }  
};  
int main() {  
    Complex c1(10, 5), c2(2, 4);  
    Complex c3 = c1 + c2;  
    c3.print();  
}
```



Операторні функції такі ж, як і звичайні функції. Єдині відмінності полягають у тому, що ім'я операторної функції завжди є ключовим словом **operator**, за яким слідує символ оператора, і операторні функції викликаються тоді, коли використовується відповідний оператор.

12 + i9

# Обмеження при перевантаженні операторів

Перевантажені функції не можуть змінювати пріоритет операторів.

Кількість операндів фіксована: *жодного, один чи два*.

Значення операндів не можна задавати за замовчуванням.

# Перевантаження операторів

Більшість операторів C++ допускають перевантаження

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

Не перевантажують такі оператори

::   .   .\*   :?

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



# Важливо відмітити

Щоб перевантаження операторів працювало, принаймні один з операндів повинен бути об'єктом користувацького класу.

Компілятор автоматично створює оператор присвоєння за замовчуванням для кожного класу. Оператор присвоєння за замовчуванням копіює всі члени з правого боку до лівого і в більшості випадків працює коректно (ця поведінка схожа на конструктор копіювання).

Ми також можемо створювати оператори перетворення, які можна використовувати для перетворення одного типу в інший

# Оператор перетворення типів

```
class Fraction {  
private:  
    int num, den;  
  
public:  
    Fraction(int n, int d)  
    {  
        num = n;  
        den = d;  
    }  
    operator float() const  
    {  
        return float(num) / float(den);  
    }  
};  
  
int main()  
{  
    Fraction f(2, 5);  
    float val = f;  
    cout << val << '\n';  
    return 0;  
}
```

Оператори перетворення повинні бути методам-членами. Інші оператори можуть бути як методами-членами, так і глобальними методами.

0.4

# Оператор перетворення типів

- ❖ Випадки явного і неявного перетворення
  - явне і неявного приведення як аналог перетворення для вбудованих типів
  - у виразах при підборі реалізацій для перевантажених функцій компілятор сам може виконати (неявно) допустиме перетворення
- ❖ 2 способи перетворення:
  - конструкторами **до типу X**

```
X::X(const Y &);  
X::X(const Y &, int =0);
```
  - операторами перетворення **типу X** до типу **Y**

```
X::operator Y() const{ return Y(...);}
```
- ❖ Проблеми неявного перетворення:
  - приховані виклики і виконання конструкторів та операторів перетворень
  - виникнення неоднозначних шляхів перетворень

Перевага : зменшення кількості перевантажених функцій

# Неявне перетворення конструктором

```
class One {
public:
    One() { std::cout << "in One()"; }
};

class Two {
public:
    Two(const One&) { std::cout << "in Two(const One&)"; }
    Two(const Two& t) { std::cout << "in Two(const Two&)"; }
};

void f(Two) { std::cout << "in f(Two)"; }

int main() {
    One one;
    f(one); // Wants a Two, has a One
           // in One()in Two(const One&)in f(Two)
}
```

# explicit конструктор

- ❖ **explicit** оголошення конструктора – заборона неявного перетворення (неявного автоматичного виклику конструктора)

```
class Two {  
public:  
    explicit Two(const One&)  
    { std::cout << "in Two(const One&);"}  
  
    Two(const Two& t)  
    { std::cout << "in Two(const Two&); }  
};  
  
void f(Two) { std::cout << "in f(Two)"; }  
  
int main() {  
    One one;  
    f(one);  
    // error C2664: 'f' : cannot convert parameter 1 from 'One' to 'Two'  
    // No user-defined-conversion operator available  
    f(Two(one));  
}
```

# Оператор = та this

```
#include <iostream>
#include <string>

using namespace std;

class MyClass {
    string data;

public:
    // Constructor
    MyClass(const string& d = "") : data(d) {}

    // Overloaded Assignment Operator
    MyClass& operator=(const MyClass& other) {
        if (this != &other) { // Check for self-assignment
            data = other.data;
        }
        return *this;
    }

    // Display method
    void display() const {
        cout << data << endl;
    }
};


int main() {
    MyClass obj1("Hello");
    MyClass obj2;

    obj2 = obj1; // Assignment operator is called

    obj1.display();
    obj2.display();

    return 0;
}
```

- ❖ **this** – неявний аргумент кожного нестатичного методу;
- ❖ Містить адресу об'єкта, через який викликано метод (посилання на себе);
- ❖ Це спосіб доступу в методі до даних цього об'єкта;
- ❖ \*this – значення об'єкта, через який викликано метод



Hello  
Hello

# Оператор ++ (префіксний та постфіксний)

```
Point& Point::operator++() {           // prefix increment
    ++x; // Increment x
    ++y; // Increment y
    return *this;
}
```

```
Point Point::operator++(int) {         // postfix increment
    Point old(*this);
    ++*this;
    return old;
}
```

Фіктивний параметр -  
“повідомлення” компілятору,  
що дана функція перевантажує  
постфіксну форму оператора

# Оператори вводу та виводу

```
#include <iostream>
#include <string>

using namespace std;

class Person {
    string name;
    int age;

public:
    // Constructor
    Person(const string& n = "", int a = 0) : name(n), age(a) {}

    // Overloaded << operator for output
    friend ostream& operator<<(ostream& out, const Person& p) {
        out << "Name: " << p.name << ", Age: " << p.age;
        return out;
    }

    // Overloaded >> operator for input
    friend istream& operator>>(istream& in, Person& p) {
        cout << "Enter name: ";
        in >> p.name;
        cout << "Enter age: ";
        in >> p.age;
        return in;
    }
};

int main() {
    Person person;

    // Using >> operator to get input
    cin >> person;

    // Using << operator to display output
    cout << person << endl;

    return 0;
}
```



# Дякую!



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY