

C++

## 程式語言（二）

Introduction to Programming (II)

Function Overloading & Overriding

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

# Platform/IDE

- Dev-C++



<https://www.pngegg.com/en/search?q=Dev-C>

- Codeblocks



<https://icons8.com/icons/set/code-blocks>

- OnlineGDB (<https://www.onlinegdb.com/>)



- Real-Time Collaborative Online IDE (<https://ide.usaco.guide/>)



# Textbooks (We focusing on C++11)

- *Learn C++ Programming by Refactoring* ( 由重構學習 C++ 程式設計 ). Pang-Feng Liu ( 劉邦鋒 ). NTU Press. 2023.
- *C++ Primer. 5th Edition.* Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. 2019.
- *Effective C++.* Scott Meyers. O'Reilly. 2016.
- *Thinking in C++. Vol. 1: Introducing to Standard C++.* 2nd Edition. Bruce Eckel. Prentice Hall PTR. 2000.

# Useful Resources

- Tutorialspoint
  - <https://www.tutorialspoint.com/cplusplus/index.htm>
  - Online C++ Compiler
- Programiz
  - <https://www.programiz.com/cpp-programming>
- LEARN C++
  - <https://www.learncpp.com/>
- MIT OpenCourseWare - Introduction to C++
  - <https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/pages/lecture-notes/>
- Learning C++ Programming
  - <https://www.programiz.com/cpp-programming>
- GeeksforGeeks
  - <https://www.geeksforgeeks.org/c-plus-plus/>



# Function Overloading & Overriding

# Function Overloading vs. Function Overriding

	Function Overloading	Function Overriding
Timing	achieved at compile time	achieved at run time
Parameter, Data types, Return Types, ...	Changed	Unchanged (i.e., the same)
Class Hierarchy	Can be done in base and derived classes.	Can only be done in derived classes.
Scope	The same scope	Difference scopes
Function Behavior	The same; depending on passed parameters	Could be different (i.e., added jobs)

# An Example of Function Overloading

<https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/>

```
#include <iostream>
using namespace std;

// overloaded functions
void fuct(int);
void fuct(float);
void fuct(int, float);

int main() {
    int a = 4;
    float b = 5.5;

    fuct(a);
    fuct(b);
    fuct(a, b);

    return 0;
}
```

```
// Method 1
void fuct(int var){
    cout << "Integer number: " << var
        << endl;
}

// Method 2
void fuct(float var){
    cout << "Float number: " << var << endl;
}

// Method 3
void fuct(int var1, float var2){
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

# Another Example of Function Overloading

```
#include <iostream>
using namespace std;

// overloaded functions
void add(float, float);
void add(float, float, float);

int main() {
    float a = 4.5;
    float b = 5.5;

    add(a, b);
    add(a, b, b);

    return 0;
}
```

```
// add two float numbers
void add(float var1, float var2){
    cout << "sum: " << var1+var2 << endl;
}

// add three float numbers
void add(float var1, float var2, float var3) {
    cout << "sum: " << var1+var2+var3 << endl;
}
```



# Inheritance & Member Functions

- A base class must distinguish the functions **it expects its derived classes to override** from those that **it expects its derived classes to inherit without change**.
- The base class expects its derived classes to override its member functions defined as `virtual` in the base class.
- A base class specifies that a member function should be **dynamically** bound by preceding its declaration with the keyword `virtual`.
- Any **nonstatic** member function, **other than a constructor**, may be virtual.
- The `virtual` keyword appears only on the declaration **inside** the class and may not be used on a function definition that appears outside the class body.

# An Example of Function Overriding

<https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/>

```
class A {  
public:  
    virtual void display() {  
        cout << "hello" << endl;  
    }  
};  
  
class B : public A {  
public:  
    void display() {  
        cout << "bye" << endl;  
    }  
};
```

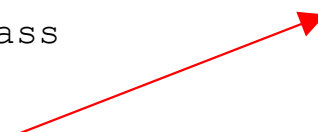
```
int main()  
{  
    A objA;  
    B objB;  
  
    objA.display();  
    objB.display();  
    return 0;  
}
```

# Another Example of Function Overriding

You can also refer to <https://www.programiz.com/cpp-programming/function-overriding>

```
#include<iostream>
using namespace std;

class BaseClass
{
public:
    virtual void Display() {
        cout << "\nThis is Display() method"
              " of BaseClass";
    }
    void Show() {
        cout << "\nThis is Show() method "
              "of BaseClass";
    }
};
```



Try to remove this key word  
and run the program and see  
what will happen.

```
class DerivedClass : public BaseClass
{
public:
    // Overriding method
    void Display() {
        cout << "\nThis is Display() method"
              " of DerivedClass";
    }
};
```

```
int main() {
    DerivedClass dr;
    BaseClass &bs = dr;
    bs.Display();
    bs.BaseClass::Display();
    dr.Show();
}
```

# Recall for the `const` function

- <https://onlinegdb.com/OLYAN3DlT>

```
class MyClass {
private:
    int myData = 0;
public:
    MyClass() = default;
    MyClass(int num): myData(num) {}
    int modMyData1() const { return ++myData; } // error
    int modMyData2() const { return myData + 1; } // fine
    int modMyData3()      { return ++myData; } // fine
};
```

# Example: Book Total Sales with Discount

<https://onlinegdb.com/c3nQoyaVk>

An object of  
Bulk\_quote

inherited from  
Quote

defined by  
Bulk\_quote

bookNo  
price

min\_qty  
discount

```
#include <iostream>
//using namespace std;

// Below we define the base class
class Quote {
public:
    Quote() = default;
    Quote(const std::string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    //returns the total sales price for the specified number of items
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default; // dynamic binding for the destructor
private:
    std::string bookNo;
protected:
    double price = 0.0; // this is protected because we want it to be used by
                        // the derived classes
};

//Below we define a class derived from the base class Quote
class Bulk_quote : public Quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string &book, double p, std::size_t qty, double disc):
        Quote(book, p), min_qty(qty), discount(disc) { }
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; // minimum purchase for the discount to apply
    double discount = 0.0; // the discount to apply
};
```

`std::size_t total = 0;`

Add a member function to  
accumulate the total sold volumes.

Add some statements for the  
total sold volumes.

```
double Bulk_quote::net_price(size_t cnt) {
    if (cnt >= min_qty)
        return cnt * (1-discount) * price;
    else
        return cnt * price;
}
```

# Exercise: Book Total Sales with Discount

An example of the main function for testing:

```
int main()
{
    Quote item("NTOU in Love", 100.0);
    std::cout << "BOOK: " << item.isbn();
    std::cout << ", total cost: " << item.net_price(10) << std::endl;
    Bulk_quote bulk("NTOU No.1", 100.0, 5, 0.2);
    std::cout << "BOOK: " << bulk.isbn();
    std::cout << ", total cost: " << bulk.net_price(10) << std::endl;
    std::cout << "BOOK: " << bulk.isbn();
    std::cout << ", total cost: " << bulk.net_price(20) << std::endl;
    std::cout << "Total sell: " << bulk.Sell_total() << std::endl;
    return 0;
}
```

Sample output:

```
BOOK: NTOU in Love, total cost: 1000
BOOK: NTOU No.1, total cost: 800
BOOK: NTOU No.1, total cost: 1600
Total sell: 30
```

# Preventing Inheritance

```
class NoDerived final { /* */ }; // NoDerived can't be a base class
class Base { /* */ };
// Last is final; we cannot inherit from Last
class Last final : Base { /* */ }; // Last can't be a base class
class Bad : NoDerived { /* */ }; // error: NoDerived is final
class Bad2 : Last { /* */ }; // error: Last is final
```

# Circumventing the Virtual Mechanism

- In some cases, we want to **prevent dynamic binding** of a call to a virtual function.
- We can use the scope to do so.

```
baseP *Quote = Bulk_quote("NTOU in Love", 100.0, 10, 0.2);  
double undiscounted = baseP->Quote::net_price(42);
```



# Pure Virtual Function

- Sometimes we'd like to prevent users from creating objects of a derived class at all.

```
class Disc_quote : public Quote {  
public:  
    Disc_quote() = default;  
    Disc_quote(const std::string& book, double price,  
               std::size_t qty, double disc):  
        Quote(book, price),  
        quantity(qty), discount(disc) { }  
    double net_price(std::size_t) const = 0;  
protected:  
    std::size_t quantity = 0; // purchase size for the discount to apply  
    double discount = 0.0; // fractional discount to apply  
};
```

➡ The function body must be defined **outside** the class.

# Example of Applying the Pure Virtual Function

```
// Disc_quote declares pure virtual functions,  
// which Bulk_quote will override  
Disc_quote discounted; // error: can't define a Disc_quote object  
Bulk_quote bulk; // ok: Bulk_quote has no pure virtual functions
```

# Class Scope under Inheritance

- Name Collisions and Inheritance:

```
struct Base {  
    Base(): mem(0) { }  
protected:  
    int mem;  
};  
struct Derived : Base {  
    Derived(int i): mem(i) { } // initializes Derived::mem to i  
                                // Base::mem is default initialized  
    int get_mem() { return mem; } // returns Derived::mem  
protected:  
    int mem; // hides mem in the base  
};
```

```
Derived d(42);  
cout << d.get_mem() << endl; // what's the output?
```

# Class Scope under Inheritance

- Name Collisions and Inheritance:

```
struct Base {  
    Base(): mem(0) { }  
protected:  
    int mem;  
};  
struct Derived : Base {  
    Derived(int i): mem(i) { } // initializes Derived::mem to i  
                                // Base::mem is default initialized  
    int get_mem() { return mem; } // returns Derived::mem  
protected:  
    int mem; // hides mem in the base  
};
```

```
int get_mem() { return Base::mem; }
```

```
Derived d(42);  
cout << d.get_mem() << endl; // what's the output?
```

# Name Lookup

```
struct Base {  
    int memfcn();  
};  
struct Derived : Base {  
    int memfcn(int); // hides memfcn in the base  
};  
  
Derived d;  
Base b;  
  
b.memfcn(); // calls Base::memfcn  
d.memfcn(10); // calls Derived::memfcn  
d.memfcn(); // error: memfcn with no arguments is hidden  
d.Base::memfcn(); // ok: calls Base::memfcn
```

# Name Lookup (solution: overloading)

<https://onlinegdb.com/fsDlqIITR>

```
struct Base {  
    int memfcn();  
};  
struct Derived : Base {  
    using Base::memfcn;  
    int memfcn(int); // here comes overloading of memfcn  
};  
  
Derived d;  
Base b;  
  
b.memfcn(); // calls Base::memfcn  
d.memfcn(10); // calls Derived::memfcn  
d.memfcn(); // OK now! memfcn with no arguments is not hidden  
d.Base::memfcn(); // ok: calls Base::memfcn
```

# Virtual Functions and Scope

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    // D1 inherits the definition of Base::fcn()
    int fcn(int); // parameter list differs from fcn in Base
    virtual void f2(); // new virtual function that does not exist in Base
};
class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn(); // overrides virtual fcn from Base
    void f2(); // overrides virtual f2 from D1
};
```

# Virtual Functions and Scope

```
Base bobj;  
D1 d1obj;  
D2 d2obj;  
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;  
  
bp1->fcn(); // virtual call, will call Base::fcn at run time  
bp2->fcn(); // virtual call, will call D1::fcn at run time  
bp3->fcn(); // virtual call, will call D2::fcn at run time  
  
D1 *d1p = &d1obj;  
D2 *d2p = &d2obj;  
  
bp2->f2(); // error: Base has no member named f2  
d1p->f2(); // virtual call, will call D1::f2() at run time  
d2p->f2(); // virtual call, will call D2::f2() at run time
```



# Revised Solution

<https://www.onlinegdb.com/cY2jxm4dm>

```
class Base {
public:
    virtual int fcn() {
        cout << "Base fcn()" << endl;
        return 0;
    }
};

class D1 : public Base {
public:
    using Base::fcn;
    // Bring Base::fcn() into scope to
    // prevent it from being hidden

    int fcn(int a) {
        cout << "D1::fcn(int)" << endl;
        return 0;
    }
    virtual void f2() {
        cout << "D1::f2()" << endl;
    }
};
```

```
class D2 : public D1 {
public:
    using D1::fcn;
    // Bring D1::fcn(int) into scope
    int fcn(int a) { return 0; }
    // Hides D1::fcn(int)
    int fcn() {
        // Overrides Base::fcn()
        cout << "D2::fcn()" << endl;
        return 0;
    }
    void f2() { cout << "D2::f2()" << endl; }
};
```

```
Base bobj;
D1 d1obj;
D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d2obj;
bp1->fcn();
bp2->fcn(); // Calls D2::fcn()
D1 *d1p = &d1obj;
D2 *d2p = &d2obj;
d1p->f2(); // Calls D1::f2()
d2p->f2();
```

```
Base fcn()
D2::fcn()
D1::f2()
D2::f2()
```

# Exercise

```
#include <iostream>
using namespace std;

class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
protected:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
int main() {
    Derived derived1;
    Base bObj;
    Base* ptr = &derived1;
    Base &obj = derived1;
    ptr->print();
    obj.print();
    return 0;
}
```

## Problem:

Please modify class Base so that the main function generates the output as below

```
Derived Function
Derived Function
```

# Implicit Type Conversion

```
class Employee {
private:
    string name;
    int empID;
public:
    Employee() = default;
    Employee(string n, int e):
        name(n), empID(e) {}
    void displayInfo() {
        cout << "Employee ID: " << empID
            << endl;
        cout << "Name: " << name
            << endl;
    }
    ~Employee() = default;
};
```

```
int main() {
    Employee e = "Alice";
    //Implicit conversion!!
    e.display();
    return 0;
}
```

## What's the output?

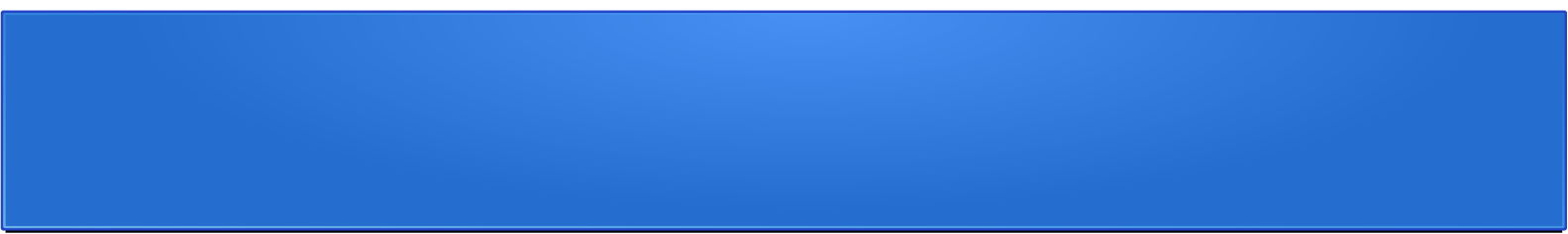
- Employee expects two parameters: string, int.
- Since int can have a default value of 0, C++ implicitly converts "Alice" into an Employee object with empID = 0.

# explicit: Preventing implicit type conversion

```
class Employee {
private:
    string name;
    int empID;
public:
    Employee() = default;
    Explicit Employee(string n, int e):
        name(n), empID(e) {}
    void displayInfo() {
        cout << "Employee ID: " << empID
            << endl;
        cout << "Name: " << name
            << endl;
    }
    ~Employee() = default;
};
```

```
int main() {
    Employee e1 = "Alice"; //Error!!
    e1.display();
    Employee e2("Bob", 101);
    e2.display(); // Correct!
    return 0;
}
```

- Use “explicit” to
  - Prevent implicit, unintended type conversions.
  - Avoid accidental misuse



# Discussions & Questions