

C++

程式語言（二）

Introduction to Programming (II)

Friend Functions & Friend Classes

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

Platform/IDE

- Dev-C++



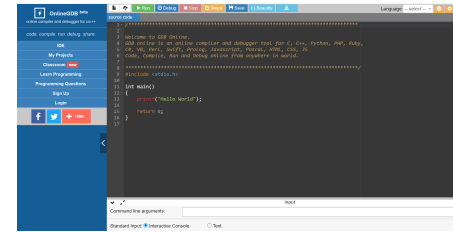
<https://www.pngegg.com/en/search?q=Dev-C>

- Codeblocks

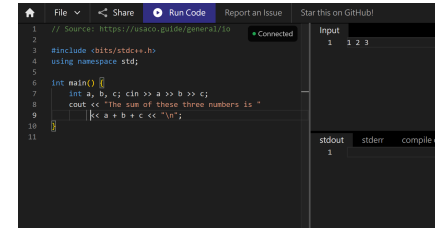


<https://icons8.com/icons/set/code-blocks>

- 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/>



Friend Functions

Friend Functions

Refer to the material at <https://www.programiz.com/cpp-programming/friend-function-class>

- A **friend function** can access the **private** and **protected** data of a class.
- We declare a friend function using the **friend** keyword **inside** the body of the class.

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

class Distance {
    private:
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance() : meter(0) {}
};
```

```
// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Exercise

<https://onlinegdb.com/wvkTzVLw5>

- Try to make the friend function be "protected" or "public".
- See what we will get.

Another Example (Accessing two classes)

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

// forward declaration
class ClassB;

class ClassA {
public:
    // constructor to initialize numA to 12
    ClassA() : numA(12) {}

private:
    int numA;
    // friend function declaration
    friend int add(ClassA, ClassB);
};
```

```
class ClassB {
public:
    // constructor to initialize numB to 1
    ClassB() : numB(1) {}

private:
    int numB;

    // friend function declaration
    friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```




Friend Classes

Friend Classes

Refer to the material at <https://www.programiz.com/cpp-programming/friend-function-class>

- Take a whole class as a friend.

```
class ClassB;

class ClassA {
    // ClassB is a friend class of ClassA
    friend class ClassB;
    ... ..
}

class ClassB {
    ... ..
}
```

- When a class is declared a friend class, all the member functions of the friend class become friend functions.

Example

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

// forward declaration
class ClassB;

class ClassA {
private:
    int numA;

    // friend class declaration
    friend class ClassB;

public:
    // constructor to initialize numA to 12
    ClassA() : numA(12) {}
};
```

```
class ClassB {
private:
    int numB;

public:
    // constructor to initialize numB to 1
    ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};
```

```
int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Another Example

```
#include <iostream>

using namespace std;

class Box {
    double width;

    public:
        friend void printWidth(Box box);
        void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
    width = wid;
}
```

```
void printWidth( Box box ) {
    cout << "Width of box : " << box.width
        <<endl;
}

// Main function for the program
int main() {
    Box box;
    box.setWidth(10.0);
    // Use friend function to print the width.
    printWidth(box);
    return 0;
}
```

An Example for Book Sales

<https://onlinegdb.com/PSz0YMg0R>

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

// Below we define the base class
class Quote {
    friend bool book_compare(Quote &item1, Quote &item2);
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)
        { return n * price; }
    virtual ~Quote() = default; // dynamic binding for the destructor
private:
    std::string bookNo;
protected:
    double price = 0.0;
};

//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) 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; // a account 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;
}

bool book_compare(Quote &item1, Quote &item2) {
    return item1.price > item2.price;
}

int main()
{
    Quote item("NTOU is Love", 100.0);
    std::cout << "BOOK: " << item.isbn();
    std::cout << ", total cost: " << item.net_price(10) << std::endl;
    Bulk_quote bulk("NTOU Forever", 100.0, 5, 0.2);
    std::cout << "BOOK: " << bulk.isbn();
    std::cout << ", total cost: " << bulk.net_price(10) << std::endl;
    if (book_compare(item, bulk))
        std::cout << bulk.isbn() << " is cheaper!" << std::endl;
    else
        std::cout << bulk.isbn() << " is cheaper!" << std::endl;
    return 0;
}
```

Friend Function Can NOT be Inherited

<https://onlinegdb.com/SzMrngR-02>

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

class MyBaseClass {
protected:
    int x;
public:
    MyBaseClass() {
        x = 20;
    }
    friend void display();
};

class MyDerivedClass : public MyBaseClass {
private:
    int y;
public:
    MyDerivedClass() {
        x = 40;
    }
};
```

```
void display() {
    MyDerivedClass derived;
    cout << "The value of x of Base class is: "
          << derived.x << endl;
    cout << "The value of y of Derived class is: "
          << derived.y << endl;
}
```

```
int main() {
    display();
    return 0;
}
```

Exercise

```
class rectangle {
public:
    typedef double unit;
    unit area();
    void set(unit wd, unit ht);
private:
    unit width;
    unit height;
};
```

```
class triangle {
public:
    typedef int unit;
    unit area();
    void set(unit wd, unit ht);
private:
    unit width;
    unit height;
};
```

```
int main() { // DO NOT modify main()
    rectangle::unit x1, y1;
    rectangle obj1;
    cin >> x1 >> y1;
    triangle::unit x2, y2;
    triangle obj2;
    cin >> x2 >> y2;
    obj1.set(x1,y1);
    obj2.set(x2,y2);
    cout << area_sum(obj1, obj2) << endl;
    return 0;
}
```

Sample input:

```
2 5
3 6
```

Sample output:

```
19
```



Supplementary:

Can a constructor be
"private"?

Create an object using a private constructor...

```
class MyClass {  
private:  
    MyClass() {}    // private constructor  
};  
  
// In main():  
MyClass obj;    // Error: 'MyClass::MyClass()' is private
```

Scenarios (some design patterns)

- Singleton pattern
 - Ensure **only one object** is created.
- Static-only **utility** class
 - Prevent anyone from creating an instance.

```
class MathUtils {  
private:  
    MathUtils() {} // Prevent instantiation  
public:  
    static int square(int x) { return x * x; }  
};
```

- Base class constructor as **protected**.
 - So that **only the derived classes** can call it.

Motivation for “Singleton”

- **Multiple** components of the system need access to **the same instance**.
- We don't want multiple copies, because that would lead to inconsistencies or waste of resources.
 - Logging to a file
 - Database connections
 - ...

Example: Logger

```
class Logger {  
public:  
    void log(string msg) { /* write to a log file */  
}  
};
```

```
Logger log1;  
Logger log2;  
  
log1.log("System started");  
log2.log("User logged in");
```



- If each Logger object writes to its own file, your logs get scattered.
- We lose centralized tracking

Example: Logger

<https://onlinegdb.com/ssmsm44ww>

<https://onlinegdb.com/aNnFWR4yZ>

```
class Logger {
private:
    // Private constructor
    Logger() { cout << "[Logger created]" << endl;}

    // Private copy constructor and
    // assignment operator
    Logger(const Logger&) {
        cout << "[Copy constructor called]"
            << endl;
    }
    Logger& operator=(const Logger&) {
        cout << "[Assignment operator called]"
            << endl;
        return *this;
    }
    // Static instance pointer
    static Logger* instance;
```

```
public:
    // Public method to access the
    // instance
    static Logger* getInstance() {
        if (instance == nullptr) {
            instance = new Logger();
        }
        return instance;
    }

    void log(const string& message) {
        cout << "[LOG]: " << message
            << endl;
    }
};

// Remember to define the static member //
// outside the class
Logger* Logger::instance = nullptr;
```

Example: Logger (**x** move constructor)

<https://onlinegdb.com/WtqtrF5-B>

```
class Logger {
private:
    // Private constructor
    Logger() { cout << "[Logger created]" << endl;}

    // Private copy constructor, move constructor
    // and assignment operator
    Logger(const Logger&) {
        cout << "[Copy constructor called]" << endl;
    }
    Logger& operator=(const Logger&) {
        cout << "[Assignment operator called]" << endl;
        return *this;
    }
    Logger(Logger&&) { // not working since it's private
        cout << "[Move constructor called]" << endl;
    }
    Logger& operator=(Logger&&) {
        cout << "[Assignment operator called]" << endl;
        return *this;
    }
    // Static instance pointer
    static Logger* instance;
```

```
public:
    // Public method to access the
    // instance
    static Logger* getInstance() {
        if (instance == nullptr) {
            instance = new Logger();
        }
        return instance;
    }

    void log(const string& message) {
        cout << "[LOG]: " << message
            << endl;
    }
};

// Remember to define the static
// member outside the class
Logger* Logger::instance = nullptr;
```

Example: Logger

```
[Logger created]
[LOG]: System started
[LOG]: User logged in
logger1 and logger2 are the same instance.
```

```
int main() {
    Logger* logger1 = Logger::getInstance();
    logger1->log("System started");

    Logger* logger2 = Logger::getInstance();
    logger2->log("User logged in");

    // Check if logger1 and logger2 are the same object
    if (logger1 == logger2) {
        cout << "logger1 and logger2 are the same instance."
              << endl;
    }
}
```

Example: Logger (cleaner)

```
class Logger {
private:
    // Private constructor
    Logger() { cout << "[Logger created]" << endl;}

    // 'delete' the copy and move constructors and
    // the use of assignment
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    Logger(Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) = delete;

    // Static instance pointer:-
    // static Logger* instance;
};
```

<https://onlinegdb.com/sp0wcNBNe>

```
public:
    // create the instance once and
    // reuse it forever
    static Logger* getInstance() {
        static Logger instance;
        return instance;
    }

    void log(const string& message) {
        cout << "[LOG]: " << message
            << endl;
    }
};
// Remember to define the static
// member outside the class
Logger* Logger::instance = nullptr;
```


Example: Logger (cleaner case)

```
[Logger created]
[LOG]: System started
[LOG]: User logged in
logger1 and logger2 are the same instance.
```

```
int main() {
    Logger& logger1 = Logger::getInstance();
    logger1.log("System started");

    Logger& logger2 = Logger::getInstance();
    logger2.log("User logged in");

    // Check if logger1 and logger2 are the same object
    if (&logger1 == &logger2) {
        cout << "logger1 and logger2 are the same instance."<< endl;
    }
}
```

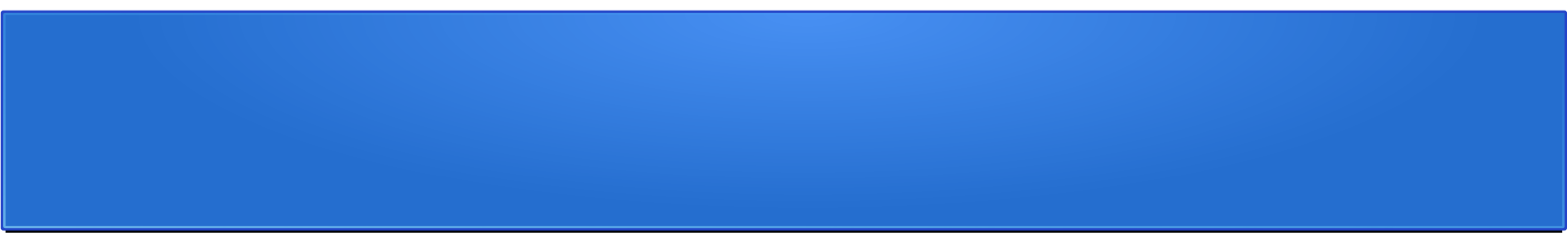
Example: Logger

- Effect achieved:

Prevent external creation	⇒	private constructor
Ensure only one instance	⇒	static (pointer) <code>Logger::instance</code>
Controlled access	⇒	<code>getInstance</code> returns that one object
Safe from accidental copies	⇒	<code>delete</code> (or <code>private</code>) copy constructor & assignment

- Compile-time errors:

```
Logger x = *Logger::getInstance(); // copy
Logger y; // direct constructor
```



Discussions & Questions