

C++

程式語言（二）

Introduction to Programming (II)

Constructors & Destructors

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

Platform/IDE

- Dev-C++



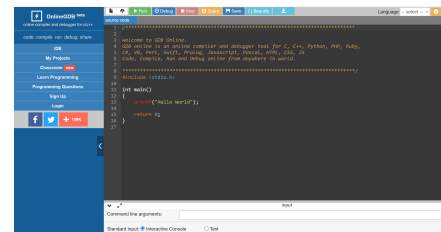
<https://www.pngegg.com/en/search?q=Dev-C>

- Codeblocks

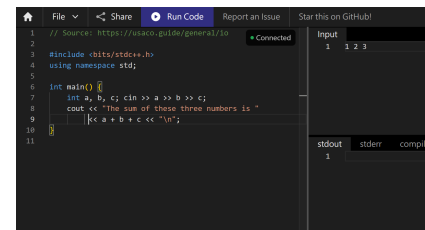


<https://icons8.com/icons/set/code-blocks>

- 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/>



Constructors and Destructors

Constructors

- Each class defines how objects of its type can be **initialized**.
- Classes control object initialization by defining one or more special member functions known as **constructors**.
 - **NO** return type.
- A constructor: **initialize the data members of a class object**.
 - A constructor is run whenever an object of a class type is created.
- It's very useful for setting initial values for certain member variables.

Default Constructors

- The compiler generates a default constructor, called **synthesized default constructor**, automatically only if a class declares no constructors.
- **Note:** for some classes, the synthesized default constructor does the **wrong** thing.

Constructors

Refer to: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

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

class Line {
public:
    void setLength( double len );
    double getLength();
    Line(); // the constructor
private:
    double length;
};
```

```
Line::Line() {
    cout << "Object is being created\n";
}
void Line::setLength(double len) {
    length = len;
}
double Line::getLength() {
    return length;
}
```

```
int main() {
    Line line;

    line.setLength(6.0); // set line length
    cout << "Length of line: " << line.getLength() << endl;

    return 0;
}
```

Object is being created
Length of line : 6

Constructors

Refer to: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

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

class Line {
public:
    void setLength( double len );
    double getLength();
    Line() = default;
    // synthesized default constructor
private:
    double length;
};
```

```
void Line::setLength(double len) {
    length = len;
}
double Line::getLength() {
    return length;
}
```

```
int main() {
    Line line;

    line.setLength(6.0); // set line length
    cout << "Length of line: " << line.getLength() << endl;

    return 0;
}
```

Length of line : 6

Parameterized Constructors

Refer to: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

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

class Line {
public:
    void setLength( double len );
    double getLength();
    Line(double len);
    // constructor with parameters
private:
    double length;
};
```

```
int main() {
    Line line(10.0);

    cout << "Length of line: " << line.getLength() << endl;
    line.setLength(6.0);
    cout << "Length of line: " << line.getLength() << endl;
    return 0;
}
```

```
Line::Line(double len) {
    cout << "Object is being created, "
    << "length = " << len << endl;
    length = len;
}

void Line::setLength(double len) {
    length = len;
}

double Line::getLength() {
    return length;
}
```

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Constructor_INITIALIZER List

Refer to: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

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

class Line {
public:
    void setLength( double len );
    double getLength();
    Line() = default;
    Line(double len): length(len) {};
private:
    double length;
};
```

```
void Line::setLength(double len) {
    length = len;
}
double Line::getLength() {
    return length;
}
```

Constructor_INITIALIZER List

=> **Initialize** the data members, instead of **assigning** their values afterwards

```
int main() {
    Line line1, line2(10.0);
    cout << "Length of line1: " << line1.getLength() <<endl;
    cout << "Length of line2: " << line2.getLength() <<endl;
    line1.setLength(6.0);
    cout << "Length of line1: " << line1.getLength() <<endl;
    return 0;
}
```

```
Length of line1: 4.68426e-310
Length of line2: 10
Length of line1: 6
```

const or references must be initialized

- For example,

<https://onlinegdb.com/7x8SHvUZ1>

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci; // must be initialized
    int &ri; // must be initialized
};

ConstRef::ConstRef(int ii) { // assignment...
    i = ii; // ok
    ci = ii; // error: cannot assign to a const
    ri = i; // error: ri is a reference and was
            // never initialized...
}
```

Destructors (1/3)

- **Destructors** do whatever work is needed to **free** the resources used by an object and **destroy** the **nonstatic data members** of the object.
- The destructor is a member function with the name of the class prefixed by a tilde (~).
- It has **no return value** and takes **no parameters**.
 - Cannot be overloaded.
 - There is always only one destructor for a given class.

```
class Foo {  
public:  
    ~Foo(); // destructor  
    // ...  
};
```

Destructors (2/3)

- A destructor also has a function body and a destruction part.
- In a destructor:
 - The function body is executed first, and then the members are destroyed.
 - Members are destroyed in reverse order from the order in which they were initialized.
- The function body of a destructor does whatever operations the class designer wishes to have executed subsequent to the last use of an object.
 - Typically, the destructor **frees resources** an object allocated during its lifetime.

Destructors (3/3)

- The destruction part is implicit.
 - What happens when a member is destroyed depends on the type of the member.
 - Members of class type are destroyed by running the member's own destructor.
- The built-in types do not have destructors, so nothing is done to destroy members of built-in type.

Destructor Examples

Refer to: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

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

class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line();    // constructor
    ~Line();   // destructor

private:
    double length;
};
```

```
int main() {
    Line line;
    line.setLength(6.0);
    cout << "Length of line : "
          << line.getLength() << endl;
    return 0;
}
```

```
Line::Line(void) {
    cout << "Object is being created"
          << endl;
}
Line::~~Line(void) {
    cout << "Object is being deleted"
          << endl;
}
void Line::setLength(double len) {
    length = len;
}
double Line::getLength(void) {
    return length;
}
```

```
Object is being created
Length of line : 6
Object is being deleted
```


Destructor Examples

Refer to: <https://onlinegdb.com/QK8YB6RBP>

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

class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line();    // constructor
    ~Line() { cout << "An object with length " << length << " is out!" << endl;} ;

private:
    double length;
};
```

```
void Line::setLength(double len) {
    length = len;
}

double Line::getLength(void) {
    return length;
}
```

```
int main() {
    Line line1, line2(10.0);
    cout << "Length of line1: " << line1.getLength() << endl;
    cout << "Length of line2: " << line2.getLength() << endl;
    line1.setLength(6.0);
    cout << "Length of line1: " << line1.getLength() << endl;
    return 0;
}
```

```
Length of line2: 10
Length of line1: 6
An object with length 10 is out!
An object with length 6 is out!
```

Exercise

- Add constructor(s) and a destructor to the following class

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

```
void rectangle::set(unit wd, unit ht)  
{  
    width = wd;  
    height = ht;  
}
```

```
void rectangle::area()  
{  
    cout << "The area: " << width * height << endl;  
}
```

```
int main() // DO NOT modify main()  
{  
    rectangle obj, obj2(2,5); //creating object of rectangle class  
    rectangle::unit x, y;  
    cin >> x;  
    cin >> y;  
    obj.set(x, y);  
    obj.area();  
    obj2.area();  
    return 0;  
}
```

Copy Constructor

- Reference:
 - <https://courses.cs.washington.edu/courses/cse333/12su/lectures/lec11.pdf>
- An Example of "**Person**":
 - <https://onlinegdb.com/8EeWdA3zv>

When and Why make a copy constructor?

- Timing:
 - e.g., variable assignment.
 - `some_class obj2(obj1);`
 - `some_class obj3 = obj1;`
- Why?
 - Assigning all fields (data members) of a class may NOT be what you really want.

Point (An Easy Example)

<https://onlinegdb.com/tBIRDQbrH>

```
class Point {
public:
    double x, y;
    Point(): x(0.0), y(0.0) {
        cout << "default constructor"
              << endl;
    }
    Point(double nx, double ny): x(nx), y(ny) {
        cout << "2-parameter constructor"
              << endl;
    }
};
```

```
int main() {
    Point q(1.0, 2.0);
    // 2-parameter constructor
    Point r = q;
    cout << "(" << r.x << ", "
          << r.y << ")" << endl;
    return 0;
}
```

2-parameter
constructor
(1, 2)

* Using the default copy constructor.

Point (An Easy Example)

<https://onlinegdb.com/Dpr2qSKcK>

```
class Point {
public:
    double x, y;
    Point(): x(0.0), y(0.0) {
        cout << "default constructor"
              << endl;
    }
    Point(double nx, double ny): x(nx), y(ny) {
        cout << "2-parameter constructor"
              << endl;
    }
    Point(Point &o): x(o.x), y(o.y) {
        //x = o.x; y = o.y;
        cout << "custom copy constructor"
          << endl;
    }
};
```

```
int main() {
    Point q(1.0, 2.0);
    // 2-parameter constructor
    Point r = q;
    cout << "(" << r.x << ", "
          << r.y << ")" << endl;
    return 0;
}
```

2-parameter constructor
custom copy constructor
(1, 2)

Copy constructor with a “specific” use

```
class Person {  
    double height;  
    double weight;  
    int ID;  
public:  
    Person() = default;  
    Person(double h, double w, int id): height(h), weight(w), ID(id) {}  
    Person(Person &p) {  
        this->height = p.height;  
        this->weight = p.weight;  
        this->ID = p.ID+1; // not simply "copy", but increase the id  
    }  
    void showInfo();  
    ~Person() {  
        /* do nothing */  
    };  
};
```

Another Example

-from cplusplus.com

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

class Example {
    string* ptr;
public:
    // constructors:
    Example(): ptr(new string) {}
    Example (const string& str): ptr(new string(str)) {}
    // destructor:
    // since we dynamically allocate a string
    ~Example () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

int main () {
    Example foo;
    Example bar ("NTOU CSE IS THE BEST!");
    cout << "bar's content: " << bar.content() << endl;
    return 0;
}
```


More on the Copy Constructor

<https://www.cplusplus.com/doc/tutorial/classes2/>

```
MyClass::MyClass (const MyClass&);
```

If a class has no custom copy nor move constructors (or assignments) defined, an *implicit copy constructor* is provided.

This copy constructor simply performs a copy of its own members.
For example,

```
class MyClass {  
public:  
    int a, b;  
    string c;  
};
```

An implicit copy constructor is automatically defined and is equivalent to

```
MyClass::MyClass(const MyClass& x) :  
    a(x.a), b(x.b), c(x.c) {}
```

When is the copy constructor called?

<https://www.cplusplus.com/doc/tutorial/classes2/>

```
MyClass foo;  
MyClass bar {foo};           // object initialization: copy constructor called  
MyClass baz = foo;           // object initialization: copy constructor called  
foo = bar;                   // object already initialized: copy assignment called
```

```
MyClass& operator= (const MyClass& x) {  
    delete ptr;  
    ptr = new string (x.content());  
    return *this;  
}
```

Operator overloaded
(We will discuss about it in the future.)

Another Example (Destructor + Copy Constructor)

-from cplusplus.com

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

class Example {
    string* ptr;
public:
    // constructors:
    Example(): ptr(new string) {}
    Example (const string& str): ptr(new string(str)) {}
    Example (const Example& x): ptr(new string(x.content())) {}
    // destructor:
    ~Example () {delete ptr;}
    // access content:
    const string& content() const {return *ptr;}
};

void main () {
    Example foo("NTOU CSE IS THE BEST!");
    Example bar = foo;

    cout << "bar's content: " << bar.content() << '\n';
}
```

*Move Constructor

<https://www.cplusplus.com/doc/tutorial/classes2/>

```
MyClass (MyClass&&);           // move-constructor  
MyClass& operator= (MyClass&&); // move-assignment
```

- Similar to copying, moving also uses the value of an object to set the value to another object.
- But, unlike copying, the content is actually transferred from one object (the source) to the other (the destination):
 - **The source loses that content**, which is taken over by the destination.
 - This moving only happens when the source of the value is an *unnamed* object.

```
MyClass fn();           // function returning a MyClass object  
MyClass foo;           // default constructor  
MyClass bar = foo;      // copy constructor  
MyClass baz = fn();    // move constructor  
foo = bar;              // copy assignment  
baz = MyClass();        // move assignment
```



Notes on the "reference" &

Example

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

int main () {
    int i, j;
    int &ref1 {i};
    int &ref2 {ref1};

    cin >> i;
    cout << "i = " << i << endl;
    cout << "&i = " << &i << endl;
    cout << "&j = " << &j << endl;
    cout << "ref1 = " << ref1 << endl;
    cout << "&ref1 = " << &ref1 << endl;
    cout << "ref2 = " << ref2 << endl;
    cout << "&ref2 = " << &ref2 << endl;

    ref1 = 8;
    cout << "i = " << i << endl;
    j = ref2 + 3;
    cout << "j = " << j << endl;
    return 0;
}
```

```
5
i = 5
&i = 0x7ffff9ff72c0
&j = 0x7ffff9ff72c4
ref1 = 5
&ref1 = 0x7ffff9ff72c0
ref2 = 5
&ref2 = 0x7ffff9ff72c0
i = 8
j = 11
```

<https://onlinegdb.com/l3PeEt-0jT>

Example (Loop)

```
#include <iostream>
using namespace std;
#define N 5

int main () {
    int nums[N];
    int sum{0}; //initialization
    for (int &v: nums)
        cin >> v;
    for (int &v: nums)
        sum += v;
    cout << "sum = " << sum << endl;
    return 0;
}
```

```
1 2 3 4 5
sum = 15
```

Example (Loop)

```
#include <iostream>
using namespace std;
#define N 5

int main () {
    int nums[N];
    int sum{0}; //initialization
    for (int &v: nums)
        cin >> v;
    for (int v: nums)
        sum += v;
    cout << "sum = " << sum << endl;
    return 0;
}
```

```
1 2 3 4 5
sum = 15
```



Do we need the '&' here?



Do we need the '&' here?

Pass by Reference

<https://github.com/pangfengliu/Cplusplus-refactor/blob/main/structure/rational-ref.cc>

```
struct Rational {
    int q, p; // q/p
};
int gcd(int a, int b) {
    if (a % b == 0)
        return b;
    else
        return gcd(b, a % b);
}
void simplify(Rational &a) {
    int factor {
        gcd(abs(a.p), abs(a.q))
    };
    a.p /= factor;
    a.q /= factor;
    if (a.p < 0) {
        a.p = -a.p;
        a.q = -a.q;
    }
}
```

```
Rational add(Rational &a, Rational &b) {
    Rational sum {b.p * a.q + b.q * a.p, b.p * a.p};
    simplify(sum);
    return sum;
}
```

```
void print(Rational &a) {
    cout << a.q << '/' << a.p << endl;
}
```

```
int main() {
    Rational a, b;
    cin >> a.q >> a.p >> b.q >> b.p;
    Rational c {add(a,b)};
    print(c);
    return 0;
}
```

2 3 1 6
5/6