C++

程式語言（二）

Introduction to Programming (II)

Operator Overloading

Joseph Chuang-Chieh Lin

Dept. CSE, NTOU

Platform/IDE

- Dev-C++

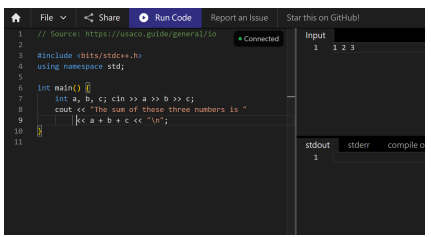
https://www.pngegg.com/en/search?q=Dev-C

- 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/

Operator Overloading

Operator Overloading

- Similar to function overloading.
- Give similar meaning to an existing operator.
  - Not all operators can be overloaded.
- Compile-time polymorphism.
- Operations for class objects.

Operators for built-in types

```
class Vect {
    int n1, n2;
public:
    Vect(int a, int b):
        n1(a), n2(b) {}
    ~Vect() = default;
};
```

```
int main() {
    int x = 5, y = 3, z;
    z = x + y ; // using = and +
    int a[10], b[10], c[10];
    c = a + b ; // is this valid?
    Vect f1(3, 5), f2(2, 5), f3;
    f3 = f1 + f2;
    // (3,5)+(2,5)=(5,10) is it valid?
    return 0;
}
```

Operators for built-in types

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
};
```

```
int main() {
    int x = 5, y = 3, z;
    z = x + y ; // using = and +
    int a[10], b[10], c[10];
    c = a + b ; // is this valid?
    Vect f1(3, 5), f2(2, 5), f3;
    f3 = f1 + f2;
    // (3,5)+(2,5)=(5,10) is it valid?
    return 0;
}
```

We need to design functions for such operations...

Example (Binary Operator)

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
};
```

```
int main() {
    Vect o1(1,2), o2(3,4), o3;
    o3 = o1 + o2; // o3:(4,6)
    o3.print() ;
    //later, we will try: cout << o3 ;
    return 0;
}
```

Example (Binary Operator)

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
    void set(Vect r);
    Vect add(Vect r);
};
```

```
int main() {
    Vect o1(1,2), o2(3,4), o3;
    o3.set(o1.add(o2)); // o3:(4,6)
    o3.print() ;
    //later, we will try: cout << o3;
    return 0;
}
```

Exercise

Example (Binary Operator)

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
    void set(Vect r);
    Vect add(Vect r);
};
```

```
Vect Vect::add(Vect r) {
    Vect temp;
    temp.x = x + r.x;
    temp.y = y + r.y;
    return temp;
}
void Vect::set(Vect r) {
    x = r.x;
    y = r.y;
}
void Vect::print() {
    cout << x << "/" << y << endl;
}
```

Example (Binary Operator)

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
    void operator=(Vect r);
    Vect operator+(Vect r);
};
```

```
int main() {
    Vect o1(1,2), o2(3,4), o3;
    o3 = o1 + o2; // o3:(4,6)
    o3.print() ;
    return 0;
}
```

```
Vect Vect::operator+(Vect r) {
    Vect temp;
    temp.x = x + r.x;
    temp.y = y + r.y;
    return temp;
}
void Vect::operator=(Vect r) {
    x = r.x;
    y = r.y;
}
void Vect::print() {
    cout << x << "/" << y << endl;
}
```

Step-by-step

```
o3 = o1 + o2;
```

```
o3 = o1.operator+(o2);
```

```
o3.operator=(o1.operator+(o2));
```

Another safer way: **Call by Reference**

```
class Vect {
    int x, y;
public:
    Vect(int a, int b):
        x(a), y(b) {}
    ~Vect() = default;
    void operator=(const Vect& r);
    Vect operator+(const Vect& r);
};
```

```
Vect Vect::operator+(const Vect& r) {
    Vect temp;
    temp.x = x + r.x;
    temp.y = y + r.y;
    return temp;
}
void Vect::operator=(const Vect& r) {
    x = r.x;
    y = r.y;
}
void Vect::print() {
    cout << "(" << x << ", " << y
         << ")" << endl;
}
```

Consecutive Additions/Assignments

```
Vect Vect::operator+(const Vect& r) {
    Vect temp;
    temp.x = x + r.x;
    temp.y = y + r.y;
    return temp;
}
```

```
o1 + o2 + o3;
o1.operator+((o2.operator+(o3)));
```

Vect

```
void Frac::operator=(const Vect& r) {
    x = r.x;
    y = r.y;
    return *this;
}
```

```
o1 = o2 = o3;
o1.operator=((o2.operator=(o3)));
```

Relational Operators

```
bool Vect::operator==(const Vect& r) {
    return (this->x == r.x) && (this->y == r.y);
}
```

Usage:

```
int main() {
    Vect o1(1,2), o2(3,4);
    if (o1==o2)
        cout << "equal" << endl;
    else
        cout << "unequal: << endl;
    return 0;
}
```

Exercise

```
bool Vect::operator>(const Vect& r) {
    /* complete the function body */
}
```

Usage:

```
int main() {
    Vect o1(1,2), o2(3,4);
    if (o1 > o2)
        cout << "larger" << endl;
    else
        cout << "not larger" << endl;
    return 0;
}
```

Unary Operators

```
Vect Vect::operator++() {
    x++; y++;
    return *this;
}

Vect Vect::operator++(int) {
    Vect temp = *this;
    x++;
    y++;
    return temp;
}
```

no extra meaning;
only for distinguishing prefix and postfix

$$(n, m) \Rightarrow (n+1, m+1)$$

```
// requirement
int main() {
    Vect o1(1,2), o2, o3;
    o2 = ++o1; // prefix
    o3 = o1++; // postfix
    return 0;
}
```

A scenario of using friend functions

```
int main() {
    Vect o1(1,2), o2;
    o2 = 7 + o1; // is it "7.operator+(o1)"??
    return 0;
}
```

Here’s the catch

- For `o1 + 7`:
  - `o1` is a `Vect`, so its member `operator+(const Vect&)` is a candidate.
  - The literal `7` is implicitly converted to `Vect(7)`.
  - So it compiles and calls `o1.operator+(Vect(7))`.
- For 11 + o2:
  - The left operand is an `int`, so the compiler does not look for `Vect::operator+`.
  - Member functions can only be called on objects of their own class (or derived).
  - No user-defined conversion is applied to turn the `int` into a `Vect` before selecting member operators.
  - The compiler falls back to searching for free (non-member) `operator+` functions in scope that can take (`int`, `Vect`) or, via conversions, (`Vect`, `Vect`).
  - Without your friend overload, there is no such free function, so `7 + o2` is invalid.

Using friend functions

```
class Vect {
    int x, y;
public:
    ...
    Vect(int a){ x = a; y = a; }
    friend Vect operator+(Vect obj1, Vect obj2);
};
Vect operator+(Vect obj1, Vect obj2) {
    return Vect(obj1.x+obj2.x, obj1.y+obj2.y);
}
```

It resembles the broadcasting in Python.

```
int main() { // Try it!
    Vect o1(1,2), o2;
    o2 = 7 + o1;
    return 0;
}
```

Try it by yourself!

Overloading >> and <<

<<

- Left operand must be type of `ostream &`
- For example,

```
cout << obj1;
```

>>

- Left operand must be type of `istream &`
- For example,

```
cin >> obj1;
```

Note:

We cannot overload them as a member function of a class.

- Use "`friend`".

Overloading >> and <<

- Prototype (inside the class definition):

```
friend ostream& operator<<(ostream&, const someClass&);
friend istream& operator>>(istream&, someClass&);
```

- Function definition (outside the class):

```
ostream& operator<<(ostream& output, const someClass& obj) {
    output << obj.data << ...
    return output;
}
```

```
istream& operator>>(ostream& input, someClass& obj) {
    input >> obj.data >> ...
    return input;
}
```

Exercise

```
class Vect {
    int x, y;
public:
    Vect() = default;
    ~Vect() = default;
    friend ostream& operator<<(ostream& os, const Vect& r);
    friend istream& operator>>(istream& is, Vect& r);
};
```

```
int main() { //sample main()
    Vect obj;
    cin >> obj;
    cout << obj;
    return 0;
}
```

```
ostream& operator<<(ostream& os, const Vect& r) {
    ... // complete it
}

istream& operator<<(istream& is, Vect& r) {
    ... // complete it
}
```

Overall Wrap-up

- https://onlinegdb.com/g4l6Inzyh

Appendix or Further Resources

Revisit `sales_item.h` in C++ Primer:

https://github.com/amidvidy/learning/blob/master/cpp-primer/Sales_item.h

Arithmetic operators (can be overloaded)

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operator name		Syntax	C++ prototype examples	
			As member of K	Outside class definitions
Addition		`a + b`	`R K::operator +(S b);`	`R operator +(K a, S b);`
Subtraction		`a - b`	`R K::operator -(S b);`	`R operator -(K a, S b);`
Unary plus (integer promotion)		`+a`	`R K::operator +();`	`R operator +(K a);`
Unary minus (additive inverse)		`-a`	`R K::operator -();`	`R operator -(K a);`
Multiplication		`a * b`	`R K::operator *(S b);`	`R operator *(K a, S b);`
Division		`a / b`	`R K::operator /(S b);`	`R operator /(K a, S b);`
Modulo (integer remainder)[a]		`a % b`	`R K::operator %(S b);`	`R operator %(K a, S b);`
Increment	Prefix	`++a`	`R& K::operator ++();`	`R& operator ++(K& a);`
	Postfix	`a++`	`R K::operator ++(int);`	`R operator ++(K& a, int);`
			Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and postfix increment operators.	
Decrement	Prefix	`--a`	`R& K::operator --();`	`R& operator --(K& a);`
	Postfix	`a--`	`R K::operator --(int);`	`R operator --(K& a, int);`
			Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and postfix decrement operators.	

Relational operators (can be overloaded)

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operator name	Syntax	Included in C	Prototype examples	
			As member of K	Outside class definitions
Equal to	`a == b`	Yes	`bool K::operator ==(S const& b) const;`	`bool operator ==(K const& a, S const& b);`
Not equal to	`a != b` `a not_eq b` [b]	Yes	`bool K::operator !=(S const& b) const;`	`bool operator !=(K const& a, S const& b);`
Greater than	`a > b`	Yes	`bool K::operator >(S const& b) const;`	`bool operator >(K const& a, S const& b);`
Less than	`a < b`	Yes	`bool K::operator <(S const& b) const;`	`bool operator <(K const& a, S const& b);`
Greater than or equal to	`a >= b`	Yes	`bool K::operator >=(S const& b) const;`	`bool operator >=(K const& a, S const& b);`
Less than or equal to	`a <= b`	Yes	`bool K::operator <=(S const& b) const;`	`bool operator <=(K const& a, S const& b);`
Three-way comparison[c]	`a <=> b`	No	`auto K::operator <=>(const S &b);`	`auto operator <=>(const K &a, const S &b);`
			The operator has a total of 3 possible return types: `std::weak_ordering`, `std::strong_ordering` and `std::partial_ordering` to which they all are convertible to.	

Logical operators (can be overloaded)

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

All logical operators exist in C and C++ and can be overloaded in C++, albeit the overloading of the logical AND and logical OR is discouraged, because as overloaded operators they behave as ordinary function calls, which means that *both* of their operands are evaluated, so they lose their well-used and expected short-circuit evaluation property.[2]

Operator name	Syntax	C++ prototype examples	
		As member of K	Outside class definitions
Logical negation (NOT)	`!a` `not a` [b]	`bool K::operator !();`	`bool operator !(K a);`
Logical AND	`a && b` `a and b` [b]	`bool K::operator &&(S b);`	`bool operator &&(K a, S b);`
Logical OR	`a || b` `a ??!??! b` [d][e] `a or b` [b]	`bool K::operator ||(S b);`	`bool operator ||(K a, S b);`

Bitwise operators (can be overloaded)

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operator name	Syntax	Prototype examples	
		As member of K	Outside class definitions
Bitwise NOT	`~a` `??-a` [d] `compl a` [b]	`R K::operator ~();`	`R operator ~(K a);`
Bitwise AND	`a & b` `a bitand b` [b]	`R K::operator &(S b);`	`R operator &(K a, S b);`
Bitwise OR	`a | b` `a ??! b` [d] `a bitor b` [b]	`R K::operator |(S b);`	`R operator |(K a, S b);`
Bitwise XOR	`a ^ b` `a ??' b` [d] `a xor b` [b]	`R K::operator ^(S b);`	`R operator ^(K a, S b);`
Bitwise left shift[f]	`a << b`	`R K::operator <<(S b);`	`R operator <<(K a, S b);`
Bitwise right shift[f][g]	`a >> b`	`R K::operator >>(S b);`	`R operator >>(K a, S b);`

Assignment operators (can be overloaded)

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operator name	Syntax	C++ prototype examples	
		As member of K	Outside class definitions
Direct assignment	`a = b`	`R& K::operator =(S b);`	—
Addition assignment	`a += b`	`R& K::operator +=(S b);`	`R& operator +=(K& a, S b);`
Subtraction assignment	`a -= b`	`R& K::operator -=(S b);`	`R& operator -=(K& a, S b);`
Multiplication assignment	`a *= b`	`R& K::operator *=(S b);`	`R& operator *=(K& a, S b);`
Division assignment	`a /= b`	`R& K::operator /=(S b);`	`R& operator /=(K& a, S b);`
Modulo assignment	`a %= b`	`R& K::operator %=(S b);`	`R& operator %=(K& a, S b);`
Bitwise AND assignment	`a &= b` `a and_eq b` [b]	`R& K::operator &=(S b);`	`R& operator &=(K& a, S b);`
Bitwise OR assignment	`a |= b` `a ??!= b` [d] `a or_eq b` [b]	`R& K::operator |=(S b);`	`R& operator |=(K& a, S b);`
Bitwise XOR assignment	`a ^= b` `a ??'= b` [d] `a xor_eq b` [b]	`R& K::operator ^=(S b);`	`R& operator ^=(K& a, S b);`
Bitwise left shift assignment	`a <<= b`	`R& K::operator <<=(S b);`	`R& operator <<=(K& a, S b);`
Bitwise right shift assignment[g]	`a >>= b`	`R& K::operator >>=(S b);`	`R& operator >>=(K& a, S b);`

Member and pointer operators

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Operator name	Syntax	Can overload in C++	Included in C	C++ prototype examples	
				As member of K	Outside class definitions

Subscript	`a[b]`
	`a<:b:>`
	`a??(b??)` [d][h]
		Yes	Yes	`R& K::operator [](S b);`
				`R& K::operator [](S b, ...); // since C++23`
					—

Indirection ("object pointed to by *a*")	`*a`	Yes	Yes	`R& K::operator *();`	`R& operator *(K a);`
Address-of ("address of *a*")	`&a` `bitand a` [b][i]	Yes[j]	Yes	`R* K::operator &();`	`R* operator &(K a);`
Structure dereference ("member *b* of object pointed to by *a*")	`a->b`	Yes	Yes	`R* K::operator ->();` [k]	—
Structure reference ("member *b* of object *a*")	`a.b`	No	Yes	—	
Member selected by pointer-to-member *b* of object pointed to by *a*[l]	`a->*b`	Yes	No	`R& K::operator ->*(S b);`	`R& operator ->*(K a, S b);`
Member of object *a* selected by pointer-to-member *b*	`a.*b`	No	No	—	

Exercise: Fractional Arithmetic Operations

- Use friend functions to overload arithmetic operations of fractional numbers.
- Define a class `Fractional` with **private** members

```
int numerator, denominator;
```

- Provide a constructor

```
Fractional(int r = 0, int i = 0).
```

- Declare these friend functions inside `Fractional`:

```
Fractional operator+(const Fractional&, const Fractional&);
Fractional operator-(const Fractional&, const Fractional&);
Fractional operator*(const Fractional&, const Fractional&);
Fractional operator/(const Fractional&, const Fractional&);
```

- `void printFrac(const Fractional&);` // print out a fractional number

Exercise: Fractional Arithmetic Operations

Sample input:

```
2 5 + 8 5
3 4 - 5 2
7 3 / 6 9
1 2 * 2 3
```

Sample output:

```
2
-7/4
7/2
1/3
```

Note: You MUST use the friend functions (i.e., the operator overloading) to implement the fractional number arithmetic operations.

Discussions & Questions