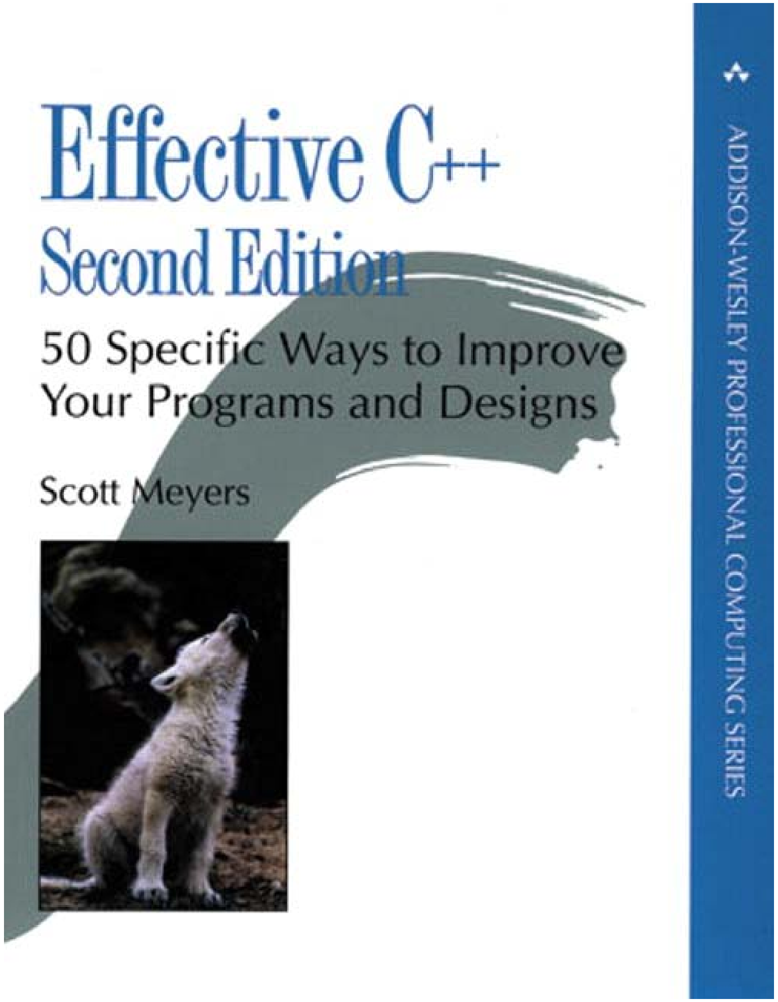
Advanced C++

Effective C++:
50 Specific Ways to Improve Your Programs and Designs,
Scott Meyers,
2nd Edition,
Addison Wesley,
ISBN: 0-201-92488-9.



1. “const” or “#define”

- Use "***const***" and "***inline***" instead of "***#define***".
- Prefer the ***compiler*** to the ***preprocessor***.
- Symbolic name is never seen by the compiler (not in the symbol table), this can be confusing if you get an error.
- Solution is to define a "***const***".
- Pointer better be declared "***const***", in addition to what the pointer points to.

"inline“ or “#define” (2)

- Drawbacks of macro definition.

```
#define MAX(a, b) ( (a) > (b) ? (a) : (b) )
int a = 1, b = 0;

MAX(++a, b);         // a is incremented twice
MAX(++a, b+10);      // a is incremented once
MAX(a, "Hello");     // comparing int and ptr
```

"inline“ or “#define” (3)

- ***inline int MAX(int a, int b)***
  { return a > b ? a : b; }

  or even better

- ***template<class T>***
  inline T& MAX(T& a, T& b)
  { return a > b ? a : b; }

"inline“ or “#define” (4)

- ***#define GENERATE_MAX(T) ***
  ***inline T& MAX(T& a, T& b) ***
  { return a > b ? a : b; }

- ***GENERATE_MAX(int);***
  // generate MAX for ints
- ***GENERATE_MAX(double);***
  // generate MAX for doubles

2. "iostream.h" or "stdio.h"

- Prefer "iostream.h" to "stdio.h".
- Type safety and type extensibility.

- ***friend ostream&***
  operator<<(ostream& s,
  const class-type &x);

3. “new” or “malloc”

- Use "new" and "delete" instead of "malloc" and "free".
- "malloc" and "free" know nothing about constructors and destructors.
- Combining "new" and "delete" with "malloc" and "free" is a bad idea.

4. Prefer C++-Style Comments

- ***/* int a;*** /* declare a */

  int b;

  ****/***

#define PI 3.1416 **//** defined constant

- Given a preprocessor unfamiliar with C++, the comment at the end of the line becomes part of the macro!

Memory Management

- The biggest headaches – potential memory leaks.

5. Calls to “new” and “delete”

- Use the same form in corresponding calls to "new" and "delete".
- What is wrong with this picture?

class String { char *data; .....}

String *stringArray = new String[100];

delete stringArray;

- 99 of the 100 String objects are unlikely properly destroyed, because their destructors will probably never be called.

Calls to “new” and “delete” (2)

- If you do not use brackets in your call to "delete", "delete“ assumes that a single object is pointed.

Calls to “new” and “delete” (3)

typedef string AddressLines[4];

string *pa1 = new AddressLines;

- Must be matched with the array form of delete:

```
delete pa1;          // undefined!
delete [ ] pa1;      // fine
```

- Better off define Addresslines to be vector<string>.

6. Call "delete" on Pointer Members

- Forget to initialize a pointer in a constructor, or forget to handle it inside the assignment operator, the problem usually becomes apparent fairly quickly – not too worry.

- Failing to delete the pointer in the destructor, however, often exhibits no obvious external symptoms – a big concern.

Call "delete" on Pointer Members (2)

- Deleting a null pointer is always safe.

- Not to call "delete" on a pointer that was never initialized with a call to "new", and "almost" never delete a pointer that was passed to you in the first place.

Call "delete" on Pointer Members (3)

- In other words, your class destructor usually should not be using ***delete*** unless your class members were the ones who used ***new*** in the first place.

7. Check with Return Value of "new"

- ***#define NEW(PTR, TYPE) { ***
```
#define NEW(PTR, TYPE) { \
    (PTR) = new TYPE;     \
    assert ((PTR)!= 0); }
```

- However, there are other calling forms.
```
new T;
new T(constructor's arguments);
new T[size];
```

Check with Return Value of "new“ (2)

- In <new.h>,

  extern

  void (*set_new_handler (void (*) ())) ();
- A function that takes one argument and returns one result.
- Both the argument and the result are themselves pointers to functions, each of which takes no arguments and returns nothing.

Check with Return Value of "new“ (3)

```
// function to call if "new" cannot allocate enough memory
void noMoreMemory() {
    cerr << "Unable to satisfy request for memory“
          << endl;
    abort();
}

main() {
    set_new_handler(noMoreMemory);
    char *bigString = new char[100000000];
}
```

Constructors, Destructors, and Assignment Operators

- Control the fundamental operations of bringing a new object into existence and making sure it is initialized;

- getting rid of an object and making sure it has been properly cleaned up; and

- giving an object a new value.

11. Copy Constructor and Assignment Operator

- ***String*** is a class with dynamically allocated memory.
- ***String a("Hello");*** // declare and construct *a*

  { // open new scope

  String b("World"); // declare and construct *b*

  b = a; // execute default op=,

  // lose *b*'s memory

  } // close scope, call *b*'s destructor

  String c = a; // *c.data* is undefined

  // *a.data* is already deleted

Copy Constructor and Assignment Operator (2)

- ***void doNothing(String localString) {}***
  String s = "Goodbye";
  doNothing(s);
- Default copy constructor makes *localString* have a copy of the pointer that is inside *s*.
- When *localString* goes out of scope, its destructor is called.
- *s* contains a pointer to memory that has already been deleted.

Copy Constructor and Assignment Operator (3)

- Even if *s* is never used again, there could be a problem when it goes out of scope.
- The solution is to write your own "**copy constructor**" and the "**assignment operator**" (copying actual content) if you have any pointers in your class.

  ```
  char *copy= new char[strlen(data) + 1];
  strcpy(copy, data);
  return copy;
  ```

- Safer, slower, callers must remember to use delete on this returned pointer.

12. Initialization or Assignment

- Prefer initialization to assignment in constructors.
- ***class NameData {***

```
class NameData {
        String name;
        void *data;

    public:
        NameData(const String& initName,
                        void *dataPtr);
}
```

Initialization or Assignment (2)

- Which one of the following is better?

1. ***NameData::NamaData(***

   const String& initName, void *dataPtr)

   : name(initName), data(dataPtr) {}

   // Use the the member initialization list.

2. ***NameData::NamaData(***

   const String& initName, void *dataPtr)

   { name = initName; data = dataPtr; }

   // Make assignments in the constructor body.

Initialization or Assignment (3)

- If

  const String name; // or ***String& name;***

  void * const data;

  // const and reference members

  // can only be initialized,

  // never assigned.

Initialization or Assignment (4)

- Efficiency consideration for the original class, the one contains no const or reference members.

1. Assignment inside the construction, two calls to *String* member functions: the "default constructor" and one more for the "assignment".
2. Member initialization: only a single function call, the "copy constructor".

Initialization or Assignment (5)

- In other words, initialization via member initialization list is always legal, is never less efficient than assignment inside the body of the constructor, and is often more efficient.
- The exception is when you have a large number of data members of built-in types, and you want them all initialized the same way in each constructors.

13. Order of Member Initialization

- List members in an initialization list in the order in which they are declared.
- ***class Array {***

```
    int *data;                 // ptr to actual array data
    unsigned size;             // # of elements in array
    int lBound, hBound;        // lower bound, higher bound
public:
    Array(int lowBound, int highBound)
    : size(highBound – lowBound + 1),
          lBound(lowBound), hBound(highBound),
          data(new int(size)) {}              }
```

Order of Member Initialization (2)

- Regardless of what "*new*" returns, you have absolutely no idea how much memory "*data*" points to.

- Class members are initialized in the order of their declarations in the class; the "order of members in an initialization list" is ignored.

Order of Member Initialization (3)

- Base class data members are initialized before derived class data members, so if you are using inheritance, you should list base class initializers at the very beginning of your member initialization lists.

14. Virtual Destructors

- Sometimes it is convenient for a class to keep track of how many objects of its type exist. The straightforward way to do this is to create a static class member for counting the objects.

- Delete a *derived* class object through a *base* class pointer and the base class has a nonvirtual destructor, the results are **undefined**.
- Make destructors virtual in base classes.

Virtual Destructors (2)

- By declaring the destructor virtual in the base class, you tell the compiler that it must examine the object being deleted to see where to start calling destructors.

```
class Array{
        int *data;
public:
        ~Array();
};
```

Virtual Destructors (3)

```
class NamedArray : public Array {
    const char * const arrayName;
public:
    ~NamedArray();
};

NamedArray *pna = new NamedArray(.....);
Array *pa = pna;                 // NamedArray* -> Array*
delete pa;     // NamedArray destructor will never be called.
          // arrayName memory will never be deallocated.
```

Virtual Destructors (4)

- However, when a class is not intended to be used a base class, making the destructor virtual is usually a bad idea.

```
class Point {
    int x, y;
};
```

- If the "*Point*" class contains a virtual function, objects of that type will be implicitly larger in size, from two 32-bit ints to two 32-bit ints plus 32-bit vtpr (virtual table pointer).

Virtual Destructors (5)

- If a class does not contain any virtual functions, that is often an indication that it is not meant to be used as a **base** class.

- A good rule: declare a virtual destructor in a class if and only if that class contains at least one virtual function.

15. Return of Operator=

- Have operator= return a reference to *this.
- Chain assignment together like:

  w = x = y = z = 0;

  w = x = y = z = "Hello";

or

  w = (x = (y = (z = "Hello")));

Return of Operator= (2)

- The return type of operator= must also be acceptable as an input to itself.
- Which of the following is correct?

```
String& operator=(const String& rhs) {
    .....
    return *this;
            // return reference to left-hand object.
    return rhs;
            // return reference to right-hand object.
}
```

Return of Operator= (3)

- The version returning "*rhs*" will not compile.
- That is because rhs is a reference-to-**const-String**, but operator= returns a reference-to-**String**.

1. 避免a3 = (a1 = a2);
   cannot convert from 'const A' to 'A &'
2. 希望能夠寫 (a1 = a2) = a3;

- Easy, re-declare operator= like this:

  String& String::operator=(String& rhs) {…}

- ***x = "Hello"*** is a char array, not a **String**.

Return of Operator= (4)

- ***x = "Hello";*** // same as ***x.op=("Hello");***

  is equivalent to

  String temp("Hello"); // create temporary

  x = temp; // pass temporary to op=

- What is the life-span of the temporary that compiler generated?
- The temporary object is const.

Return of Operator= (5)

- Prevents you from accidentally passing a **temporary** into a function that modifies its parameter (i.e., this **temporary**).

- If that were allowed, only the compiler-generated **temporary** was modified, not the argument they actually provided at the call site.

16. Assignment using Operator=

- Assign to all data members in operator=.
- Can you let C++ generate a default assignment operator and let you selectively override those parts you do not like? No such luck.
- If you want to take control of any part of the assignment process, you must do the entire thing yourself.

Assignment using Operator= (2)

- Assignment operators must be updated if new data members are added to the class.

- Examine the following:

```
class A {
    int x;
public:
    A(int i) : x(i) {}
};
```

Assignment using Operator= (3)

- ***class B : public A {***

```
class B : public A {
        int y;
public:
        B(int i) : A(i), y(i) {}        // okay
}
```

- Erroneous assignment operator:

```
B& B::operator=(const B& rhs) {
        if (this == &rhs) return *this;
        y = rhs.y;      // x is unaffected by this assignment
        return *this;
}
```

Assignment using Operator= (4)

- ***main() {***

  B b0(0); // *b0.x* = 0, *b0.y* = 0.

  B b1(1); // *b1.x* = 1, *b1.y* = 1.

  b0 = b1; // *b0.x* = 0, *b0.y* = 1.

  }

Assignment using Operator= (5)

- Correct assignment operator should add

  ((A&) *this) = rhs;

  // call operator= on *A* part of *this.

  // reference must be a reference to an *A*

  or

  A& A::operator=(const A& rhs); // A's assign op

  and

  A::operator=(rhs); // call this->*A*::operator=

17. Assigning to Self

- Check for assignment to self in operator=.

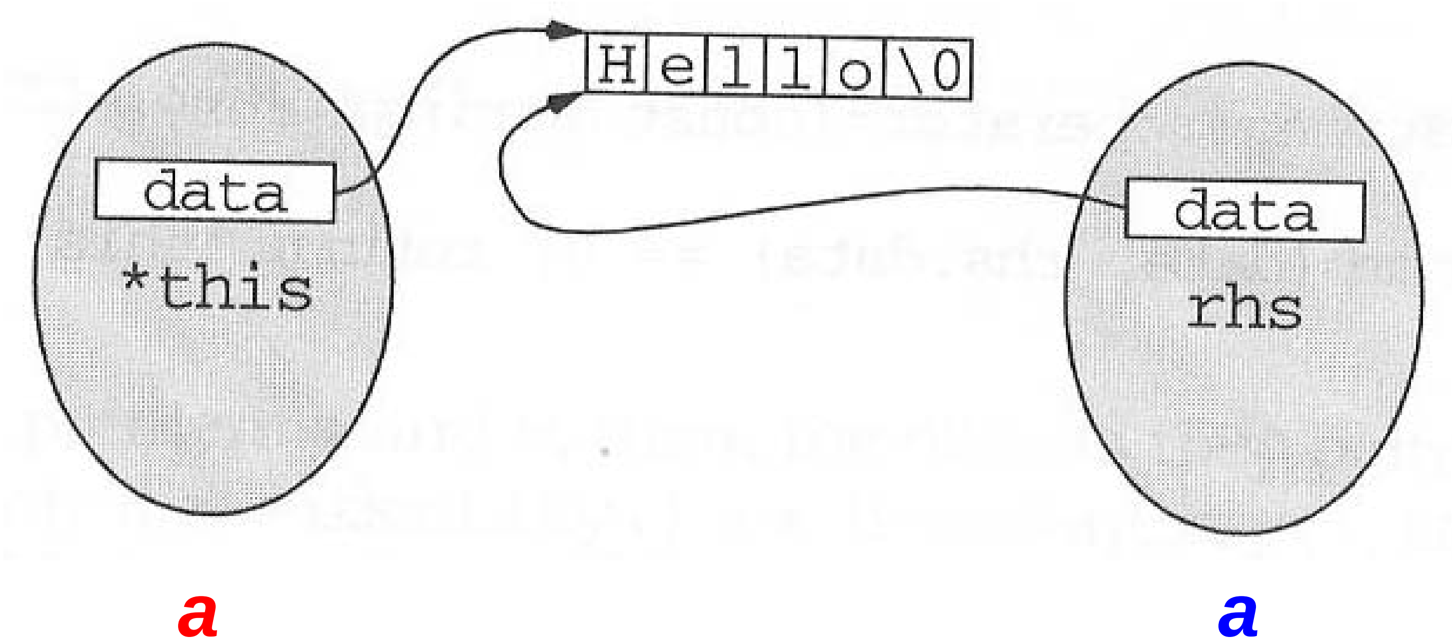
```
X a;
X &b = a;
a = a;
a = b;
```

- **Efficiency**.
- **Correctness**: free the resources allocated to an object before it can allocate the new resources corresponding to its new value.

Assigning to Self (2)

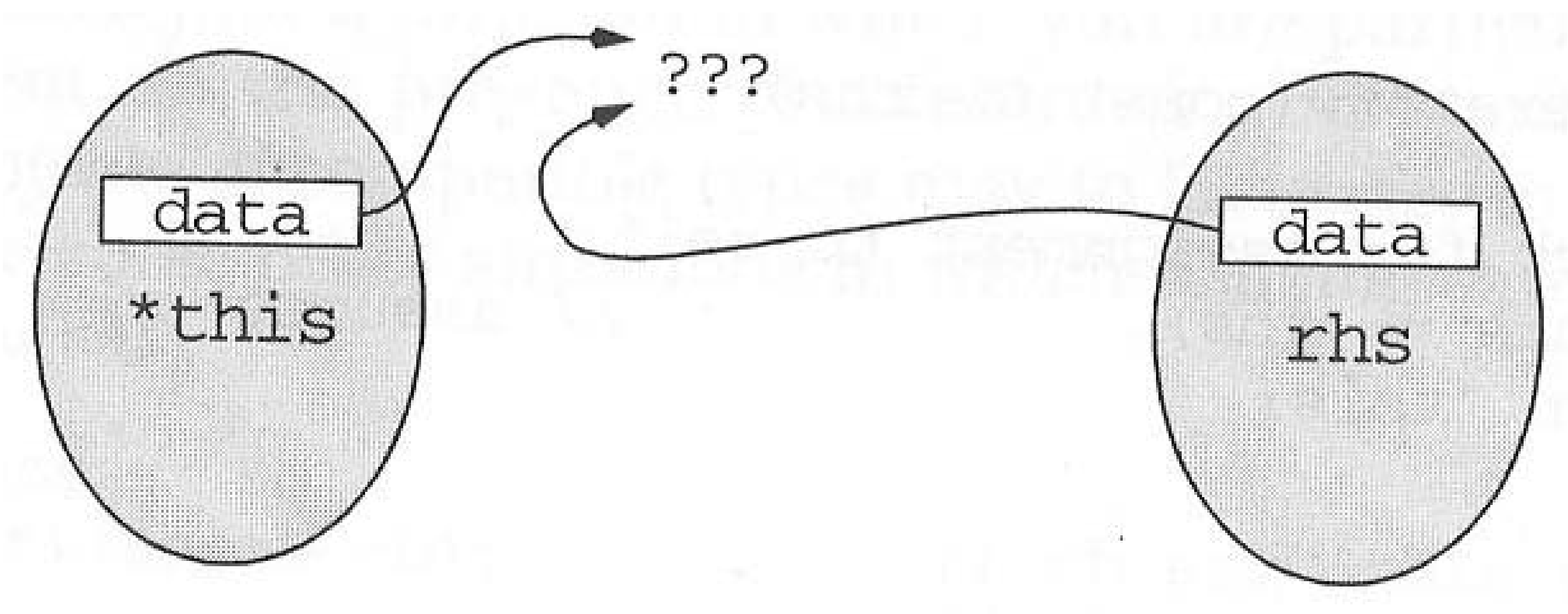
String a = “Hello”;

a = a;



Assigning to Self (3)

- Without check for assignment to self, the first thing assignment operator does is use ***delete*** on ***data***, and the result is the following state of affairs.



Assigning to Self (4)

```
class A {};
class B : public A {};
class C : public A {};
class D : public B, public C {};

D d;
D* PD1 = &d;
B* PD2 = &d;
C* PD3 = &d;
A* PD4 = (B*) &d;
A* PD5 = (C*) &d;
```