Efficient C++

Contents

1 Accustoming yourself to C++

1.1 Prefer `const, enum, inline` to `#define`

- `#define` will be confusing if you get an error during complication.
- Use of `constant` yield smaller code than using a `#define`.

When replacing `#defines` with constants, there are two spectial cases:

- Defining constant pointers": `const char* const authorName = "xxx''`
- Concerning class-specific constants. To limit the scope od a constant to a class, you must make it a member, to ensure there is at most one copy of the constant, you must make it a `static` member.

  ```
  class Game{
  private:
    static const int Nums = 5;
    int scores[Nums];
    ...
  };
  ```

  If it is an integral type (integers, chars, bools), we do not need provide a definition. Otherwise we need a separate definition in **implementation file**:

  ```
  const int Game::Nums; // definition of Nums, no initial value here.
  ```

  There is no way to carete a class-specific constant using a `#define`.
- In another way, we can use "enum hack":

  ```
  class Game{
  private:
    enum {Nums = 5};
    int scores[Nums];
    ...
  };
  ```

  - The enum hack behaves in some ways more like a `#define`, for example, it's legal to take the address of a const but not to an enum.
  - Also, enums never result in unnecessary memory allocation.
  - the enum hack is a fundamental technique of template metaprogramming. (see Section **??**)

Another common misuse of the `#define` is using it to implement macros that look like functions but that do not incur the overhead of a function call. For example:

```
#define CALL_WITH_MAX(a,b) f( (a) > (b) ? (a) : (b) )
```

You have to remember to parenthesize all the arguments in the macro body. Oherwise you will run into trouble. However, even you get that right, the weird things can happen:

```
int a = 5, b = 0;
CALL_WITH_MAX(++a, b);     // a is incremented twice
CALL_WITH_MAX(++a, b+10); // a is incremented once
```

Now we can use a template for an inline function in Section **??**

```
template <class T>
inline void callWithMax(const T& a, const T &b){
  f(a > b ? a : b);
} // because we donot know what T is, we pass by reference-to-const.
```

1.2 Use const whenever possible

The const keyword is remarkably versatile. For pointers, you can specify whethwe the pointer itself is const or the data is const:

```
char greeting[] = "Heello";
const char *p = greeting;  //non-const pointer, const data
char *const p = greeting;  //const pointer, non-const data
const std::vector<int>::iterator iter = vec.begin(); // iter acts like a T *const;
++ iter;  // error! iter is const.
std::vector<int>::const_iterator cIter = vec.begin(); // iter acts like a const T*
*cIter = 10; // error! *cIter is const.
```

Some of the most powerful uses of const stem from its application to function declarations:

- **Having a function return a constant value** sometimes can reduce the incidence of the clinent errors without giving up safety or efficiency. For example: In Section **??**

  ```
  const Rational operator*(const Rational &lhs, const Rational &rhs);
  ```

  We can avoid the unintentional error like `if(a * b = c)`. Such code would be illegal if a and b were of a built-in type. **One of the hallmarks of good user-defined types is that they avoid gratuitous incompatibilities with built-in types.** (see Section **??**)

- The purpose of **const on member functions** is to identify which member functions may be invoked on const objects, which are important for two reasons:
  - They make the interface of a class easier to understand. It is important to know which functions may modify an object.
  - They make it possible to work with const objects. There is a fundamental ways to improve performance, which is **pass objects by reference-to-const**, which will explains in Section **??**.

- **Member functions differing only in** constness can be overloaded, which is an important feature of C++. For example:

```
class TextBlock{
public:
  const char &operator[](std::size_t position) const;
  char &operator[](std::size_t position);
};
```

By overloading and giving the different versions different return types, you can have `const` and non-`const` TextBlocks handled differently.

- There are two prevailing notions of `const`: **bitwise const** and **logical const**. Bitwise constness is easy to understand, for logical constness, here is an example"

```
class CTextBlock{
public:
  std::size_t length() const;
private:
  char *pText;
  std::size_t textLength;
  bool lengthIsValid;
};
std::size_t CTextBlock::length() const{
  if(!lengthIsValid){
    textLength = std::strlen(pText); // error!
    lengthIsValid = true;   // error!
  }
  return textLength;
}
```

Now the solution is simple: `mutable` frees non-static data members from the constraints of bitwise constness.

- **Avoiding Duplication in `const` and `non-const` member functions**. Sometimes `operator[]` in `TextBlock` not only returns a reference to the character, it also performed bounds checking, logged access information, etc. Putting all this in both functions yields more compilation time, maintenance and code-bloat. It is possible to move all codes into a seperate member function (private).

There is an another way. That is, you want to have one version of `operator[]` call the other one.

```
class TextBolock{
public:
  ...
  const char &operator[](std::size_t position) const{ ... }
  char &operator[](std::size_t position){
    return const_cast<char&>(static_cast<const TextBlock&>(*this)[position]);
  }
```

The one that removes `const` can be accomplished only via `const_cast`. Though casting is such a bad idea as a general rule (see **??**), but code duplication is no picnic either. It is determined by you, but this technique is worth knowing.

1.3 Make sure that objects are initialized before they are used

If you are in the C part of C++ and initialization would probably incur a runtime cost, initialization is not guaranteed to take place. This explains why array (from C part of C++) isn't necessarily guaranteed to have its contents initialized but a vector is. The best way is to **always initialize objects before you use them.**

- For built-in types, you need to do this manually, for else, the responsibility for initialization falls on constructors. However, **do not confuse assignment with initialization.**

```
class ABEntry{
public:
  ABEntry(const std::string &name, const stdLLstring &address);
private:
  std::string theName;
  std::string theAddress;
  int numTimesConsulted;
};
ABEntry::ABEntry(const std::string &name, const std::string &address){
  theName = name;          \\ these are all assignments, not initializations.
  theAddress = address;
  numTimesConsulted = 0;
}
```

Their default constructors were automatically called prior to entering the ABEntry constructor. But this is not true for numTimesConsulted because it is a built-in type. For it, there is no guarantee it was initialized at all prior to its assignment.

A better way is to use the member initialization list instead of assignments:

```
ABEntry::ABEntry(const std::string &name, const std::string &address)
: theName(name), theAddress(address), numTimesConsulted(0)
{} // these are now all initializations.
```

It is more efficient because default consturctions were wasted before.

For objects of built-in type, ifor consistency, it is often best to initialize everthing via member initialization. There is a policy of **always listing every data member on the initialization list**.

```
ABEntry::ABEntry(): theName(), theAddress(), numTimesConsulted(0)
{}
```

Sometimes initialization list must be used for built-in types. For example: data members that are const or references that must be initialized. (See 2.1).

When class has multiple constructors, we can omit entries in the lists for data members where assignment works as well as true initialization, moving the assignments to a single function that all constructors call.

One aspect of C++ that is not fickle is **the order in which an object's data is initialized**. This order is always the same: base classes, derived clasees, data members in the order in which they are declared. (**When initialize an array, declare the size first.**)

- **The order of initialization of non-local static objects difined in different translation units** is important.

  A static **object** is one that exists from the time it's constructed until the end of the program, including global objects, objects declared static inside classes, functions or at file scope. Static objects inside functions are known as local static objects.

  If initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, **the object it uses could be uninitialized.** For example,

```
class FileSystem{
public:
  std::size_t numDisks() const;
  ...
};
extern FileSystem tfs;
```

  In another file, there is

```
class Directory{
public:
  Directory(params);
  ...
};
Directory::Directory(params){
  std::size_t disks = tfs.numDisks();
  ...
}
```

  Further suppose the client decides to create a single Diretory object tempDir, tfs and tempDir are created by different people at different times in different source files. How can you be sure that tfs will be initialized before tempDir? You can't because **the relatice order of initilization of non-local static objects defined in different translation units is undefined.**

  Fortunately, a small design change eliminates the problem entirely. All that has to be done is to **move each non-local static object into its own function, where it's declared** static. This approach is founded on C++'s guarantee that local static objects are initilized when the object's definition is first encountered during a call to that function.

```
FileSystem &tfs(){
  static FileSystem fs;
  return fs;
}
Directory &tempDir(){
  static Directory td(params);
  return td;
}
```

The reference-returning functions dictated by this scheme are always simple, which makes them excellent candidates for inlining. On the other hand, it make them problematic in multithreaded systems. One way to deal with it is to invoke all the reference-returning functions during the single-threaded startup portion of the program.

To avoid using objects before they are initialized, you need to do three things:

- manually initialize non-member objects of built-in types
- use member initialization lists to initialize all parts of an object
- design around the initialization order uncertainty that afflicts non-local static objects defined in separate translation units.

2 Constructors, destructors, assignment operators

2.1 Know what functions C++ silently writes and calls

Compilers will declare their own versions of a copy constructor, copy assignment, destructor if you don't declare them. Further more, if you declare no constructors, compliers will also declare a default constructor. All these functions are `public and inline` (See **??**).

```
class Empty
public:
  Empty() { ... } // default constructor
  Empty(const Empty& rhs) { ... } // copy constructor
  ~Empty() { ... } // destructor | see below for whether it's virtual
  Empty& operator=(const Empty& rhs) { ... } // copy assignment operator
};
```

- The generated destructor is **non-virtual** (See 2.3) unless it's for a class inheriting from a base class that declares a virtual destructor,
- The copy constructor and assignment simply copy each non-static data member.
- If you want to support copy assignment in a class **containing a reference or const member**, you must define the copy assignment operator yourself.
- Compilers reject implicit copy assignment operators in derived classes that inherit from **base classes declaring the copy assignment operator** `private`.

2.2 Explicitly disallow the use of compilergenerated functions you do not want.

Use `=delete` or declare `private` and give no implementations if you don't want a class to support a particular kind of copy functionality.

2.3 Declare destructors virtual in polymorphic base classes.

A particluar example is get function of a factory, we will get a pointer to a base class. If we declare destructors non-virtual, **the object will be partially destroyed**.

- If a class does not contain virtual functions, that often indicates it is not meant to be used as a base class, so making the destructor virtual is usually a bad idea because the size of the class will increase. **Declare a virtual destructor in a class if and only if that class contains at least one virtual function.**
- The implementation of virtual functions requires that objects carry information that can be used at runtime to determine which virtual functions should be invoked on the object. This information typically takes the form of a pointer called a `vptr`. The `vptr` points to an array of function pointers called a `vtbl`; each class with virtual functions has an associated `vtbl`. When a virtual function is invoked on an object, the actual function called is determined by following the object's `vptr` to a `vtbl` and then looking up the appropriate function pointer in the `vtbl`.

- Not all base classes are designed to be used polymorphically. **Neither the standard string type, for example, nor the STL container types are designed to be base classes at all,** much less polymorphic ones.
- If we want an abstract base class, we can declare a pure virtual destructor, but **we must provide a definition for the pure virtual destructor.**

2.4 Prevent exceptions from leaving destructors

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

2.5 Never call virtual functions during construction or destruction

- **During base class construction of a derived class object, the type of the object is that of the base class.** Not only do virtual functions resolve to the base class, but the parts of the language using runtime type information (e.g., `dynamic_cast` (see **??**) and `typeid`) treat the object as a base class type.
- The same reasoning applies during destruction.
- It's not always so easy to detect calls to virtual functions during construction or destruction.

```
class Transaction {
public:
  Transaction() { init(); } // call to non-virtual...
  virtual void logTransaction() const = 0;
  ...
private:
void init() {
  ...
  logTransaction(); // ...that calls a virtual!
}
};
```

2.6 Have assignment operators return a reference to `*this`

One of the interesting things about assignments is that you can chain them together:

```
int x, y, z;
x = y = z = 15; // chain of assignments.
x = (y = (z = 15)); //equivalent.
```

The way this is implemented is that **assignment returns a reference to its left-hand argument**. This convention applies to all assignment operators.

```
class Widget {
public:
  ...
  Widget &operator=(const Widget &rhs){
    ...
    return *this;
  }
  Widget& operator+=(const Widget& rhs) // the convention applies to
  {                                     // +=, -=, *=, etc.
    ...
    return *this;
  }
Widget& operator=(int rhs)              // it applies even if the
  {                                     // operator's parameter type
  ...                                   // is unconventional
  return *this;
  }
};
```

2.7 Handle assignment to self in operator=

If you follow the advice of 3.1 and **??**, you'll always use objects to manage resources, and you'll make sure that the resource-managing objects behave well when copied.
If you try to manage resources yourself, however (which you'd certainly have to do if you were writing a resourcemanaging class), you can fall into the trap of **accidentally releasing a resource** before you're done using it. For example,

```
Widget& Widget::operator=(const Widget& rhs) // unsafe impl. of operator=
{
  delete pb; // stop using current bitmap
  pb = new Bitmap(*rhs.pb);                  // start using a copy of rhs's bitmap
  return *this;
}
```

Now, the self-assignment problem here is that itself holds a pointer to a deleted object! There are three ways to prevent this error.

- check for assignment to self via an identity test at the top of operator=.

  ```
  Widget& Widget::operator=(const Widget& rhs){
    if (this == &rhs) return *this; // identity test: if a self-assignment, do nothing
    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
  }
  ```

  This works but there is also exception-unsafe. If the "new Bitmap" expression yields an exception (either because there is insufficient memory for the allocation or because Bitmap's copy constructor throws one), the Widget will end up holding a pointer to a deleted Bitmap. You can't safely delete them. You can't even safely read them.

- **??** explores exception safety in depth, but in this Item, it suffices to observe that in many cases, a careful ordering of statements can yield exception-safe code.

  ```
  Widget& Widget::operator=(const Widget& rhs)
  {
    Bitmap *pOrig = pb; // remember original pb
    pb = new Bitmap(*rhs.pb); // point pb to a copy of rhs's bitmap
    delete pOrig; // delete the original pb
    return *this;
  }
  ```

  Now, if "`new Bitmap`" throws an exception, `pb` remains unchanged.

- If you're concerned about efficiency, use the technique known as "**copy and swap**", which is described in **??**.

  ```
  class Widget {
    ...
    void swap(Widget& rhs); // exchange *this's and rhs's data, see Item 29 for details
  };
  Widget& Widget::operator=(const Widget& rhs){
    Widget temp(rhs); // make a copy of rhs's data
    swap(temp); // swap *this's data with the copy's
    return *this;
  }
  ```

- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

2.8 Copy all parts of an object

When you're writing a copying function, be sure to

- copy **all** local data members,

  ```
  class Customer {
  public:
    ...
    Customer(const Customer& rhs);
    Customer& operator=(const Customer& rhs);
    ...
  private:
    std::string name;
    Date lastTransaction;
  };
  ```

```
Customer::Customer(const Customer& rhs)
: name(rhs.name){} // copy rhs's data  Data has been forgotten!!!

Customer& Customer::operator=(const Customer& rhs)
{
  name = rhs.name; // copy rhs's data
  return *this;    //  DATE!
}
```

- invoke the appropriate copying function in **all base classes**.

```
class PriorityCustomer: public Customer { // a derived class
public:
  ...
  PriorityCustomer(const PriorityCustomer& rhs);
  PriorityCustomer& operator=(const PriorityCustomer& rhs);
  ...
private:
  int priority;
};
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: Customer(rhs), // invoke base class copy ctor
  priority(rhs.priority){}
PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  Customer::operator=(rhs); // assign base class parts
  priority = rhs.priority;
  return *this;
}
```

- if you find that your copy constructor and copy assignment operator have similar code bodies, **eliminate the duplication by creating a third member function that both call**.

3 Resource Management

This chapter begins with a straightforward object-based approach to resource management built on C++'s support for constructors, destructors, and copying operations. Experience has shown that disciplined adherence to this approach can all but eliminate resource management problems. The chapter then moves on to Items dedicated specifically to memory management. These latter Items complement the more general Items that come earlier, because objects that manage memory have to know how to do it properly.

3.1 Use objects to manage resources