【Effective CPP】Day1

Book	Effective C++
Author	
Summary	
Date	@2022/05/01

【Ch1】Accustoming Yourself to C++

Item 1: View C++ as a federation of languages

To make sense of C++, we have to recognize its primary sublanguages. Fortunately, there are only 4:

- **C**
- **Object-Oriented C++**. C with classes: constructors and destructors, encapsulation, inheritance, polymorphism, virtual functions.
- **Template C++**. This is the generic programming part of C++.
- **The STL**. Containers, iterators, algorithms, function objects and etc.

Note: Rules for effective C++ programming vary, depending on the part of C++ we are using.

Item2: Prefer consts, enums, and inlines to #defines

This item might better be called “prefer the comipler to the preprocessor,” because `#define` may be treated as if it’s not part of the language.

When we do something like this:

```
#define ASPECT_RATIO 1.653
```

the symbol name `ASPECT_RATIO` may never be seen by compilers. It may be removed by the preprocessor before the source code ever gets to a compiler.

This can be confusing if we get an error during compilation involving the use of the constant, because the errror message may refer to 1.653, not `ASPECT_RATIO`.

The solution is to replace the macro with a constant:

```
const double AspectRatio = 1.653;
```

As a language constant, `AspectRatio` is definitely seen by compilers and is certainly entered into their symbol tables.

In addition, in the case of a floating point constant, use of the constant may yield smaller code than using a `#define`. That's because the preprocessor's blind substitution of the macro name `ASPECT_RATIO` with 1.653 could result in multiple copies of 1.653 in the object code.

The use of the constant `AspectRatio` should never result in more than one copy.

When replacing `#define` with constants, two special cases are worth mentioning.

The first is defining constnat pointers. To define a constant `char*`-based string in a header file, we have to write `const` twice:

```
const char* const authorName = "Scott Meyers";
```

However, the string objets are generally preferable to the char*-based progenitors. `authorName` is often and better defined this way:

```
const std::string authorName("Scott Meyers");
```

The second special case concerns class-specific constants. To limit the scope of a constant to a class, we must make it a member.
And to ensure there's at most one copy of the constant, we must make it a `static` member:

```
class GamePlayer {
private:
  static const int NumTurns = 5; // Constant declaration
  int scores[NumTurns]; // Use of constant
};
```

In-class initialization is allowed only for integral types and only for constants. In cases where the above syntax cannot be used, we put the initial value at the point of definition:

```
class CostEstimate {
private:
  static const double FudgeFactor; // Declaration of static class constant; goes in header
};

const double CostEstimate::FudgeFactor = 1.35; // Definition of static class constant; goes in the implementation file
```

However, this technique does not work if the value of the constant need to be known during compilation such as the declaration of the array `GamePlayer::scores` above.

We can use another technique called the enum hack instead. GamePlayer could just as well be defined like following:

```
class GamePlayer {
private:
  enum { NumTurns = 5 };
  int scores[NumTurns];
};
```

The enum hack behaves in some ways more like a `#define`. We cannot take the address of an enum, just like we cannot take the address of `#define`.

When we write macros like functions:

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

We can run into trouble when somebody calls the macro with an expansion:

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

Fortunately, we can just replace this with a function template:

```
template <typename T>
ineline void callMax(const T& a, const T& b) {
  f(a > b ? a : b);
}
```

Things to remember.

1. For simple constants, prefer `const` objects or `enums` to #define
2. For funtion-like macros, prefer inline functions to #define