【Effective CPP】Day2

Book	Effective C++
Author	
Summary	
Date	@2022/05/08

【Ch1】Accustoming to C++

Item 3: Use const Whenever Possible

`const` allows us to specify a constraint-a particular object should not be modified.

```
char greeting[] = "Hello";
const char* p1 = greeting; // Const-data; non-const pointer
char* const p2 = greeting; // Non-const data; const pointer
const char* const p3 = greeting; // Const data and const pointer
```

If const appears to the left of the asterisk, what's pointed to is constraint; if the word const appears to the right of the asterisk, the pointer itself is constant.

When what's pointed to is constant, some programmers list const before the type and some after the type. There is no difference in meaning:

```
void f1(const Widget*);
void f2(Widget const*);
```

STL iterators are modeled on pointers, so an iterator acts much like a `T*` pointer. When we declare a `const vector<int>::iterator`, it is like `T* const`: the iterator isn't allowed to point to something different, but the thing it points to may be modified.

If we want an iterator that points to something that cannot be modified, we should use `const_iterator`:

```
std::vector<int> vec(3, 0);
const std::vector<int>::iterator iter = vec.begin(); // Acts like a T* const
*iter = 10; // Ok
++iter; // Error!!! iter is const

std::vector<int>>::const_iterator cIter = vec.cbegin(); // cIter acts like a const T*
*cIter = 10; // Error!!! cIter is const T*
++cIter; // Ok
```

Having a function return a constant value is generally inappropriate, but sometimes doing so can reduce the incidence of client errors without giving up safety or efficiency.

For example, consider the declaration of the `operator*` function for rational numbers:

```
class Rational{...};
const Rational operator*(const Rational &lhs, const Rational &rhs);
```

Clients may write some code like this if we do not return `const Rational`

```
Rational a, b, c;
if(a * b = c)
```

Such code would be an error but can hardly be detected by the compiler.

const Member Functions

The purpose of `const` on member functions is to identify which member functions may be invoked on `const` objects.

Many people think that two member functions differing only on their constness can be overloaded, but see the following example:

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

  char& operator[](std::size_t position) {
    return text[position];
  }

private:
  std::string text;
};
```

Textblock's `operator[]` can be used as following. We can have const and non-const TextBlocks handle differently:

```
TextBlock tb("hello");
const TextBlobk ctb("Hi");

std::cout << tb[0]; // OK: reading a non-const object
std::cout << ctb[0]; // OK: reading a const object
tb[0] = 'A'; // OK: writing a non-const object
ctb[0] = 'A'; // Error: writing a const object
```

Note that the error here hassonly to do with the return type of the `operator[]`. The error arises out of an attempt to make an assignment to a `const char&`, because that is the return type from the const version of `operator[]`.

A const member function might modify some of the bits in the object on which it's invoked, but only in ways that clients cannot detect.

We can name some members `mutable`. `mutable` frees non-static data members from the constraints of bitwise constness

```
class CTextBlock {
public:
  ...
  std::size_t length() const;

private:
  char *pText;
  mutable std::size_t textLength;
  mutable bool lengthIsValid;
};
```

```
std::size_t CTextBlock::length() const {
  if(!lengthIsValid) {
    lengthIsValid = true;
    textLength = std::strlen(pText);
  }
  return textLength;
}
```