【C++】Day six

Class	C++
Date	@November 23, 2021
Material	
Series Number	
Summary	Compound Type Declaration&Const

【Ch2】Quantifiers

2.3.3 Understanding Compound Type Declarations

A single definition might define variables of different types:

```
int i = 1024, *ptr = &i, &refI = i;
```

A reference is not an object. Hence, we may not have a pointer to a reference.

However, because a pointer is an object, we can define a reference to a pointer.

```
int i = 42;
int *p;
int *&r = p; //r is a reference to the pointer p
r = &i; // r refers to a pointer; assigning &i to r makes p point to i
*r = 0; //dereferencing r yields i, the object to which p points
```

To understand the type of r is to read the definition right to left.

1. The symbol closest to the name of the variable(in this case the `&` in `&r`) is the one that has the most immediate effect on the variable's type.
2. Thus, we know that r is a reference.
3. The rest of the declarator determines the type to which r refers.
4. The next symbol, `*` in this case, says that the type r refers to is a pointer type.
5. The base type of the declaration says that r is a reference to a pointer to an int.

2.4 Const Qualifier

We can make a variable unchangeable by defining the variable's type as `const`:

```
const int bufSize = 512; //input buffer size
```

Now, any attempt to change the value of `bufSize` is an error.

Because we cannot change the value of a `const` object after we create it, it must be initialized.

```
const int k; //error: k is uninitialized const
```

Note: By default, `const` objects are local to a file.

The compiler will usually replace uses of the variable with its corresponding value during compilation. That is, the compiler will generate code using the value at the place where our variable is called.

To substitute the value of the variable, the compiler has to see the variable's initializer. When we split a program into multiple files, every file that uses the `const` must have access to its initializer. Thus, to define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration:

```
extern const int bufSize = 512; //File_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize; //same bufSize as defined in file_1.cc
```

Note: To share a `const` object among multiple files, you must define the variable as `extern`.

2.4.1 References to const

```
const int ci = 20;
const int &ref1 = ci; //ok: both reference and underlying object are const
int &ref2 = ci; //error: non-const reference to a const object
```

We can initialize a reference to `const` from any expression that can be converted to the type of the reference. In particular, we can bind a reference to `const` to a `nonconst` object, a literal, or a more general expression

```
int i = 42;
const int &r1 = i; //we can bind a const int& to a plain int object.
const int &r2 = 42; //ok: r1 is a reference to const
const int &r3 = r1 * 2; //ok: r3 is a reference to const
int &r4 = r * 2; //error: r4 is a plain, non-const reference
```

The easiest way to understand this difference in initialization rules is to consider what happens when we bind a reference to an object of a different type:

```
double dval = 3.14;
const int &ri = dval;
```

Here `r`i refers to an int. Operations on `ri` will be integer operations, but `dval` is a floating-point number, not an integer. To ensure that the object to which `ri` is bound is an `int`, the compiler transforms this code into something like

```
const int temp = dval;
const int &ri = temp;
```

In this case, `ri` is bound to a temporary object. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression.

Summary:

1. A `non-const` reference cannot be declared to a `const` variable
2. A `const` reference can be bound to either a `const` or `non-const` variable
3. A const reference can take any expression which can be converted to the type of the reference.(It will create a temporary object to do so).