Lecture 5: `const` Qualifier

Constant Modifier

Note: The `const` only works on the variable it declares. For the simplest example,

```
int a = 0;
const int *P = &a
```

`P` is a pointer to a constant integer. It only means that you cannot change the value of integer `a` by pointer `P`. It doesn't mean that piece of memory is read-only or unchangeable. So, you can still change the value of `a` by another pointer, or even just by `a` itself.

`const` in C Vs. `const` in C++

- Fake constant
- Symbol table

> Do `const` variables in C++ have address?

Not always. For example, when you first declare and initialize a:

```
const int a = 18;
```

Here a is stored in symbol table instead of memory. (There is no symbol table in C)
But when you use "&" to get the address of a:

```
const int *p = &a
```

Then a piece of memory will be alloced to store an integer whose value is 18.
After that, no matter how you change the value stored in this piece of memory, whenever you use `a` as an `int`, the value you use will always be 18.

Variants of const in C++

1. `Const` & Data variables

3) Special Case: const_cast and "double value"

```
#include <iostream>
using namespace std;
int main()
{
    const int a = 10;
    const int* p = &a;
    int* q;
    q = const_cast<int*>(p);
    *q = 20;
    cout << a << " " << *p << " " << *q << endl;
    cout << &a << " " << p << " " << q << endl;
    return 0;
}
```

Result:

```
10 20 20
0x7ffeca5eeea4 0x7ffeca5eeea4 0x7ffeca5eeea4
```

This is an example of changing the integer value stored in `&a`. But when you print `a`, it will still print 10.

2. `Const` & Function Arguments

3.`Const` & Pointer

ONE PRINCIPLE!!!

> const applies to the thing left of it. If there is nothing on the left then it applies to the thing right of it.

Exercises: What do these `const` apply to? They are all valid!

```
const int* a                // a pointer to a constant integer
int const * a               // a pointer to a constant integer
int* const a                // a constant pointer to an integer
const int* const a          // a constant pointer to a constant integer
int const * const a         // a constant pointer to a constant integer


int const * const * a       // a pointer to a constant pointer to a constant
integer
int const * const * const a  // a constant pointer to a constant pointer to a
constant integer
```

For the last two variables, you can first ignore the modifier `const`, in order to identify them more easily.
So, the type of `a` is obviously `int**`, which means `a pointer to a pointer to an integer`.
When applying the principle, you should first notice that :

1. `int` refers to the `integer` value.
2. The first `*` refers to the `pointer to an integer`.
3. The second `*` refers to the `pointer to a pointer to an integer`, which is a.

Then, apply that principle. For the second last line, `int` is on the left of the first `const`, and the first `*` is on the left of the second `const`. So, correspondingly, the `integer` value and the `pointer to an integer` is constant. But the value of `a` itself is not constant. So, a is a pointer to a constant pointer to a constant integer

4. `Const` & Class and Class Member

```
const Class object;

class MyClass
{
    int a;
    void Foo() const; // function member
    int get_a();
    MyClass(int _a);
};
```

For function member with `const`, you can think that, in this function, this class is declared `const`. In this function, add a `const` modifier to all the members in this class. In other words, `int a;` becomes `const int a;` and `int get_a();` becomes `int get_a() const;`

Note: `const` after `Foo()` works on pointer `this`, which is a pointer to this class. So you cannot change data members of this class but you can still change variables beyond this class.

5. `Const` & References

Const Reference vs Non-const Reference

There is something special about const references: (IMPORTANT!!!)

- Const reference are allowed to be bind to right values;
- Normal references are not allowed to.

Exercises:

Consider the following program. Which lines cannot compile?

```
int main(){
    // Which lines cannot compile?
    int a = 1;
    const int& b = a;
    const int c = a;
    int &d = a;
    const int& e = a+1;
    const int f = a+1;

    int &g = a+1;      // x
```

Here `a + 1` is a Right-Value.
By the way, in general, if we cannot get the address of something, it is a Right_Value. Reference has the same address with the origin variable.

Let's go on. `Normal references are not allowed to bind with right values.` So, here g cannot be bind to `a + 1`;
But if you change it into `const int &g = a + 1`. It will become valid and it is equivalent to `const int &g = 2` here.

```
    int r1 = 42;
    const int &r2 = 42;
    const int &r3 = r1 * 2;
    // These two are both R-value, including an expression.
    // So, only `const &` is valid here.


    // What if they have different types?
    double dval = 3.14;
    const int & r1 = dval;
    // What if r1 is not a constant?
```

If they have different types, it works like this

```
int tmp = (int) dval;
const int & r1 = tmp;
```

Note: this is only a simulation. We cannot access `tmp`, which is not a variable and has no name. It is just a piece of memory. So it is the same as `const int & r1 = 3` here.

Normally, if a const reference is bind to a right value, the const reference is no difference to a simple const.

Why we don't use a const pointer ?

- const reference -> rvals can be passed in.

For example, when we pass an integer to `void Foo(const &int a)`, we can dericly write `Foo(3)`.
But if the function declaration is `void Foo(const int *p)`, we have to declare another `int a = 3` and pass `a` to the function.

Const and Types

Function Pointer

```
typedef int (*MYFUN)(int, int);
```

How to understand this `typedef`?

Here is a small skill:

Let's begin with the easiest example:

```
1. typedef int *p;
```

For this simple `p`, we first ignore `typedef`.

```
int *p;
```

Then the type of `p` is `"pointer to an integer"`.
Adding `typedef`, then `p` is an alias name for the type `"pointer to an integer"`.

```
2. typedef double *Dp;
```

we first ignore `typedef`.

```
double *Dp;
```

Then in this line, the type of `p` is `"pointer to a double"`.
Adding `typedef`, then `p` is an alias name for the type `"pointer to a double"`.

Then, let's move to the pointer to function.

```
typedef int* Func(int); // Func *fptr; fptr is a pointer to function
```

Ignore `typedef`, we have `int* Func(int);`
Then this is a function declaration. The type of `Func` is : a function with type signature

```
(int*) (int)
```

So, adding `typedef`, `Func` means an alias name for `"the functions with type signature(int*) (int)"`.
Functions have different types. Type signature determines their type. If you are confused with type signature, please refer to Lecture 6 below.

Similarly, for

```
typedef int (*PFunc)(int);
// PFunc fptr; fptr here is also a pointer to function
```

Ignoring `typedef`, we have `int (*PFunc)(int);`, which declares a function pointer `PFunc`. The type of `Pfunc` is `"pointer to functions with type signature (int) (int)"`.
So, `PFunc` is an alias name for `"pointer to functions with type signature (int) (int)"`

How to use pointer to function?

For example

```
struct point
{
    int x;
    int y;
};
bool comp_F1(point a, point b);
bool comp_F2(point a, point b);
bool comp_F3(point a, point b);
bool comp_F4(point a, point b);
bool comp_F5(point a, point b);
bool comp_F6(point a, point b);
// Assume here are 6 functions to do comparison with different methods.
typedef bool (*Pfunc)(point, point);
int main()
{
    Pfunc funcs[6];
    funcs[0] = comp_F1;
    funcs[1] = comp_F2;
    funcs[2] = comp_F3;
    funcs[3] = comp_F4;
    funcs[4] = comp_F5;
    funcs[5] = comp_F6;
    vector<point> TO_Sort;
    for (int i = 0; i < 6; i++)
    {
        sort(TO_Sort.begin(), TO_Sort.end(), funcs[i]);
    }
}
```

You can also use

```
typedef bool Func(point, point);
Func *funcs[6];
```

Lecture 6: Procedural Abstraction

Type signature is different from function signature.

Type signature

- The type signature of a function can be considered as part of the abstraction.
- Type signature includes return type, number of arguments and the type of each argument.
- Type signature does not include the name of the function.

Function signature in C++

- Function signature includes function name, argument type, number of arguments, order, and the class and namespace in which it is located
- Two overloaded functions must not have the same signature.
- The return value is not part of a function's signature.

Function signatures are used for functions to identify different functions.

```
  int Divide (int n, int m) ;

  double Divide (int a, int b) ;
```

These two have the same function signature.

```
Divide(int, int);
```

So, compiler regard them as the same function, so they cannot be compiled together successfully. For `overload` functions, they should have differences except the return type.

But these two have different type signatures

```
int (int, int);
double (int, int);
```

So, if we `typedef`

```
typedef int fff1(int, int);
typedef double fff2(int, int);
```

`fff1` works for the `Divide` in the first line, and `fff2` works for `Divide` in the second line.

Exercise 3. Do any two of following functions have the same function signature?

```
int func(int);
float func(float);

class C {
    int func(int);
    class C2
    {
        int func(int);
    };
};

namespace N
{
    int func(int);
    class C
    {
        int func(int);
    };
}
```

None of them are the same. Because they are in different classes and namespaces. (Here namespace is not covered in VE280, you can ignore this exercise)

> If you have any questions, or if you find any error in my handouts, please come to me. Contact me by e-mail(cyril-chenyn@sjtu.edu.cn) or by WeChat(cyncfyy).

Credit

FA21 VE280
TA Chen Yunuo
Lecture 5 & 6