Big RC Part 2

Lecture 5: `const` Qualifier

Constant Modifier

`const` in C++ makes a particular data entity, a constant one i.e. during the scope of the entire program, the value of the const variable remains the same.

In the actual scenarios, the const keyword is very essential to lock a particular value as static and constant throughout its use.

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

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

> constant value should be initialized in the meanwhile of declaration

Immutability and Compiler

Whenever a type something is `const` modified, it is declared as immutable and is added to symbol table.

Remember this immutability is enforced by the compiler at compile time. This means that `const` in fact does not guarantee immutability, it is an intention to. The compiler does not forbid you from changing the value intentionally. Consider the following program:

```
int main(){
    const int a = 10;
    auto *p = const_cast<int*>(&a);
    *p = 20;
    cout << a << endl;
}
```

What's the output? This is actually an UB. It depends on your compiler and platform whether the output is 10 or 20. Therefore, be careful of undefined behaviors from type casting, especially when `const` is involved.

> Be very careful with `const_cast` in both your exam and daily programming.

Variants of const in C++

1. `Const` & Data variables

1) General Case

```
const data-type variable = value;
```

Error example: `error: assignment of read-only variable`

```
#include<iostream>
using namespace std;
int main()
{
    const int A = 10;
    A += 10;
    cout << A;
}
```

What if I do `int* p = &A`?

2) Const Global

Say when we declare a string for jAccount username, and want to ensure that the max size of the string is 32.

```
int main(){
    char jAccount[32];
    cin >> jAccount;
    for (int i = 0; i < 32; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

This is bad, because the number 32 here is of bad readability, and when you want to change 32 to 64, you have to go over the entire program, which, chances are that, leads to bugs if you missed some or accidentally changed 32 of other meanings.

This is where we need constant global variables.

```
int MAX_SIZE = 32;
int main(){
    char jAccount[MAX_SIZE];
    cin >> jAccount;
    for (int i = 0; i < MAX_SIZE; ++i){
        if (jAccount[i] == '\0'){
            cout << i << endl;
            break;
        }
    }
}
```

Note that const globals must be initialized, and cannot be modified after. For good coding style, use UPPERCASE for const globals.

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

IMPORTANT: Type Coercion!

The following are the **Const Prolongation Rules**.

- `const type&` to `type&` is incompatible.
- `const type*` to `type*` is incompatible.
- `type&` to `const type&` is compatible.
- `type*` to `const type*` is compatible.

In one word, only from non-const to const is allowed.

Exercise:

Consider the following example:

```
void reference_me(int &x){}
void point_me(int *px){}
void const_reference_me(const int &x){}
void main() {
    int x = 1;
    const int *a = &x;
    const int &b = 2;
    int *c = &x;
    int &d = x;

    // which lines cannot compile?
    int *p = a;              // x
    point_me(a);             // x
    point_me(c);
    reference_me(b);        // x
    reference_me(d);
    const_reference_me(*a);
    const_reference_me(b);
    const_reference_me(*c);
    const_reference_me(d);
}
```

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 data members in this class. In other words, `int a;` becomes `const int a;` and they cannot be changed in this scope. All the non-const functions need to be ignored as they may change the value of those const data members.

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 derictly 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.

Type define and rename

> Do replacement in your code.

Special: 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

Abstraction

> Abstraction is the principle of separating what something is or does from how it does it.

Properties:

- Provide details that matters (what)
- Eliminate unnecessary details (how)
- Only need to know what it does, not how it does it

2 types of abstractions:

- Data Abstraction
- Procedural Abstraction

The product of procedural abstraction is a procedure, and the product of data abstraction is an abstract data type (ADT).

Different roles in programming:

- The author: who implements the function
- The client: who uses the function
- In individual programming, you are both.

> An Example of client: you use cout to output, which is written by author of C++. You don't need to worry about how cout works.

Procedural Abstraction

Properties

Functions are mechanism for defining procedural abstractions.

Difference between *abstraction* and *implementation*:

- Abstraction tells what and implementation tells how.
- The same abstraction could have different implementations.

There are 2 properties of proper procedural abstraction *implementation*:

- Local: the implementation of an abstraction does not depend of any other abstraction implementation.
- Substitutable: Can replace a correct mplementation with another.

Composition

There are 2 parts of an abstraction:

- Type signature
- Specification

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.

Exercise:

1. What are the signatures of the following function pointers `fPtr`?

```
bool (*fPtr)(int, int);
```

```
(bool) (int, int); // type signature
```

2. Do these two functions have the same type signature ? Do these two functions have the same function signature ?

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

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

Specifications

Used to the describe abstraction (not implementation).

> How to describe implementation?
> natrual language / pseudo code / real code

There are 3 clauses in the specification comments:

- REQUIRES: preconditions that must hold, if any
- MODIFIES: how inputs will be modified, if any
- EFFECTS: what the procedure is computing, if any

```
void log_array(double arr[], size_t size)
// REQUIRES: All elements of `arr` are positive
// MODIFIES: `arr`
// EFFECTS: Compute the natural logarithm of all elements of `arr`
{
    for (size_t i = 0; i < size; ++i){
        arr[i] = log(arr[i]);
    }
}
```

Complete and Partial Functions

Completeness of functions are defined as follows:

- If a function does not have any `REQUIRES` clauses, then it is valid for all inputs and is complete.
- Else, it is partial.
- You may convert a partial function to a complete one.

> Note: Specifications are just comments. You cannot really prevent clients from doing stupid things, unless you use exception handling. While in VE280, you can always assume the input is valid if there is a REQUIRES comment.

Lecture 13: Abstract Data Type

Type

Type is a rather abstract concept. It can be defined independently of any programming languages. A type is always defined in two aspects:

- The set of values that can be represented by items of the type
- The set of operations that can be performed on items of the type.

For example, `int` in cpp actually refers to signed integers, which supports basic arithmetic operations, such as add, subtract and NOR.

struct in C++

- `struct` in C:
  - Provide a way to define complex data type
  - Simply organize data of different types together for simplicity and easier management
  - No inheritance. No polymorphism. No member functions. No access control (public, private, protected).
- `struct` in C++:
  - Enhanced version of C struct
  - All member variables/functions are public (no access control)
  - Support member function, function overloading, constructor/destructor, inheritance
  - However, it is still recommended to use `class` for complex types and behaviors. `struct` is more suitable for simple types defination.

Abstract Data Type

What is ADT?

ADTs provide an **abstract description** of **values** and **operations**. In short, to define an ADT, we only need to know:

- What values it represents: a mobile phone that can make and receive calls
- What can it do to these values (operations): turn on/off, make/receive call, text message, play games...

Why use ADT?

Abstraction hides implementation detail and makes users' life easier.

ADTs provide two advantages:

- Information hiding: We don't need to know the details (how messages travel across the air to reach our phone?)
  1. The user do not need to know (and should not need to know) how the object is represented (Is IntSet represented by array or linked list? We don't know, but we can use it well. The user do not need to know when using it!).

2. The user do not need to know (and should not need to know) how the operations on the object are implemented.

- Encapsulation: the objects and their operations are defined in the same place (You don't need to buy the screen, the circuit board, the wifi module...You just buy a phone)

  > combine both data and operations in one entity.

ADTs has several benefits:
Similary to the 2 properties of proper procedural abstraction implementation.

- **local**: the implementation of other components of the program does not depend on the implementation of ADT (5G base stations don't care how the phones will process the signals)
- **substitutable**: you can change the implementation and no users of that type can tell (iPhone, Huawei, Xiaomi can all make phone calls, but their hardwares/softwares are not the same)

ADTs are implemented with `class` in C++.

> Attention: The definition of ADTs itself has no relation with C++. C++ `class` is only one way to implement ADTs.

C++ class

C++ "class" provides a mechanism for both information hiding (private/protected access specifier) and encapsulation (member functions/methods).

> Understand the concepts with the example of IntSet in the lecture notes.

Defination & Declaration

Seperated files

Typically, we write only the declarations of a class in header files (.h/.hpp) and definitions in source files (.cpp). Header files and source files always appear in pairs. They are always named with class names. Each file describes only one class (Not forced in C++ but in Java).

Detail Hiding

When you want to use a library written by others (or cpp standard libraries), it is very likely that you will only get header files (.h/.hpp) and some static libraries (.a/.lib) or dymamic libraries(.so/.dll). The detailed implementation of these libraries are hidden, and you only need to read the documentations to learn how to use them.

However, writing function definations in header files do has its advantages. For functions defined in header files, it will be compiled as almost inline functions to improve performance.

Getters & Setters

These are two concepts widely used in OOP programming. Typically, getters & setters refer to those **member functions used to get or modify private members**. These methods allow outer codes to access certain attributes of the class.

Also, using getters/setters allows you to add extra operations when getting/modifying values. For example, validating the values when modifying an attribute:

```
class Student {
    int    score;
```

```
  public:
    // A getter of score, qualified as const
    int    getScore() const {return this->score;};

    // A setter of score. New scores lower than 0 is regarded as illegal.
    void   setScore(int newScore) {
      if (newScore < 0) {
        cout << "How is that possible?" << endl;
        return;
      }
      this->score = newScore;
    };
};
```

This validation cannot be done by directly assigning values.

Representation Invariant

This invariant is a rule that the representation must obey both **immediately before** and **immediately after** any method's execution. In simpler language:

- An ADT must still be legal no matter how your customer use it
- This is achieved by carefully writing each methods
- **"immediately before"** guarantees your following operations can be done correctly (if you are given a set like {1,1,2,2,3}, then even if you write the method `remove()` correctly, the result will not be correct)
- **"immediately after"** guarantees your operations by far are legal. This ensures that further operations can be done correctly.

More on C++ class

Initialization List

```
ClsName::ClsName() : base(..), m1(..), m2(..) {
       // Code for the some other operations need to be done during construction
}
```

- The order of initialization **is the order they are defined in the class**
- The performance (both time and memory) can be better than assigning to each values.
- A member that don’t have a default constructor must be initialized in the initialization list.
- const members and references can only be initialized in the initialization list.

const Member Functions

- A `const` qualifier after **member functions** promises that this member function will not modify this object.

```
class Sample {
    int val;
public:
    void setVal() const { val = 0; }    // Compile error
};
```

- Also, inside a `const` member function, non-const member functions (as well as other functions that may modify the object) cannot be called (to ensure that the object will not be modified).

```
void ordinary_func (int &d) {
    d = 666;
}
class Sample {
public:
    int a;
    int b;
public:
    int getA() const { return a; };
    int getB() { return b; }
    void setVal() const {
        int tmp1 = getA();
        int tmp2 = getB();  // not OK, getB() should be qualified as const
        ordinary_func(a);
        // not OK, "a" is automatically cast into "const int"
        // in this member function
    }
};
```

- This qualifer tells the compiler to check. It protects the object by casting all member variabales into `const`, as well as `this`.

Last Suggestion

Go back to the slides.

Good Luck!

Credit

FA21 VE280
Lecture 5 & 6 & 13