# VE280 2021SU Midterm Review Part3

## L5: Constant Qualifier

### Immutability of `const`

Whenever a type something is `const` modified, it is declared as immutable.

```
struct Point{
    int x;
    int y;
};

int main(){
    const Point p = {1, 2};
    p.y = 3; // compiler complains
    cout << "(" << p.x << "," << p.y << ")" << endl;
}
```

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?  As you can see, we use `const_cast` to cast a "pointer to const" to a normal pointer, and try to modifies `a` by that normal pointer. And this would not cause a compiler error. 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 in type casting, especially when `const` is involved.

### 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.

```
const 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 and '_' for delimiter for const globals.

## Const References

### Const Reference vs. Non-const Reference

There are many ways to define a constant reference, which are all identical.

- `const int& iref`
- `(const int)& iref`
- `int& (const iref)`
- `const int& (const iref)`

And const references has some special properties:

- Const reference are allowed to be bind to r-values (those values that could only appears right to the `=`), while normal references are not allowed to.

Consider the following program. Which lines cannot compile?

```
int main(){
    // Which lines cannot compile?
    int a = 1;
    const int& b = a; // line 4
    const int c = a; // line 5
    int &d = a;
    const int& e = a+1;
    const int f = a+1;
    int &g = a+1;
    b = 5;
    c = 5;
    d = 5;
}
```

const ref一旦绑定，就和普通的恒常变量一样了

Normally, if a const reference is bind to an r-value, the const reference **is the same as a simple const variable.** In the above example,  line 4 and 5 are actually identical.

### Usage - Argument Passing

Then **when do we need const references?** One important usage of const reference is argument passing. See the following example.

```
class Large{
    // I am really large.
    int a[200]
};

int utility(const Large &l){
    // ...
}
```

Reasons to use a constant reference:

- Passing by reference -> avoid expensive copy-by-value;
- `const` -> avoid unintentionally changes the input object - the compiler will catch the error
- const reference -> r-values (likes constants `utility(3)`, and expression `utility(3+4)` ) can be passed in.
    - That's why we don't use a const pointer.

## Const Pointers

There are many ways to define a `const` pointer, which are NOT identical. Personally my trick of identification is to **read from the right hand side.**

- Pointer to Constant (PC): `const int *ptr;`
- Constant Pointer (CP): `int *const ptr` or `int *(const ptr);`
- Constant Pointer to Constant (CPC): `const int *const ptr` or `const int *(const ptr)`.

| Type | Can change the value of pointer? | Can change the object that the pointer points to? |
|---|---|---|
| Pointer to Constant | Yes | No |
| Constant Pointer | No | Yes |
| Constant Pointer to Constant | No | No |

See the following example. Which lines cannot compile?

```
int main(){
    // Which lines cannot compile?
    int a = 1;
    int b = 2;
    const int *ptr1 = &a;
    int *const ptr2 = &a;
    const int *const ptr3 = &a;
    *ptr1 = 3;
    ptr1 = &b;
    *ptr2 = 3;
    ptr2 = &b;
    *ptr3 = 3;
    ptr3 = &b;
}
```

## Const and Typedef

### Type Definition

When some compound types have long names, you probably don't want to type them all. This is when you need `typedef`.

The general rule is `typedef real_type alias_name`.

For example, you probably want:

```
typedef std::unordered_map<std::string, std::priority_queue<int,
std::vector<int>, std::greater<int> > > string_map_to_PQ;
```

Mind that you can define a type based on a defined type. See following example. Which lines cannot compile?

```
typedef int * int_ptr_t;
typedef const int const_int_t;
typedef const int_ptr_t Type1;
typedef const_int_t* Type2;
typedef const Type2 Type3;

int main(){
    // Which lines cannot compile?
    int a = 1;
    int b = 2;
    Type1 ptr4 = &a;
    Type2 ptr5 = &a;
    Type3 ptr6 = &a;
    *ptr4 = 3;
    ptr4 = &b;
    *ptr5 = 3;
    ptr5 = &b;
    *ptr6 = 3;
    ptr6 = &b;
}
```

### 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 type coercion from non-const to const is allowed. Whenever you need a const object, you could always pass a non-const object, while the opposite is invalid.**

Consider the following example:

```
void reference_me(int &x){}
void point_me(int *px){}
void const_reference_me(const int &x){}

int main() {
    int x = 1;
    const int *a = &x;
    int *b = &x;
    const int &c = 2;
    int &d = x;

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

# L8: Enum

## Why `enum`

`enum` is a type whose values are restricted to a set of values.

Consider an example,

```
enum Error_t {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

The advantages of using an `enum` here are:

- Compared to constant `string`: more efficient in memory.
  - Even the minimum size of a `std::string` is larger than an `int`.
- Compared to constant `int` or `char`: more readable and limit valid value set.
  - Numbers or chars are less readable than a string.

## Properties

Enum values are actually represented as an integer types (`int` by default). If the first enumerator does not have an initializer, the associated value is zero.

```
enum Error_t {
    INVALID_ARGUMENT,   // 0
    FILE_MISSING,       // 1
    CAPACITY_OVERFLOW,  // 2
    INVALID_LOG,        // 3
};
```

Making use of this property, one usually writes:

```
int main(){
    const string error_info[] = {
            "The argument is invalid!",
            "The file is missing!",
            "Reaching the maximum capacity!",
            "The log is invalid!"
    };
    enum Error_t err = INVALID_LOG;
    cout << error_info[err] << endl;
}
```

Here are a few important properties of `enum`.

**Comparability:**

Enum values are comparable. This means that you can perform <, >, ==, >=, <=, != on enum values.

**Designable:**

The constant values can also be chosen by programmer.

Consider the following example, what would the output be?

```
enum QUQ {
    a, b, c = 3, d, e = 1, f, g = f+d
};

int main(){
    QUQ lovelyCat = g;
    cout << static_cast<int>(lovelyCat) << endl;
}
```

Note that if you directly use `cout << err << endl`, the compiler would complain. And it is always a good habit to cast the enum value to a printable type by like `static_cast<int>`.

The underlying integer type can be modified as well. If the range of `int` is either too big or too small, you can always use a different underlying integer type.

```
enum Error_t : char {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

Note that `char` is an integer type of 1 Byte (Recall your knowledge about ASCII).

## Enum Class

You may also define the `enum` as an enum class, *i.e.*

```
enum class Error_t {
    INVALID_ARGUMENT,    // 0
    FILE_MISSING,        // 1
    CAPACITY_OVERFLOW,   // 2
    INVALID_LOG,         // 3
};
```

When defining an variable, do not forget about `Error_t::`.

```
enum Error_t err = Error_t::INVALID_LOG;
```

# L9: Program Arguments

Program arguments are passed to the program through `main()`:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    cout << argc << endl;
    for(int i = 0; i < argc; ++i){
        cout << argv[i] << endl;
    }

    return 0;
}
```

`argv`: all arguments including program name; an array of C-strings

`argc`: the number of strings in `argv`

Call:

```
g++ -o test test.cpp
./test ve280 midterm
```

Output:

```
3
./test
ve280
midterm
```

Manipulate C-strings:

```
int integer = atoi(argv[1]);

char carr_err[strlen(argv[1])] = argv[1]; // incorrect
char *carr = new char[strlen(argv[1])+1];
strcpy(carr, argv[1]); // #include <cstring>
delete[] carr;

std::string str = argv[1]; // recommended
```

Many Linux commands are programs that take arguments:

```
diff file1 file2
g++ test.cpp
```

You could take a look at `/usr/bin` and `/bin` directories if you like.

# L10: I/O Streams

## `cin` , `cout` & `cerr`

### >> and <<

In C++, streams are **unidirectional**, which means you could only `cin >>` and `cout <<`.

If we look into `cin`, it's an object of class `istream` (input stream). `operator>>` (the extraction operator) is one of it's member function.

Check `std::istream::operator>>`, similar for `cout` & `cerr` ( `ostream` )

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (long long& val);
istream& operator>> (unsigned long long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

Many type of parameter it takes -> it knows how to convert the characters into values of certain type

Return value also a reference of `istream` -> it can be cascaded like `cin >> foo >> bar >> baz;`

Some other useful functions:

```
istream& getline (istream& is, string& str);
std::ios::operator bool // member of istream -> if(cin), while(cin)
istream& get (char& c); // member of istream
```

### Stream Buffer

`cout` and `cin` streams are buffered (while `cerr` is not).

You need to run `flush` to push the content in the buffer to the real output.

`cout << std::endl` is actually equivalent to `cout << '\n' << std::flush`

## File Stream

```
#include <fstream>

ifstream iFile; // inherit from istream
ofstream oFile; // inherit from ostream
iFile.open("myText.txt");


iFile >> bar;
while(getline(iFile, line)) // simple way to read in lines
iFile.close(); // important

oFile << bar;
```

## String Stream

```
#include <sstream>

istringstream iStream; // inherit from istream
iStream.str(line); // assigned a string it will read from, often used for
getline
iStream >> foo >> bar;
iStream.clear(); // Sometimes you may find this useful for reusing iStream
iStream.close();

ostringstream oStream; // inherit from ostream
oStream << foo << " " << bar;
string result = oStream.str(); // method: string str() const;
```

Convert the message with ',' and **arbitrary number of spaces (could be 0)** as the delimiter

```
Ann ,Peter , Owen,Alice , Bob, Tim
```

to

```
Ann
Peter
Owen
Alice
Bob
Tim
6 // number of names
```

## Solution

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
```

```
{
    string line;
    getline(cin, line);
    for (auto it = line.begin(); it != line.end(); it++) // you could also use
line[i]
        *it = (*it == ',') ? ' ' : *it; // change ',' to ' '
    stringstream ss(line);
    int count = 0;
    string name;
    while (ss >> name)
    {
        cout << name << endl;
        count++;
    }
    cout << count << endl;
}
```

# L11: Testing

## Difference between testing and debugging

- Testing: discover a problem
- Debugging: fix a problem

## Five Steps in testing:

1. Understand the specification
2. Identify the required behaviors (program that outputs differently on 1/2 arguments)
3. Write specific tests (for each required behavior)
4. Know the answers in advance
5. Include stress tests

## General steps for Test Driven Development

1. Think about task specification carefully
2. Identify behaviors
3. Write one test case for each behavior
4. Combine test cases into a unit test
5. Implement function to pass the unit test
6. Repeat above for all tasks in the project

## A simple example

Step 1: Specification

```
Write a function to calculate factorial of non-negative integer,
return -1 if the input is negative
```

Step 2: Behaviors

```
Normal: return 120 for input = 5
Boundary: return 1 for input = 0
Nonsense: return -1 for input = -5
```

Step 3: Test Cases

```
void testNormal() {
    assert(fact(5) == 120);
}

void testBoundary() {
    assert(fact(0) == 1);
}

void testNonsense() {
    assert(fact(-5) == -1);
}
```

Step 4: Unit Test

```
void unitTest() {
    testNormal();
    testBoundary();
    testNonsense();
}
```

## Another example

Step 1: Specification

```
Given 2 non-negative integers n>=1 and 0<=r<=n, write a function to calculate all
the binomial coefficient starting from C(n,r) to C(n,n), output -1 if the input
is invalid
```

Step 2: Behaviors

```
Normal: output 6, 4, 1 for input (n,r) = (4,2)
Boundary_1: output 1 / 1, 1 for input n = 1 (testing n = 1)
Boundary_2: output C(n,0), ..., C(n,n) for input r = 0 (testing r = 0)
Boundary_3: output 1 for input r = n (testing r = n, only one output)
Nonsense_1: return -1 for input n < 1; (n == 0)
Nonsense_2: return -1 for input r < 0; (r == -1)
Nonsense_3: return -1 for input r > n; ((n,r) = (2,3))
```

# L12: Exception

## `try` & `catch`

1. the **first** catch block with the **same** type as the thrown exception object will handle the exception
2. `catch(...) {}` is the default handler that matches any exception type
3. It is always good coding styles to specify in the EFFECTS clause that when a function would throw an exception:

```
int factorial (int n);
    // EFFECTS: if n>=0, return n!; otherwise throws n
```

## Backpropagation: locate the handler

1. browse through the remaining part of the function, if not found, proceed to step 2
2. browse through caller of the function, if not found, proceed to step 3
3. browse through caller of the caller...

   Once found: run all the commands **in and after** the catch block **that catch the exception.**

```
void foo()
{
    try
    {
        throw 'a'; // try int and double
    }
    catch (char c)
    {
        cout << "Char " << c << " caught!" << endl;
    }
    cout << "Foo executed." << endl;
}

void bar()
{
    try
    {
        foo();
    }
    catch (int i)
    {
        cout << "Int " << i << " caught!" << endl;
    }
    cout << "Bar executed." << endl;
}

int main()
{
    try
    {
        bar();
    }
    catch (...)
    {
        //default constructor
        std::cout << "Default catch!" << endl;
    }
    cout << "Main executed." << endl;
    return 0;
}
```

## Exam Logistics

- 16:00 –17:40,  June 22nd, 2021 (Tuesday)
- Close book
  - L2 Linux Commands
  - L3 Develop C++ program
  - L4&L5 C++ Basics (Pointers, Reference, Const Qualifier)
  - L7 Function Pointers & Function Call Mechanism
  - L8 Enum
  - L10 I/O Streams
  - L11 Testing
  - L12 Exception
  - L13 ADT
- General tips (ranked by priority):
  - Go over the lecture slides and midterm review RC notes carefully
  - Review your project 2 code
  - Practice writing some codes by hand (you may take a look at lab exercises)

## Reference

[1] Paul, Wang, Weikang, Qian. VE280 Lecture 5, 8-12.

[2] Ziqiao, Ma. VE280 Midterm Review Slides. 20SU.

[3] Changyuan, Qiu. VE280 Midterm Review Slides. 20FA.