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

- Fake constant
- Symbol table

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

Immutability and Compiler

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

```
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? 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.

Variants of const in C++

The `const` keyword in C++ can be used alongside different data entities of programming such as:

- Data variables

- Function arguments
- Pointers
- Class and class member
- Reference

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
```

2. `Const` & Function Arguments

```
data-type function(const data-type variable)
{
  //body
}
```

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

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

int add(const int x)
{
    x=x+100;
    return x;
}
int main()
{
  int ans = add(50);
  cout<<"Addition:"<<ans;
}
```

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

```
data-type const *variabl;
```

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

```
#include<iostream>
using namespace std;
int main()
{   int age = 21;
    int const *point = & age;
    *point = *point+10;
    cout << "Pointer value: " << *point;

}
```

There are many ways to define a `const` reference, which are NOT identical.

- 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
#### Exercise:		
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;      // x
    ptr1 = &b;
    *ptr2 = 3;
    ptr2 = &b;      // x
    *ptr3 = 3;      // x
    ptr3 = &b;      // x
}
```

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*
int const *
int* const
const int* const
int const * const
int const * const *
int const * const * const
```

4. `Const` & Class and Class Member

```cpp
const Class object;

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

Error example 1: `passing 'const Test' as 'this' argument of 'int Test::getValue()' discards qualifiers`

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
    int getValue() {return value;}
};

int main() {
    const Test t;
    cout << t.getValue();
    return 0;
}
```

Error example 2:

```cpp
class MyClass
{
    const int a; // data member
    void Foo() const; // function member
    MyClass(int _a);
};
MyClass::MyClass(int _a)
{
    this->a = _a;
}
```

Correction:

```
MyClass::MyClass(int _a):a(_a){}
```

5. `Const` & References

Const Reference vs Non-const Reference

There is something special about const references:

- 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
    b = 5;              // x
    c = 5;              // x
    d = 5;




    const int ci = 100;
    const int &r1 = ci;
    r1 = 11              // x
    int &r2 = ci         // x


    int i= 42;
    const int &r1 = i;
    const int &r2 = 42;
    const int &r3 = r1 * 2;


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

Normally, if a const reference is bind to a right value, the const reference is no difference to a simple const. In above example, you may consider line 4 and 5 identical.

Why do we need const references ?

See the following example.

```
class Large{
    // I am really large.
};
int utility(const Large &l){
    // ...
}
```

Reasons to use a constant reference:

- Passing by reference -> avoids copying;
- `const` -> avoids changing the structure.

Why we don't use a const pointer ?

- const reference -> rvals can be passed in.

Const and Types

Type Definition

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

```
typedef real_name alias_name
```

Example:

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

Actually, size_t is `typedef`ed from long unsigned int.

Advantages?

Improve the portability and readability of your code.

Exercise:

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

```
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;      // x
    *ptr5 = 3;      // x
    ptr5 = &b;
    *ptr6 = 3;      // x
    ptr6 = &b;      // x
}
```

Function Pointer

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

How to understand this `typedef`?

Let's begin with the easiest example:

```
int *p;
typedef int *p;
```

Let's move to

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

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.
- 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 implementation 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 function name, number of arguments and the type of each argument.
- Type signature is also known as function prototype.
- Two overloaded functions must not have the same signature.
- The return value is not part of a function's signature.

Exercise:

Do these two functions have the same 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]);
    }
}
```

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.

Credit

FA21 VE280 Teaching Group
Lecture 5 & 6