Lecture 5: `const` Qualifier

`Const` qualifier is used to avoid changing. Any changing to `const` variants will be warned by compiler.

NOTE: pointer to `const` type is a little bit special.

Example:

```
int a = 0;
const int b; //x
const int c=0; //√
const int *P; //√
P = &a;
*p=1; //x
a=1; //√
```

- Only pointer to `const` variable like `P` can be not initialized when declared.
- The `const` only works on the variable it declares: can point to non-const variants & no changing through that pointer only.

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 function members
- Reference

1. `Const` & Data variables

1) General Case 1

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

Error example: 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**.

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;
        }
    }
}
```

For good coding style, use UPPERCASE for `const` globals.

2. `Const` & Function Arguments

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

Error example: assignment of read-only parameter

```
int add(const int x)
{
    x=x+100;
    return x;
}
```

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 coercion 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
```

*4.`Const` & Class Function Member

```
const Class object;

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

For `Foo()`, `int a` becomes `const int a` and `int get_a()` becomes `int get_a() const`. This is because "`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; //any left value is right value
const int c = a;
int &d = a;
const int& e = a+1;
const int f = a+1;
int &g = a+1; // x
int &g = b; // x
b = 5; // x
c = 5; // x
d = 5; //a = 5
}
```

Normally, if a const reference is bind to a right value, the const reference is no difference to a simple const. `b` and `c` is similar.

Why do we need `const` references or pointers ?

See the following example.

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

Reasons to use a constant reference:

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

Advantage of `const` reference over pointer:

Passing rvals directly.

```
add(1,2);
frequency("absdfjad", "a");
```

`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. Typedef is just an alias name. It will improve the portability and readability of your code.

```
typedef real_name alias_name
```

Typedef may nest.

Exercise:

```
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
}
```

type1=const ptr to int, type2=ptr to const int, type3=const_ptr to const_int

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)

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.

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).

Focus: Procedural Abstraction

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

- 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 type** is not part of a function's signature

Example: Do these two functions have the same signature ?

```
int Divide (int n, int m) ;
double Divide (int a, int b) ; //x

main(){
    ...
    a=Divide(n,m);
}
```

Specifications

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

```
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 by exception handling.

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 7: Recursion; Function Pointers; Function Call Mechanism

Recursion

Recursion simply means to refer to itself. Its idea is to divide and conquer. It will loop until the boundary, or base case, is reached.
For any recursion problem, you may focus on the 2 compositions:

- Base cases: There is (at least) one “trivial” base or “stopping” case.
- Recursive step: All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.

A trivial example would be:

```
i
nt factorial (int n) {
// REQUIRES: n >= 0
// EFFECTS: computes n!
    if n == 0
        return 1; // Base case
    else
        return n * factorial(n-1); // Recursive step
}
```

Sometimes it is hard to implement a recursive function directly due to lack of function arguments.
In this case, you may find a **helper function** useful.
Instead of

```
recursion(...){
    ...
    recursion(...)
    ...
}
```

One may use

```
recursion(...){
    ...
    recursion_helper(...)
    ...
}
recursion_helper(...){
    ...
    recursion_helper(...)
    ...
}
```

where recursion_helper keeps updating the extra arguments, eg. `is_palindrome_helper(string s, int begin, int end)` in lecture slides keeps increasing `begin` and deceasing `end`.

Function Pointers

Variables that store the address of functions are called function pointers. By using them, we could pass functions into functions, return them from functions, and assign them to variables.

Consider when you only need to change one step in a larger function, like changing “adding” all elements in the matrix to “multiplying” all the elements. It is a waste of time and space to repeat the code, thus programmers would consider using a function pointer.

```
int avg(int arr[], size_t size) {
    int average = 0;
    for (size_t i = 0; i < size; i++){
        average += arr[i];
    }
    average /= size;
    return average;
}
int get_stats(int arr[], size_t size, int (*foo)(int[], size_t)){
    ...
    foo(arr, size);
    ...
}
int main(){
    int arr[] = {1,2,3,4,5};
    cout << get_stats(arr, 5, min) << endl;
    cout << get_stats(arr, 5, max) << endl;
    cout << get_stats(arr, 5, avg) << endl;
}
```

Mind the difference between function pointer and other pointer. Do not use "&" when assigning and do not use "*" when calling. (Although they actually both work, it is a convention and it is easier.) You can think of assigning as, for example, telling compiler to substitute `foo` with `avg` when codes call `foo`.

Function Call Mechanism

To fully understand function call mechanism, you need to understand the memory organization of system, which is beyond the scope of VE280 (Learn more about it in VE370). For now, you only have to get familiar with the concept of **call stack**.
At each function call, the program does the following:

1. Evaluate the actual arguments.
   For example, your program will convert `y = add(1*5, 2+2)` to `y = add(5, 4)`.
2. Create an **activation record** (stack frame)
   The activation record would hold the formal parameters and local variables.
   For example, when `int add(int a, int b) { int result = a+b; return result; }` is called, your system would create an activation record to hold:
   ○a , b (**formal parameters**)
   ○result (**local variable**)
3. Copy the actual values from step 1 to the memory location that holds formal values.
4. Evaluate the function locally.
5. Replace the function call with the result.
   For the same example, your program will convert `y = add(5, 4)` to `y = 9` .

6. Destroy the activation record.

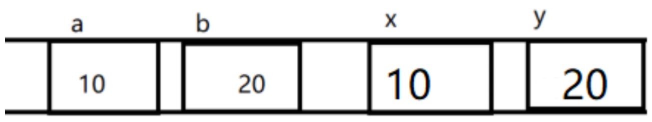
Typically, we come across situations with multiple functions are called and multiple activation records are to be maintained. To store these records, your system applies a data structure called stack.

Actual parameters and formal parameters

- Actual parameters: what you fill in the brackets to call the function. They exist outside the called function.
- Formal parameters: what are used in called function. They have same value with actually parameters (copying), but occupy different memory and only exist in call stack and are only used by the function. After the function calling, they are destroyed.

Example

```
int add(int x, int y){...}
main(){
    int c, a=10, b=20;
    c=add(a,b);//Actual parameters:a, b; Formal parameters:x, y
}
```



- Changing formal parameters will not affect actual parameters.
- **Passing pointer/reference**: outside variables will be modified

Credit

2021 RC slides
Lecture 5, 6, 7