C++ Basic Course

Chapter 1 —— Hello, C++!

What is C++?

> ***C++ = C + Object-Oriented Programming + Generic Programming***

C Programming Philosophy

Let's talk about C programming philosophy first. Like most mainstream language when C was created, C is a ***procedural*** language. That means it emphasizes the algorithm side of programming. Conceptually, procedural programming consists of figuring out the actions a computer should take and then using the programming language to implement those actions. In short, ***Data + algorithms = program***. BTW, procedural programming, AKA ***Procedural-Oriented Programming***.

Earlier procedural languages, such as FORTRAN and BASIC, ran into organizational problems as problems as programs grew larger. Many older programs had such tangled routing (called "**spaghetti programming**") that is virtually impossible to understand a program by reading it, and modifying such a program was an invitation to disaster. In response, computer scientists developed a more disciplined style of programming called ***structured programming***. C includes features to facilitate this approach. The primary idea behind structured programming is to avoid the use of undesirable control flow structures such as infinite loops, unrestricted goto statements, and deeply nested conditional statements, and instead focus on using clear, organized control structures.

Structured programming has the following characteristics:

1. Sequential Structure: Programs execute statements sequentially, from top to bottom, one after the other.
2. Selection Structure: Conditional statements (e.g., if statements) are used to choose different code blocks for execution based on specific conditions.
3. Iteration Structure: Loop statements (e.g., for loops, while loops) are used to repeat a section of code until a specific condition is met.

What else need to be mentioned is the ***Top-down design***. With C, the idea is to break a large program into smaller, more manageable tasks. If one of these tasks is still too broad, you divide it into yet smaller tasks. You continue this process until the program is compartmentalized into small, easily programmed modules. C's design facilitates this approach, encouraging you to develop program units called ***functions*** to represent individual task modules. As you may have noticed, **the structured programming techniques reflect a procedural mind-set, thinking of a program in terms of the actions it perform.**

The C++ Shift: Object-Oriented Programming

Although the principle of structured programming improved the clarity, reliability, and ease of maintenance of programs, large-scale programming still remains a challenge. ***Object-Oriented Programming*** brings a new approach to that challenge. Unlike procedural programming, which emphasizes algorithms, OOP emphasizes the data. Rather than try to fit a problem to the procedural approach of language, OOP attempts to fit the language to the problem. The idea is to

design data forms that correspond to the essential features of a problem. That data forms are called "***Classes***".

The OOP approach to program design is to first design classes that accurately represent those thing with which the program deals. Then you would proceed to design a program, suing objects of those classes. The process of going from a lower level of organization, such as classes, to a high level, such as program design, is called ***bottom-up programming***. However, top-down programming and bottom-up programming are **not contradictory**.

C++ and Generic Programming

Generic programming is yet another programming paradigm supported by C++. It shares with OOP the aim of making it simpler to reuse code and the technique of abstracting general concepts. But whereas OOP emphasizes the data aspect of programming, generic programming emphasizes independence from a particular data type. And its focus is different. OOP is a tool for managing large projects, whereas generic programming provides tools for performing common tasks, such as sorting data or merging lists. The term generic refers to code that is type independent. C++ data representations come in many types— integers, numbers with fractional parts, characters, strings of characters, and user-defined compound structures of several types. If, for example, you wanted to sort data of these various types, you would normally have to create a separate sorting function for each type. Generic programming involves extending the language so that you can write a function for a generic (that is, an unspecified) type once and use it for a variety of actual types. C++ templates provide a mechanism for doing that.

Basic syntax

Our goal is to completely understand the code below!

```
#include <iostream>
#include <typeinfo>

using namespace std;

void swap(int, int);
void swap(int*, int*);
void swapPointer(int*, int*);
void swapReference(int&, int&);

void output(int p, int q) {
    cout << "a: " << p << endl
         << "b: " << q << endl << endl;
}

int main() {
    int a, b;
    cin >> a >> b;

    cout << "Before swap: " << endl;
    output(a, b);

    swap(a, b);
    cout << "After swap(int, int): " << endl;
```

```
    output(a, b);

    swap(&a, &b);
    cout << "After swap(int*, int*)" << endl;
    output(a, b);

    swapPointer(&a, &b);
    cout << "After swapPointer(int*, int*)" << endl;
    output(a, b);

    swapReference(a, b);
    cout << "After swapReference(int&, int&)" << endl;
    output(a, b);

    return 0;
}

void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(int* a, int* b) {
    auto tmp = a;
    a = b;
    b = a;

    cout << typeid(tmp).name() << endl;
}

void swapPointer(int* a, int*b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void swapReference(int& a, int& b) {
    auto tmp = a;
    a = b;
    b = tmp;

    cout << typeid(tmp).name() << endl;
}
```

- ***The C++ Preprocessor***

  C++, like C, uses a ***preprocessor***. This is a program that processes a source file before the main compilation takes place. You don't have to do anything special to invoke this preprocessor. It automatically operates when you compile the program.

```
#include <iostream>     // a PREPROCESSOR directive
```

This directive causes the preprocessor to add the contents of the iostream file to your program. This is a typical preprocessor action: adding or replacing text in the source code before it's compiled. In essence, the contents of the iostream file replace the `#include <iostream>` line in the program. **Your original file is not altered, but a composite file formed from your file and iostream goes on to the next stage of compilation.**

- ***header***

Kind of header	Convention	Example	Comments
C++ old style	Ends in `.h`	`iostream.h`	Usable by C++ programs
C old style	Ends in `.h`	`math.h`	Usable by C and C++ programs
C++ new style	No extension	`iostream`	Usable by C++ programs, uses `namespace std`
Converted C	`c` prefix, no extension	`cmath`	Usable by C++ programs, might use non-C features, such as `namespace std`

- ***namespace***

Namespace support is a C++ feature designed to simplify the writing of large programs and of programs that combine pre-existing code from several vendors and to help organize programs. One potential problem is that you might use two prepackaged products that both have, say, a function called `wanda()`. If you then use the `wanda()` function, the compiler won't know which version you mean. The namespace facility lets a vendor package its wares in a unit called a ***namespace*** so that you can use the name of a namespace to indicate which vendor's product you want. So Microflop Industries could place its definitions in a namespace called `Microflop`.Then `Microflop::wanda()` would become the full name for its wanda() function. Similarly, `Piscine::wanda()` could denote Piscine Corporation's version of wanda(). Thus, your program could now use the namespaces to discriminate between various versions:

```
Microflop::wanda("go dancing?");        // use Microflop namespace version
Piscine::wanda("a fish named Desire")   // use Piscine namespace version
```

The using directive like `using namespace std` makes all the names in the `std` namespace available. Modern practice regards this as a bit lazy and potentially a problem in large projects. The preferred approaches are to use the std:: qualifier or to use something called a using declaration to make just particular names available:

```
using std::cout;
using std::cin;
using std::endl;
```

However, for me, it is more customary not to use the using directive.

In addition, the using directive can also give aliases to types:

```
using i64 = long long;
```

It is somewhat similar to `#define LL long long`, but still different:

- ***Scope:*** "using i64 = long long;" is the syntax for creating a type alias in C++, and its scope is limited to the current namespace or scope. On the other hand, "#define i64 long long" is a preprocessing directive in C and C++ that performs text replacement during the preprocessing phase, and its scope extends throughout the entire source file.
- ***Type Safety:*** "using i64 = long long;" preserves type safety when creating a type alias. "i64" and "long long" are essentially the same type with just a different name. This means that when using "i64," the compiler performs type checking to ensure that only "long long" operations are allowed.

  "#define i64 long long" is a simple text replacement and lacks type checking. When using "i64," the preprocessor replaces all occurrences of "i64" with "long long" without performing type checks.
- ***Life time:*** The type alias created with "using i64 = long long;" remains valid for the entire lifetime of the program and can be used anywhere in the code. In contrast, the text replacement performed by "#define i64 long long" is only effective within the current source file and does not affect other files.

- ***comments***

  The ***double slash (//)*** introduces a C++ comment. A comment is a remark from the programmer to the reader that usually identifies a section of a program or explains some aspect of the code. The compiler ignores comments.

  - C++ also recognizes C comments, which are enclosed between /* and */ symbols:

    ```
    #include <iostream> /* a C-style comment */
    ```

- ***function***

  Because functions are the modules from which C++ programs are built and because they are essential to C++ OOP definitions, you should become thoroughly familiar with them. Some aspects of functions are advanced topics, so the main discussion of functions comes later (maybe).

  However, if we deal now with some basic characteristics of functions, you'll be more at ease and more practiced with functions later.

  - ***function form***

    ```
    returnType functionName(argumentList) {
        statements;
    }
    ```

    If return type is `void`, that means the function returns nothing, you can use `return;` statement to end the function, or let the program run to the end of the function automatically. Similarly, if the argument is `void`, then it means that the arguments are not accepted, and `foo(void)` and `foo()` have the

same meaning. However, it should be noted that in C, the argumentList is empty, which means silence to accept arguments, rather than not accepting the arguments, unless explicitly declared as `foo(void)`

```
...
void foo() { printf("Function running\n"); }
...
foo(1);
...
// In C++
error: too many arguments to function 'void foo()'
     foo(1);
note: declared here
void foo() { printf("Running\n"); }

/* In C */
Function running
```

- ***function header***

  The `output()` function in code has this header

```
void output(int p, int q);
```

As we said before, the initial void means that little poor `output()` has no return value. So you can't use it this way:

```
auto var = output(a, b);

error: 'void var' has incomplete type
   21 |     auto var = output(a, b);
```

The `int p, int q` within the parentheses means that you are expected to use `output()` with two arguments of type int. The `p, q` are new variables assigned the value passed during a function call.

What's more, `a, b` in the function call are `argument / actual parameter`, `p, q` in the function header are `parameter / formal parameter`. In function prototypes, the name of the `parameter / formal parameter` can be ignored, you just need to keep the types.

- ***function polymorphism / function overloading***

In C, you are not allowed to write the code like this:

```
...
void swap(int, int);
void swap(int*, int*);
```
```
swap(a, b);
swap(&a, &b);
...
```

You'll get a bunch of error:

```
error: conflicting types for 'swap'; have 'void(int *, int *)'
    4 | void swap(int*, int*);
      |      ^~~~
note: previous declaration of 'swap' with type 'void(int, int)'
    3 | void swap(int, int);
      |      ^~~~
In function 'main':
warning: passing argument 1 of 'swap' makes pointer from integer without a cast [-Wint-conversion]
    9 |     swap(a, b);
      |          ^
      |          |
      |          int
note: expected 'int *' but argument is of type 'int'
    4 | void swap(int*, int*);
      |           ^~~~
warning: passing argument 2 of 'swap' makes pointer from integer without a cast [-Wint-conversion]
    9 |     swap(a, b);
      |             ^
      |             |
      |             int
note: expected 'int *' but argument is of type 'int'
    4 | void swap(int*, int*);
      |                 ^~~~
At top level:
error: conflicting types for 'swap'; have 'void(int *, int *)'
   19 | void swap(int* p, int* q) {
      |      ^~~~
note: previous definition of 'swap' with type 'void(int, int)'
   15 | void swap(int p, int q) {
      |      ^~~~
```

But just like you saw before, it's not only allowed in C++, but also very very very widely used.

Function polymorphism is a neat C++ addition to C's capabilities. Whereas default arguments let you call the same function by using varying numbers of arguments, function polymorphism, also called ***function overloading***, lets you use multiple functions sharing the same name. The word polymorphism means having many forms, so function polymorphism lets a function have many forms. Similarly, the expression function overloading means you can attach more than one function to the same name, thus overloading the name. Both expressions boil down to the same thing, but we'll usually use the expression function overloading—it sounds harder working. You can use function overloading to design a family of functions that do essentially the same thing but using different argument lists.

The key to function overloading is a function's argument list, also called the ***function signature***. If two functions use the same number and types of arguments in the same order, they have the same signature; the variable names don't matter. C++ enables you to define two functions by the same name, provided that the functions have different signatures. The signature can differ in the number of arguments or in the type of arguments, or both. For example, you can define a set of print() functions with the following prototypes:

```
void print(const char* str, int width); // #1
void print(double d, int width);        // #2
void print(long l, int width);          // #3
void print(int i, int width);           // #4
void print(const char* str);            // #5
```

When you then use a print() function, the compiler matches your use to the prototype that has the same signature:

```
print("Pancakes", 15);      // use #1
print("Syrup");             // use #5
print(1999.0, 10);          // use #2
print(1999, 12);            // use #4
print(1999L, 15);           // use #3
```

When you use overloaded functions, you need to be sure you use the proper argument types in the function call. For example, consider the following statements:

```
unsigned int year = 2023;
print(year, 6);             // ambiguous call
```

Which prototype does the print() call match here? It doesn't match any of them! A lack of a matching prototype doesn't automatically rule out using one of the functions because C++ will try to use standard type conversions to force a match. If, say, the only print() prototype were #2, the function call print(year, 6) would convert the year value to type double. But in the earlier code there are three prototypes that take a number as the first argument, providing three different choices for converting year. Faced with this ambiguous situation, C++ rejects the function call as an error.

```
error: call of overloaded 'print(unsigned int&, int)' is
ambiguous
   19 |     print(year, 6);
      |     ~~~~~^~~~~~~~~
note: candidate: 'void print(double, int)'
   10 | void print(double d, int width);
      |      ^~~~~
note: candidate: 'void print(long int, int)'
   11 | void print(long l, int width);
      |      ^~~~~
note: candidate: 'void print(int, int)'
   12 | void print(int i, int width);
      |      ^~~~~
```

Some signatures that appear to be different from each other nonetheless can't coexist. For example, remember the code we gave at first? Can I change the name of `swapReference()` to `swap()`? The answer is **no**! We'll explain it later when we talk about reference.

Keep in mind that the signature, not the function type, enables function overloading. For example, the following two declarations are incompatible:

```
long gronk(int n, float m);     // same signatures
double gronk(int n, float m);   // hence not allowed

error: ambiguating new declaration of 'double gronk(int,
float)'
   10 | double gronk(int n, float m);
      |        ^~~~~
note: old declaration 'long int gronk(int, float)'
    9 | long gronk(int n, float m);
```

Therefore, C++ doesn't permit you to overload gronk() in this fashion. You can have different return types, but only if the signatures are also different:

```
long gronk(int n, float m);
double gronk(double n, float m);
```

New types

- ***type alias***

  we said before

- ***const && constexpr***

- the ***auto*** keyword

C++11 introduces a facility that allows the compiler to deduce a type from the type of an initialization value. For this purpose it redefines the meaning of auto, a keyword dating back to C, but one hardly ever used. (We'll discuss the previous meaning of auto later, or not.) Just use auto instead of the type name in an initializing declaration, and the compiler assigns the variable the same type as that of the initializer:

```
auto n = 100;       // n is int
auto x = 1.5;       // x is double
auto y = 1.3e12L;   // y is long double
```

However, this automatic type deduction isn't really intended for such simple cases. Indeed, you might even go astray. Only use when the type is obvious or when the type is annoyingly verbose to write out.

```
#include <iostream>
#include <string>
#include <map>
#include <unordered_map>
#include <vector>

int main() {
    std::map<std::string, std::vector<std::pair<int,
std::unordered_map<char, double> > > complexType;

    // What does this do? We'll find out in the iterators lecture!
    std::map<std::string, std::vector<std::pair<int,
std::unordered_map<char, double> > >::iterator it =
complexType.begin();;

    // vs
    auto it = complexType.begin();

    return 0;
}
```

- ***array***
- ***pointer***
    - ***void ****
- ***reference***
- ***struct***
- ***class***