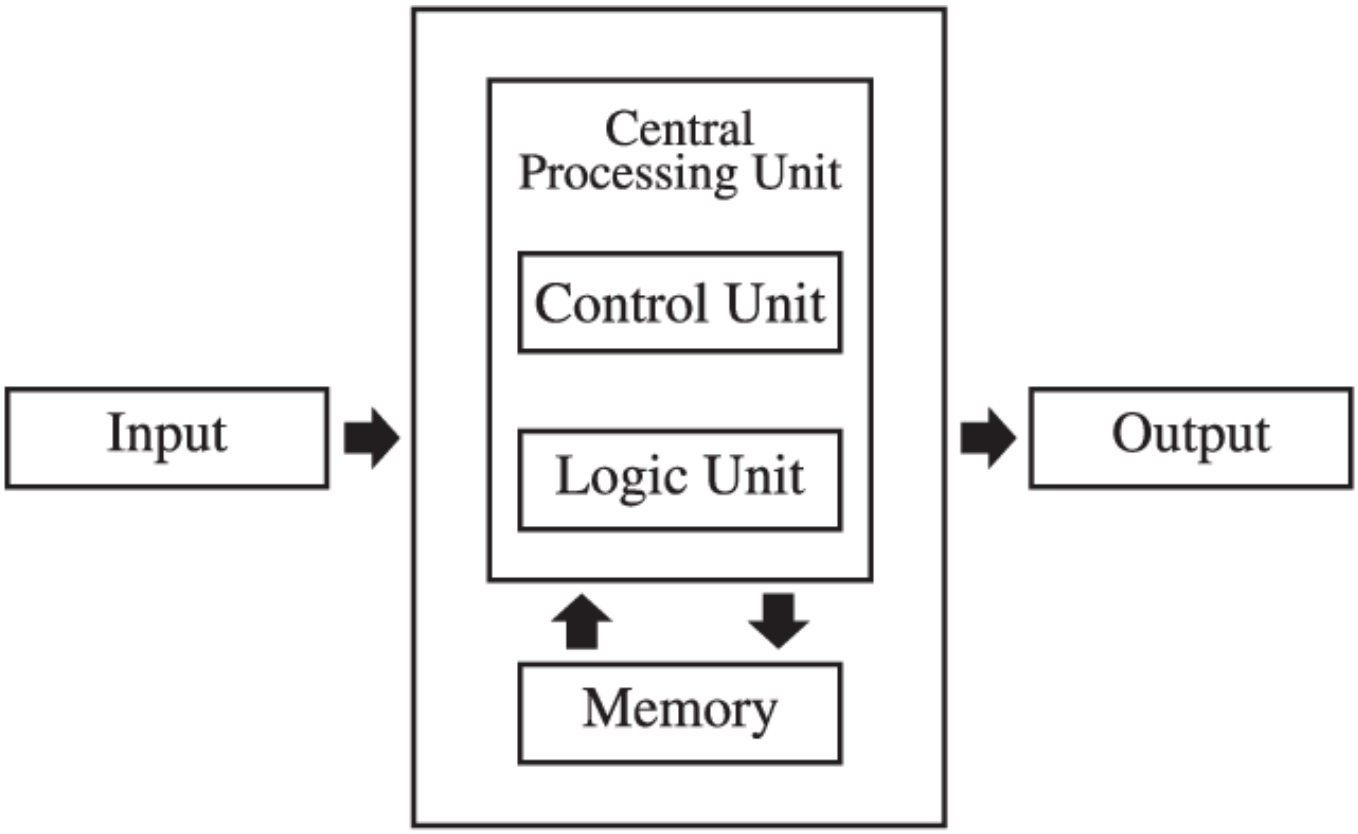
Fundamentals of Programming with C++

Jacy(家才)

Chapter 01 Introduction of Programming

1. The Working Model of Computer

1.1 Von Neumann Structure



1.2 The Component of Computer

> A computer is composed of hardware and software. If hardware is the blood and flesh of a computer, the software is its soul.
>
> To some extent, the performance of a computer is mainly determined by its hardware, while the function of a computer is mostly decided by its software.

- **hardware**
  - **Central Processing Unit, CPU**
    - **Controller** gets the instructions from memory and give order to other parts.
    - **ALU (Arithmetic Logic Unit)** implement the arithmetic and logic computations described by instructions.
    - **Register** temporarily stores the data needed to do the computation and the outcome of the computation, the state of current implementation and the address of the next instruction.
  - **memory**

    > Memory is used to store the ongoing program and the data used by the program.
    >
    > The capacity of **memory** is much larger than **register**, but the access to register is much faster than memory.

    - **Read Only Memory, ROM**

        - **Random Access Memory, RAM**

            > If we turn off the computer, the content of ROM is still there, but the content of RAM will be lost.

    - **peripheral device**
        - **input/output device**
        - **External storage** permanently stores anything you want to keep while the computer is turned off
- **software**

    > Software can be divided into **system software**, **application software** and **supporting software**.

    - **Program** is the description of processing objects (**data structure**) and processing **rules**(**algorithm**)
    - **Document** is the explanation materials that helps understand the program.

> A computer with only hardwares is called **bare machine**.
>
> When software is added to the computer, it evolves into a more functional computer called **virtual machine**.

1.3 How Information is Represented in Computer

> All information in the computer is eventually represented as a sequence of 0 and 1.
>
> The basic unit of information is called **bit**, which means a **binary digit**.

2. Programming

2.1 Programming Paradigm

- **Imperative programming** emphasizes **how to do it** and gives the computer explicit descriptions of the definite and specific procedure.
    - **Procedural programming**
    - **Object-oriented programming**
- **Declarative programming** emphasizes **what to do** and leave everything else to the computer.
    - **Functional programming**
    - **Logic programming**

2.2 Programming Procedure

- **requirement analysis**
- **system design**
- **system implementation**, also called **coding**
- **testing** and **debugging**
- **maintenance**

2.3 language

- **low-level language**
    - **Machine language** can be understand by computer directly, which is composed of 0 and 1.
    - **Assembly language** highly corresponds to machine language and can also be understood by human though with difficulty. But it must be translated by **assembler** into machine language for computers to actually understand.
- **high-level language**, also called **programming language**

    > High-level language must be translated into machine language or assembly language then machine language for computers to understand.
    >
    > The main two ways to translate high-level language is **compiling** by **compiler** or **interpreting** by **interpreter**.

> Terminology:
>
> **Syntax** refers to rules.
>
> **Semantics** refers to meanings.
>
> **Pragmatics** refers to occasions and effects.

3. Introduction of C++

> A C++ program must have a function named `main` defined in only one source document.

3.1 Lexer in C++

1. **symbol set**
    - letters
    - numbers
    - special characters
2. **word**
    - **Identifier** is composed of letters, numbers and underline. A legal identifier can't start with a number and can't be keyword.

        > - Capital letter and lowercase letter is different.
        > - There will be some common practice in defining a user-defined identifier. Following them will make your code more easier to be understood by others.

    - **Keyword** is predefined by the language itself and conveys some special and fixed meanings, which also has its own pattern of usage.
    - **Literal** is composed of numbers, characters and strings.
    - **Operator** describes the operation on data, which is also called operands.
    - **Punctuation** plays a role in syntax and semantics.

    > When editing a C++ code, the words above sometimes need to be separated using **white-space character**.

> **Comment** is also a kind of white-space character, which does some illustration and demonstration to help the reader or other coder under stand the content.
>
> There are two ways of writing comments:
>
> ```
> //till the end of this line.
> /* between thiese two special forms, and can include multiple lines*/
> ```
>
> `\` is called **continuation character**, which allows you two write your code on the next line when the current line is not long enough.

3.2 Procedure for the Running of a C++ Program

- **editing**
  - Use an **editor** to edit **source code** and store them in a document with a filename extension of `.h` or `.cpp`.
- **compiling**
  - Use a **compiler**(including a **preprocessor**, which will execute some part of the source code which is not actually a part of the C++ program) to translate the source code into the **object code** and store them in a document with a filename extension of `.obj`.
- **linking**
  - Use a **linker** to link all the files that make up the whole program and output an **executive code** stored in an executive document with a filename extension of `.exe`.
- **executing/running**

Chapter 02 Simple Description of Data

1. Data Type

> Data type is composed of **value set** and **operation set** and can be divided into **primitive data type** and **compound data type**.

In terms of the requirement of declaration on data type, programming language can be classified as:

- **Statically typed language** call for declaration on type of each data and usually executed by compiling.
- **Dynamically typed language** doesn't need to claim data type and usually executed by interpreting.

2. Fundamental Data Type

> Fundamental data type is predefined by the language and is also called **standard type** or **built-in type**.
>
> Fundamental data type is mostly **arithmetic type**.

2.1 Integral Type

1. integer type

- `int`
- `short int` or `short`, ranging from -32768 to 32767, occupying 2 byte.
- `long int` or `long`

> Range(short int) <= Range(int) <= Range(long int)
>
> Unsigned integer is twice the range of usual integer type.

2. character type

- `char`

> Character is stored in computer as **ASCII**, which means **American Standard Code for Information Interchange**.

3. boolean type

- `bool`

> Boolean type has only two value: `true` or `false`, while `true` is often stored as `1` and `false` is often stored as `0`.

2.2 Real Type

> Real type is also called **float point type** in C++.

- `float`
- `double`
- `long double`

> Range(float) < Range(double) <= Range(long double)

2.3 Void Type

- `void`

> Void type is used to describe the type of the return value of a function that actually doesn't return a value.

2.4 Some Operations

- We can use function `sizeof(<type-name>)` to get the space(byte) that needed to store this type of data.
- We can use keyword `typedef <existing-type-name> <new-name>` to rename an existing type, for example:

```
typedef unsigned int Uint;
//Remember that typedef just rename an existing data type, rather than creating a new data type.
```

3. Forms of Data

> In a program, data usually appears as **constant** or **variable**.

3.1 Constant

> **Constant** refers to the data which remains the same or can't be modified during the execution of the program.

1. literal

> **literal** refers to the constant represented just by its value and doesn't have a unique name.

Scientific Notation:

We use E(or e) as 10 when using scientific notation in C++, for example `4.5678E2 = 456.78`.

2. symbolic constant

> **Symbolic constant** refers to the constants which has a unique name.
>
> Using symbolic constant in tour code will make it more readable, more consistent and easier to maintain.

Definition of Symbolic Constant:

```
const <data-type> <constant-name> = <value>;
//or
#define <constant-name> <value>
//Example:
const double PI = 3.1415926;
#define PI 3.1415926
```

3.2 Variable

> Each variable has a **name**, which is an identifier in C++, belongs to a specific and unique data type and has a value that corresponds to its data type. When running the program, each variable will get a space with a concrete address in the memory.

Definition of Variable:

```
<data-type> <name>; //or
<data-type> <name> = <initial-value>; //or
<data-type> <name> (<initial-value>);
//Example:
int a = 0; //a is an integer type variable, which can also be written like: int a(0);
int b = a + 1; //initial value can also be an expression
double x; //x is a double type variable with out initial value
```

- Definition of multiple variables with the same data value can be defined in one line:

```cpp
int a = 0, b = a + 1;
```

- Remember that we must give the variable a value before we using the it. A variable without value makes no sense.

3.3 How to Input a Value?

```cpp
#include <iostream> //include some declaration of basic operations
using namespace std; //C++ built-in functions are defined in the namespace std

int main(){
    int i;
    double d;
    cin >> i; //input an integer from keyboard and assign the number to the variable i
    cin >> d; //input a real number from keyboard and assign the number to the variable d
    ......
    return 0;
}
```

> ```cpp
> cin >> i;
> cin >> j;
> ```
>
> can also be written in one line like:
>
> ```cpp
> cin >> i >> j;
> ```
>
> `cin` use white-space character as a division for the input.

4. Operator

> **Operator** describes basic operation on data, which is also called **operand(s)**.
>
> Operands can be both variables or constants or expressions that evaluate to a value.
>
> In C++, some operator will change the value of operands, such as `++` `--` `=` . This is called **side effect**.

4.1 Arithmetic Operator

name	symbol
add/ positive	`+`
minus/ negative	`-`
multiply	`*`
divide	`/`
self-increase	`++`
self-decrease	`--`

1. About `/`

- When the operands of `/` are both integer, `/` will turn into `//`.

2. About `++`/`--`

- `++` and `--` can both be put before or behind the operand, there will be some difference if the whole expression is used as an operand of other operator.
- `++`/`--` put before the operand means first self-increase/self-decrease then use; `++`/`--` put behind the operand means first use then self-increase/self-decrease.
- `++`/`--` not only evaluates to a value but also change the value of the operand, so they are operator with side effects.

> The data type of the return value of an arithmetic operator is usually the same as the operand(s).
>
> The problem of **overflow** will occur in the process of doing arithmetic operations with vary large operands, which need to be taken into consideration.

4.2 Boolean Operator

1. Relational Operator

> Relationship operator is used to compare the size of two operands, whose return value is boolean type.

name	symbol
more than	`>`
less than	`>`
no less than	`>=`
no more than	`<=`
equal	`==`
`different`	`!=`

> Characters will be compared according to their ASCII.
>
> Double type can't be transformed into floats without any error, so we should avoid comparing two floats directly using relational operator. Instead, we should compare their distance with a extremely small number, for example:
>
> ```
> fabs(x - y) < 1e-6; //x == y
> fabs(x - y) > 1e-6; //x != y
> ```

2. Logic Operator

> Logic operator is used to manipulate boolean types or expressions that evaluate to boolean types, whole return value is the value of its last evaluation.

name	symbol
and	`&&`
or	`||`
not	`!`

> Logic Operator follows the rule of **Short-circuit Evaluation** . That is to say when one operand is true, there's no need to evaluate the remaining operands of `||` and when one operand is false, there's no need to evaluate the remaining operands of `&&`.
>
> The rule of short-circuit evaluation on one hand improves the efficiency of computing, on the other hand provide a **guard** for the remaining operands, which means the former operand can help avoid raising exceptions when evaluating the latter operands.

4.3 Bitwise Operator

> Bit wise operator manipulates two binary numbers digit by digit.

1. Logical Bitwise Operator

name	symbol
bitwise not	`~`
bitwise and	`&`
bitwise or	`|`
bitwise differ	`^`

> Operator `^` has a special feature: `(x^a)^a = x`.

2. Shift Operation

name	symbol
left-shift	`<<`
right-shift	`>>`

4.4 Assignment Operator

> The value of a variable can be modified, either by input or by assignment statement.

1. Atomic Assignment Operator

`=` needs two operands, and the former one must be an expression whose value is editable. The operator will assign the value of the expression to the right of it to the former operand.

> In C++, unlike other programming language, assignment is provided as an operator rather than a statement, which has its own return value that is the left operand of `=`. So the change of the value of its left operand is `=`'s side effect.
>
> So, `(a = b) * c` means first give the value of `b` to `a`, then multiply `a` by `b`.

2. Compound Assignment Operator

`a #= b` is the same as `a = a # b`, here `#` is a binary operator.

4.5 Other Operators

1. Conditional Operator

name	symbol
conditional	`<d1> ? <d2> : <d3>`

> **Conditional operator** is *a ternary operator*, whose syntax is:
>
> ```
> <d1> ? <d2> : <d3>
> //Here <d1> is usually a boolean expression whose return value is true or false.
> //if <d1> evaluates to true, then the return value of this expression will be thevalue of <d2>;
> //Otherwise, the return value of this expression will be the value of <d3>.
> (a > b) ? a : b
> //The return value of the expression above is the maximun of a and b.
> ```

2. Comma Operator

name	symbol
comma	`<exp1>, <exp2>`

> The meaning of `,` is to do each computation from left to right.
>
> ```
> x = a + b, y = c + d, z = x + y;
> //is the same as
> z = a + b + c + d;
> ```

3. Sizeof Operator

name	symbol
sizeof	`sizeof<data-type>` or `sizeof<expression>`

> **sizeof** computes the space it needs in the memeory by `byte`.

5. Type Conversion of Operands

5.1 Implicit Type Conversion

- **Usual Arithmetic Conversions**

> In arithmetic expressions, operands of different data type in the same expression will be conversed to the same data type, which is the data type **with maximum value set**.
>
> All integral type including integer, character, boolean will be conversed into integer when appearing in arithmetic expressions.

- **Relational Conversions**

> Just like usual arithmetic conversions, different data type will be conversed into the data type with maximum value set.
>
> Boolean expressions can also be used as operands with its return value true as 1 and false as 0.

- **Logic Conversions**
- **Bitwise Conversions**
- **Assignment Conversions**

> The operand to the right of `=` will be conversed to correspond with the operand to the left of `=`.

- **Conditional Conversions**

5.2 Explicit Type Conversion

Syntax:

```
<type-name> (<operand>)
//or
(<type-name>) <operand>
```

> Note that we'll encounter some error or lost some data if we converse a data type with a smaller value set to a data type with a larger value.

6. Basic Computation of Data - Expressions

6.1 Component of Expression

- Expression consists of operators, operands and parentheses, which make up the primitive computation unit in a program.
- Constants and Variables alone make up a special form of expression called primitive expression or atomic expression.
- If there are two operators live next to each other, to make it not ambiguous, we use whitespace to separate them apart.

6.2 Types of Expression

1. According to Their Return Value

- **Arithmetic Expression** evaluates to integer type or real type.
- **Relational/Boolean** Expression evaluates to boolean type.
- **Address Expression** evaluates to an address.

2. According to Whether its Value is Determined When Compiling

- **Constant Expression**'s value is determined when compiling because its value doesn't need the implementation of the whole program to define.

3. According to Whether it has an Explicit Address in the Memory

- **Lvalue Expression** has an address representing a room in the memory to store its value and its addressed can be accessed using `&`.
  - Lvalue expression can also be divided into modifiable and unmodifiable because a symbolic constant also has its address but its value can't be modified.
- **Rvalue Expression**'s value is stored in temporary store unit whose address can't be accessed during the program using `&` and when the expression is evaluated over, the space will be released.

6.3 Precedence and Associativity of an Operator

> Each operator has its built-in associativity and precedence.
>
> The precedence defines the priority of two nearby operator while the associativity defines who came first when their are of the same priority.

Basic Rule for Precedence:

- unary > binary > ternary > assignment
- arithmetic > bitwise shift > relational > bitwise logic > logic

6.4 Output

```cpp
#include <iostream>
using namespace std;
int main(){
    ......
    cout << a + b * c;
    cout << a;
    cout << b;
    cout << endl; //another line
    //or
    cout << a + b * c << a << b << endl;
}
```

Chapter 03 Control Flow

1. Algorithm

> **Algorithm** describes the control of the flow, such as which expression computes first, whether to compute some expressions or repeating computing some expressions several times.
>
> We use **statement** to describe and implement algorithm.
>
> Sometimes we choose to first draw a **flowchart** to help us clarify the algorithm before we actually write some code to implement it.

2. Sequential Execution

- **Expression Statement**

> An **expression statement** is just an expression with `;` behind it, which does the same thing as the expression but is a complete statement.

- **Compound Statement**

> **Compound statement** is composed of several statements, with `{` before them and `}` behind them, which is also called **block**.

- **Blank statement**

> Blank statement is just `;`, which does nothing at all. What it only does is that it plays a grammar role or syntax role at a place where it needs nothing but a statement to make it grammatically appropriate.

3. Selection Control

> **Selection control** also called **branching control** allows us to execute some statements selectively according to some conditional judgments, which is implemented using **if-statement** or **switch-statement**.

3.1 If-statement

```
if (<boolean-expression>) <statement(s)>
if (<boolean-expression>) <statement(s)1> else <statement(s)2>
```

> C++ doesn't take writing style into consideration when reading your code.
>
> Unlike python, indent makes no sense in C++ other than make your code easier for other coder to understand.
>
> Each `else` is coupled with its **nearest** `if`.

3.2 Switch-statement

```
switch (<integral-expression>){
    case <integral-constant-expression1>: <statement-sequence1>
    case <integral-constant-expression2>: <statement-sequence2>
    ......
    case <integral-constant-expressionn>: <statement-sequencen>
    [default: <statement-sequencen+1>]
}
```

> Integral constant expressions' values can't be the same with each other.
>
> Statement sequence can either be a statement or a block without parentheses, and regularly end with a break statement.

Rules for execution:

- Evaluate the integral expression to get an integer value.
- Compare the value with all the following cases one by one until the integral constant expression with the same value.
- Execute the statement sequence behind it until it reach the end or encounter a break statement.
- If there's no corresponding case, execute the statement sequence behind default if there exists one.

4. Loop Control

> A loop control usually contains four parts:
>
> - Loop initialization
> - Loop condition
> - Loop body
> - Next loop preparation

> Please always remember to check the loop condition and next loop preparation so that your loop can end within control.

4.1 While-statement

```
while (<conditional-expression>) <statement(s)>
```

> Keep executing statement(s) while conditional expression evaluates to true value.
>
> First judge, then execute.

4.2 Do-while-statement

```
do <statement(s)> while (<conditional-expression>);
```

> Keep executing statements(s) until conditional expression evaluates to false value.
>
> First execute, then judge.
>
> Don't leave out the `;` at the end.

4.3 For-statement

```
for (<expression-1>; <expression-2>; <expression-3>) <statement(s)>
```

> Here expression-1, expression-2 and expression-3 can be any expression and can all be left out.
>
> Usually expression-1 is assignment; expression-2 is boolean expression; expression is self-increase or self-decrease expression.

The Implementation Rules:

1. Evaluate expression-1, usually as loop initialization
2. Evaluate expression-2, usually as loop condition, if false, get out of the loop
3. Implement the statements
4. Evaluate expression-3, usually as preparation for next loop.

> Variables defined in expression-1 only lives in the for-loop.
>
> The usage of expression-1 reveals that the designer of C++ advocates that we define a variable just where we begin to use it, rather than defining it in the very beginning, which might be difficult for human to understand.

4.4 Types of Loop

- **Counter-Controlled Loop**, in which you know the times of loop ahead of time and use a **loop control variable** to count the times until it reach the predefined times.
- **Event-Controlled Loop**, in which you don't know the exact time of loop. The loop end when some condition is reached when executing the loop. So it is also called **conditional loop**.

> When choosing implementing statements, we choose for-statement for counter-controlled loop, while-statement for event-controlled loop and do-while-statement for loops that have to be implemented at least once.

4.5 Examples of Iteration

> The problems below might be easy, so try your best to make your solution easier and simpler.

Fibonacci

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin >> n;
    int fib_1 = 1, fib_2 = 2; //the first two fibonacci number
    for (int i = 3; i <= n; i++){
        int temp = fib_1 + fib_2; //compute the next fibonacci number
        fib_1 = fib_2; //remember last fibonacci number;
        fib_2 = temp; //remember the new fibonacci number;
    }
    cout << "The " << n << "th fibonacci number is: " << fib_2 << "." << endl;
    return 0;
}
```

> The body for the iteration can also be written like this:
>
> ```cpp
> fib_2 = fib_1 + fib_2;
> fib_1 = fib_2 - fib_1;
> ```

Newton's Iteration Method For Cubic Root of 'a'

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    const double EPS = 1e-6; //EPS means epsilon, which is a very small number.
    double a, x1, x2; //x1, x2 is used to store x(n) and x(n+1)
    cout << "Please input a number: ";
    cin >> a;
    x2 = a; //the first number is a
    do{
      x1 = x2; //remember the previous number
      x2 = (2 * x1 + a / (x1 * x1)) / 3; //compute the new number
    } while (fabs(x1 - x2) >= EPS);
    cout << a << "'s cubic rooy is " << x2 << " ." << endl;
    return 0;
}
```

Output All the Prime Numbers No Larger Than 'n'

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    int n, count = 1;
    cin >> n; //input a number from the keyboard
    if (n <= 2) return -1;
    cout << 2 << ", "; //cout the first prime number

    for (int i = 3; i <= n; i += 2) { //even number can't be prime number, so we just need to test the odd numbers.
        int j = 2, k = (int)sqrt((double)i); //we don't need to test every number less than i, try everything to make the steps less
        while (j <= k && i % j != 0) j++;
        if (j > k) {
            cout << i << ", ";
            count ++;
            if (count % 6 == 0) cout << endl; //make sure each line have 6 numbers
        }
    }
    cout << endl;
    return 0;
}
```

Sum 1 + x + x^2/2! + x^3/3! + x^4/4! + ... + x^n/n!

```cpp
#include <iostream>
using namespace std;
int main(){
    double x;
    int n;
    cin >> x >> n;
    double item = x, //item is used to store each item of the series
          sum = 1 + x; //sum stores the sum, initialized as the summation od the first two items
    for (int i = 2; i < n; i++){ //compute each item and add them to the sum
        item *= x / i; //compute the next item
//inatead of compute the factorial and power separately, using recursion is more convenient
        sum += item;
    }
    cout << "x = " << x << ", n = " << n << ", sum = " << sum << endl;
    return 0;
}
```

5. Unconditional Jump

5.1 Goto Statement

```
goto <statement-label>;
......
<statement-label>: <statements>; //here statements is called labeled-statement
```

> `goto` means jumping to the labeled-statement corresponding to its statement-lebel.
>
> Remember that we're not allowed to use `goto` to jump through function definition or over variable definition.
>
> And we seldom use `goto` to jump from outside a compound statement to inside a compound statement.

5.2 Break Statement

```
break;
```

> **Break statement** has two main functions:
>
> 1. End a branch of the switch statement;
> 2. End the iteration body containing it.

- Break statement only jumps out of the inner layer of iteration. If you want to jump out of multiple layers of iteration, please use goto statement or use a **flag**, which is a variable placed along with loop condition using `&&`.

5.3 Continue Statement

```
continue;
```

> **Continue statement** can only be used in the iteration body, which means end the current loop and jump to the next loop.

6. Programming Style

> Usually, when we judge a program, in addition to **correctness** and **efficiency**, **readability** is also a quite important criteria, which makes the program easy to maintain and indirectly influenced its correctness.

6.1 Structured Programming, SP

> **Structured Programming** refers to a series of programming method which improves the readability and maintenance.
>
> It demands the program itself is well-structured as well as the process of designing the program.

To a program, well-structured refers to:

- Each program should feature only one entrance and only one exit.
- There's no statements in the program which will never stop implementing, which means the program must exit in a limited time.
- There's no useless statements in the program, which means each statement in the program should be implemented or at least has a chance to be implemented.

6.2 About `goto`

> `goto` is the most primitive and classic control-statement, which is widely used in the early programs. But it will destroyed the one-entrance-one-exit feature of the program if used improperly, which betrays the principle of structured programming.
>
> And it has been proved that goto-statement is not necessary and irreplaceable, so we should try our best to avoid usage of goto-statement to move **forward** or **backward**.

Chapter 04 Procedural Abstraction - Function

1. General Description

> When people design a complex program, they usually uses the method of **function decomposition** and **function mix**.

- **Function decomposition** refers to designing a program in a **top-down** and **step-wise** way, which means dividing a large goal into several smaller ones and dividing these smaller ones into some tiny ones if possible until we can solve the problem easily.
- **Function mix** refers to apply the outcome of each atomic problems to finally get to the initial goal in a **bottom-up** way.

> In the process of decomposition and mix, we need a system of abstraction, such as **procedural abstraction** and **functional abstraction**, which indicates that we don't have to know clearly how to do it in the every beginning, we just need to know what to do and what we'd like to achieve.

2. Sub-Program

2.1 What is Sub-program?

> Sub-program is a block of code that has its unique name and in the program, where these code is need will be placed the name. And this practice will help us avoid repeating many unnecessary codes as well as accomplishing procedural abstraction.
>
> In addition to procedural abstraction, sub-program also plays a role of **encapsulation** and **information hiding**.

2.2 Data Transfer Between Sub-programs

> When defining a function, we need to set its **parameters** and when making a function call, we need to input some **arguments** according to its parameters, in which process we transfer data to the function and get a return value.

> The most elementary two ways are **call-by-value** and **call-by-reference**, the latter one is more faster and efficient but will have some side effects.

3. Function

3.1 Function Definition

```
<type-of-return-value> <function-name> (<parameter-list>) <function-body>
```

- `<type-of-return-value>` describes the data type the function will return. If it is void, the function won't return anything.
- `<function-name>` is the name of the function, which is a identifier.
- `<parameter-list>` contains the names of parameters, in the form of `<data-type> <parameter-name>` separated by commas.
- `<function-body>` is a compound statement which usually contains a **return statement** which means end the function and return the value of the expression comes after it.

> Notice that we are not allowed to jump out of the function body using `goto` statement.
>
> Remember that each C++ program must have a function called `main`, which describes the whole procedure of the program and call some other functions help solve the problem.

3.2 Function Call

```
<function-name> (<argument-list>)
```

> Argument-list as a list of arguments separated by commas and correspond to the definition of the function.

The execution rule for function call is that:

- Evaluate the arguments and get the values of them while which is evaluated is not regulated.
- Pass the value of each argument to the corresponding parameter.
- Execute the function body.
- Return the value when encountering the return statement if there is one.

> Notice that a function call is also an expression in which the function name works as an operator and arguments works as operands.
>
> As to a function call that has no return value, we must match it with a `;`, so that we get a statelemt.

3.3 Function Delcaration

```
<type-of-return-value> <function-name> (<parameter-list>);
```

> A function must be defined or declared before we call it since C++ program is top-down.
>
> We use **function prototype** to declare a function and we can just list types of parameters in the parameter-list without declaring their name.

3.4 Call-by-Value

Now we'll use an example to demonstrate how values are passed to a function and returned.

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

double power(double x, int k){
    if (x == 0) return 0;
    double product = 1.0;
    if (n >= 0)
        while (n > 0){
            product *= x;
            n--;
        }
    else
        while (n < 0){
            product /= x;
            n++;
        }
    return product;
}

int main(){
    double a = 3.0, c;
    int b = 4;
    c = power(a, b);
    cout << a << "," << b << "," << c << endl;
    return 0;
}

```

1. When executing `main`, the memory will allocate space to variable `a`, `b` and `c`;
2. When calling function `power`, the memory will also allocate space to `x`, `n`, `product` and use `a`, `b`, `1.0` to initialize them;
3. When function `power` ends its execution, `x = 3.0; n = 0; product = 81.0`;
4. The return value `81.0` will be passed to `c` and the space of `x`, `n`, `product` will be released.

> From the process above we can conclude that the arguments `a` and `b` won't be affected during the function call, which indicates that call-by-value will do nothing to the outside scope and have no side effects other than output which actually is not a side effect in C++ programs.

3.5 Global Variable and Local Variable

> In addition to abstraction, function also features the role of encapsulation, which means everything in the function body is invisible to outside the function body, which is also called information hiding.

- **Local Variable** refers to variables defined in a compound statement, which means this variable is only effective in this compound statement.

> Function body is also a compound statement, so the parameters are only effective in the function body.
>
> Local variable shows the function's feature of encapsulation and information concealment, which indicates that variables in different scopes could have the same name.

- **Global Variable** refers to the variables defined outside any function including the `main` function, which means they can be accessed and used by any function in this program.

  > The global variable must be defined before we use it or we should give it a declaration.
  >
  > ```
  > extern <data-type> <variable-name>;
  > ```
  >
  > Variable definition is also a kind of declaration called definitional declaration in comparison with non-definitional declaration.
  >
  > The difference between variable definition and variable declaration is that a variable definition will allocate some space from memory to the variable and give it a initial value if you want, while a variable declaration does nothing but give just a declaration or an announcement.
  >
  > Be careful when using global variable because it can be accessed and modified by any function through function side-effect, which provides an opportunity that what one function does to the global variable will disturb the smooth and correct implementation of other functions.

4. Scope and Lifetime

4.1 Multi-Module Structure

- A **module** is made up of several relevant definitions of program bodies. And a module usually consists of two parts:
  - **Module interface** gives some definition and declaration on functions and variables that will be used in other modules and we store this part of codes in a **header file** with `.h` as a filename extension.
  - **Module implementation** is the main part and we store these codes in a **source file** with `.cpp` as a filename extension. And if we want to use a module interface, we need to use **compile preprocessing commands** like below.

    ```
    #include <file_name.h>
    ```

- The basic principle for designing modules is that we should increase **cohesion** inside the module as much as we can and decrease **coupling** between modules as much as possible, which is based on structured programming.

4.2 Scope for Identifier

name	meaning
local scope	only effective from the definition to the end of the compound statement including it
potential scope	a local scope excluding another inner local scope with the same name
global scope	a kind of linkage that can be used globally and if we want to use it in the inner local scope which has the same name, we need to put a **global scope resolution operator** `::` before it.
file scope	if we want a variable only effective in this file, we can put `static` before its definition.
function scope	a variable defined in a function can only be accessed in this function body
function prototype scope	the parameter of a function declaration in only effective in this function prototype

4.3 Namespace

```
namespace <name-of-namespace> {
    <definition-of-global-variables/constants/functions>;
}
```

> If we want to use the name in a namespace outside this namespace, we should use namespace + **scope resolution operator** `::` to restrict it.
>
> And if we want use names from one namespace from now on, we could use the following instruction:
>
> ```
> using namespace <name-of-namespace>;
> ```
>
> Moreover, the following two bricks of code is the same:
>
> ```
> namespace {//a namespace without a name can only be used in this file
>  int x,y;
> }
> //-----------
> static int x, y;
> ```

4.4 Lifetime and Memory Allocation

> Lifetime of a variable refers to how long this variable occupies part of the memory.

- **Static lifetime**
    - Variables with static lifetime will get a room in the **static data** part of the memory in the beginning of the program which won't be withdrawn until the end of the program.

    - The keyword `static` has two meanings:
        - Change the scope of a global variable into a file scope
        - Give a local variable a static life time, which means this local variable has the features of a global variable to this function. That is to say, you can still use the value at the time of last function call of this variable in the current function call because it is preserved during multiple calls.
- **Automatic lifetime**
    - Variables with automatic lifetime get a room in the **stack** part of the memory only when it is defined and the room will be withdrawn when reaching the end of the compound statement containing it, that is to say when its lifetime in the local scope is over.
    - A local variable's default lifetime is automatic, but we can also use keyword `auto` to assign it explicitly.
- **Dynamic lifetime**
    - The lifetime of **dynamic variables** is begin with keyword **new** or function **malloc** and end with keyword **delete** or function **free**.
    - Variables with dynamic lifetime will be given a room in the **heap** part of the memory also called **free store**.

> There's another area in the memory called **code**, which stores the instructions. And remember that each part in the memory has maximum size which is relevant to the hardware of the computer.
>
> The keyword `register` means to store the variable in the register of CPU, which makes accessing to it much faster.

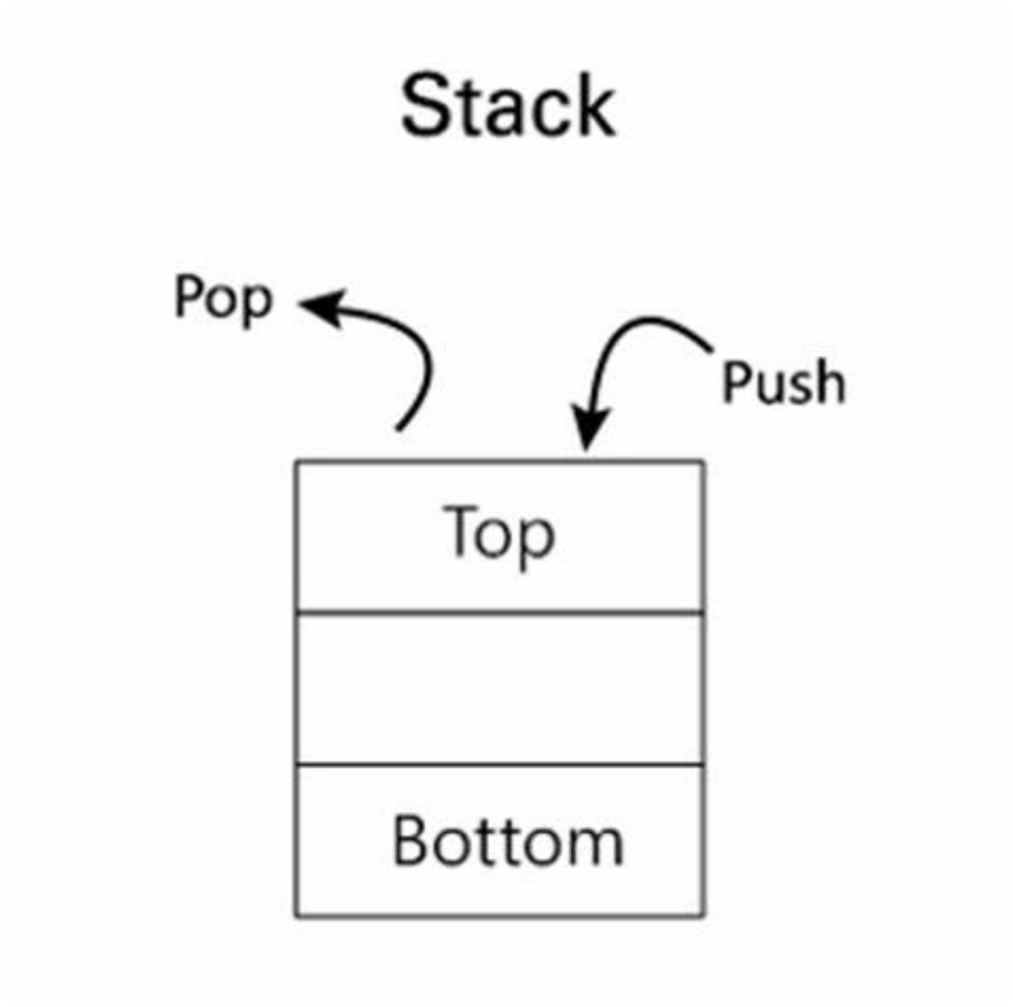
Example: Linear Congruential Method for Random Number Production

> r[k] = (multiplier * r[k - 1] + increment) % modulus
>
> r[k - 1] is the previous random number, r[k] is the current random number. Modulus points out the range of random numbers, if we choose multiplier and increment carefully and properly, we'll get quite well-versed random sequence.
>
> The next random number is based on the last random number, so it is necessary to conserve that.

```
unsigned int random(){
    static unsigned int seed = 1;
    const int modulus = 65536, multiplier = 25173, increment = 13849;
    seed = (multiplier * seed + increment) % modulus;
    return  seed;
}
```

4.5 Function Call Based on Stack

> Stack is a liner data structure whose number of elements is editable and the addition and delete of its elements can only be done on one end of the stack. Stack has an important feature: **Last In First Out (LIFO)**.



- You can just compare stack in C++ to environment paradigms in Python.

5. Recursive Function

> A recursive function is a function that calls itself directly or indirectly in its body.

5.1 Divide and Conquer

> When defining a recursive function, we must clarify **general case** and **base case**.

Example: Fibonacci

```cpp
int fib(int n) {
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

> When choosing between **iteration** and **recursion**, if there isn't many steps, recursion can reduce code lines while there is may steps, maximum recursion depth will be exceeded so we have to choose iteration.

> However, when we lean further, we can solve the problem of maximum recursion depth exceeding by **dynamic programming**.

Example: Power

```cpp
double power(double x, int n){
    if (x == 0) return 0;
    if (n == 0)
        return 1;
    else if(n > 0)
        return x * power(x, n - 1);
    else
        return 1 / power(x, -n);
}
```

Example: Greatest Common Divisor

```cpp
int gcd(int x, int y){
    return (y == 0) ? x : gcd(y, x % y);
}
```

Example: Hanoi

```cpp
void hanoi(char x, char y, char z, int n) {
    if (n == 1)
        cout << "1: " << x << "->" << y << endl;
    else {
        hanoi(x, z, y, n - 1);
        cout << n << ": " << x << "->" << y << endl;
        hanoi(z, y, x, n - 1);
    }
}
```

5.2 Standard Library

> Some bonus from the language designer and developer.

6. Further Discussion on C++ Functions

6.1 Macro with Parameters

```cpp
#define <macro-name> (<parameter-list>) <expression>
//example:
#define max(a, b) (((a) > (b)) ? (a) : (b))
```

> Macro is inherited from C and can solves the problem of inefficiency of frequent call to small functions.

> The implementing rule for macro with parameters is direct substitution without any checking, which might bring some problems.

6.2 Inline Function

```
inline <a-function-definition>
```

> Inline function is defined just put the keyword `inline` before the regular function definition. Which has similar function to macro but will check the data type, which is safer than macro.
>
> But the keyword `inline` just suggest the compiler to deal with when compiling, whether the suggestion is going to be taken depends on the compiler.
>
> And we usually put the definition of inline function in the header file.

6.3 Parameters with Default Value

> We can give the parameter a default value when defining or declaring it, but there are some rules to follow:
>
> - Assign default value from right to left;
> - A parameter can only be assigned one default value in the same file.

6.4 Function Overloading

> In C++, the two function can share the same name if they've got some difference in parameters in terms of number of parameters or their data types, which is called **function overloading**.

6.5 Anonymous Function - Lambda Expression

```
[<environment-announcement>] (<parameter-list>) -> <return-value-type> <function body>;
//example: call expression include a lambda expression
int n = [] (int x, int y)->int {return x + y;} (3, 4);
```

About the environment-announcement:

- ` `: We can't use variables in the outer scope;
- `&`: We can use variables in the outer scope as a quota, which means you can modify them;
- `=`: We can use variables in the outer scope by value, which means you can't modify them.

> More information will be talked later when we discuss passing in a function as an argument.

Chapter 05 Compound Data Type - Constructed Type

1. Enumeration Type

- **Type Construction**

```
enum <enumeration-type-name> {<enumerration-value-list>};
```

> Enumeration type name is an identifier, which identifies the name of an enumeration type, which can be left out.
>
> Enumeration value list is a list of values separated by commas, and the values in it are integral symbolic constants, and each of them has a corresponding integer value which by default start from 0.
>
> Enumeration variable definition:
>
> ```
> <enumeration-type-name> <variable-list>;
> //or
> enum <enumeration-type-name> <variable-list>;
> ```
>
> We can also define a enumeration type as well as defining a enumeration variable.
>
> ```
> enum Day {SUN, MON, TUE, WED, THU, FRI, SAT} d1, d2, d3;
> ```
>
> If we leave out the name of the enumeration type, we must define the variable just as we define the type or we'll never have access to the type again because it's anonymous.

- **Manipulation of Enumeration Type**
  - **Assignment**
    - We can't use assignment between variables of different enumeration types
    - If we assign an integer to a enumeration type variable, we assign the value in its value set corresponding to the integer. If there's none, an error will be raised.
  - **Comparison**
    - Compare the integer corresponding to it.
  - **Output**
    - We can't input a value to an enumeration variable
    - But we can output an enumeration value in the form of its corresponding integer.
    - So, if we want to input a enumeration value, there should be some special methods:

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

enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};

int main(){
    Day d;
    int i;
    cin >> i;
    switch (i){
```

```
            case 0: d = SUN; break;
            case 1: d = MON; break;
            case 2: d = TUE; break;
            case 3: d = WED; break;
            case 4: d = THU; break;
            case 5: d = FRI; break;
            case 6: d = SAT; break;
            default: cout << "Input Error!" << endl; return -1;
        }
        return 0;
    }
```

```
//another way
if (i < 0 || i > 6){
    cout << "Input Error!" << endl;
    return -1;
}
else
    d = (Day)i; //exeplicit type reversion
```

2. Array Type

> Array Type is a user-defined compound data type to describe a sequence of elements.

2.1 Single Dimensional Array

> **Single dimensional array** can describes data structure called **linear list**.

- **Definition of Single Dimensional Array**

```
<type-of-the-elements> <name-of-the-array> [<number-of-elements>];
```

> Here type of **elements** is called **base type**, and it can be any type in C++, atomic or compound, but not void. Name of the array is an identifier which can represent the array. Number of elements is a integral **constant** expression, remember: **constant**.

```
typedef <type-of-the-elements> <alias-of-the-array-type> [<number-of-elements>];
```

> The keyword `typedef` gives an alias to a certain kind of data, compound or atomic, and the we can refer to this type using alias and use it to define variables as we always do : `<data-type> <variable-name>;`.
>
> Notice that in a C++ program, the length of an array is fixed and unchangeable.

- **Initialization for Variables in Single Dimensional Arrays**

> When initializing an newly defined array, we use a pair of brace to include them, and the number of our initial value can't exceed the length of the array. If we haven't provided enough initial value, the other elements in the array will be initialized as 0.

> Moreover, we are allow to leave out the number of elements if we define an array whose length is exactly the number of initial values we give.
>
> Here are some examples:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int b[10] = {1, 2, 3, 4}; //The other element execpt the first four will be initialized
int c[] = {1, 2, 3}; //The length of c[] is 3.
```

- **Manipulation for Single Dimensional Arrays**
  - **get item**

    ```
    <array-name> [<index/subscript>];
    ```

    > We can have access to an element of an array with its name and a subscript giving the elements index.
    >
    > Remember that the elements of an array is indexed from `0` and don't exceed its maximum length.

  - **element manipulation**

    > Manipulation for an element of an array is not on the same scale as manipulation for the array.
    >
    > As to the elements, you can do what you can do to a normal variable to an element of the array such as assignment, self-increase or self-decrease, using it as an operand and so on.

  - We should use iteration or loop to travel the array so that we could get access to each element.

  > Arrays are not first class in C++ programs. They are discriminated, so if we want to do something to it, we must one element at a time.

- **How The Single Dimensional Array are Stored in The Memory**

  Let us use the following definition as an example:

  ```
  int a[10];
  ```

  > First, the memory will give the array ten rooms next to each other whose addresses is constant.
  >
  > Then, put each initial value in its corresponding room if there is one initial value.
  >
  > Moreover, we can get each room's address using the following formula:
  >
  > ```
  > <first-element-address> + <index> * sizeof(<base-data-type>)
  > ```
  >
  > Having a good understanding of how an array stored will help me understand how to use the **pointer** to manipulate an array better.

- **How to Pass a Single Dimensional Array to a Function**

> We usually need to pass the name of an array and its length of it to a function to manipulate it or refer to it.
>
> In the parameter list, the syntax for a single dimensional array is `<base-type> <name>[]`.
>
> Notice that the array or any other compound data type is called by reference, which means the function calling it could have some side effects.

2.2 Single Dimensional Character Array - A Implementation for Strings

> Each Character Array must end with `\0`, which marks the end of the string. The length of the string depends on the place of `\0` and its maximum length is the length of the array minus one because the last character must be `\0`.

- **Definition of a Character Array**

  ```c
  char s[10] = {'h', 'e', 'l', 'l', 'o', '\0'};
  char s[10] = {"hello"};
  char s[10] = "hello";
  char s[] = "hello";
  //The last three statements will add a '\0' in the end of the string by default.
  ```

- Manipulation of a character array is just the same as regular arrays.

Small Functions of Character Array

```c
int strlen(char str[]){
    int i = 0;
    while (str[i] != '\0') i++;
    return i;
}
```

```c
int str_to_int(char str[]){
    int n = 0;
    for (int i = 1; str[i] != '0'; i++)
        n = n*10 + (str[i] - '0');
    return n;
}
```

```c
int find_substr(char str[], char sub_str[]){
    int len = strlen(str), sub_len = strlen(sub_str);
    for (int i = 0; i <= len - sub_len; i++) {
        int j = 0;
        while (j < sub_len && sub_str[j] == str[i + j]) j++;
        if (j == sub_len) return i;
    }
    return -1;
}
```

```
void reverse(char str[]) {
    int len = strlen(str);
    for (int i = 0, j = len - 1; i < j; i++, j--){
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
```

2.3 Double Dimensional Array

> **Double dimensional array** describes a kind of data structure that has rows and columns, such as a matrix. Each element of it can be fixed exactly with its row index and column index.

- **Definition of a Double Dimensional Array**

  ```
  <element-type> <array-name> [<row-number>][<column-number>];
  //or
  typedef <element-type> <alias> [<row-number>][<column-number>];
  <alias> <array-name>;
  ```

  > If we give a single dimensional array type an alias and use it as the base type to construct another single dimensional array, actually this array is double dimensional.

- **Initialization of a Double Dimensional Array**

  > Just observe the examples below, I'm sure you'll find the rules and syntax.

  ```
  int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
  int a[2][3] = {1, 2, 3, 4, 5, 6};
  //the two statements above works the same
  int a[2][3] = {{1, 2}, {3, 4}};
  int a[2][3] = {1, 2, 3, 4};
  //the two statements above works differently
  //uninitialized elements will be assigned 0 by default
  int a[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  //the number of column can't be left out anyway while the number of rows can be left out.
  ```

- **Manipulation for Double Dimensional Array**
  - **get item**

    ```
    <array-name> [row-index][column-index]
    ```

  - As is always the case, array in C++ is not first class, so any manipulation should be done element by element.
  - So if we want to travel a double dimensional array, we should use nested iteration.

- **Store of Double Dimensional Array**

  > It is just the same as the single dimensional array if we see it as a single dimensional array with its base type to be single dimensional array.
  >
  > If you understand the above, all nth dimensional arrays can be viewed as a special kind of single dimensional array.

- **Pass a Double Dimensional Array to a Function**

  > In the parameter list, a double dimensional array is described as `<base-type> <array-name>[][column-number]`.
  >
  > We should leave out the number of rows as we preserve the number of columns.
  >
  > We can't leave out the number of rows because the address of `a[i][j]` is computed by the following formula:
  >
  > ```
  > addr(a[i][j]) = addr(a[1][1]) + i * number-of-columns + j
  > //addr(a[1][1]) is stored in the name a.
  > ```
  >
  > We must the number of columns to help get the item.

2.4 Application

- **Matrix Multiplication**

```
#include <iostream>
using namespace std;
int main(){
    const int M = 2, N = 3, T = 4;
    int a[M][N], b[N][T], c[M][T];
    cout << "Please input matrix A(" << M << " * " << N << "): " << endl;
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            cin >> a[i][j];
    cout << "Please input matrix B(" << N << " * " << T << "): " << endl;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < T; j++)
            cin >> b[i][j];
    for (int i = 0; i < M; i++) {
        for (int j = 0; i < N; j++) {
            c[i][j] = 0;
            for (int k = 0; k < N; K++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
    cout << "The multiplication of matrix A and B is C(" << M << " * " << T << "):"
<< endl;
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < T; j++)
```

```
            cout << c[i][j] << " ";
      cout << endl;
    }
    return 0;
}
```

- **Josephus Problem**

```
#include <iostream>
using namespace std;
const int N = 20, M = 5;

int main(){
    bool in_circle[N];
    int num_of_children_remained = N, index;
    for (index = 0; index <= N - 1; index++)
        in_circle[index] = true;
    index = N - 1;
    while (num_of_children_remained > 1) {
        int count = 0;
        while (count < M) {
            index = (index + 1) % N;
            if (in_circle[index])
                count++;
        }
        in_circle[index] = false;
        num_of_children_remained--;
    }
    for (index = 0; index < N; index++)
        if (in_circle[index]) break;
    cout << "The winner is No." << index << ". \n";
    return 0;
}
```

- **Selection Sort**

```
void sel_sort(int x[], int n) {
    for (; n > 1; n--) {
      int i_max = 0;
      for (int i = 1; i < n; i++)
          if (x[i] > x[i_max]) i_max = i;
        int temp = x[i_max];
        x[i_max] = x[n - 1];
        x[n - 1] = temp;
    }
}
```

- **Quick Sort**

```c
int split(int x[], int first, int last){
    int split_point = first, pivot = x[first];
    for (int unknown = first + 1; unknown <= last; unknown++)
        if (x[unknown] < pivot) {
            split_point++;
            int temp = x[split_point];
            x[split_point] = x[unknown];
            x[unknown] = temp;
        }
    x[first] = x[split_point];
    x[split_point] = pivot;
    return split_point;
}

void quick_sort(int x[], int first, int last) {
    if (first < last) {
        int split_point = split(x, first, last);
        quick_sort(x, first, split_point - 1);
        quick_sort(x, split_point + 1, last);
    }
}
```

3. Structure Type

> **Structure type** is a kind of compound data type constructed by the user, which is composed of several members known as its **member**.

3.1 Definition of Structure Type

```c
struct <name-of-structure-type> {<members-list>};
```

> Name of structure type is an identifier, and member list is description for each member, which is quite similar to variable definition. The data type of the structure's members can be any data type except void and itself. Different structure type can have member of the same name.
>
> Definition for the variable of a structure type is just as usual.
>
> ```c
> <type-name> <variable-name>;
> ```
>
> **Initialization** for a structure type is quite similar to initialization for an array. The example below will perfectly demonstrate how to define and initialize a structure type variable. Remember that we can't initialize a member of a structure type when defining it because memory will not allocate room when we define a data type, which means there's no room to store the initial value.
>
> Moreover, of course we can define a structure type without name, but if so we have to define some variables at the same time or the definition of structure type is useless.

```c
enum Sex {MALE, FEMALE};
```

```
struct Date {
  int year, month, day;
};
enum Major {MATHEMATICS, PHYSICS, CHEMISTRY, COMPUTER, GEOGRAPHY, ASTRONOMY, ENGLISH, CHINESE, PHILOSIPHY};
struct Student {
    int no;
    char name[20];
    Sex sex;
    Date birth_date;
    char birth_place[40];
    Major major;
};
Date today, yesterday, some_date;
Student monitor, best_student;
Student some_student = {2, "Jacy", Male, {2002, 5, 1}, "Yancheng", COMPUTER};
```

3.2 Manipulation for Structure Type

- **Access to Members**

  ```
  <variable-name>.<member-name>
  ```

  > **Dot expression** is a universal kind of expression used for access to members of a structure type variable or attributes of an instance or an object. The following examples are legal and via observation you'll find the rules.

  ```
  best_student.no = 1;
  strcpy(best_studenr.name, "Jerry");
  best_student.sex = MALE;
  best_student.birth_date = some_date;
  strcpy(best_student.birth_place, "Nanjing");
  best_student.major = COMPUTER;
  ```

- **Assignment for Structure Type**

  > Variables of the same structure type can be assigned to each other.

- **Passing a Structure Type to a Function**

  > Just the same as usual: `<type-name> <parameter-name>` in the function signature and `<argument-name>` in the function call.

- **How a Structure Type Stored in the Memory**

  > Just like an array, while the only difference is that the sizes of rooms are various according to the types of its members and we use dot expression instead of subscript to get access to them.

3.3 Application of Structure Type

- **Name-table and its Research**

```
const int NAME_LEN = 20;
const int TABLE_LEN = 100;
struct TableItem { //element-type of a name-table
    char name[NAME_LEN]; //key
    ...... //other information
} ;
TableItem name_table[TABLE_LEN]; //name-table
```

```
#include <cstring>
using namespace std;
int linear_search(char key[], TableItem t[], int num_of_items){
     int index;
    for (index = 0; index < num_of_items; index++)
        if (strcmp(key, t[index].name) == 0) break;
    if (index < num_of_items)
        return index;
    else
        return -1;
}
```

```
#include <cstring>
using namespace std;
int binary_research(char key[], TableItem t[], int num_of_items) { //if the nametable has been sorted by key in advance
    int index, first = 0, last = num_of_items - 1;
    while (first <= last) {
        index = (first + last) / 2;
        int r = strcmp(key, t[index].name);
        if (r == 0)
            return index;
        else if (r > 0)
            first = index + 1;
        else
            last = index - 1;
    }
    return -1;
}
```

4. Union Type

> **Union Type** is a kind of compound data type that can represent several types of data. As we known, each data has its unique type and the memory will allocate to it a corresponding kind of room with appropriate size and whatever needed. While the union type is composed of several base types, the memory will give it a room with the size of the maximum of these base types so that each of them can live in it but only one of them will actually live in it.
>
> So we can consider the union type as a multi-functional type but can just run one of its all functions.

Basic manipulation for the union type is just like former types that have been introduced, so I'll just give an example to help you confirm your assumption.

```
struct Line{
    double x1, y1, x2, y2;
};
struct Rectangle{
    double left, top, right, bottom;
};
struct Circle {
    double x, y, r;
};
union Figure {
    Line line;
    Rectangle rect;
    Circle circle;
}
const int MAX_NUM_OF_FIGURES = 100;
Figure figures[MAX_NUM_OF_FIGURES];
//representing a group of figures including line, rectangle and circle.
```

5. Pointer Type - Description of Address in Memory

> **Pointer Type** is a user-defined data type used to describe addresses in the memory. As has been introduced before, we can call anything in the program either by its name or by its address, and pointer type is how we describe the address.

5.1 Definition and Initialization of A Pointer

```
<base-type> *<pointer-variable>;
```

> Here `*` is a symbol that indicates that the variable here is a pointer type.
>
> The base type of a pointer tells what kind of data this pointer can point to.
>
> If the base type of a pointer is `void`, it is a general pointer and can point to any data type.
>
> A pointer is also a kind of data, so it needs a room in the memory too, which also means that a pointer itself can be pointed to by another pointer.

```
<base-type> *<pointer> = &<variable>;
<base-type> *<pointer> = <another-pointer>;
```

> Operator `&` gets the address of a variable and we can assign it to a pointer type whose base type is the type of this variable.
>
> Or we can just use another pointer with the same base type to do the initialization.
>
> On the other hand, there exists an operator `*` which can get the value stored in an address.
>
> We use the symbolic constant `NULL` to represent an empty pointer, which is the macro of 0 and can represent any type of pointer that is empty.

5.2 Manipulation for Pointer Type

- **Assignment**

  ```
  <pointer> = <another-pointer>;
  ```

  > A pointer can be assigned to another pointer if it is NULL or if they share the same base type.

  ```
  <pointer> = &<variable>;
  ```

  > Put the address of the variable into the pointer.

- **Access**

  ```
  *<pointer>
  ```

  > The expression above evaluates to the value in the address that the pointer points to.
  >
  > If the pointer points to a structure type, we can use the following two ways to point to its members.

  ```
  (*<pointer>).<member>
  //or
  <pointer>-><member>
  //the two ways above are exactly the same
  ```

- **Computing**

  ```
  <pointer>[<integral-expression>] <=> *(<pointer> + <integral-expression>)
  <pointer> + <integer> = <address> + <integer> * sizeof(<base-type>)
  ```

  > When we use a pointer to add or minus an integer, it means to move the pointer several rooms forward or backward, and the size of the room depends on its base type.

  ```
  <pointer1> - <pointer2> = (<address1> - <address2>) / sizeof(<base-type>)
  ```

> The offset of two pointer describes how many rooms are there between the places these two pointers point to.
>
> And if we compare two pointer, we compare their offset with zero.

- **Output**

  > If we output a pointer, we output the address it points to except the pointer of a character array, in which case we output the whole string.
  >
  > If you want to output the value, use the `*` operator.
  >
  > If you want to output the address a character pointer points to, use explicit type conversion.

> A pointer can also play the role of parameter or return value. If a pointer is passed to a function as an argument, the function is call by reference, which means it could have side effects.

5.3 Dynamic Variable

1. Creation

```
new <type-name>
```

> Keyword **new** allocates room in the heap to a dynamic variable and return its address as a pointer, and if you want to use it later, you must use a pointer to store it.

```
new <type-name> [integral-expression]...[integral-expression]
```

> The expression above create a dynamic array and return the address of its first element.

```
#include <cstdlib>
void* malloc(unsigned int size);
```

> Built-in function `malloc` allocated a room of input size in the heap and return its start address as a general pointer, which needs a explicit type reversion to your target type. Here are some examples.

```
int *p1 = (int *)malloc(sizeof(int)); //create a int dynamic variable
int *p2 = (int *)malloc(sizeof(int) * n); //create a int dynamic array with n rooms
```

2. Revoke

> Usually, we use keyword `delete` to revoke dynamic variables created by keyword `new` and function `free` to revoke dynamic variables created by function `malloc`.

```
delete <pointer>;
//revoke the dynamic variable pointed by the pointer
delete []<pointer>;
//revoke the dynamic array pointed by the pointer
#include <cstdlib>
free(<pointer>);
//revoke the dynamic variable or array pointed by the pointer
```

> Notice that if you want to revoke a dynamic array, the pointer must point to its first element.
>
> When we have revoked a dynamic variable, the pointer is still there pointing to its room which is empty now. So the pointer has become a **dangling pointer**.
>
> Moreover, if there's a dynamic variable has not been revoked and the pointer previously pointing at it has pointed somewhere else or ended its lifetime, the dynamic variable here will become an orphan - no longer able to be accessed while continue to occupy the room, which is a waste of the memory. This phenomenon is called **memory leak**.

3. Application - Dynamic Array

> Use dynamic array is a method to solve the problem that the regular array must be defined with fixed length.

```
//if we know the number of numbers
int n, *p;
cin >> n;
p = new int[n];
for (int i = 0; i < n; i++) cin >> p[i];
sort(p, n);
delete []p;
```

```
//if end with a flag
int max_len = 20, count = 0, n;
const int INCREMENT = 10;
int *p = new int[max_len];
cin >> n;
while (n != -1) {
    if (count >= max_len) {
        max_len += INCREMENT;
        int *q = new int[max_len];
        for (int i = 0; i < count; i++) q[i] = p[i];
        delete []p;
        p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p, count);
```

```
delete []p;
```

4. Application - Linked List

> **Linked List** is a linear data structure, whose implementation is a structure type containing a pointer whose base type is itself as its member. The data is stored in the structure type's other members.

```
struct Node {
    int data;
    Node *next;
};
```

- **Create a Linked List by Inserting Nodes into the Head**

```
const int N = 10;
Node* InsCreate(){
    Node* head = NULL;
    for (int i = 0; i < N; i++){
        Node* p = new Node;
        p->data = i; //cin >> p->data; is also Okay.
        p->next = head;
        head = p;
    }
    return head;
}
```

> Notice that `InsCreate` creates a linked list with a reversed order contrary to your input order.
>
> If you want the regular order, try the following method.

- **Create a Linked List by Inserting Nodes into the Tail**

```
const int N = 10;
Node* AppCreate(){
    Node* head = NULL,* tail = NULL;
    for (int i = 0; i < N; ++i){
        Node* p = new Node;
        p->data = i; // cin >> p->data;
        p->next = NULL;
        if (head = NULL)
            head = p;
        else
            tail->next = p;
        tail = p;
    }
    return head;
}
```

- **Output of a Linked List - Travel of the Linked List**

```
void PrintList(Node* head) {
    if (!head)
        cout << "List is empty. \n";
    else {
        while (head) {
            cout << head->data << " ";
            head = head->next;
        }
        cout << endl;
    }
}
```

- **Insert a Node into a Linked List**

```
void InsertNode (Node* head, int i) {
    Node* current = head;
    int j = 1;
    while (j < i && current->next != NULL) {
        current = current->next;
        ++j;
    }
    if (j == i) {
        Node* p = new Node;
        cin >> p->data;
        p->next = current->next;
        current->next = p;
    }
    else
        cout << "There's no ith node. \n";
}
```

- **Delete a Node of a Linked List**

```
Node* DeleteNode(Node* head, int i) {
    if (i == 1) {
        Node* current = head;
        head = head->next;
        delete current;
    }
    else {
        Node* previous = head;
        int j = 1;
        while (j < i - 1 && previous->next != NULL) {
            previous = previous->next;
            ++j;
        }
        if (previous->next != NULL) {
            Node* current = previous->next;
            previous->next = current->next;
            delete current;
        }
        else
            cout << "There's no ith node. \n";
    }
    return head;
}
```

- **Delete the Whole List**

```
void DeleteList(Node* head) {
    while (head) {
        Node* current = head;
        head = head->next;
        delete current;
    }
}
```

- **Insertion Sort**

```
Node* Insert(Node* h, Node* p) {
    if(p->data < h->data) {
        p->next = h;
        h = p;
        return h;
    }
    Node* cur = h;
    Node* prev;
    while (cur) {
        if(p->data < cur->data)
            break;
        prev = cur;
        cur = cur->next;
    }
    p->next = prev->next;
    prev->next = p;
    return h;
}

Node* InsSort(Node* head) {
    if (head == NULL || head->next == NULL)
        return head;
    Node* cur = head->next;
    head->next = NULL;
    while (cur) {
        Node* prev = cur;
        cur = cur->next;
```

```c
            head = Insert(head, prev);
        }
        return head;
}
```

> When dealing with problems concerning linked lists, pay attention to the following conditions:
>
> - an empty linked list.
> - a linked list with only one node.
> - manipulating the first node
> - manipulating the last node
> - Next of the last node is NULL.
> - The pointer manipulating the linked list has pointed to the last element.
>
> And we must reserve a head pointer because the linked list is single-directional.

5.4 Pointer and Array

- The name of an array is **a constant pointer** pointed at the first element of the array whose value is the address of the first element of an array.
- Arrays with more than one dimensions can be vied as single dimensional arrays as has been introduced before and we can use higher order pointer to manipulate them since pointer is also a data type which they can also be pointed at.
- Some times `main` function also has its arguments.

5.5 Pointer for a Function

- **Definition**

  ```c
  <return-type> (*<pointer>) (<parameter-list>);
  //or
  typedef <return-type> (*<type-name>) (<parameter-list>);
  <type-name> <pointer>;
  ```

- **Call using a Pointer**

  ```c
  (*<function-pointer>)(<argument-list>);
  //or
  <function-pointer>(<argument-list>);
  ```

- **Assignment**

  ```c
  fp = &f; //we can use & to get the address of a function
  //or
  fp = f; //the compiler will convert the name of a function into an address implicitly
  ```

Example:

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

const int MAX_LEN = 8;
typedef double (*FP) (double);
FP func_list[MAX_LEN] = {sin, cos, tan, asin, acos, atan, log, log10};

int main(){
    int index;
    double x;
    do{
        cout << "Please input the function you want to compute (0:sin 1:cos 2:tan 3:asin" << endl;
        cout << "4:acos 5:atan 6:log 7:log10):";
        cin >> index;
    } while (index < 0 || index > 7);
    cout << "Please input the argument:";
    cin >> x;
    cout << "The outcome is:" << (*func_list[index])(x) << endl;
    return 0;
}
```

- **Pass a function to another function**

> Juse refer to the higher-order function we've learnt in Python, and combine it with call-by-reference, you will understand the working principle for the functional pointer passing a function. The following example will show the process.

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

const int N = 1e6;

double Integrate(double (*pfun)(double x), double x1, double x2) {
    double s = 0;
    int i = 1, n;
    while(i <= n){
        s += (*pfun)(x1 + (x2 - x1) / n * i);
        ++i;
    }
    s *= (x2 - x1) / n;
    return s;
}
double square(double x){
    return x * x;
}

```

```cpp
int main(){
    cout << Integrate(cos, 1, 2) << endl;
    cout << Integrate([](double x)->double {return x * x}, 0, 1) << endl; //lambda expression for anonymous function
    cout << Integrate(square, 0, 1) << endl;
    return 0;
}
```

> Since we've make it clear that pointer is also a kind of data type, it is stored in the memory with a unique address, which indicates that we can use anothor pointer to point at it and use the third pointer to point at this one. As a result, we will get higher-order pointer.

6. Reference Type

> **Reference Type** doesn't have a room in the memory, it is just an alia of a currently existing variable, which makes refering to the original variable easier and gives a function the ability to make differences to the outside variable passed in as a reference.

- **Definition and Initialization**

  ```cpp
  <base-type> &<reference-variable-name> = <variable>;
  //We must initialize the reference variable when we define it.
  ```

  ```cpp
  int x = 0;
  int &y = x;
  //x and y are exactly the same variable
  ```

> Reference type is faster and more direct than pointer type and it is also called **hidden pointer**, which has similar functions to the pointer type.