C++

极夜酱

目录

1 Hello World! 1

1.1 Hello World! 1

1.2 数据类型 5

1.3 输入输出 8

1.4 表达式 12

2 分支 16

2.1 逻辑运算符 16

2.2 if 18

2.3 switch 22

3 循环 24

3.1 while 24

3.2 for 30

3.3 break or continue? 36

4 数组 39

4.1 数组 39

4.2 字符串 46

4.3 string 54

5 函数 56

5.1 函数 56

5.2 作用域 61

5.3 默认参数 64

5.4 递归 66

6 预处理 74

6.1 预处理 74

6.2 多文件编译 78

7 指针 **82**

7.1 指针 82

7.2 指针与数组 87

7.3 指针与字符串 91

7.4 二级指针 93

7.5 引用 97

7.6 动态内存申请 99

8 文件 **105**

8.1 文件 105

8.2 文件 I/O 107

8.3 fstream 116

9 结构体 **118**

9.1 枚举 118

9.2 联合体 119

9.3 结构体 120

10 面向对象 **123**

10.1 封装 123

10.2 构造函数 128

10.3 友元 133

10.4 运算符重载 135

10.5 继承 138

10.6 抽象类 144

10.7 多态 147

11 异常 **151**

11.1 异常 151

11.2 自定义异常 155

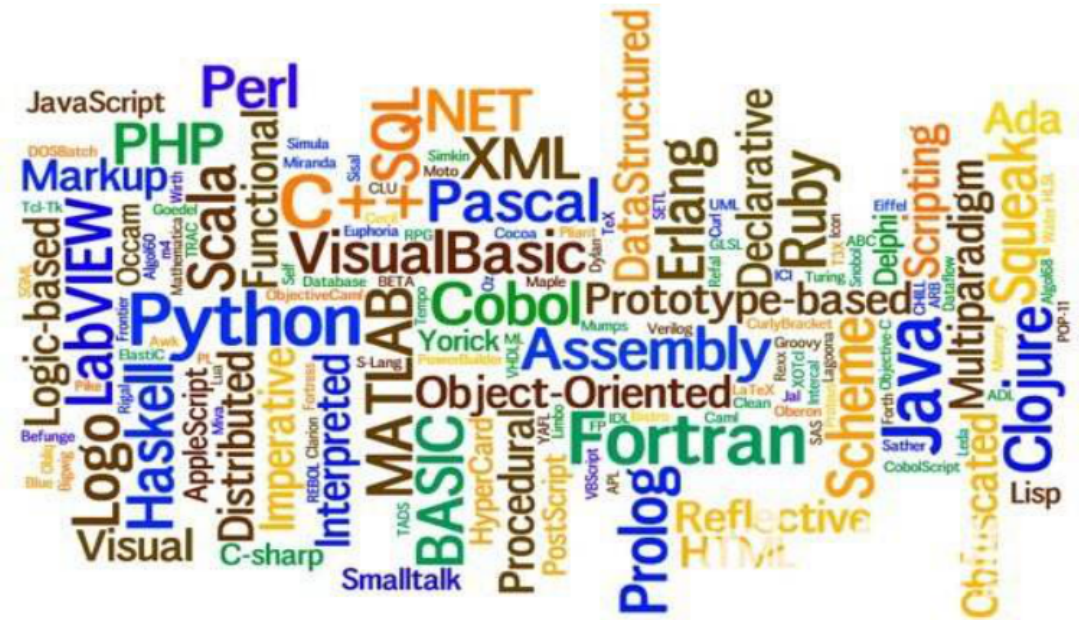
Chapter 1 Hello World!

1.1 Hello World!

1.1.1 编程语言（Programming Language）

程序是为了让计算机去解决某些问题，它由一系列指令构成。但是计算机并不能理解人类的语言，即使是最简单的，例如“计算一下 1+2 是多少”。

计算机采用的是二进制（binary），也就是只能够理解 0 和 1，因此编程语言用于作为人类与计算机之间沟通的桥梁。



通过使用编程语言来描述解决问题的步骤，从而让计算机一步一步去执行。流程图（flow chat）成为了一种程序的图形化表示方式。

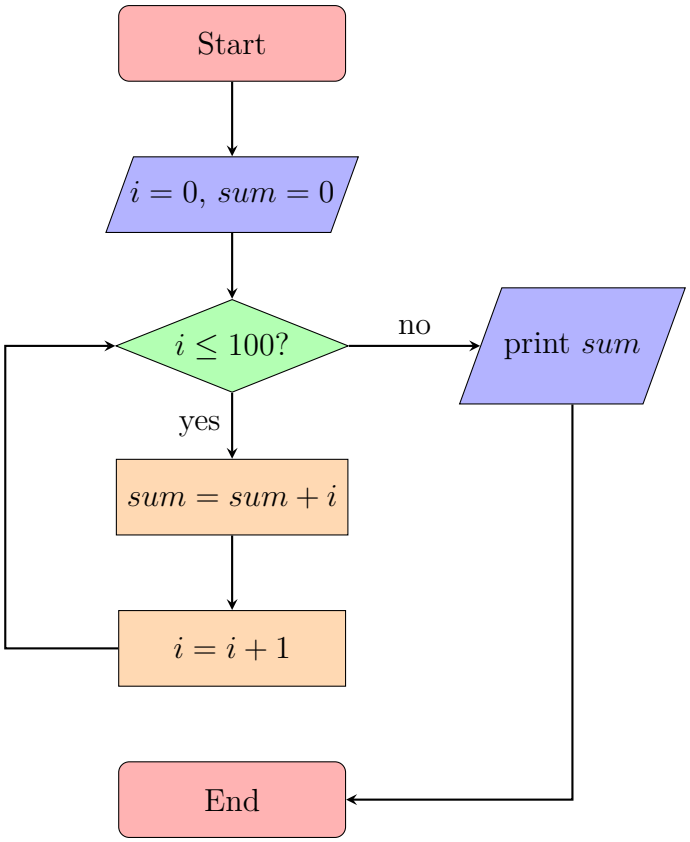


图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

1.1.2 Hello World!

Hello World 是学习编程的第一个程序，它的作用是向屏幕输出"Hello World!"。

Hello World!

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

运行结果

```
Hello World!
```

#include <iostream> 用于包含输入输出库的头文件（header file），这样才能够在程序中进行输入输出相关的操作。

using namespace std 表示使用 std 命名空间。

main() 是程序的入口，程序运行后会首先执行 main() 中的代码。cout 的功能是在屏幕上输出数据，endl 表示输出一个换行符。最后的分号用于表示一条语句的结束，注意不要使用中文的分号。

return 0 表示 main() 运行结束，返回值为 0，一般返回 0 用于表示程序正常结束。

不同编程语言的 Hello World 写法大同小异，可以看出编程语言的基本结构是相似的。

C

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
```

```
    }
}
```

Python

```
print("Hello World!")
```

1.1.3 注释（Comment）

注释就是对代码的解释和说明，它并不会程序所执行。注释能提高程序的可读性，让人更加容易了解代码的功能。

注释一般分为单行注释和多行注释：

1. 单行注释：以//开头，该行之后的内容视为注释。
2. 多行注释：以/* 开头，*/结束，中间的内容视为注释。

注释

```
/*
* Author: Terry
* Date: 2022/11/16
*/

#include <iostream>      // header file

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

1.2 数据类型

1.2.1 数据类型（Data Types）

在计算机中，每个数据一般都有一个对应的类型，基础数据类型包括：

1. 整型
    - 短整型 short
    - 整型 int
    - 长整型 long
    - 长长整型 long long
2. 浮点型
    - 单精度浮点数 float
    - 双精度浮点数 double
3. 字符型 char

不同的数据类型所占的内存空间大小不同，因此所能表示的数值范围也不同。

数据类型	大小	取值范围
short	2 字节	$-2^{15} \sim 2^{15}-1$
int	4 字节	$-2^{31} \sim 2^{31}-1$
long	4 字节	$-2^{31} \sim 2^{31}-1$
long long	8 字节	$-2^{63} \sim 2^{63}-1$
float	4 字节	7 位有效数字
double	8 字节	15 位有效数字
char	1 字节	$-128 \sim 127$

1.2.2 变量（Variable）

变量是用来存储数据的内存空间，每个变量都有一个类型，变量中只能存储对应类型的数据。

```
1 int num = 10;
2 double salary = 8232.56;
```

变量的命名需要符合规范：

1. 由字母、数字和下划线组成，不能以数字开头
2. 不可以使用编程语言中预留的关键字
3. 使用英语单词，顾名思义

关键字是编程语言内置的一些名称，具有特殊的用处和意义，因此不应该作为变量名，防止产生歧义。

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum
extern	float	for	friend	goto
if	inline	int	long	new
operator	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	throw
try	typedef	union	unsigned	virtual
void	volatile	while		

表 1.1: 关键字

1.2.3 常量（Constant）

变量的值在程序运行过程中可以修改，但有一些数据的值是固定的，为了防止这些数据被随意改动，可以将这些数据定义为常量。

在数据类型前加上 const 关键字，即可定义常量，常量一般使用大写表示。如果在程序中尝试修改常量，将会报错。

常量

```
#include <iostream>

using namespace std;

int main()
{
    const double PI = 3.1415;
    PI = 4;
    return 0;
}
```

运行结果

error: assignment of read-only variable "PI"

1.3 输入输出

1.3.1 cout

cout 是输出流对象，用来向屏幕输出数据。但是有些需要输出的字符在编程语言中具有特殊含义，因此这些特殊的字符，需要经过转义后输出。

转义字符	描述
\\	反斜杠 \
\'	单引号 '
\"	双引号 "
\n	换行
\t	制表符

表 1.2: 转义字符

转义字符

```
#include <iostream>

using namespace std;

int main()
{
    cout << "\"Hello\nWorld\"" << endl;
    return 0;
}
```

运行结果

```
"Hello
World"
```

在对变量的值进行输出时，可以使用格式控制符改变输出的格式。

长方形面积

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int length = 10;
    int width = 5;
    double area = length * width;

    cout << "Area = " << length << " * " << width << " = "
            << fixed << setprecision(2) << area << endl;
    return 0;
}
```

运行结果

```
Area = 10 * 5 = 50.00
```

1.3.2 printf()

printf() 是 C 语言中的输出函数，包含在头文件 <cstdio> 中，用于向屏幕输出指定格式的文本，使用对应类型的占位符可以更加方便地输出变量的值。

数据类型	占位符
int	%d
float	%f
double	%f
char	%c

表 1.3: 占位符

```
printf("Area = %d * %d = %.2f\n", length, width, area);
```

1.3.3 cin

有时候一些数据需要从键盘输入，cin 可以读取对应类型的数据，并赋值给相应的变量。

在使用 cin 前，通常会使用 cout 先输出一句提示信息，告诉用户需要输入什么数据。

圆面积

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

int main()
{
    const double PI = 3.14159;
    double r;
    double area;

    cout << "Radius: ";
    cin >> r;

    area = PI * pow(r, 2);
    cout << "Area = " << fixed << setprecision(2) << area << endl;

    return 0;
}
```

运行结果

```
Radius: 5
Area = 78.54
```

头文件 <cmath> 中定义了一些常用的数学函数，例如 pow(x, y) 可用于计算 x 的 y 次方。

1.4 表达式

1.4.1 算术运算符

大部分编程语言中的除法与数学中的除法意义不同。

当相除的两个数都为整数时，那么就会进行整除运算，因此结果仍为整数，例如 $21 \ / \ 4 = 5$。

如果相除的两个数中至少有一个为浮点数时，那么就会进行普通的除法运算，结果为浮点数，例如 $21.0 \ / \ 4 = 5.25$。

取模（modulo）运算符% 用于计算两个整数相除之后的余数，例如 $22 \ \% \ 3 = 1$、$4 \ \% \ 7 = 4$。

逆序三位数

```cpp
#include <iostream>

using namespace std;

int main()
{
    int num;
    int a, b, c;

    cout << "Enter a 3-digit integer: ";
    cin >> num;

    a = num / 100;
    b = num / 10 % 10;
    c = num % 10;

    cout << "Reversed: " << c*100 + b*10 + a << endl;
    return 0;
```

```
19 }
```

运行结果

```
Enter a 3-digit integer: 520
Reversed: 25
```

1.4.2 复合运算符

使用复合运算符可以使表达式更加简洁。例如 `a = a + b`可以写成 `a += b`，-=、*=、/=、%= 等复合运算符的使用方式同理。

当需要给一个变量的值加/减 1 时，除了可以使用 `a += 1`或 `a -= 1`之外，还可以使用 ++ 或--运算符，但是 `++`和 `--`可以出现在变量之前或之后：

表达式	含义
a++	执行完所在语句后自增 1
++a	在执行所在语句前自增 1
a--	执行完所在语句后自减 1
--a	在执行所在语句前自减 1

表 1.4: 自增/自减运算符

自增/自减运算符

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int n = 10;
8
```

```
9     cout << n++ << endl;
10    cout << ++n << endl;
11    cout << n-- << endl;
12    cout << --n << endl;
13
14    return 0;
15 }
```

运行结果

```
10
12
12
10
```

1.4.3 隐式类型转换

在计算机计算的过程中，只有类型相同的数据才可以进行运算。例如整数 + 整数、浮点数/浮点数等。

但是很多时候，我们仍然可以对不同类型的数据进行运算，而并不会产生错误，例如整数 + 浮点数。这是由于编译器会自动进行类型转换。在整数 + 浮点数的例子中，编译器会将整数转换为浮点数，这样就可以进行运算了。

编译器选择将整数转换为浮点数，而不是将浮点数转换为整数的原因在于，浮点数相比整数能够表示的范围更大。例如整数 8 可以使用 8.0 表示，而浮点数 9.28 变为整数 9 后就会丢失精度。

隐式类型转换最常见的情形就是除法运算，这也是导致整数/整数 = 整数、整数/浮点数 = 浮点数的原因。

1.4.4 显式类型转换

有些时候编译器无法自动进行类型转换，这时就需要我们手动地强制类型转换。

显式类型转换

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int total = 821;
    int num = 10;
    double average = (double)total / num;
    cout << "Average = " << fixed << setprecision(2) << average << endl;
    return 0;
}
```

运行结果

```
Average = 82.10
```

Chapter 2 分支

2.1 逻辑运算符

2.1.1 关系运算符

编程中经常需要使用关系运算符来比较两个数据的大小，比较的结果是一个布尔值（boolean），即 True（非 0）或 False（0）。

在编程中需要注意，一个等号 = 表示赋值运算，而两个等号 == 表示比较运算。

数学符号	关系运算符
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=
$=$	==
$\neq$	!=

2.1.2 逻辑运算符

逻辑运算符用于连接多个关系表达式，其结果也是一个布尔值。

1. 逻辑与 &&：当多个条件全部为 True，结果为 True。

条件 1	**条件 2**	**条件 1 && 条件 2**
T	T	T
T	F	F
F	T	F
F	F	F

2. 逻辑或 ||：多个条件至少有一个为 True 时，结果为 True。

条件 1	**条件 2**	**条件 1 || 条件 2**
T	T	T
T	F	T
F	T	T
F	F	F

3. 逻辑非!：条件为 True 时，结果为 False；条件为 False 时，结果为 True。

条件	**! 条件**
T	F
F	T

2.2 if

2.2.1 if

if 语句用于判断一个条件是否成立，如果成立则进入语句块，否则不执行。

年龄

```
#include <iostream>

using namespace std;

int main()
{
    int age;
    cout << "Enter your age: ";
    cin >> age;
    if(age > 0 && age < 18)
    {
        cout << "Minor" << endl;
    }
    return 0;
}
```

运行结果

```
Enter your age: 17
Minor
```

2.2.2 if-else

if-else 的结构与 if 类似，只是在 if 语句块中的条件不成立时，执行 else 语句块中的语句。

闰年

```cpp
#include <iostream>

using namespace std;

int main()
{
    int year;
    cout << "Enter a year: ";
    cin >> year;

    /*
        * A year is a leap year if it is
        * 1. exactly divisible by 4, and not divisible by 100;
        * 2. or is exactly divisible by 400
        */
    if((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    {
        cout << "Leap year" << endl;
    }
    else
    {
        cout << "Common year" << endl;
    }

    return 0;
}
```

运行结果

```
Enter a year: 2020
Leap year
```

2.2.3 if-else if-else

当需要对更多的条件进行判断时，可以使用 if-else if-else 语句。

字符

```cpp
#include <iostream>

using namespace std;

int main()
{
    char c;
    cout << "Enter a character: ";
    cin >> c;

    if(c >= 'a' && c <= 'z')
    {
        cout << "Lowercase" << endl;
    }
    else if(c >= 'A' && c <= 'Z')
    {
        cout << "Uppercase" << endl;
    }
    else if(c >= '0' && c <= '9')
    {
        cout << "Digit" << endl;
    }
    else
    {
        cout << "Special character" << endl;
    }

    return 0;
}
```

运行结果

```
Enter a character: T
Uppercase
```

2.3 switch

2.3.1 switch

switch 结构用于根据一个整数值，选择对应的 case 执行。需要注意的是，当对应的 case 中的代码被执行完后，并不会像 if 语句一样跳出 switch 结构，而是会继续向后执行，直到遇到 break。

计算器

```
#include <iostream>

using namespace std;

int main() {
    int num1, num2;
    char op;

    cout << "Enter an expression: ";
    cin >> num1 >> op >> num2;

    switch (op)
    {
    case '+':
        cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;
        break;
    case '-':
        cout << num1 << " - " << num2 << " = " << num1 - num2 << endl;
        break;
    case '*':
        cout << num1 << " * " << num2 << " = " << num1 * num2 << endl;
        break;
    case '/':
        cout << num1 << " / " << num2 << " = " << num1 / num2 << endl;
        break;
    default:
```

```
27          cout << "Error! Operator is not supported" << endl;
28          break;
29      }
30
31      return 0;
32  }
```

运行结果

```
Enter an expression: 5 * 8
5 * 8 = 40
```

Chapter 3 循环

3.1 while

3.1.1 while

while 循环会对条件进行判断，如果条件成立，就会执行循环体，然后再次判断条件，直到条件不成立。

while 循环的次数由循环变量的变化决定，因此 while 循环一般都包括对循环变量的初值、判断和更新。

```
1 int i = 1;          // initial value
2 while(i <= 5)       // condition
3 {
4     cout << "In loop: i = " << i << endl;
5     i++;            // update
6 }
7 cout << "After loop: i = " << i << endl;
```

while 循环的特点是先判断、再执行，因此循环体有可能会执行一次或多次，也有可能一次也不会执行。

平均身高

```
1 #include <iostream>
2 #include <iomanip>
3
4 #define NUM_PEOPLE 5
5
6 using namespace std;
7
8 int main()
```

```
{
    double height;
    double total = 0;

    int i = 1;
    while (i <= NUM_PEOPLE)
    {
        cout << "Enter person " << i << "'s height: ";
        cin >> height;
        total += height;
        i++;
    }

    double average = total / NUM_PEOPLE;
    cout << "Average height: "
         << fixed << setprecision(2) << average << endl;

    return 0;
}
```

运行结果

```
Enter person 1's height: 160.8
Enter person 2's height: 175.2
Enter person 3's height: 171.2
Enter person 4's height: 181.3
Enter person 5's height: 164
Average height: 170.50
```

统计元音、辅音数量

```
#include <iostream>

using namespace std;

```

```cpp
int main()
{
    char c;
    int vowel = 0;
    int consonant = 0;

    cout << "Enter an English sentence: ";

    while((c = cin.get()) != '\n')
    {
        if (c == 'a' || c == 'A' ||
            c == 'e' || c == 'E' ||
            c == 'i' || c == 'I' ||
            c == 'o' || c == 'O' ||
            c == 'u' || c == 'U')
        {
            vowel++;
        }
        else if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
        {
            consonant++;
        }
    }

    cout << "Vowel = " << vowel << endl;
    cout << "Consonant = " << consonant << endl;
    return 0;
}
```

运行结果

```
Enter an English sentence: Hello World!
Vowel = 3
Consonant = 7
```

3.1.2 do-while

do-while 循环是先执行一轮循环体内的代码后，再检查循环的条件是否成立。如果成立，则继续下一轮循环；否则循环结束。

do-while 循环是先执行、再判断，因此它至少会执行一轮循环。do-while 一般应用在一些可能会需要重复，但必定会发生一次的情景下。例如登录账户，用户输入账户和密码后，检查是否正确，如果正确，那么就成功登录；否则继续循环让用户重新输入。

需要注意，do-while 循环的最后有一个分号。

```
do {
    // code
} while(condition);
```

整数位数

```
#include <iostream>

using namespace std;

int main()
{
    int num;
    int n = 0;

    cout << "Enter an integer: ";
    cin >> num;

    do
    {
        num /= 10;
        n++;
    } while(num != 0);
```

```
18
19      cout << "Digits: " << n << endl;
20      return 0;
21  }
```

运行结果

```
Enter an integer: 123
Digits: 3
```

猜数字

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <ctime>
4
5   using namespace std;
6
7   int main()
8   {
9       srand(time(NULL));          // set random seed
10
11      // generate random number between 1 and 100
12      int answer = rand() % 100 + 1;
13      int num = 0;
14      int cnt = 0;
15
16      do
17      {
18          cout << "Guess a number: ";
19          cin >> num;
20          cnt++;
21
22          if(num > answer)
23          {
```

```
24              cout << "Too high" << endl;
25          }
26          else if(num < answer)
27          {
28              cout << "Too low" << endl;
29          }
30      } while(num != answer);
31
32      cout << "Correct! You guessed " << cnt << " times." << endl;
33      return 0;
34  }
```

运行结果

```
Guess a number: 50
Too high
Guess a number: 25
Too low
Guess a number: 37
Too low
Guess a number: 43
Too high
Guess a number: 40
Too high
Guess a number: 38
Too low
Guess a number: 39
Correct! You guessed 7 times.
```

3.2 for

3.2.1 for

while 循环将循环变量的初值、条件和更新写在了三个地方，但是这样不容易明显地看出循环变量的变化。

for 循环将循环变量的初值、条件和更新写在了一行内，中间用分号隔开。对于指定次数的循环一般更多地会采用 for 循环，而对于不确定次数的一般会采用 while 循环。

```
1 for(int i = 0; i < 5; i++)
2 {
3     cout << "i = " << i << endl;
4 }
```

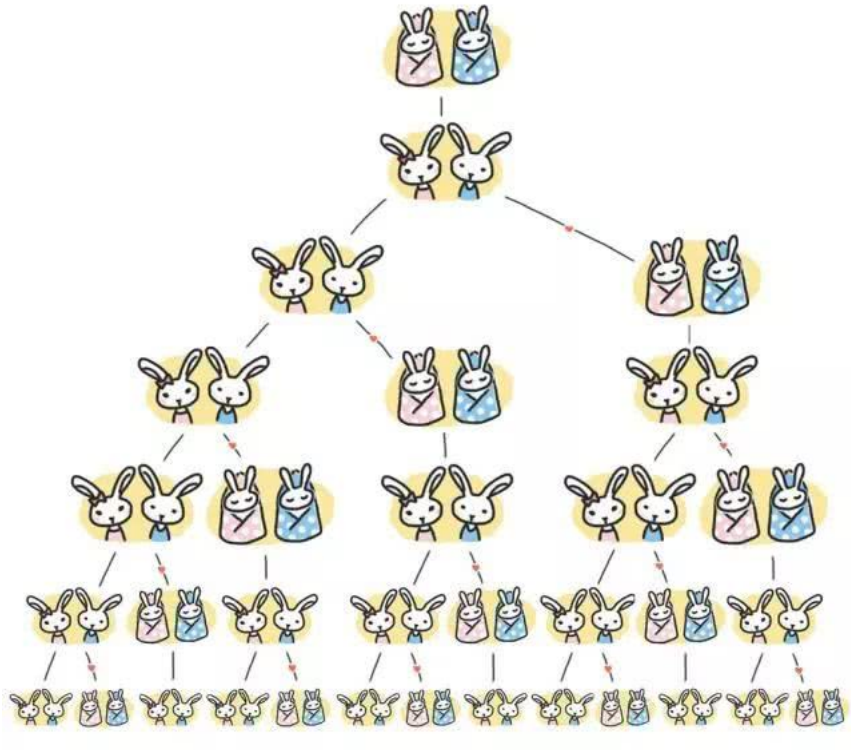
累加

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int sum = 0;
8      for(int i = 1; i <= 100; i++)
9      {
10         sum += i;
11     }
12     cout << "Sum = " << sum << endl;
13     return 0;
14 }
```

运行结果

```
Sum = 5050
```

斐波那契数列



```
#include <iostream>

using namespace std;

int main()
{
    int n;
    cout << "Enter the number of terms: ";
    cin >> n;

    if(n == 1)
    {
        cout << "1" << endl;
```

```
    }
    else if(n == 2)
    {
        cout << "1, 1" << endl;
    }
    else
    {
        int num1, num2, val;
        num1 = 1;
        num2 = 1;
        cout << "1, 1";

        for(int i = 3; i <= n; i++)
        {
            val = num1 + num2;
            cout << ", " << val;
            num1 = num2;
            num2 = val;
        }
        cout << endl;
    }

    return 0;
}
```

运行结果

```
Enter the number of terms: 10
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

3.2.2 嵌套循环

循环也可以嵌套使用，外层循环每执行一次，内层循环就会执行多次。

```
for(int i = 0; i < 2; i++)
```

```
{
    for(int j = 0; j < 3; j++)
    {
        cout << "i = " << i << ", j = " << j << endl;
    }
}
```

运行结果

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

九九乘法表

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

```
#include <iostream>

using namespace std;
```

```
4
5  int main()
6  {
7      for(int i = 1; i <= 9; i++)
8      {
9          for(int j = 1; j <= 9; j++)
10         {
11             cout << i << "*" << j << "=" << i*j << "\t";
12         }
13         cout << endl;
14     }
15     return 0;
16 }
```

打印图案

```
1  *
2  **
3  ***
4  ****
5  *****
```

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      for(int i = 1; i <= 5; i++)
8      {
9          for(int j = 1; j <= i; j++)
10         {
11             cout << "*";
12         }
13         cout << endl;
```

```
14      }
15      return 0;
16  }
```

3.3 break or continue?

3.3.1 break

break 可用于跳出当前的 switch 或循环结构。在一些情况下，在循环的中途已经完成了某个目标，没有必要再进行剩余的循环，这时就可以使用 break 跳出循环。

例如在判断一个数 n 是否为素数时，利用循环逐个判断 $2 \sim n-1$ 之间的数是否能整除 n。只要发现其中有一个数能整除 n，就证明 n 不是素数，可以跳出循环，不必再进行剩余的检查。

素数

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int n;
    cout << "Enter an integer: ";
    cin >> n;

    bool is_prime = true;
    for(int i = 2; i <= sqrt(n); i++)
    {
        if(n % i == 0)
        {
            is_prime = false;
            break;
        }
    }

    if(is_prime)
```

```
23      {
24          cout << n << " is a prime number" << endl;
25      }
26      else
27      {
28          cout << n << " is not a prime number" << endl;
29      }
30
31      return 0;
32  }
```

运行结果

```
Enter an integer: 17
17 is a prime number
```

3.3.2 continue

continue 与 break 使用方法类似，但是它并不是跳出循环，而是跳过本轮循环，直接开始下一轮循环。

正数平方和

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int n = 10;
8       cout << "Enter " << n << " integers: ";
9
10      int sum_square = 0;
11      for(int i = 0; i < n; i++)
```

```
12      {
13          int num;
14          cin >> num;
15          if(num < 0)
16          {
17              continue;
18          }
19
20          sum_square += num * num;
21      }
22
23      cout << "Sum of squares of positive integers: "
24              << sum_square << endl;
25
26      return 0;
27  }
```

运行结果

```
Enter 10 integers: 5 7 -2 0 4 -4 -9 3 9 5
Sum of squares of positive integers: 205
```

Chapter 4 数组

4.1 数组

4.1.1 数组（Array）

数组能够存储一组类型相同的元素，数组在声明时必须指定它的大小（容量），数组的大小是固定的，无法在运行时动态改变。数组通过下标（index）来访问某一位置上的元素，下标从 0 开始。

```
1 int arr[5] = {3, 6, 8, 2, 4};
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]

如果在声明数组时没有指定数组的大小，那么将根据初始化的元素个数来确定。

```
1 int arr[] = {3, 6, 8, 2, 4, 0, 1, 7};
```

通过下标可以访问数组中的元素，下标的有效范围是 0 ～ 数组的长度 - 1，如果使用不合法的下标就会导致数组越界。

```
1 cout << arr[0] << endl;     // 3
2 cout << arr[3] << endl;     // 2
3 cout << arr[7] << endl;     // 7
```

当数组的容量比较大时，可以使用循环来初始化数组。

```
1 int arr[10];
2
3 for(int i = 0; i < 10; i++) {
4     arr[i] = i + 1;
5 }
```

查找数据

```cpp
#include <iostream>

using namespace std;

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    int arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int key;
    cout << "Enter the key: ";
    cin >> key;

    bool found = false;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            found = true;
            break;
        }
    }

    if (found) {
        cout << key << " exists." << endl;
    } else {
        cout << key << " not found!" << endl;
    }

    return 0;
}
```

运行结果

```
Enter the number of elements: 5
Enter the elements: 4 8 9 2 3
Enter the key: 2
2 exists.
```

最大值/最小值

```cpp
#include <iostream>

using namespace std;

int main() {
    int num[] = {7, 6, 2, 9, 3, 1, 4, 0, 5, 8};
    int n = sizeof(num) / sizeof(num[0]);
    int max = num[0];
    int min = num[0];

    for (int i = 1; i < n; i++) {
        if (num[i] > max) {
            max = num[i];
        }
        if (num[i] < min) {
            min = num[i];
        }
    }

    cout << "Max = " << max << endl;
    cout << "Min = " << min << endl;
    return 0;
}
```

运行结果

```
Max = 9
Min = 0
```

4.1.2 for-each

for-each 循环是一种更加简洁的 for 循环，可以用于遍历访问数组中的每一个元素。

平方和

```cpp
#include <iostream>

using namespace std;

int main() {
    int arr[5] = {7, 6, 2, 9, 3};
    int sum = 0;
    for (int elem : arr) {
        sum += elem * elem;
    }
    cout << "Square sum = " << sum << endl;

    return 0;
}
```

运行结果

```
Square sum = 179
```

4.1.3 二维数组（2-Dimensional Array）

二维数组由行和列两个维度组成，行和列的下标同样也都是从 0 开始。在声明二维数组时，需要指定行和列的大小。二维数组可以看成是由多个一维数组组成的，因此二维数组中的每个元素都是一个一维数组。

```
1 int arr[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]
arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]
arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]

在初始化二维数组时，为了能够更直观地看出二维数组的结构，可以将每一行单独写在一行中。

```
1 int arr[3][4] = {
2     {1, 2, 3, 4},
3     {5, 6, 7, 8},
4     {9, 10, 11, 12},
5 };
```

对于容量较大的二维数组，可以通过两层循环进行初始化。

```
1 int arr[3][4];
2
3 for(int i = 0; i < 3; i++) {
4     for(int j = 0; j < 4; j++) {
5         arr[i][j] = 0;
6     }
7 }
```

矩阵运算

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-0 \\ 1-7 & 0-5 \\ 1-2 & 2-1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    int A[3][2] = {
        {1, 3},
        {1, 0},
        {1, 2}
    };
    int B[3][2] = {
        {0, 0},
        {7, 5},
        {2, 1}
    };
    int C[3][2];

    cout << "Matrix Addition" << endl;
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 2; j++) {
            C[i][j] = A[i][j] + B[i][j];
            cout << setw(3) << C[i][j];
        }
        cout << endl;
```

```
    }

    cout << "Matrix Subtraction" << endl;
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 2; j++) {
            C[i][j] = A[i][j] - B[i][j];
            cout << setw(3) << C[i][j];
        }
        cout << endl;
    }

    return 0;
}
```

运行结果

```
Matrix Addition
  1  3
  8  5
  3  3
Matrix Subtraction
  1  3
 -6 -5
 -1  1
```

4.2 字符串

4.2.1 ASCII

美国信息交换标准代码 ASCII（American Standard Code for Information Interchange）一共定义了 128 个字符。

ASCII	字符	ASCII	字符	ASCII	字符	ASCII	字符
0	NUT	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w

24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

ASCII

```cpp
#include <iostream>

using namespace std;

int main() {
    for(int i = 0; i < 128; i++) {
        cout << i << " - " << (char)i << endl;
    }
    return 0;
}
```

4.2.2 字符串（String）

字符数组通常被称为字符串，字符串有两种初始化的方式。一种与普通数组的初始化类似，逐个写出每一个字符，最后需要手动添加 \0 字符，表示字符串的结束符；另一种是直接使用双引号，这种写法无需手动添加 \0。

```cpp
char str[8] = {'p', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
char str[8] = "program";
```

\0 占一个字符的大小，因此在设置字符串的大小时需要考虑 \0。

使用 cin 和 cin.getline() 都可以用于读取字符串，但是 cin 只会读取到空格为止，而 cin.getline() 会读取到回车为止。

字符串输入输出

```cpp
#include <iostream>

using namespace std;

int main() {
    char str1[32];
    cout << "Enter string 1: ";
    cin.getline(str1, 32);
    cout << str1 << endl;

    char str2[32];
    cout << "Enter string 2: ";
    cin >> str2;
    cout << str2 << endl;

    return 0;
}
```

运行结果

```
Enter string 1: hello world
hello world
Enter string 2: hello world
hello
```

字符统计

```cpp
#include <iostream>

```

```cpp
using namespace std;

int main() {
    char str[32];
    char c;

    cout << "Enter a string: ";
    cin.getline(str, 32);
    cout << "Character to search: ";
    cin >> c;

    int cnt = 0;
    int i = 0;
    while (str[i] != '\0') {
        if (str[i] == c) {
            cnt++;
        }
        i++;
    }

    cout << "'" << c << "' appears " << cnt << " times in \""
            << str << "\"." << endl;

    return 0;
}
```

运行结果

```
Enter a string: this is a test
Character to search: t
't' appears 3 times in "this is a test".
```

4.2.3 字符串函数

头文件 <cstring> 中定义了一些常用的字符串处理函数。

strlen()

计算字符串的长度。

strlen()

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char s[] = "hello world";
    cout << "Length: " << strlen(s) << endl;
    return 0;
}
```

运行结果

```
Length: 11
```

strcpy()

字符串复制，调用者需要确保字符串的大小足够。

strcpy()

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char s1[32] = "hello world";
```

```
    char s2[32] = "program";

    strcpy(s1, s2);
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    return 0;
}
```

运行结果

```
s1 = program
s2 = program
```

strcat()

字符串拼接，调用者需要确保字符串的大小足够。

strcat()

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char s1[32] = "hello";
    char s2[32] = "world";

    strcat(s1, s2);
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    return 0;
}
```

运行结果

```
s1 = helloworld
s2 = world
```

strcmp()

字符串比较，依次比较字符串中每个字符的 ASCII 码值。通过判断 strcmp() 的返回值，可以得知两个字符串比较后的结果。

- 负数：字符串 1 < 字符串 2
- 正数：字符串 1 > 字符串 2
- 0：字符串 1 == 字符串 2

strcmp()

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char s1[32] = "communication";
    char s2[32] = "compare";
    cout << strcmp(s1, s2) << endl;
    return 0;
}
```

运行结果

```
-1
```

4.2.4 字符串数组

字符串数组是一个二维的字符数组，或者可以理解为是由多个字符串组成的数组。

```
1 char str[4][12] = {"C++", "Java", "Python", "JavaScript"};
```

	0	1	2	3	4	5	6	7	8	9	10	11
0	C	+	+	\0								
1	J	a	v	a	\0							
2	P	y	t	h	o	n	\0					
3	J	a	v	a	S	c	r	i	p	t	\0	

```
1 cout << "str[0] = " << str[0] << endl;          // C++
2 cout << "str[1] = " << str[1] << endl;          // Java
3 cout << "str[0][0] = " << str[0][0] << endl;    // C
4 cout << "str[0][1] = " << str[0][1] << endl;    // +
```

4.3 string

4.3.1 string

string 是一种字符串类型，使用时需要包含头文件 <string>。

string 提供了许多便捷的方法：

方法	功能
length()	字符串长度
empty()	判断字符串是否为空
push_back()	在字符串末尾添加字符
pop_back()	删除字符串末尾的字符
append()	在字符串末尾添加字符串
insert()	在字符串指定位置处插入字符串
erase()	删除字符串指定位置起 n 个字符
replace()	替换字符串从指定位置起 n 个字符为新字符串
substr()	获取字符串从指定位置起 n 个字符
find()	查找字符串中是否存在指定字符串
compare()	比较字符串和指定字符串的大小

表 4.2: string 常用方法

string 方法

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main() {
7     string s = "Hello World!";
8     cout << "length(): " << s.length() << endl;
9     cout << "substr(): " << s.substr(0, 5) << endl;
```

```
    cout << "find(): " << s.find("World") << endl;

    s.push_back('!');
    cout << "push_back(): " << s << endl;

    s.pop_back();
    cout << "pop_back(): " << s << endl;

    s.append("Hello World!");
    cout << "append(): " << s << endl;

    s.insert(12, " ");
    cout << "insert(): " << s << endl;

    s.erase(13, 6);
    cout << "erase(): " << s << endl;

    s.replace(13, 5, "C++");
    cout << "replace(): " << s << endl;

    return 0;
}
```

运行结果

```
length(): 12
empty(): 0
push_back(): Hello World!!
pop_back(): Hello World!
append(): Hello World!Hello World!
insert(): Hello World! Hello World!
erase(): Hello World! World!
replace(): Hello World! C++!
substr(): Hello
find(): 6
```

Chapter 5 函数

5.1 函数

5.1.1 函数（Function）

数学中的函数 $y = f(x)$，通过输入 x 的值，经过计算可以得到 y 的值。计算机中的函数也是如此，将输入传给函数，经过处理后，会得到输出。

函数是一段可重复使用的代码，做了一个特定的任务。例如 printf() 和 strlen() 就是函数，其中 printf() 的功能是输出字符串，strlen() 的功能是计算字符串的长度。

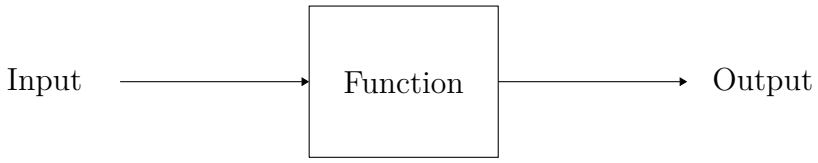


图 5.1: 函数

除了这些内置的函数以外，开发者还可以自定义函数，将程序中会被多次使用的代码或做了一件特定的任务的代码写成一个函数，这样就能避免重复写相同的代码，提高开发效率，也利于维护。

在编写函数时需要：

1. 确定函数的功能
    - 函数名
    - 确保一个函数只做一件事
2. 确定函数的输入（参数）
    - 是否需要参数
    - 参数个数

- 参数类型

3. 确定函数的输出（返回值）

   - 是否需要返回值
   - 返回值类型

最大值

```cpp
#include <iostream>

using namespace std;

int max(int num1, int num2);  // function prototype

int main() {
    cout << max(4, 12) << endl;
    cout << max(54, 33) << endl;
    cout << max(-999, -774) << endl;
    return 0;
}

int max(int num1, int num2) {
    // if(num1 > num2) {
    //     return num1;
    // } else {
    //     return num2;
    // }

    return num1 > num2 ? num1 : num2;
}
```

运行结果

```
12
54
-774
```

函数也可以没有返回值，因为它执行完函数中的代码，并不需要将结果返回给调用者，此时函数的返回值类型为 void。

棋盘

```cpp
#include <iostream>

using namespace std;

void print_board() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            cout << "   |";
        }
        cout << endl;

        if (i < 2) {
            cout << "---+---+---" << endl;
        }
    }
}

int main() {
    print_board();
    return 0;
}
```

运行结果

```
   |   |
---+---+---
   |   |
---+---+---
   |   |
```

5.1.2 函数调用

当调用函数时，程序会记录下当前的执行位置，并跳转到被调用的函数处执行。当被调用的函数执行结束后，程序会回到之前的位置继续执行。

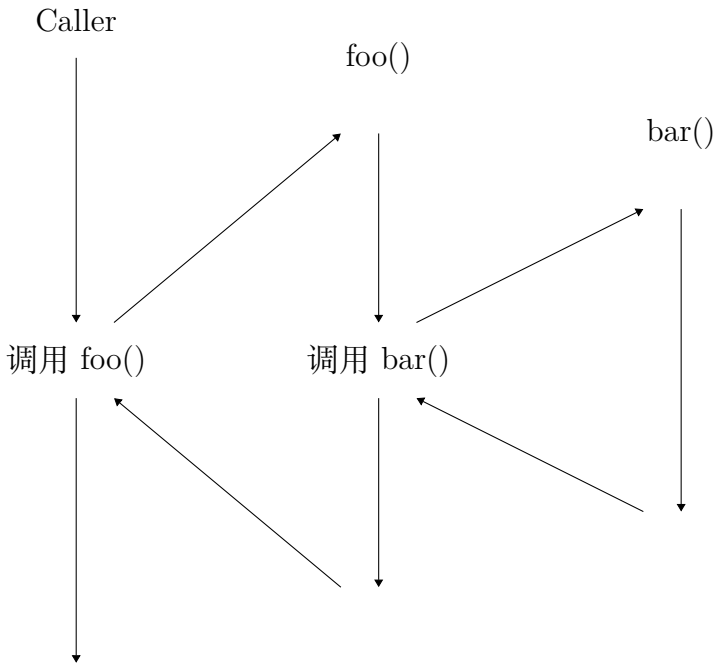


图 5.2: 函数调用

两点间距离

```
1 #include <iostream>
2 #include <cmath>
```

```cpp

using namespace std;

double square(double x) {
    return x * x;
}

double distance(double x1, double y1, double x2, double y2) {
    return sqrt(square(x1 - x2) + square(y1 - y2));
}

int main() {
    double x1, y1, x2, y2;
    cout << "Enter (x1, y1): ";
    cin >> x1 >> y1;
    cout << "Enter (x2, y2): ";
    cin >> x2 >> y2;

    cout << "Distance: " << distance(x1, y1, x2, y2) << endl;
    return 0;
}
```

运行结果

```
Enter (x1, y1): 0 0
Enter (x2, y2): 3 4
Distance: 5.00
```

5.2 作用域

5.2.1 局部变量（Local Variable）

定义在块中的变量称为局部变量，在进入块时变量才会被创建，当离开块时变量就会被销毁。因此，局部变量的生命周期为从声明时开始到所在块结束。

例如有些变量只在程序的某一段代码中使用，而在其它地方不会被使用。这时就可以将这些变量定义在一个块（if、for、函数等）中，这样可以避免变量名冲突的问题。最典型的一个例子就是在 for 循环中，循环变量 i 被定义被块中，因为 i 的作用仅用于控制循环次数，在离开循环后就没有存在的必要了。

```
1 for(int i = 0; i < 5; i++)
```

块与块之间的局部变量是互相独立的，即使变量名相同，它们也不是同一个变量。

例如在函数调用中，函数的参数也是局部变量，它们的作用域仅限于函数内。

例如一个用于交换两个变量的函数 swap()，在 main() 中的变量 a 和 b 与 swap() 中的 a 和 b 并不是同一个变量。在调用 swap() 时，是将 main() 中的 a 和 b 的值复制给 swap() 中的 a 和 b。swap() 交换的是其内部的局部变量，并不会对 main() 中的 a 和 b 产生任何影响。

局部变量

```
1 #include <iostream>
2
3 using namespace std;
4
5 void swap(int a, int b) {
6     int temp = a;
7     a = b;
8     b = temp;
9     cout << "swap(): a = " << a << ", b = " << b << endl;
```

```
}

int main() {
    int a = 1;
    int b = 2;

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

    return 0;
}
```

运行结果

```
Before: a = 1, b = 2
swap(): a = 2, b = 1
After: a = 1, b = 2
```

5.2.2 全局变量（Global Variable）

全局变量拥有比局部变量更长的生命周期，它的生命周期贯穿整个程序。全局变量可以被程序中所有函数访问。

全局变量一般用于：

- 定义在整个程序中都会被使用到的常量（例如数组容量）
- 被函数间共享的变量（例如计数器）

全局变量

```
#include <iostream>

```

```cpp
using namespace std;

int a, b;

void swap() {
    int temp = a;
    a = b;
    b = temp;
    cout << "swap(): a = " << a << ", b = " << b << endl;
}

int main() {
    a = 1;
    b = 2;

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

    return 0;
}
```

运行结果

```
Before: a = 1, b = 2
swap(): a = 2, b = 1
After: a = 2, b = 1
```

5.3 默认参数

5.3.1 默认参数

函数参数可以有默认值，如果在调用函数时不指定某个参数的值，则使用默认值。默认参数必须放在参数列表的最后。

日期

```cpp
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>

using namespace std;

string format_date(int year=1970, int month=1, int day=1) {
    ostringstream oss;
    oss << setfill('0') << setw(4) << year << "/"
        << setw(2) << month << "/"
        << setw(2) << day;
    return oss.str();
}

int main() {
    cout << format_date(2022, 12, 16) << endl;
    cout << format_date(2022, 12) << endl;
    cout << format_date(2022) << endl;
    cout << format_date() << endl;

    return 0;
}
```

运行结果

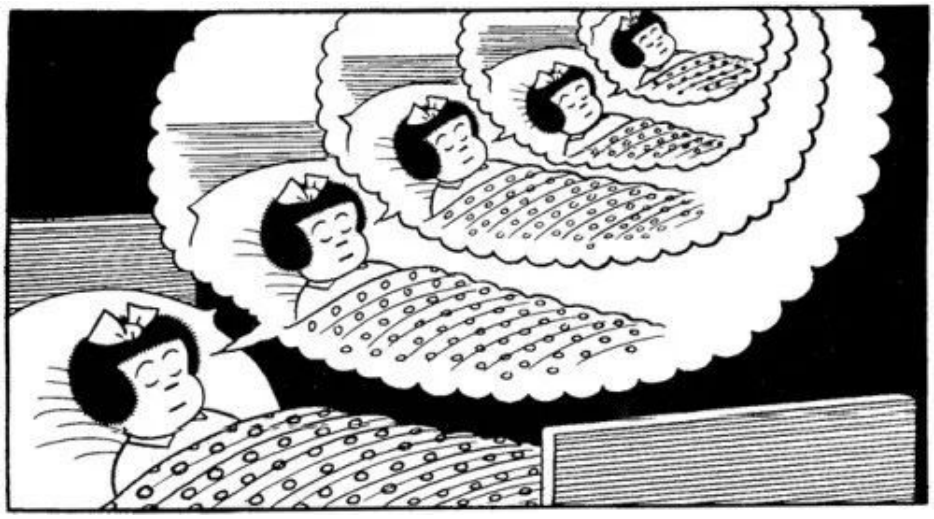
```
2022/12/16
2022/12/01
2022/01/01
1970/01/01
```

5.4 递归

5.4.1 递归（Recursion）

要理解递归，得先理解递归（见5.4章节）。

一个函数调用自己的过程被称为递归。递归可以轻松地解决一些复杂的问题，很多著名的算法都利用了递归的思想。



讲故事

```
#include <iostream>
#include <string>

using namespace std;

void tell_story() {
    string story;
    story += "从前有座山，山里有座庙\n";
    story += "庙里有个老和尚\n";
    story += "老和尚在对小和尚讲故事：\n";
    cout << story;

    tell_story();
```

```
}

int main() {
    tell_story();
    return 0;
}
```

运行结果

```
从前有座山，山里有座庙
庙里有个老和尚
老和尚在对小和尚讲故事：
从前有座山，山里有座庙
庙里有个老和尚
老和尚在对小和尚讲故事：
从前有座山，山里有座庙
庙里有个老和尚
老和尚在对小和尚讲故事：
...
```

一个永远无法结束的递归函数最终会导致栈溢出。因此递归函数需要确定一个结束条件，确保在递归过程中能在合适的地方停止并返回。

阶乘

```
#include <iostream>

using namespace std;

int factorial(int n) {
    if(n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n-1);
```

```
}

int main() {
    cout << "5! = " << factorial(5) << endl;
    return 0;
}
```

运行结果

```
5! = 120
```

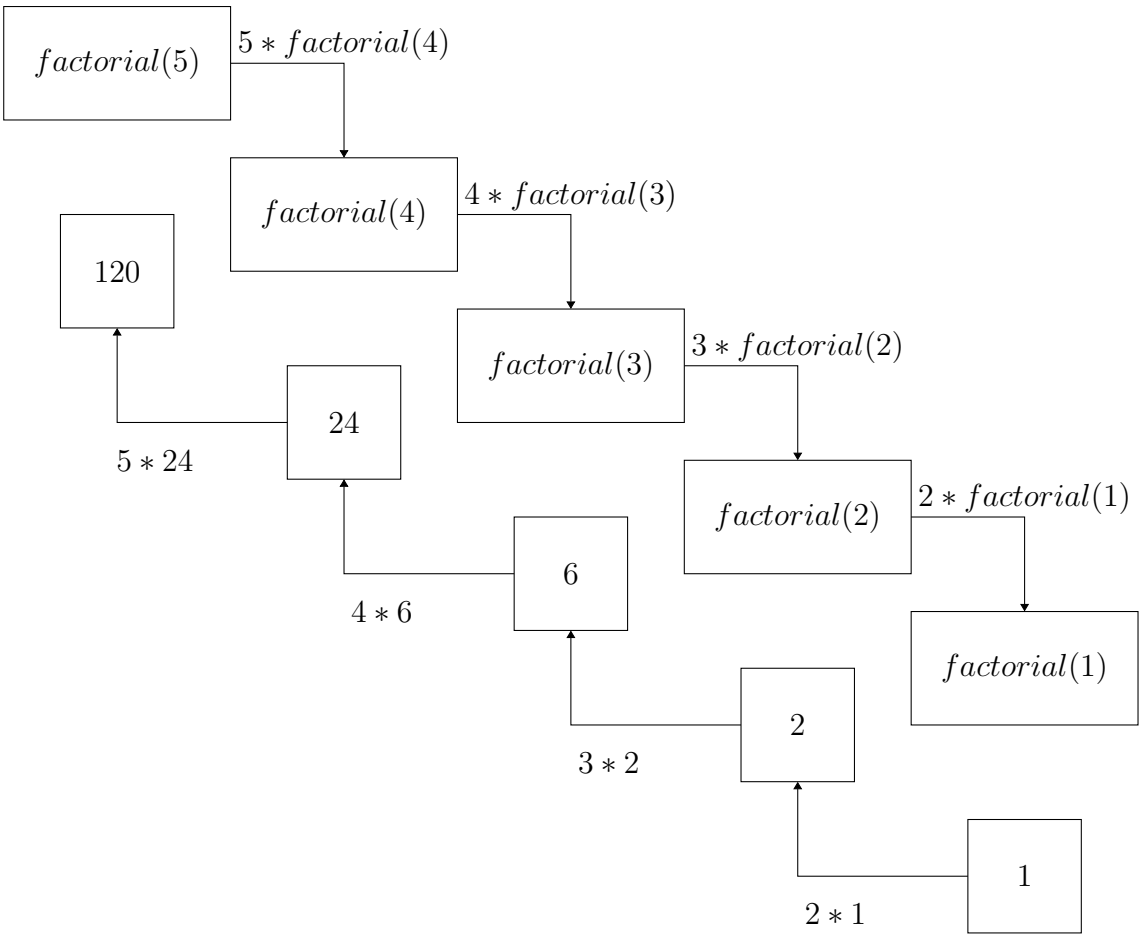


图 5.3: 阶乘

斐波那契数列

```cpp
#include <iostream>

using namespace std;

int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return n;
    }
    return fibonacci(n - 2) + fibonacci(n - 1);
}

int main() {
    int n = 7;
    cout << fibonacci(n) << endl;
    return 0;
}
```

运行结果

```
21
```

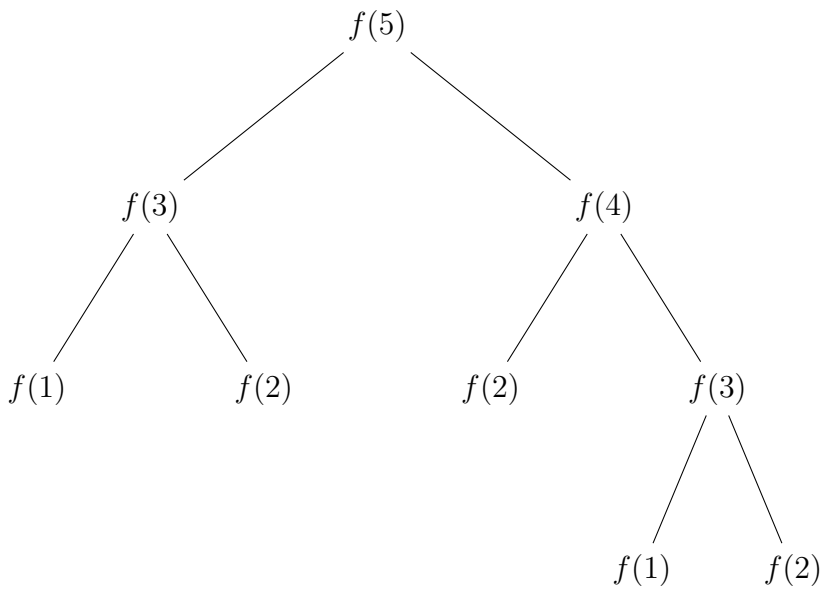


图 5.4: 递归树

递归的特点就是将一个复杂的大问题逐步简化为一个可以解决的小问题，然后再

逐步计算出大问题的解。

递归的优点在于代码简洁易懂，但是缺点也很明显，就是效率很低。每次递归都会产生函数调用，而函数调用的开销是很大的，不适合用来解决大规模的问题。

例如在计算斐波那契数列的第 40 项时，递归需要花费大量时间，因为其中包含了大量的重复计算。相比而言，使用循环的方式能够节省大量的时间。因此像阶乘和斐波那契数列这样的情况，通常会采用循环，而不是递归进行计算。

然而还存在很多问题不得不使用递归的思想才能解决。

阿克曼函数

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, n = 0 \\ A(m-1, A(m,n-1)) & m > 0, n > 0 \end{cases}$$

```cpp
#include <iostream>

using namespace std;

int A(int m, int n) {
    if (m == 0) {
        return n + 1;
    } else if (m > 0 && n == 0) {
        return A(m - 1, 1);
    } else {
        return A(m - 1, A(m, n - 1));
    }
}

int main() {
    cout << A(3, 4) << endl;
```

```
17      return 0;
18  }
```

运行结果

```
125
```

汉诺塔

有三根柱子 A、B、C，A 柱子上从下到上套有 n 个圆盘，要求将 A 柱子上的圆盘移动到 C 柱子上。每次只能移动一个圆盘，且大圆盘始终不能叠在小圆盘上面。

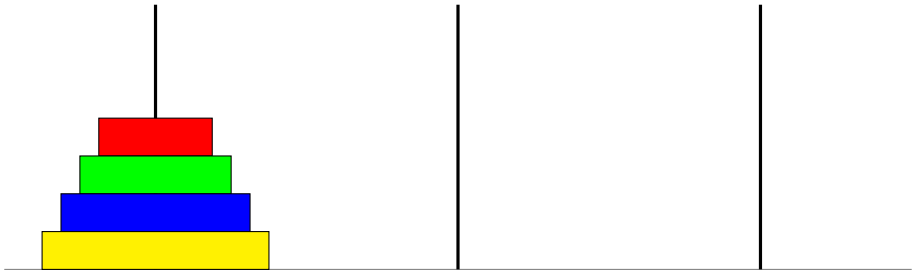


图 5.5: 汉诺塔

递归算法求解汉诺塔问题：

1. 将 n-1 个圆盘从 A 借助 C 移到 B。
2. 将第 n 个圆盘从 A 移到 C。
3. 将 n-1 个圆盘从 B 借助 A 移到 C。

```
1   #include <iostream>
2
3   using namespace std;
4
5   int moves = 0;
6
7   void hanoi(int n, char src, char mid, char dst) {
```

```
    if (n == 1) {
        cout << src << " -> " << dst << endl;
        moves++;
    } else {
        // move top n-1 disks from src to mid
        hanoi(n - 1, src, dst, mid);
        cout << src << " -> " << dst << endl;
        moves++;
        // move top n-1 disks from mid to dst
        hanoi(n - 1, mid, src, dst);
    }
}

int main() {
    hanoi(3, 'A', 'B', 'C');
    cout << "Moves: " << moves << endl;
    return 0;
}
```

运行结果

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
Moves: 7
```

假设每次移动花费 1 秒，解决一个 64 层的汉诺塔问题大约需要 5800 亿年。


吓得我抱起了
抱着抱着抱着我的小鲤鱼的我的我的我

Chapter 6 预处理

6.1 预处理

6.1.1 宏（Macro）

宏是一种简单的文本替换工具，可以用于定义一个特定的常量或表达式，一般用大写表示。宏定义使用 #define 指令，在编译期间，编译器会将程序中所有的宏替换为其内容。

与变量的定义不同的是，宏没有类型，也不占内存空间。

圆

```
#include <iostream>
#define PI 3.14159

using namespace std;

double perimeter(double r) {
    return 2 * PI * r;
}

double area(double r) {
    return PI * r * r;
}

int main() {
    double radius;
    cout << "Enter radius: ";
    cin >> radius;

    cout << "Perimeter: " << perimeter(radius) << endl;
```

```
20      cout << "Area: " << area(radius) << endl;
21
22      return 0;
23  }
```

运行结果

```
Enter radius: 5
Perimeter: 31.42
Area: 78.54
```

宏也可以像函数一样传递参数，但是宏的参数不会进行类型检查，宏最终同样也会在编译期间被展开。

但是由于宏定义的内容在编译时会被替换到代码中，有时候会导致运算的优先级发生改变。

```
1  #define SQUARE x * x
```

例如 SQUARE(2 + 3) 会被展开为 2 + 3 * 2 + 3，而不是 (2 + 3) * (2 + 3)。因此，最好在宏中使用括号来避免这种情况。

```
1  #define SQUARE (x * x)
```

6.1.2 条件编译

条件编译是一种在编译时根据宏的定义来决定是否编译某段代码的方法。

斐波那契数列

```
1  #include <iostream>
2
3  using namespace std;
4
```

```cpp
#define RECURSION

#ifdef RECURSION
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
#else
int fibonacci(int n) {
    int seq[n];
    seq[0] = 0;
    seq[1] = 1;

    for (int i = 2; i <= n; i++) {
        seq[i] = seq[i - 1] + seq[i - 2];
    }

    return seq[n];
}
#endif

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << fibonacci(n) << endl;
    return 0;
}
```

运行结果

```
Enter n: 7
13
```

6.2 多文件编译

6.2.1 编译（Compile）

集成开发环境 IDE（Integrated Development Environment）包含了文本编辑器、编译器、调试器和其它工具，可以很方便地进行开发。但是对于大型项目，使用命令行编译更加灵活和高效。

交换

```cpp
#include <iostream>

using namespace std;

#define SWAP(a, b) {int t; t = a; a = b; b = t;}

int main() {
    int a = 1;
    int b = 2;

    cout << "Before: a = " << a << ", b = " << b << endl;
    SWAP(a, b);
    cout << "After: a = " << a << ", b = " << b << endl;

    return 0;
}
```

```
g++ -Wall swap.cpp -o swap
./swap
```

其中 g++ 表示编译器的名称，-Wall 表示要输出所有警告信息，swap.cpp 为需编译的源文件，-o 用于指定输出的可执行文件的名称为 swap。编译成功后使用./swap 即可运行。

一个完整的编译过程包含 4 个步骤：

1. 预处理：将头文件、宏定义等展开

```
g++ -E swap.cpp -o swap.i
```

2. 编译：将预处理后的代码转换为汇编代码

```
g++ -S swap.i -o swap.s
```

3. 汇编：将汇编代码转换为机器码

```
g++ -c swap.s -o swap.o
```

4. 链接：将目标文件链接为可执行文件

```
g++ swap.o -o swap
```

6.2.2 多文件编译

模块化编程的目的是为了将程序分解成多个独立、可重用的部分。当程序变得复杂时，分成多个文件可以使得程序逻辑更加清晰、易于维护。

在多文件中，每个模板一般都分为.h 和.cpp 两部分，其中.h 文件用于声明函数原型，.cpp 文件用于实现函数。这样其它文件只需要包含.h 文件即可使用这些函数，就像包含头文件 iostream 一样，只不过自定义的头文件一般使用双引号包含。

由于一个头文件可以被多个源文件包含，为了避免重复定义，一般在头文件的开头使用条件编译来判断是否已经被包含。

面积

geometry.h

```
#ifndef _GEOMETRY_H_
#define _GEOMETRY_H_

```

```
4 double circle_area(double radius);
5
6 double triangle_area(double base, double height);
7
8 #endif
```

geometry.cpp

```
1  #include "geometry.h"
2
3  #define PI 3.1415926
4
5  double circle_area(double radius) {
6      return PI * radius * radius;
7  }
8
9  double triangle_area(double base, double height) {
10     return base * height / 2;
11 }
```

area.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  #include "geometry.h"
4
5  using namespace std;
6
7  int main() {
8      cout << "Area of circle: "
9           << fixed << setprecision(2) << circle_area(5) << endl;
10
11     cout << "Area of triangle: "
12          << fixed << setprecision(2) << triangle_area(5, 10) << endl;
13
14     return 0;
15 }
```

```
g++ -Wall geometry.cpp area.cpp -o area
./area
```

运行结果

```
Area of circle: 78.54
Area of triangle: 25.00
```

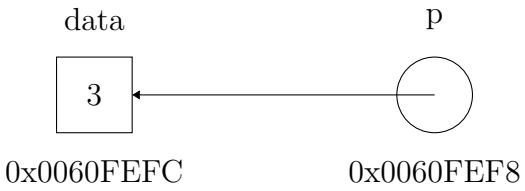
Chapter 7 指针

7.1 指针

7.1.1 指针（Pointer）

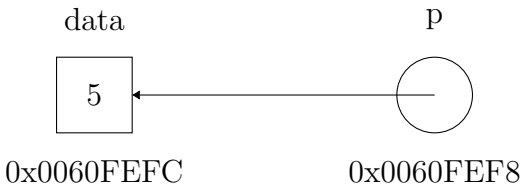
每个变量都会在内存中占用一定的空间，不同类型的变量占用的空间大小也不同。每个空间都有一个地址，一般采用十六进制表示，如 0x0060FEFC。

有时候需要通过变量的地址对变量进行操作，这时候就需要将变量的地址保存起来，保存地址的变量就成为指针。



* 用于声明一个指针变量，例如 int *p 表示 p 是一个指针，指向一个 int 类型的变量的地址。通过取地址运算符 & 可以获取变量的地址，占位符%p 能够以十六进制的形式输出地址。

既然指针保存了另一个变量的地址，那么通过指针就可以访问到那个变量上的数据。在指针变量前使用 * 运算符，就可以获取到指针所指向的变量的值。



指针

```
1 #include <iostream>
2
```

```cpp
using namespace std;

int main() {
    int data = 3;
    int *p = &data;

    cout << "Value of data: " << data << endl;
    cout << "Address of data: " << &data << endl;

    cout << "Value of p: " << p << endl;
    cout << "Address of p: " << &p << endl;
    cout << "Value of data pointed by p: " << *p << endl;

    *p = 5;

    cout << "Value of data: " << data << endl;
    cout << "Value of data pointed by p: " << *p << endl;

    return 0;
}
```

运行结果

```
Value of data: 3
Address of data: 0x61fe1c
Value of p: 0x61fe1c
Address of p: 0x61fe10
Value of data pointed by p: 3
Value of data: 5
Value of data pointed by p: 5
```

为什么不直接修改变量的值，还要多此一举通过指针修改呢?

例如需要实现 swap() 用于交换两个变量的值，由于传递参数是按值传递（pass by value），所以交换的仅仅只是 swap() 中局部变量的值。

这种情况下就需要使用指针，将需要交换的变量的地址传递给 swap()，然后在 swap() 中交换这两个地址上的值。

交换

```cpp
#include <iostream>

using namespace std;

void swap(int *data1, int *data2) {
    int temp = *data1;
    *data1 = *data2;
    *data2 = temp;
}

int main() {
    int a = 3;
    int b = 5;

    cout << "Before: a = " << a << ", b = " << b << endl;
    swap(&a, &b);
    cout << "After: a = " << a << ", b = " << b << endl;
    return 0;
}
```

运行结果

```
Before: a = 3, b = 5
After: a = 5, b = 3
```

函数最多只能返回一个值，但如果需要有多个值需要返回，就可以使用指针将数据带回。

一元二次方程

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

/**
 * Solve quadratic equation ax^2 + bx + c = 0.
 * @param a coefficient of x^2
 * @param b coefficient of x
 * @param c constant
 * @param x1 pointer to the first root
 * @param x2 pointer to the second root
 * @return true if the equation has real roots, false otherwise.
 */
bool solver(double a, double b, double c, double *x1, double *x2) {
    double delta = b * b - 4 * a * c;
    if (delta < 0) {
        return false;
    }
    *x1 = (-b + sqrt(delta)) / (2 * a);
    *x2 = (-b - sqrt(delta)) / (2 * a);
    return true;
}

int main() {
    double a, b, c;
    double x1, x2;

    cout << "Quadratic equation ax^2 + bx + c = 0" << endl;
    cout << "Enter coefficients a, b, c: ";
    cin >> a >> b >> c;

    if (solver(a, b, c, &x1, &x2)) {
        cout << fixed << setprecision(2)
             << "x1 = "  << x1 << ", x2 = " << x2 << endl;
```

```
37      } else {
38          cout << "No real roots" << endl;
39      }
40
41      return 0;
42  }
```

运行结果

```
Quadratic equation ax^2 + bx + c = 0
Enter coefficients a, b, c: 1 -9 20
x1 = 5.00, x2 = 4.00
```

7.1.2 NULL

如果一个变量声明时没有初始化，那么它的值是不确定的。声明指针时如果不对指针进行初始化，那么它就会指向一块不确定的内存地址，这种指针被称为野指针。

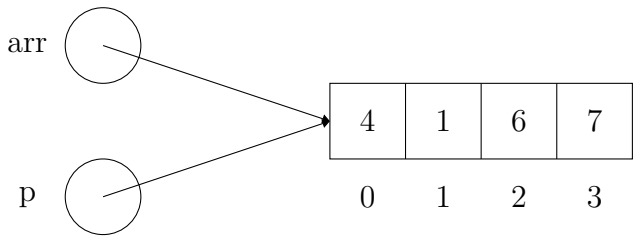
使用野指针可能会导致程序崩溃，因为它可能指向一个不可访问的内存地址。因此，如果指针没有指向一个确定的内存地址时，应该将其赋值为空指针 NULL。

```
1  int *p = NULL;
```

7.2 指针与数组

7.2.1 指针与数组

数组名本质上就是一个指针，它指向数组的首地址。因此在获取数组的地址时，可以不使用 & 运算符。



当对一个指向数组的指针进行加减运算时（如 p++ 和 p--），并不是将地址加 1 或减 1，而是根据指针的类型加或减对应字节的长度。例如 p 是一个 int 型指针，那么 p++ 会将地址加 4（int 占 4 个字节）、p -= 2 会将地址减 8。

指针与数组

```cpp
#include <iostream>

using namespace std;

int main() {
    int arr[] = {4, 1, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int *p = arr;

    cout << "Address of arr: " << arr << endl;
    for (int i = 0; i < n; i++) {
        cout << "Address of arr[" << i << "]: " << &arr[i] << endl;
    }

    cout << "Value of p: " << p << endl;
    for (int i = 1; i < n; i++) {
```

```
17          cout << "Value of p + " << i << ": " << p + i << endl;
18      }
19
20      *p = 9;
21      *(p + 1) = 8;
22      *(p + 2) = 7;
23      *(p + 3) = 6;
24
25      for (int i = 0; i < n; i++) {
26          cout << "arr[" << i << "]: " << arr[i] << endl;
27      }
28
29      return 0;
30  }
```

运行结果

```
Address of arr: 0x61fdf0
Address of arr[0]: 0x61fdf0
Address of arr[1]: 0x61fdf4
Address of arr[2]: 0x61fdf8
Address of arr[3]: 0x61fdfc
Value of p: 0x61fdf0
Value of p + 1: 0x61fdf4
Value of p + 2: 0x61fdf8
Value of p + 3: 0x61fdfc
arr[0]: 9
arr[1]: 8
arr[2]: 7
arr[3]: 6
```

7.2.2 数组与函数

数组作为函数参数时，会将数组的地址传递给函数，函数接收到的是一个指向数组首地址的指针。由于在函数中失去了数组长度的信息，并不能通过 sizeof() 计算出数组的长度（计算得到的是一个指针变量所占的空间），因此将数组传入函数时，还需要将其长度一并作为参数传给函数。

查找

```cpp
#include <iostream>

using namespace std;

int search(int *arr, int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {4, 7, 1, 3, 9, 2};
    int n = sizeof(arr) / sizeof(arr[0]);

    int index = search(arr, n, 3);
    if (index == -1) {
        cout << "Not found" << endl;
    } else {
        cout << "Found at index " << index << endl;
    }

    return 0;
}
```

运行结果

```
Found at index 3
```

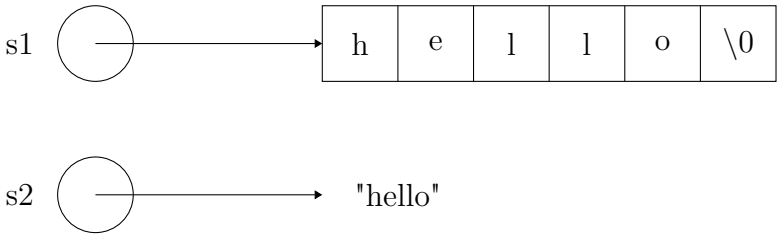
7.3 指针与字符串

7.3.1 指针与字符串

数组和指针都可以用于定义一个字符串，但是它们内存分配的方式不同，从而导致它们的使用方式也不同。

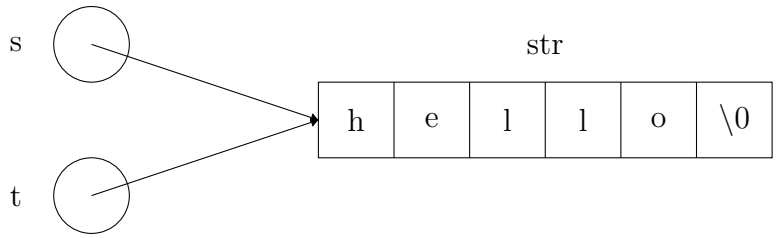
以数组形式定义的字符串，每个字符保存在一个字符数组中。这样的字符串是可以修改的，与普通的数组类似。

但是如果让一个指针指向一个字符串，那么这个字符串会被存储在常量区。常量区中的数据是不可以修改的，因此使用指针去修改字符串会导致程序崩溃。




```
1 char s1[] = "hello";
2 s1[0] = 'H';
3 cout << s1 << endl;
4
5 char *s2 = "hello";
6 s2[0] = 'H';        // error
7 cout << s2 << endl;
```

在对指向字符串的指针进行赋值操作的时候，并不会产生新的字符串，只是让两个指针都指向同一个字符串，对任意一个指针做的操作都会影响另一个指针。



指向字符串的指针

```cpp
#include <iostream>

using namespace std;

int main() {
    char str[] = "hello";
    char *s = str;
    char *t = s;

    s[0] = 'H';
    cout << "s = " << s << endl;
    cout << "t = " << t << endl;

    return 0;
}
```

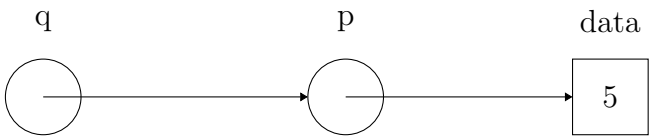
运行结果

```
s = Hello
t = Hello
```

7.4 二级指针

7.4.1 二级指针（Pointer to pointer）

既然指针也是一个变量，那么一个指针也可以指向另一个指针，这样的指针称为二级指针。



其中 p 是一个 int 型的指针（int *），指向了 data；q 是一个指向 int 型指针的指针（int **），即 q 指向了一个指向 int 的指针。

二级指针

```
#include <iostream>

using namespace std;

int main() {
    int data = 5;
    int *p = &data;
    int **q = &p;

    cout << "data = " << data << endl;
    cout << "*p = " << *p << endl;
    cout << "**q = " << **q << endl;

    cout << "Address of data = " << &data << endl;
    cout << "Address of p = " << &p << endl;
    cout << "Address of q = " << &q << endl;

    cout << "Value of p = " << p << endl;
    cout << "Value of q = " << q << endl;

```

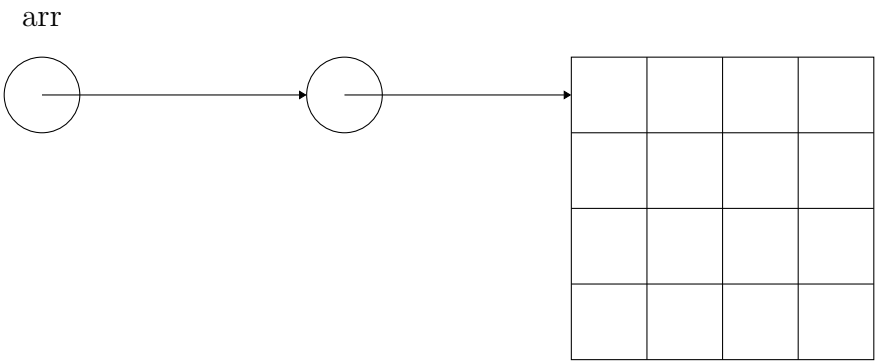
```
21      return 0;
22  }
```

运行结果

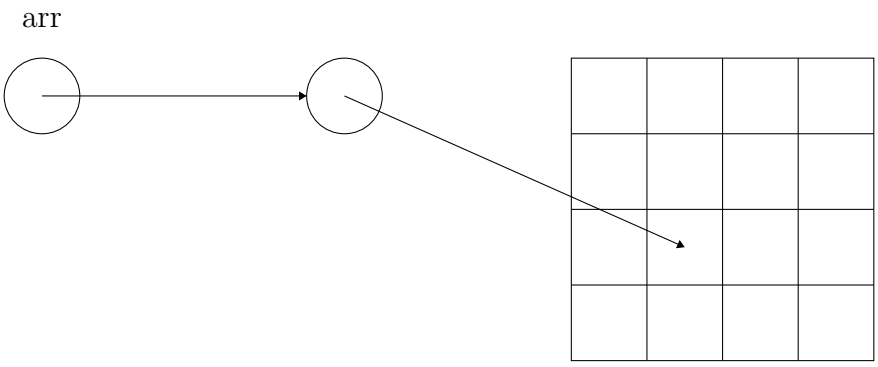
```
data = 5
*p = 5
**q = 5
Address of data = 0x61fe1c
Address of p = 0x61fe10
Address of q = 0x61fe08
Value of p = 0x61fe1c
Value of q = 0x61fe10
```

7.4.2 指针与二维数组

二级指针还可以用于表示二维数组。与一维数组类似，可以使用下标或 * 运算符来访问二维数组中的元素。



例如，arr[2][1] 也可以写成 *(*(arr + 2) + 1)。



当如果命令行运行时，可以向 main() 函数传递命令行参数。传递参数的数量和值可以通过 argc（argument count）和 argv（argument vector）来获取。argv 为一个二维数组，每个元素都是一个字符串，其中 argv[0] 为当前程序的名称。

命令行参数

```
#include <iostream>
#include <cctype>
#include <cstring>

using namespace std;

char *lower(char *s) {
    char *p = s;
    while (*p) {
        *p = tolower(*p);
        p++;
    }
    return s;
}

char *upper(char *s) {
    char *p = s;
    while (*p) {
        *p = toupper(*p);
        p++;
    }
    return s;
```

```
}

void usage(const char *program) {
    cout << "Usage: " << program << " [option] [string]" << endl;
    cout << "--lower: convert string to lower case" << endl;
    cout << "--upper: convert string to upper case" << endl;
}

int main(int argc, char **argv) {
    if (argc != 3) {
        usage(argv[0]);
        return 1;
    }

    char *option = argv[1];
    char *string = argv[2];

    if (strcmp(option, "--lower") == 0) {
        cout << "Lower: " << lower(string) << endl;
    } else if (strcmp(option, "--upper") == 0) {
        cout << "Upper: " << upper(string) << endl;
    } else {
        usage(argv[0]);
        return 1;
    }

    return 0;
}
```

```
g++ -Wall command_line.cpp -o command_line
./command_line --upper "Hello World!"
```

运行结果

```
Upper: HELLO WORLD!
```

7.5 引用

7.5.1 引用（Reference）

引用可以看作是变量的别名，对引用的操作与对变量的操作完全一样。因此引用可以用于传递参数和返回值，避免了复制较大变量的开销。

声明引用时必须对其初始化，并且之后不能再将该引用作为其它变量的别名。

交换

```cpp
#include <iostream>

using namespace std;

void swap(int &data1, int &data2) {
    int temp = data1;
    data1 = data2;
    data2 = temp;
}

int main() {
    int a = 3;
    int b = 5;

    cout << "Before: a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "After: a = " << a << ", b = " << b << endl;
    return 0;
}
```

运行结果

```
Before: a = 3, b = 5
After: a = 5, b = 3
```

	指针	引用
内存	占用空间	不占用空间
大小	4 字节/8 字节	被引用对象的大小
能否为空	可以为 NULL	引用对象必须存在
能否改变指向	可以	不可以
使用	通过 * 对指向的变量操作	直接对变量操作
层级	可以有多级指针	只有一级

表 7.1: 指针与引用的区别

7.6 动态内存申请

7.6.1 内存管理

计算机的内存主要包括：

1. 代码区：存储程序执行时使用的指令。
2. 数据区：存储程序运行时使用的全局变量和静态变量。
3. 栈区（stack）：存储函数调用时使用的局部变量，栈中的内存由编译器自动分配和释放。
4. 堆区（heap）：存储动态分配内存的变量，需要由程序员自己分配和释放。

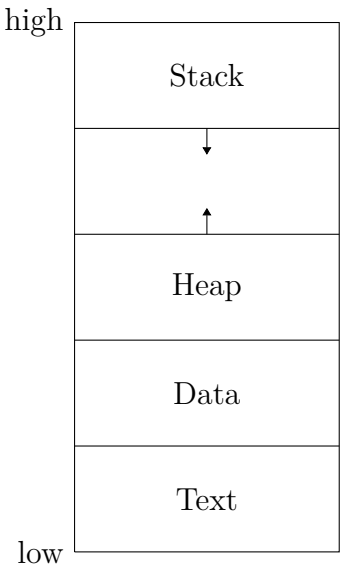


图 7.1: 内存区域

有时候需要在函数中生成一个数组，并且将数组返回给调用者。但是在函数内定义的数组是局部变量，存储于栈区。函数执行完毕后，数组所占用的内存空间会被释放。因此被返回的数组的指针指向的是一个已经被释放的内存空间，这样就会导致程序崩溃。

另一种情况是，由于数组一旦声明后，其容量就不能再改变。当需要在运行时动态改变数组容量时，就可以采用动态内存申请的方式。

7.6.2 malloc()

malloc() 函数定义在 <cstdlib> 中，用于在堆区申请一块内存空间，其函数原型为：

```
void* malloc(size_t size);
```

malloc() 接受一个参数 size，表示申请空间的大小（单位：字节），并返回指向申请到的空间的首地址的指针。如果申请失败，则返回 NULL。

malloc() 返回的是无类型指针 void *，这是由于 malloc() 只负责申请指定大小的空间，并不关心这块空间将会被存放什么类型的数据。因此，开发者需要自行将其转换为对应的类型。

内存

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    void *p;
    int cnt = 0;

    // allocate 100MB memory each time
    while ((p = malloc(100 * 1024 * 1024))) {
        cnt++;
    }
    cout << "Allocated " << cnt * 100 << " MB memory" << endl;

    return 0;
}
```

运行结果

```
Allocated 57700 MB memory
```

7.6.3 free()

通过动态申请来的内存空间需要是需要归还给操作系统，因此需要程序员自行在不需要使用时将其释放。如果不释放内存，这些动态申请的内存空间就会一直占用着，直到程序结束统一被操作系统释放。

不释放内存会导致内存泄漏（memory leak），如果一直分配内存而不是释放，最终将会耗尽所有可用的内存，导致程序运行变慢或者崩溃。

free() 函数用于释放动态申请的内存空间，其接受一个参数 ptr，表示要释放的内存空间的首地址。其函数原型为：

```
1 void free(void *ptr);
```

斐波那契数列

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int *generate_fibonacci(int n) {
7      int *arr = (int *)malloc(n * sizeof(int));
8      if (!arr) {
9          return NULL;
10     }
11
12     arr[0] = 1;
13     arr[1] = 1;
```

```
    for (int i = 2; i < n; i++) {
        arr[i] = arr[i - 1] + arr[i - 2];
    }
    return arr;
}

int main() {
    int n = 10;
    int *arr = generate_fibonacci(n);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    free(arr);
    return 0;
}
```

运行结果

```
1 1 2 3 5 8 13 21 34 55
```

7.6.4 calloc()

calloc() 与 malloc() 功能类似，也是用于动态申请内存空间的。只是 malloc() 只接受一个参数作为申请空间的大小，申请到的空间并不会进行初始化；而 calloc() 接受两个参数，可以申请多个指定大小的空间，并将这些空间初始化为 0。calloc() 的函数原型为：

```
void *calloc(size_t nitems, size_t size);
```

例如需要申请一个长度为 n 的 int 数组，并将其初始化为 0：

```
int *arr = (int *)calloc(n, sizeof(int));
```

7.6.5 realloc()

realloc() 用于对已经动态申请的内存空间进行重新分配（扩容/缩小），其函数原型为：

```
void *realloc(void *ptr, size_t size);
```

realloc() 接受两个参数，第一个参数 ptr 指向需要重新分配内存空间，第二个参数 size 表示重新分配空间的大小（单位：字节）。realloc() 会将原来内存块中的数据复制到新分配的内存块中，并返回指向新内存块的指针。如果重新分配失败，则返回 NULL。

strip

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <cctype>

using namespace std;

char *strip(char *str) {
    int i = 0;
    int j = strlen(str) - 1;

    while (isspace(str[i]) && str[i] != '\0') {
        i++;
    }

    while (isspace(str[j]) && j >= 0) {
        j--;
    }

    int k = 0;
    while (i <= j) {
        str[k++] = str[i++];
```

```
    }
    str[k] = '\0';

    str = (char *)realloc(str, (k + 1) * sizeof(char));
    return str;
}

int main() {
    int len = 32;
    char *str = (char *)calloc(len + 1, sizeof(char));

    strcpy(str, "     Hello World! \n\t ");
    cout << "Before: [" << str << "]" << endl;

    str = strip(str);

    cout << "After: [" << str << "]" << endl;

    free(str);
    return 0;
}
```

运行结果

```
Before: [     Hello World!
     ]
After: [Hello World!]
```

Chapter 8 文件

8.1 文件

8.1.1 fopen()

文件是存储数据的一种常用方式，程序可以从文件中读取和写入数据，从而实现对数据的持久化存储。

在对文件进行操作之前，首先需要使用 fopen() 函数打开文件，fopen() 的函数原型为：

```
1 FILE *fopen(const char *filename, const char *mode);
```

fopen() 接受两个参数，第一个参数是要打开的文件名，第二个参数为打开方式。fopen() 会返回一个 FILE 类型的指针，通过该指针可以对文件进行操作；如果文件打开失败，则返回 NULL。

打开方式	功能
r	只读，文件必须存在，否则打开失败
w	只写，创建一个新文件
a	追加，如果文件不存在则创建；存在则将数据追加到末尾
r+	以 r 模式打开文件，附带写的功能
w+	以 w 模式打开文件，附带读的功能
a+	以 a 模式打开文件，附带读的功能

表 8.1: 文件打开方式

8.1.2 fclose()

在对文件操作结束后，需要使用 fclose() 函数将文件关闭。fclose() 函数原型：

```
int fclose(FILE *stream);
```

文件

data.txt

```
This is a test.
```

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    FILE *fp = fopen("data.txt", "r");
    if(!fp) {
        exit(1);
    }
    fclose(fp);
    return 0;
}
```

8.2 文件 I/O

8.2.1 fprintf()

fprintf() 函数用于将数据输出到文件中，使用方法与 printf() 类似。fprintf() 的函数原型为：

```
int fprintf(FILE *stream, const char *format, ...);
```

成绩

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    FILE *fp = fopen("data1.txt", "w");
    int n;
    cout << "Enter number of students: ";
    cin >> n;

    char id[8];
    double score;
    for (int i = 0; i < n; i++) {
        cout << "Enter student " << i + 1 << "'s ID: ";
        cin >> id;
        cout << "Enter student " << i + 1 << "'s score: ";
        cin >> score;
        fprintf(fp, "ID=%s\tScore=%.2f\n", id, score);
    }

    fclose(fp);
    return 0;
}
```

运行结果

```
Enter number of students: 5
Enter student 1's ID: A001
Enter student 1's score: 92
Enter student 2's ID: A002
Enter student 2's score: 73
Enter student 3's ID: A003
Enter student 3's score: 89
Enter student 4's ID: A004
Enter student 4's score: 97
Enter student 5's ID: A005
Enter student 5's score: 65
```

运行结果 **data1.txt**

```
ID=A001 Score=92.00
ID=A002 Score=73.00
ID=A003 Score=89.00
ID=A004 Score=97.00
ID=A005 Score=65.00
```

为了统一对各种硬件的操作，不同的硬件设备也都被看作是文件进行管理。计算机中标准输入（stdin）是键盘、标准输出（stdout）是显示器、标准错误（stderr）是显示器。

因此，当使用 printf() 函数时，其实是从将数据输出到显示器上。printf() 函数是通过调用 fprintf(stdout, ...) 来实现的。

当需要输出一些错误信息时，可以通过 fprintf(stderr, ...) 将错误信息输出到标准错误 stderr 上，这样可以避免将错误信息混入到正常的输出信息中，方便查看和分析。

8.2.2 fscanf()

fscanf() 函数用于从文件中读取数据，使用方法与 scanf() 类似。fscanf() 的函数原型为：

```
1 int fscanf(FILE *stream, const char *format, ...);
```

fscanf() 函数读取成功时返回实际读取的数据个数，失败时返回文件末尾标志 EOF（End of File）。

平均分

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <iomanip>
4
5  using namespace std;
6
7  int main() {
8      FILE *fp = fopen("data1.txt", "r");
9      if (!fp) {
10         cerr << "File open failed." << endl;
11         exit(1);
12     }
13
14     char id[8];
15     double score;
16     double sum = 0;
17     int n = 0;
18
19     while (fscanf(fp, "ID=%s\tScore=%lf\n", id, &score) != EOF) {
20         sum += score;
21         n++;
22     }
23     cout << "Average = "
24          << fixed << setprecision(2) << sum / n << endl;
25
```

```
    fclose(fp);
    return 0;
}
```

运行结果

```
Average = 83.20
```

当使用 scanf() 函数时，其实是从键盘上读取数据的，scanf() 函数是通过调用 fscanf(stdin, ...) 来实现的。

8.2.3 fputc()

fputc() 函数用于将一个字符写入文件中，其函数原型为：

```
int fputc(int ch, FILE *stream);
```

fputc() 接受两个参数，第一个参数为要写入的字符（ASCII 码），第二个参数为文件指针。

当向屏幕输出一个字符时，fputc(stdout) 等价于 putchar()。

大写字母

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    FILE *fp = fopen("data2.txt", "w");

    for (int i = 0; i < 26; i++) {
        fputc('A' + i, fp);
```

```
    }

    fclose(fp);
    return 0;
}
```

运行结果 data2.txt

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

8.2.4 fgetc()

fgetc() 函数用于从文件中读取一个字符，读取成功返回字符的 ASCII 码，失败返回 EOF。fgetc() 的函数原型为：

```
int fgetc(FILE *stream);
```

当从键盘读取一个字符时，fgetc(stdin) 等价于 getchar()。

在读取文件时，除了可以通过返回值 EOF 来判断是否读取到文件末尾外，还可以使用 feof() 函数，当读到文件末尾时返回非 0 值，否则返回 0。feof() 的函数原型为：

```
int feof(FILE *stream);
```

源代码统计

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    FILE *fp = fopen("statistics.cpp", "r");
```

```
    if (!fp) {
        cerr << "File open failed." << endl;
        exit(1);
    }

    int chars = 0;
    int lines = 0;

    while (!feof(fp)) {
        char c = fgetc(fp);
        if (c == '\n') {
            lines++;
        } else {
            chars++;
        }
    }

    cout << "Characters: " << chars << endl;
    cout << "Lines: " << lines << endl;

    fclose(fp);
    return 0;
}
```

运行结果

```
Characters: 485
Lines: 29
```

8.2.5 fputs()

fputs() 函数用于将一个字符串写入文件，其函数原型为：

```
int fputs(const char *str, FILE *stream);
```

当向屏幕输出一个字符串时，fputs(stdout) 等价于 puts()。

Computer Science Quotes

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    const char *quotes[] = {
        "Talk is cheap. Show me the code.",
        "Code never lies, comments sometimes do.",
        "Stay Hungry Stay Foolish.",
    };
    int n = sizeof(quotes) / sizeof(quotes[0]);

    FILE *fp = fopen("data3.txt", "w");

    for (int i = 0; i < n; i++) {
        fputs(quotes[i], fp);
        fputc('\n', fp);
    }

    fclose(fp);
    return 0;
}
```

运行结果 data3.txt

```
Talk is cheap. Show me the code.
Code never lies, comments sometimes do.
Stay Hungry Stay Foolish.
```

8.2.6 fgets()

fgets() 函数用于从文件读取一行数据，读取成功返回指向字符串的指针，失败则返回 NULL。fgets() 的函数原型为：

```
char *fgets(char *str, int n, FILE *stream);
```

fgets() 接受三个参数，第一个参数用于保存读取到的字符串，第二个参数用于指定读取的最大字符数（包括结尾的 \0），第三个参数为文件指针。

解析单词

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <cctype>

using namespace std;

int main() {
    FILE *fp = fopen("data3.txt", "r");
    if (!fp) {
        cerr << "File open failed." << endl;
    }

    char line[128];
    while (fgets(line, sizeof(line), fp) != NULL) {
        char *token = strtok(line, " \t\n");
        while (token != NULL) {
            // remove punctuations
            int i = strlen(token) - 1;
            while (!isalpha(token[i])) {
                i--;
            }
            token[i + 1] = '\0';

            cout << token << endl;
```

```
26              token = strtok(NULL, " \t\n");
27          }
28      }
29
30      fclose(fp);
31      return 0;
32  }
```

运行结果

```
Talk
is
cheap
Show
me
the
code
Code
never
lies
comments
sometimes
do
Stay
Hungry
Stay
Foolish
```

8.3 fstream

8.3.1 fstream

fstream 库包含了三个文件 I/O 类：

1. fstream：可读写文件
2. ifstream：只能读文件
3. ofstream：只能写文件

在打开文件时可以指定文件的打开模式：

打开模式	功能
ios::in	只读
ios::out	只写
ios::app	追加
ios::ate	定位到文件尾
ios::trunc	如果文件存在，把文件长度设为 0

表 8.2: 打开模式

统计代码行数

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    ifstream in("code_lines.cpp");
    int line_count = 0;
    string line;
    while (getline(in, line)) {
```

```
        line_count++;
    }
    in.close();

    ofstream out("code_lines.txt");
    out << "File name: " << __FILE__ << endl;
    out << "Line count: " << line_count << endl;
    out.close();

    return 0;
}
```

运行结果 code_lines.txt

```
File name: code_lines.cpp
Line count: 22
```

Chapter 9 结构体

9.1 枚举

9.1.1 枚举（Enumeration）

枚举类型可以将一组相关的常量定义为一个类型，并为这些常量赋予一个可读性较高的名字。枚举类型在定义之后就可以像宏常量一样去使用。

例如需要定义一个星期：

```
1 enum Weekday {
2     SUN, MON, TUE, WED, THU, FRI, SAT
3 };
```

枚举值默认从 0 开始，因此 SUN 的值为 0、MON 的值为 1、TUE 的值为 2，以此类推。

当需要指定枚举值时，可以直接在某个枚举常量后赋值，之后的枚举常量的值会在此基础上依次加 1。

例如需要定义月份：

```
1 enum Month {
2     JAN = 1, FEB, MAR, APR, MAY, JUN,
3     JUL, AUG, SEP, OCT, NOV, DEC,
4 };
```

9.2 联合体

9.2.1 联合体（Union）

联合体允许在同一个内存位置存储不同类型的数据，联合体中多个变量共享同一块内存空间，因此联合体所占空间取决于占用空间最大的成员。这意味着在任意时刻，联合体的内存只能用于存储单个成员，这样可以有效节省内存。

联合体

```cpp
#include <iostream>

using namespace std;

union Value {
    int int_data;
    char char_data;
};

int main() {
    union Value val;

    val.char_data = 'A';
    cout << "val.int_data = " << val.int_data << endl;

    val.int_data = 97;
    cout << "val.char_data = " << val.char_data << endl;

    return 0;
}
```

运行结果

```
val.int_data = 65
val.char_data = a
```

9.3 结构体

9.3.1 结构体（Structure）

与联合体不同，结构体的成员在内存中占用不同的空间，因此结构体所占空间是所有成员占用空间的总和。

结构体通常用于存储复杂的数据类型，将一些相关的变量组合在一起。例如：

- 日期（年、月、日）

```
struct Date {
    int year;
    int month;
    int day;
};
```

- 坐标（横坐标、纵坐标）

```
struct Coordinate {
    double x;
    double y;
};
```

- 学生信息（姓名、出生日期、成绩）

```
struct Student {
    string name;
    struct Date date_of_birth;
    double score;
};
```

9.3.2 typedef

typedef 用于给数据类型定义别名，通过使用 typedef 可以简化结构体的声明，不用每次都加上 struct 关键字了。

```
typedef struct  {
    int year;
    int month;
    int day;
} Date;

typedef struct {
    string name;
    Date date_of_birth;
    double score;
} Student;
```

9.3.3 结构体指针

当结构体变量作为函数参数传递时，如果结构体变量很大，那么会消耗大量时间将结构体变量复制到函数的参数中。

为了避免这种情况，可以使用结构体指针作为函数参数，这样只需要将结构体变量的地址传递给函数，函数内部就可以直接访问结构体变量了。

使用-> 运算符可以访问结构体指针所指的结构变量中的成员。

倒数

```
#include <iostream>

using namespace std;

typedef struct {
```

```
    int numerator;
    int denominator;
} Fraction;

void reciprocal(Fraction *f) {
    if (f->numerator == 0) {
        cerr << "Error: Denominator cannot be zero." << endl;
        exit(1);
    } else {
        int temp = f->numerator;
        f->numerator = f->denominator;
        f->denominator = temp;
    }
}

int main() {
    Fraction fraction = {2, 5};  // 2/5

    cout << "Reciprocal of ";
    cout << fraction.numerator << "/" << fraction.denominator << " is ";
    reciprocal(&fraction);
    cout << fraction.numerator << "/" << fraction.denominator << endl;

    return 0;
}
```

运行结果

```
Reciprocal of 2/5 is 5/2
```

Chapter 10 面向对象

10.1 封装

10.1.1 类与对象

在面向对象编程中，把构成问题的事物分解成各个对象，每个对象都有自己的数据和行为，程序通过对象之间的交互来实现功能。

类（class）是一个模板，定义了对象的属性和方法，用来描述同一类对象的共同特征和行为。对象（object）是类的实例，它具有类定义的属性和方法。

关键字 new 实例化并返回一个指向类对象的指针，之后就可以通过访问对象的属性和方法来操作对象。使用 new 分配的对象，需要使用 delete 释放。

银行账户

```
#include <iostream>
#include <string>

using namespace std;

class BankAccount {
public:
    string owner;
    string account;
    double balance;

    void deposit(double amount) {
        balance += amount;
    }

```

```
    void withdraw(double amount) {
        balance -= amount;
    }
};

int main() {
    BankAccount account;
    account.owner = "Terry";
    account.account = "6250941006528599";
    account.balance = 50;

    cout << "Owner: " << account.owner << endl;
    cout << "Account: " << account.account << endl;
    cout << "Balance: " << account.balance << endl;

    account.deposit(100);
    cout << "Balance: " << account.balance << endl;

    account.withdraw(70);
    cout << "Balance: " << account.balance << endl;

    return 0;
}
```

运行结果

```
Owner: Terry
Account: 6250941006528599
Balance: 50
Balance: 150
Balance: 80
```

10.1.2 封装（Encapsulation）

封装是面向对象的重要原则，尽可能隐藏对象的内部实现细节。封装可以认为是一个保护屏障，防止该类的数据被外部随意访问。当要访问该类的数据时，必须通过指定的接口。合适的封装可以让代码更容易理解和维护，也加强了程序的安全性。

为了实现封装，需要对类的属性和方法进行访问权限的控制：

1. public：允许任何地方访问。
2. private：只允许在类的内部访问。
3. protected：只允许在类的内部和子类中访问。

通常会将类的属性设置为 private，然后对外提供一对 setter/getter 方法来访问该属性。

为了避免方法的参数与类的属性重名造成歧义，可以使用 this 关键字用来指代当前对象。

银行账户

```
#include <iostream>
#include <string>

using namespace std;

class BankAccount {
    private:
    const size_t ACCOUNT_DIGITS = 16;
    string owner;
    string account;
    double balance;

    public:
```

```
14      void setOwner(string owner) {
15          if (!owner.empty()) {
16              this->owner = owner;
17          }
18      }
19
20      string getOwner() {
21          return owner;
22      }
23
24      void setAccount(string account) {
25          if (account.length() == ACCOUNT_DIGITS) {
26              this->account = account;
27          }
28      }
29
30      string getAccount() {
31          return account;
32      }
33
34      void setBalance(double balance) {
35          if (balance >= 0) {
36              this->balance = balance;
37          }
38      }
39
40      double getBalance() {
41          return balance;
42      }
43
44      bool deposit(double amount) {
45          if (amount <= 0) {
46              return false;
47          }
48          balance += amount;
49          return true;
50      }
```

```
51
52      bool withdraw(double amount) {
53          if (amount <= 0 || amount > balance) {
54              return false;
55          }
56          balance -= amount;
57          return true;
58      }
59  };
60
61  int main() {
62      BankAccount account;
63      account.setOwner("Terry");
64      account.setAccount("6250941006528599");
65      account.setBalance(50);
66
67      cout << "Owner: " << account.getOwner() << endl;
68      cout << "Account: " << account.getAccount() << endl;
69      cout << "Balance: " << account.getBalance() << endl;
70
71      account.deposit(100);
72      cout << "Balance: " << account.getBalance() << endl;
73
74      account.withdraw(70);
75      cout << "Balance: " << account.getBalance() << endl;
76
77      return 0;
78  }
```

运行结果

```
Owner: Terry
Account: 6250941006528599
Balance: 50
Balance: 150
Balance: 80
```

10.2 构造函数

10.2.1 构造方法（Constructor）

构造方法是一种特殊的方法，会在创建对象时自动调用，用于创建并初始化对象。每个类可以有一个或多个构造方法，构造方法的名字必须和类名一致。构造方法没有返回值，返回值类型部分不写。

```
BankAccount() {
    owner = "admin";
    account = "0000000000000000";
    balance = 0;
}
```

如果一个类中没有写构造方法，系统会自动提供一个 public 的无参构造方法，以便实例化对象。如果一个类中已经写了构造方法，系统将不会再提供默认的无参构造方法。

```
BankAccount(string owner, string account, double balance) {
    if (!owner.empty()) {
        this->owner = owner;
    }

    if (account.length() == ACCOUNT_DIGITS) {
        this->account = account;
    }

    if (balance >= 0) {
        this->balance = balance;
    }
}
```

析构函数（Destructor）是一种特殊的方法，会在对象被销毁时自动调用，用于释放对象占用的资源。析构函数的名字必须和类名一致，前面加上 ，没有返回值，返回值类型部分不写。

```
~BankAccount() {
```

```
    // code
}
```

10.2.2 重载（Overload）

重载用于在同一个类定义多个同名方法，但是这些方法的参数列表不同。重载的主要用途是提供方法的多种版本，以便满足不同的需求。

重载还可以使代码更具可读性，因为它使得方法名更具描述性，而不必考虑特定的参数列表。

银行账户

```
#include <iostream>
#include <string>

using namespace std;

class BankAccount {
    private:
    const size_t ACCOUNT_DIGITS = 16;
    string owner;
    string account;
    double balance;

    public:
    BankAccount() {
        owner = "admin";
        account = "0000000000000000";
        balance = 0;
    }

    BankAccount(string owner, string account, double balance) {
        if (!owner.empty()) {
```

```
22              this->owner = owner;
23          }
24
25          if (account.length() == ACCOUNT_DIGITS) {
26              this->account = account;
27          }
28
29          if (balance >= 0) {
30              this->balance = balance;
31          }
32      }
33
34      void setOwner(string owner) {
35          if (!owner.empty()) {
36              this->owner = owner;
37          }
38      }
39
40      string getOwner() {
41          return owner;
42      }
43
44      void setAccount(string account) {
45          if (account.length() == ACCOUNT_DIGITS) {
46              this->account = account;
47          }
48      }
49
50      string getAccount() {
51          return account;
52      }
53
54      void setBalance(double balance) {
55          if (balance >= 0) {
56              this->balance = balance;
57          }
58      }
```

```
59
60      double getBalance() {
61          return balance;
62      }
63
64      bool deposit(double amount) {
65          if (amount <= 0) {
66              return false;
67          }
68          balance += amount;
69          return true;
70      }
71
72      bool withdraw(double amount) {
73          if (amount <= 0 || amount > balance) {
74              return false;
75          }
76          balance -= amount;
77          return true;
78      }
79
80      bool withdraw(double amount, double fee) {
81          if (amount <= 0 || amount + fee > balance) {
82              return false;
83          }
84
85          balance -= amount + fee;
86          return true;
87      }
88  };
89
90  int main() {
91      BankAccount account1;
92      cout << "Account 1 Owner: " << account1.getOwner() << endl;
93      cout << "Account 1 Account: " << account1.getAccount() << endl;
94      cout << "Account 1 Balance: " << account1.getBalance() << endl;
95
```

```
96      BankAccount account2("Terry", "6250941006528599", 50);
97      cout << "Account 2 Balance: " << account2.getBalance() << endl;
98
99      account2.withdraw(20);
100     cout << "Account 2 Balance: " << account2.getBalance() << endl;
101
102     account2.withdraw(10, 1);
103     cout << "Account 2 Balance: " << account2.getBalance() << endl;
104
105     return 0;
106 }
```

运行结果

```
Account 1 Owner: admin
Account 1 Account: 0000000000000000
Account 1 Balance: 0
Account 2 Balance: 50
Account 2 Balance: 30
Account 2 Balance: 19
```

10.3 友元

10.3.1 友元函数

封装使得类的数据对外隐藏，但是有些函数不是类的一部分，却又需要频繁访问类的数据成员，这时可以将这些函数定义为该类的友元函数。一个函数可以是多个类的友元函数，只需要在各个类中分别声明。

友元函数是可以直接访问类的私有成员和非成员函数。它是定义在类外的普通函数，它不属于任何类，但需要在类的定义声明。

友元的作用是提高程序的运行效率，但它破坏了类的封装性，使得非成员函数可以访问类的私有成员。

友元函数

```
#include <iostream>
#include <cmath>

using namespace std;

class Coordinate {
    private:
    double x;
    double y;

    public:
    Coordinate(double x, double y) : x(x), y(y) {}

    friend double distance(Coordinate &c1, Coordinate &c2);
};

double distance(Coordinate &c1, Coordinate &c2) {
    double delta_x = c1.x - c2.x;
    double delta_y = c1.y - c2.y;
```

```
20      return sqrt(delta_x * delta_x + delta_y * delta_y);
21  }
22
23  int main() {
24      Coordinate c1(0, 0);
25      Coordinate c2(3, 4);
26      cout << distance(c1, c2) << endl;
27      return 0;
28  }
```

运行结果

```
5
```

10.4 运算符重载

10.4.1 二元运算符重载

运算符重载是指在类中重新定义运算符的行为，使得运算符能够作用于类的对象上。

运算符重载通过定义运算符函数实现，函数名由关键字 operator 和要重载的运算符组成。

复数

```
#include <iostream>
#include <string>

using namespace std;

class Complex {
    private:
    double real;
    double imag;

    public:
    Complex(double real=0, double imag=0) : real(real), imag(imag) {}

    string getNumber() {
        return to_string(real) + "+" + to_string(imag) + "i";
    }

    Complex operator+(const Complex& c) {
        Complex complex;
        complex.real = this->real + c.real;
        complex.imag = this->imag + c.imag;
        return complex;
    }
```

```
};

int main() {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex result = c1 + c2;
    cout << result.getNumber() << endl;
    return 0;
}
```

运行结果

```
4.000000+6.000000i
```

10.4.2 输入输出运算符重载

通过重载输入输出流运算符可以实现对自定义类的输入输出。

复数

```
#include <iostream>
#include <string>

using namespace std;

class Complex {
    private:
    double real;
    double imag;

    public:
    Complex(double real=0, double imag=0) : real(real), imag(imag) {}

    Complex operator+(const Complex& c) {
```

```
        Complex complex;
        complex.real = this->real + c.real;
        complex.imag = this->imag + c.imag;
        return complex;
    }

    friend ostream& operator<<(ostream& os, const Complex& c);
};

ostream& operator<<(ostream& os, const Complex& c) {
    os << c.real << " + " << c.imag << "i";
    return os;
}

int main() {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex result = c1 + c2;
    cout << result << endl;
    return 0;
}
```

运行结果

```
4 + 6i
```

10.5 继承

10.5.1 继承（Inheritance）

继承指一个类可以继承另一个类的特征和行为，并可以对其进行扩展。这样就可以避免在多个类中重复定义相同的特征和行为。

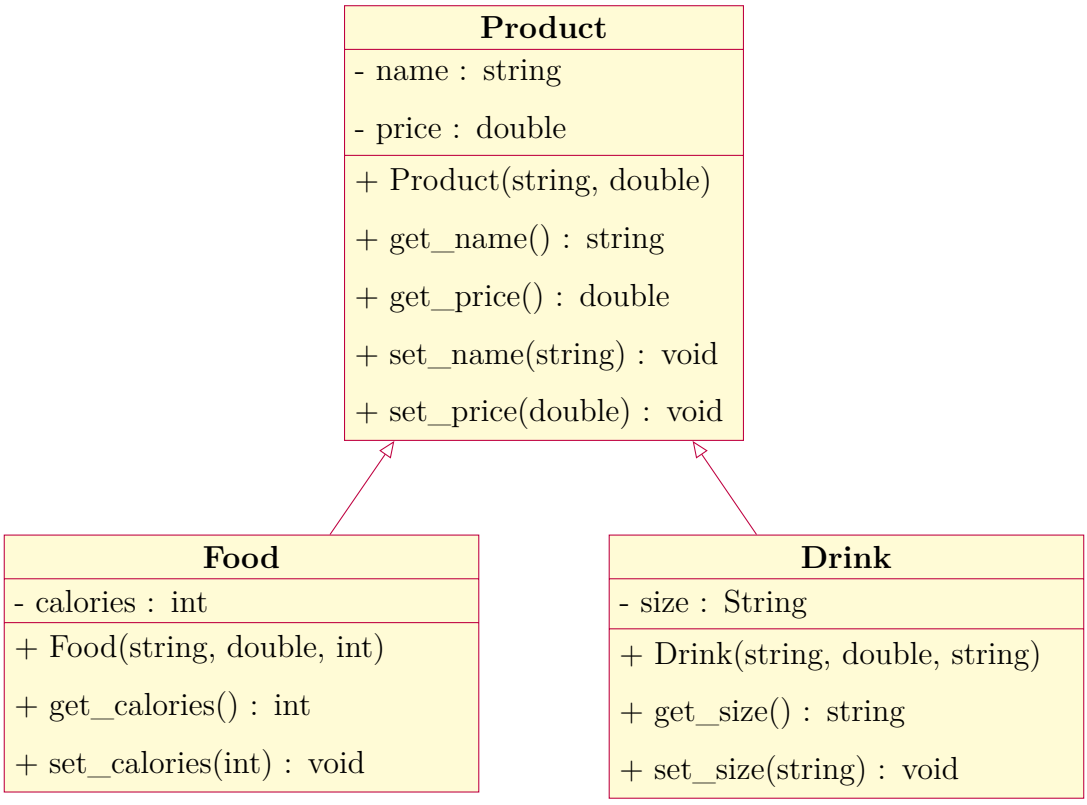


图 10.1: 继承

产生继承关系后，子类可以通过调用父类中的属性和方法，也可以定义子类独有的属性和方法。

在创建子类对象时，会先调用父类的构造方法，然后再调用子类的构造方法。因此父类中必须存在一个构造方法，否则将无法创建子类对象。

麦当劳

```
1 #ifndef _PRODUCT_H_
```

```
2  #define _PRODUCT_H_
3
4  #include <string>
5
6  class Product {
7      protected:
8      std::string name;
9      double price;
10
11     public:
12     Product(std::string name, double price);
13     std::string get_name();
14     void set_name(std::string name);
15     double get_price();
16     void set_price(double price);
17 };
18
19 #endif
```

```
1  #include "product.h"
2
3  Product::Product(std::string name, double price) {
4      this->name = name;
5      this->price = price;
6  }
7
8  std::string Product::get_name() {
9      return name;
10 }
11
12 void Product::set_name(std::string name) {
13     this->name = name;
14 }
15
16 double Product::get_price() {
17     return price;
```

```
}

void Product::set_price(double price) {
    this->price = price;
}
```

```
#ifndef _FOOG_H_
#define _FOOD_H_

#include <iostream>
#include <string>
#include "product.h"

class Food : public Product {
    private:
    int calories;

    public:
    Food(std::string name, double price, int calories);
    int get_calories();
    void set_calories(int calories);
    friend std::ostream& operator<<(std::ostream& os,
                                    const Food& food);
};

#endif
```

```
#include "food.h"

Food::Food(std::string name, double price, int calories)
    : Product(name, price) {
    this->calories = calories;
}

int Food::get_calories() {
```

```
    return calories;
}

void Food::set_calories(int calories) {
    this->calories = calories;
}

std::ostream& operator<<(std::ostream& os, const Food& food) {
    os << "Food: " << food.name
       << " ($" << food.price << ") "
       << food.calories;
    return os;
}
```

```
#ifndef _DRINK_H_
#define _DRINK_H_

#include <iostream>
#include <string>
#include "product.h"

class Drink : public Product {
    private:
    std::string size;

    public:
    Drink(std::string name, double price, std::string size);
    std::string get_size();
    void set_size(std::string size);
    friend std::ostream& operator<<(std::ostream& os,
                                    const Drink& drink);
};

#endif
```

```cpp
#include "drink.h"

Drink::Drink(std::string name, double price, std::string size)
    : Product(name, price) {
    this->size = size;
}

std::string Drink::get_size() {
    return size;
}

void Drink::set_size(std::string size) {
    this->size = size;
}

std::ostream& operator<<(std::ostream& os, const Drink& drink) {
    os << "Drink: " << drink.name
       << " ($" << drink.price << ") "
       << drink.size;
    return os;
}
```

```cpp
#include <iostream>
#include "food.h"
#include "drink.h"

using namespace std;

int main() {
    Food food("Cheeseburger", 5.45, 302);
    Drink drink("Coke", 3.7, "Large");

    cout << food << endl;
    cout << drink << endl;

    return 0;
```

```
15 }
```

运行结果

```
Food: Cheeseburger ($5.45) 302
Drink: Coke ($3.7) Large
```

10.6 抽象类

10.6.1 虚函数

有些类只能用来做继承，不能用于创建对象。例如在动物园中并不存在“动物”这个对象，只有动物的子类对象，因此动物类不应该被实例化。

抽象类是一种不能被实例化的类，它用于定义接口或公共实现，供其它类继承并实现。

有时候父类提供的方法无法满足子类不同的需求，但是如果不定义该方法，就表示该类具有该行为。

这种情况就可以将这个父类的方法定义为虚函数，这样所有的子类都必须要重写该方法，否则子类仍然为抽象类。

动物

```
#ifndef _ANIMAL_H_
#define _ANIMAL_H_

#include <string>

class Animal {
    public:
    virtual std::string sound() = 0;
};

#endif
```

```
#ifndef _DOG_H_
#define _DOG_H_

```

```
#include "animal.h"

class Dog : public Animal {
    public:
    virtual std::string sound() override;
};

#endif
```

```
#include "dog.h"
#include <iostream>

std::string Dog::sound() {
    return "Woof";
}
```

```
#ifndef _CAT_H_
#define _CAT_H_

#include "animal.h"

class Cat : public Animal {
    public:
    virtual std::string sound() override;
};

#endif
```

```
#include "cat.h"
#include <iostream>

std::string Cat::sound() {
    return "Meow";
}
```

```cpp
#include <iostream>
#include "dog.h"
#include "cat.h"

using namespace std;

int main() {
    Dog dog;
    Cat cat;

    cout << "Dog's sound: " << dog.sound() << endl;
    cout << "Cat's sound: " << cat.sound() << endl;

    return 0;
}
```

运行结果

```
Dog's sound: Woof
Cat's sound: Meow
```

10.7 多态

10.7.1 多态（Polymorphism）

多态是指对象可以具有多种形态，即同一个对象在不同时刻表现出不同的行为。例如 Dog 和 Cat 都是 Animal 的子类，因此可以将子类对象赋值给父类引用，从而产生多种形态。

```
1 Animal animal = new Dog();
```

由子类类型转型为父类类型，称为向上转型。由父类类型转型为子类类型，称为向下转型。

员工工资

```
1  #ifndef _EMPLOYEE_H_
2  #define _EMPLOYEE_H_
3
4  #include <string>
5
6  class Employee {
7      protected:
8      std::string name;
9
10     public:
11     Employee(std::string name);
12     std::string get_name();
13     virtual double get_salary() = 0;
14 };
15
16 #endif
```

```
1 #include "employee.h"
2
3 Employee::Employee(std::string name) {
```

```
    this->name = name;
}

std::string Employee::get_name() {
    return name;
}
```

```
#ifndef _FULL_TIME_EMPLOYEE_H_
#define _FULL_TIME_EMPLOYEE_H_

#include "employee.h"

class FullTimeEmployee : public Employee {
    private:
    double basic_salary;
    double bonus;

    public:
    FullTimeEmployee(std::string name,
                     double basic_salary,
                     double bonus);
    virtual double get_salary() override;
};

#endif
```

```
#include "full_time_employee.h"

FullTimeEmployee::FullTimeEmployee(std::string name,
                                   double basic_salary,
                                   double bonus)
        : Employee(name) {
    this->basic_salary = basic_salary;
    this->bonus = bonus;
}
```

```
10
11 double FullTimeEmployee::get_salary() {
12     return basic_salary + bonus;
13 }
```

```
1  #ifndef _PART_TIME_EMPLOYEE_H_
2  #define _PART_TIME_EMPLOYEE_H_
3
4  #include "employee.h"
5
6  class PartTimeEmployee : public Employee {
7      private:
8      double daily_wage;
9      int working_days;
10
11     public:
12     PartTimeEmployee(std::string name,
13                      double daily_wage,
14                      int working_days);
15     virtual double get_salary() override;
16 };
17
18 #endif
```

```
1  #include "part_time_employee.h"
2
3  PartTimeEmployee::PartTimeEmployee(std::string name,
4                                     double daily_wage,
5                                     int working_days)
6          : Employee(name) {
7      this->daily_wage = daily_wage;
8      this->working_days = working_days;
9  }
10
11 double PartTimeEmployee::get_salary() {
```

```
12      return daily_wage * working_days;
13  }
```

```
1   #include <iostream>
2   #include "full_time_employee.h"
3   #include "part_time_employee.h"
4
5   using namespace std;
6
7   int main() {
8       Employee *employees[2] = {
9           new FullTimeEmployee("Alice", 5783, 173),
10          new PartTimeEmployee("Bob", 150, 15),
11      };
12
13      for (Employee *employee : employees) {
14          cout << employee->get_name() << ": $"
15               << employee->get_salary() << endl;
16      }
17
18      return 0;
19  }
```

运行结果

```
Alice: $5956
Bob: $2250
```

Chapter 11 异常

11.1 异常

11.1.1 异常（Exception）

异常就是程序在运行过程中出现的非正常的情况，它可以被捕获并处理，以防止程序崩溃。

exception 是一个异常类，发生异常的时候会抛出一个异常对象。如果不处理异常，程序就会被中断。

异常	描述
std::exception	所有异常的父类
std::bad_alloc	内存分配失败
std::bad_cast	类型转换失败
std::logic_error	逻辑错误
std::invalid_argument	无效参数
std::out_of_range	超出有效范围
std::runtime_error	运行时错误
std::overflow_error	发生上溢
std::underflow_error	发生下溢

表 11.1: 常用异常

例如当数组访问越界时，会抛出一个 std::out_of_range 异常；当内存分配失败，会抛出一个 std::bad_alloc 异常。

11.1.2 捕获异常

try-catch 结构可以用于捕获并处理异常，将可能出现异常的代码放在 try 结构中，将异常处理的代码放在 catch 结构中。

当在 try 结构中出现异常时，程序会跳转到 catch 结构中，执行 catch 结构中的代码。一个异常被处理后，将不再影响程序的执行。

what() 是异常类提供的一个公共方法，它已被所有子异常类重载。

内存分配失败

```
#include <iostream>
#include <exception>

using namespace std;

int main() {
    try {
        int *arr = new int[0xffffffff];
        delete arr;
    } catch (bad_alloc &e) {
        cerr << e.what() << endl;
    }

```

```
14      return 0;
15  }
```

运行结果

```
std::bad_alloc
```

11.1.3 throw

throw 关键字用于抛出一个异常。

阶乘

```
1   #include <iostream>
2   #include <exception>
3
4   using namespace std;
5
6   int factorial(int n) {
7       if (n < 0) {
8           throw "Factorial of negative numbers is not defined.";
9       }
10
```

```
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;

    try {
        int fact = factorial(n);
        cout << n << "! = " << fact << endl;
    } catch (const char *e) {
        cerr << e << endl;
    }

    return 0;
}
```

运行结果

```
Enter n: -1
Factorial of negative numbers is not defined.
```

11.2 自定义异常

11.2.1 自定义异常

为了满足某些特定的需求，用户可以自定义异常，自定义异常继承于 exception 类或其子类。自定义异常的目的是为了提供更具体和有意义的错误处理。

库存

```
#ifndef _OUT_OF_STOCK_EXCEPTION_H_
#define _OUT_OF_STOCK_EXCEPTION_H_

#include <string>
#include <exception>

class OutOfStockException : public std::exception {
    private:
    std::string msg;

    public:
    OutOfStockException(std::string msg);
    virtual const char *what() const noexcept override;
};

#endif
```

```
#include "out_of_stock_exception.h"

OutOfStockException::OutOfStockException(std::string msg) {
    this->msg = msg;
}

const char *OutOfStockException::what() const noexcept {
    return msg.c_str();
}
```

```
#ifndef _PRODUCT_H_
#define _PRODUCT_H_

#include <string>
#include "out_of_stock_exception.h"

class Product {
    private:
    std::string name;
    int stock;

    public:
    Product(std::string name, int stock);
    void purchase();
};

#endif
```

```
#include "product.h"

Product::Product(std::string name, int stock) {
    this->name = name;
    this->stock = stock;
}

void Product::purchase() {
    if (stock == 0) {
        throw OutOfStockException(name + " is out of stock.");
    }
    stock--;
}
```

```cpp
#include <iostream>
#include "product.h"
#include "out_of_stock_exception.h"

using namespace std;

int main() {
    Product product("Cheeseburger", 50);

    try {
        for (int i = 0; i < 60; i++) {
            product.purchase();
        }
    } catch (OutOfStockException &e) {
        cerr << e.what() << endl;
    }

    return 0;
}
```

运行结果

```
Cheeseburger is out of stock.
```