

# 第四讲

## *Functions, Libraries*

---

薛浩

2023 年 3 月 23 日

[www.stickmind.com](http://www.stickmind.com)

- 话题 1：编程基础** 初学编程的新手，一般应该熟练使用函数和库处理字符串相关的编程任务。
- 话题 2：抽象数据类型的使用** 在尝试实现抽象数据类型之前，应该先熟练使用这些工具解决问题。
- 话题 3：递归和算法分析** 递归是一种强有力的思想，一旦掌握就可以解决很多看起来非常难的问题。
- 话题 4：类和内存管理** 使用 C++ 实现数据抽象之前，应先学习 C++ 的内存机制。
- 话题 5：常见数据结构和算法** 在熟练使用抽象数据类型解决常见问题之后，学习如何实现它们是一件很自然的事情。

# 话题 1: 编程基础

初学编程的新手，一般应该熟练使用函数和库处理字符串相关的编程任务。

- C++ 基础
- 函数和库
- 字符串和流

American Soundex		Daitch-Mokotoff Soundex	Phonetic Matching
Waagenasz	Wegonge	Bassington	Bassington
Wachenhausen	Weismowsky	Bazunachden	Vasington
Wacknocty	Weuckunas	Bechington	Washincton
Waczinjac	Wiggins	Bussington	Washington
Wagenasue	Woigemast	Fissington	
Waikmishy	Wozniak	Washington	
Washington	Wugensmid	Vasington	4 names
Washincton	...	Washincton	
Wassingtom	+ 3,900 more names	Wassington	
...			
		9 names	

Figure 1: 语音算法

**计算机如何实现抽象？**

1. 复习：库的使用
2. Function 函数
3. C++ 函数增强
4. 函数调用机制
5. 库的实现

## 复习：库的使用

---

现代编程依赖大量库的使用，当你创建一个应用软件时，真正需要编写的代码只占很小一部分。

使用 SimpleCxxLib 中的 `simpio.h` 接口可以获取用户输入：

```
int value = getInteger("Enter your value: ");  
double value = getReal("Enter your value: ");
```





# Function 函数

---

# Function 函数定义

函数定义的一般形式如下：

```
type name(parameter list) {  
    statements in the function body  
}
```

C++ 程序总是从 main 函数开始执行：

```
int main() {  
    /* ... code to execute ... */  
    return 0;  
}
```

# 练习: Perfect Number

[0, ~)

**完全数** (perfect number) 是一些特殊的自然数: 所有真因子 (即除了自身以外的约数) 的和, 恰好等于它本身。

$$6$$
$$1 \times \cancel{6}$$
$$2 \times 3 = 6$$

$$28$$
$$1 \times \cancel{28}$$
$$4 \times 7$$
$$2 \times 14$$

```
Console
Enter lower limit: 1
Enter upper limit: 10000
6
28
496
8128
```

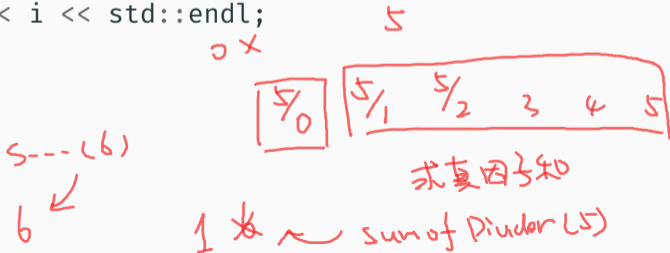
for [1, 10000]  
if (isPer-...) {  
 count ...  
}

isPerfect(int)  
true/f...

# 谓词函数

谓词函数 (predicate function) 是指返回 `bool` 类型值的函数, 例如 `isEven`, `hasChildren` 等。

```
int main() {  
    int lower = getInteger("Enter lower limit: ");  
    int upper = getInteger("Enter upper limit: ");  
    // TODO: Check user inputs  
    for (int i = lower; i <= upper; i++) {  
        if (isPerfect(i)) {  
            std::cout << i << std::endl;  
        }  
    }  
    return 0;  
}
```





**分解** (decomposition) 是将一个问题划分为可管理的片段的过程，是程序设计最基本策略。这样的设计过程被称为**逐步求精** (stepwise refinement)。

isPerfect 需要判断给定  $n$  的所有真因子的和是否恰好等于它本身：

```
bool isPerfect(int n) {  
    return (n != 0) && (n == sumOfDivisorsOf(n));  
}
```

sumOfDivisorsOf 需要计算给定  $n$  的所有真因子（即除了自身以外的约数）的和：

```
int sumOfDivisorsOf(int n) {  
    int total = 0;  
    for (int divisor = 1; divisor < n; divisor++) {  
        if (n % divisor == 0) {  
            total += divisor;  
        }  
    }  
    return total;  
}
```

# Forward Declaration 提前声明

**提前声明** (Forward Declaration) 也称函数原型，是给 C++ 编译器 提供函数信息的语句。

提前声明的语法格式如下：

```
return-type function-name(parameters);
```

例如，调用 `isPerfect` 函数之前需要这样声明：

```
bool isPerfect(int n);
```

## C++ 函数增强

---



## Overloading 重载

**重载** (Overloading) 表示如果多个不同的函数的参数不同, 则它们可以具有相同的名称。

所谓参数不同, 可以是不同的数量, 例如:

```
int sumOf(int a, int b) {  
    return a + b;  
}  
int sumOf(int a, int b, int c) {  
    return a + b + c;  
}
```

*sumOfTwo*

*sumOfThree*

可以这样使用:

```
cout << sumOf(1, 2) << endl;  
cout << sumOf(1, 2, 3) << endl;
```

## Overloading 重载

**重载** (Overloading) 表示如果多个不同的函数的参数不同, 则它们可以具有相同的名称。

所谓参数不同, 也可以是不同的类型, 例如:

```
int min(int a, int b) {  
    return (a < b) ? a : b;  
}
```

*min\_int*

```
double min(double a, double b) {  
    return (a < b) ? a : b;  
}
```

*min\_double*

可以这样使用:

```
cout << min(1, 2) << endl;  
cout << min(2.71, 3.14) << endl;
```



# Function Template 函数模板

函数模板 (Function Template) 可以用于函数体相同, 声明仅类型不同的函数。

```
int min(int a, int b) {  
    return (a < b) ? a : b;  
}  
double min(double a, double b) {  
    return (a < b) ? a : b;  
}
```

*Handwritten annotations:* Red boxes around 'a' and 'b' in the first function's return statement. Red arrows labeled 'true' point from '(a < b)' to 'a', and 'false' points from '(a < b)' to 'b'.

改写如下:

```
template <typename T>  
T min(T a, T b) {  
    return (a < b) ? a : b;  
}
```

*Handwritten annotations:* 'type T' written above the template parameter. 'int short ---' written above the function signature. A large red curly brace on the right side groups the template parameter and the function signature, with the Chinese characters '类型' (type) and '值' (value) written next to it.

## 函数调用机制

---

## Recursive Function 递归函数

最容易理解的递归示例就是函数，从定义中便清晰可见。例如，考虑阶乘函数，它可以通过以下任一方式定义：

$$\underline{n!} = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$$

$$n! = \begin{cases} 1 & \dots \text{ if } n = 0 \\ n \times \underline{(n - 1)!} & \text{ if } n > 0 \end{cases}$$

## Recursive Function 递归函数

第二种形式可以直接转换成 C++ 代码：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases}$$

```
int fact(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fact(n - 1);  
    }  
}
```

*Handwritten notes:*  
(n == 0) [?] 1 [:] n \* fact(n-1)  
true false

# Factorial

```
int main() {  
    cout << "4! = " << fact(4) << endl;  
    return 0;  
}
```

fact(4) ??

*n x ... x 1  
for*

# Factorial

```
int main() {  
    int fact(int n) {  
        int n 4  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * fact(n - 1);  
        }  
    }  
}
```

fact(3) ??

↑  
4



# Factorial

```
int main() {  
  int fact(int n) {  
    int n 4  
    int fact(int n) {  
      int n 3  
      if (n == 0) {  
        return 1;  
      } else {  
        return n * fact(n - 1);  
      }  
    }  
  }  
}
```

fact(2) ??

↑  
3

# Factorial

```
int main() {  
  int fact(int n) {  
    int n 4  
    int fact(int n) {  
      int n 3  
      int fact(int n) {  
        int n 2  
        if (n == 0) {  
          return 1;  
        } else {  
          return n * fact(n - 1);  
        }  
      }  
    }  
  }  
}
```

fact(1) ??

↑  
2

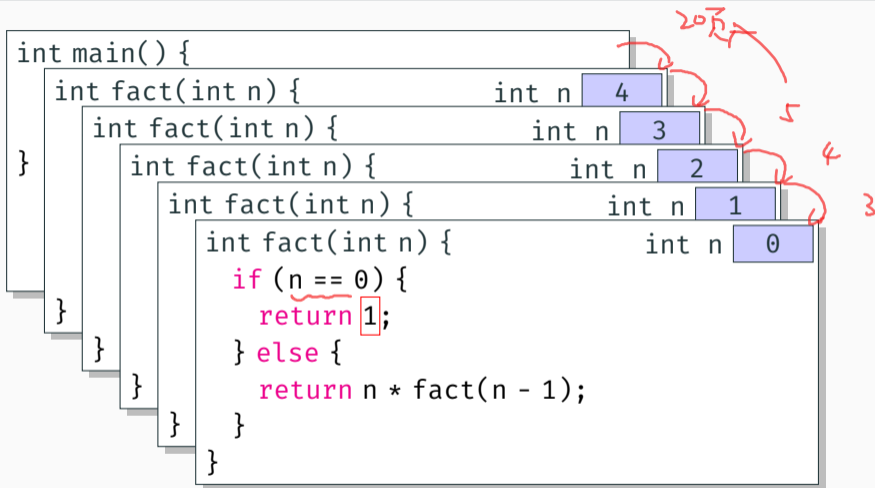
# Factorial

```
int main() {  
  int fact(int n) {  
    int n 4  
    int fact(int n) {  
      int n 3  
      int fact(int n) {  
        int n 2  
        int fact(int n) {  
          int n 1  
          if (n == 0) {  
            return 1;  
          } else {  
            return n * fact(n - 1);  
          }  
        }  
      }  
    }  
  }  
}
```

fact(0) ??

fact -- 1  
↑ fact -- 2  
fact -- 3  
fact -- 4  
main

# Factorial



# Factorial

```
int main() {  
  int fact(int n) {  
    int n 4  
    int fact(int n) {  
      int n 3  
      int fact(int n) {  
        int n 2  
        int fact(int n) {  
          int n 1  
          if (n == 0) {  
            return 1;  
          } else {  
            return n * fact(n - 1);  
          }  
        }  
      }  
    }  
  }  
}
```

The diagram illustrates the recursive call stack for calculating the factorial of 4. It shows four nested function calls for `fact(4)`, `fact(3)`, `fact(2)`, and `fact(1)`. The `fact(1)` call is highlighted with a red box around `fact(n - 1)`, a red arrow pointing to `1`, and a blue callout box containing `1`.

# Factorial

```
int main() {  
  int fact(int n) {  
    int n 4  
    int fact(int n) {  
      int n 3  
      int fact(int n) {  
        int n 2  
        if (n == 0) {  
          return 1;  
        } else {  
          return n * fact(n - 1);  
        }  
      }  
    }  
  }  
}
```

# Factorial

```
int main() {  
    int fact(int n) {  
        int n 4  
        int fact(int n) {  
            int n 3  
            if (n == 0) {  
                return 1;  
            } else {  
                return n * fact(n - 1);  
            }  
        }  
    }  
}
```

# Factorial

```
int main() {  
    int fact(int n) {  
        int n 4  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * fact(n - 1);  
        }  
    }  
}
```

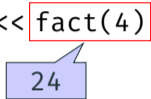
6

4



# Factorial

```
int main() {  
    cout << "4! = " << fact(4) << endl;  
    return 0;  
}
```



## 库的实现

---

*Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences. As humans, we do this all the time, as a means of coping with the world. It's what lets us talk about "chairs" without having a specific chair in mind.*

—Prabhakar Ragde

使用 `simpio.h` 接口获取用户输入时，我们已经受益于函数抽象。函数抽象允许客户调用实现者编写的函数，而不必了解其是如何实现的。

```
int value = getInteger("Enter your value: ");  
double value = getReal("Enter your value: ");
```

函数抽象创建了两个角色：客户和实现者——客户是调用函数的程序员，而实现者是编写函数的程序员。

客户和实现者的交会点称为**接口**，既是一个屏障，也是一个沟通的渠道。

## simpio.h 接口

```
// simpio.h
#ifndef _simpio_h // import guard
#define _simpio_h

#include <string>

int  getInteger(std::string prompt = "");
double getReal(std::string prompt = "");
std::string getLine(std::string prompt = "");

#endif
```

## 库的创建

在 C++ 中，接口通常以头文件 `.h` 的形式存在；而实现通常以 `.cpp` 文件形式存在。

```
OurLib
├── CMakeLists.txt
├── include
│   ├── direction.h
│   ├── error.h
│   ├── gmath.h
│   └── random.h
└── src
    ├── direction.cpp
    ├── error.cpp
    ├── gmath.cpp
    └── random.cpp
```

计算机如何实现抽象？

A word cloud of programming concepts. The words are arranged in a roughly circular pattern. The word 'Function' is the largest and most central. Other words include 'Library', 'Overloading', 'Interface', 'Recursion', 'Template', 'Call', and 'Implementation'. Handwritten annotations include a red circle around 'Library', a red star to the left of 'Function', and a red line underlining 'Function' and 'Recursion'.

Library  
Overloading  
Interface  
Function  
Recursion  
Call Template  
Implementation



问题?