第5章 函　数

本章将系统讲解C++函数的核心机制与应用方法，培养学生的模块化编程能力。具体目标包括以下几点。

(1) 掌握函数的声明、定义和调用规范，理解形参与实参的传递机制(值传递、引用传递)。

(2) 熟练运用函数重载特性，能够根据需求设计同名不同参的函数。

(3) 学习标准库函数的调用方法，认识数学库、字符串处理库等常用函数库。

(4) 初步了解递归函数的实现原理和应用场景，能够编写简单的递归算法。

(5) 培养模块化设计思维，学会将复杂问题分解为若干函数实现，提升代码的复用性。

通过本章学习，学生将具备使用函数构建结构化程序的能力，理解函数库在软件开发中的价值，为后续面向对象编程和大型项目开发奠定基础。

5.1 函数的定义、声明和调用

5.1.1 函数的定义

C++中的函数由函数头和函数体组成，其中函数头包括返回值类型、函数名称和参数列表，函数体则是由花括号括起来的一系列语句和表达式组成。函数定义的一般形式如下：

```
返回值类型 函数名(类型1 参数名1,类型2 参数名2, ...)
{
    语句代码;
    return 返回值;
}
```

其中，返回值类型是指函数代码执行完成后返回结果的数据类型。函数名代表函数的名称，命名要符合标识符定义规范。函数名后跟着的(类型1 参数1,类型2 参数2,…,类型n 参数n)是参数列表，表示运行函数时所要接收的外部数据的符号表示，因此也称形式参

数表。通过参数列表，函数可以根据不同的输入执行相应的操作，使得函数具有很高的灵活性，也提高了代码的复用性。函数体中的 return 语句用于返回函数执行结果的值。例如定义两个整数相加求和的函数：

```
int add(int a, int b){
    int sum=a+b;
    return sum;
}
```

这段代码定义了一个名为 add 的两个整数相加的函数，它有两个整型参数 a 和 b，相加的结果保存在 sum 中，计算完成后返回 sum，sum 的类型是整型，因此函数的返回值类型也是整型。

5.1.2 函数的声明

1. 函数声明的格式

C++ 中所有的变量和函数都遵循先声明后使用的原则。函数声明是告诉编译器已经存在相关的函数信息（返回值类型、参数个数和参数类型），编译器可以根据这些信息检查函数调用的正确性。在调用函数时，如果函数定义在函数调用前，则不需要额外声明；如果函数调用在函数定义之后，则需要进行声明。函数声明的语法如下：

```
返回值类型  函数名称(参数列表);
```

函数声明也称为函数原型，例如下面是对前面定义的 add()函数的声明：

```
int add(int a, int b);
```

这个声明告诉编译器有一个名为 add 的函数，接收两个 int 类型的参数 a 和 b，并返回一个 int 类型的值。有了这些信息后，编译器就可以检查函数调用的语法正确性。在函数声明中参数名称没有实际含义，是一个占位符，可以省略，所以 add()函数也可以这样声明：

```
int add(int, int);
```

2. 函数定义(声明)的可选部分

1) constexpr

constexpr 表示函数的返回值是常量值，可以在编译时进行计算。例如函数 exp()：

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

constexpr 说明符声明的函数可以在编译时对函数或变量求值，可用于需要编译常量表达式的地方。如果一个函数在编译时可以求值，则可以优化程序的执行效率。要使函数成为 constexpr 函数，必须满足以下几个要求。

- 函数体内的代码必须只包含常量表达式。
- 不能使用运行时输入的值，除非它们本身是常量表达式。
- constexpr 函数可以包含条件语句、循环等结构，但必须在编译时就能被计算出。

在以上定义的 constexpr 函数 exp()中，如果传入的参数 x 和 n 是常量表达式，则 exp()函数会在编译时求值，否则会在运行时求值（非 constexpr 函数）。

2）extern(外部函数)

当 extern 用于函数声明时，它表示该函数的定义存在于另一个源文件中。实际上在函数声明时 extern 是多余的，因为 C++ 默认将函数声明视为外部函数，但显式地加上 extern 可以增强代码的可读性，表明该函数的定义不在当前源文件中。例如：

```
extern void printMessage();                    //声明在其他文件中定义的函数
```

C++ 中的函数名通常会进行“名称修饰”，使得 C++ 编译器生成的符号名称与 C 不兼容。如果希望 C++ 函数以 C 语言风格的符号名称进行链接(即避免名称修饰)，可以使用 extern "C"。例如：

```
extern "C" {
    void myFunction();                         //使用 C 语言风格链接
}
```

3）noexcept

noexcept 用于指示一个函数不会抛出任何异常。通过在函数声明或定义前加上 noexcept，可以明确地告诉编译器该函数不抛出异常，可以允许编译器进行优化以减少异常处理相关的开销(异常处理参见 7.3 节)。例如：

```
void foo() noexcept;                           //声明 foo()函数不会抛出异常
```

4）inline(内联)

inline 关键字建议编译器将对函数的每个调用替换为函数代码本身，编译器在编译程序时，直接将函数的代码插入调用位置，避免函数调用时的开销。内联函数一般用在函数较小且调用频繁的情况下。内联函数的定义和普通函数类似，只是函数声明或定义前加上了 inline 关键字。例如：

```
inline int add(int a, int b) {
    return a + b;
}
```

add()是一个内联函数，编译器在编译时会尽可能将 add()函数的调用替换为“a + b”，从而避免了函数调用的开销。inline 是对编译器的建议，实际的行为由编译器决定。编译器会根据以下因素来决定是否执行内联函数。

- 函数的大小。如果函数非常复杂或体积较大，编译器可能不会进行内联展开，甚至忽略 inline 关键字。
- 递归函数。递归函数一般不会被内联展开，因为它涉及复杂的栈操作。
- 多次定义。内联函数可以在多个源文件中定义，并且允许不同的翻译单元包含相同的内联函数定义。这与普通函数在多个文件中的定义不同(普通函数在多个文件中定义时会引发链接错误)。

3. 函数参数的默认值

在函数中，可以为参数提供默认值，有默认值的函数参数称为默认参数。例如：

```
int print(double dvalue, int prec = 2);
```

在函数 print()中，第二个参数具有默认值 2，称为默认参数。由于在函数声明中，函数参数的名字可以省略，也可以这样声明：

```
int print(double , int = 2);
```

使用默认参数时，请注意以下几点。

（1）默认参数必须是最后的参数。如果函数有多个参数，则一个默认参数右边的其他参数必须全部是默认参数。因此以下代码有语法错误：

```
int print(double dvalue = 0.0, int prec);
                                        //dvalue 参数有默认值，但其右边的参数 prec 没有
```

（2）默认参数不能在函数的声明和定义中重复出现，需要在函数名称第一次出现时定义。因此以下代码有语法错误：

```
//print()函数声明，声明中有默认参数
int print(double dvalue, int prec = 2);
...
//print()函数定义，定义中又定义了默认参数，所以有语法错误
int print(double dvalue, int prec = 2)        //默认参数需要在函数名第一次出现时定义
{
...
}
```

5.1.3 函数的调用

1. 函数调用语法

函数调用是指在代码中使用函数名和参数来执行函数功能的过程。函数调用的语法结构如下：

```
函数名(参数值 1,参数值 2,…,参数值 n);
```

其中，“(参数值 1,参数值 2,…,参数值 n)”是与定义函数时的参数列表一一对应的具体的值，因此也称实参列表。当调用函数时，程序会将控制权传递给函数，开始执行函数体中的语句。当函数执行完毕后，程序会将控制权返回到调用函数的位置，再通过 return 语句返回函数的执行结果。例 5-1 是函数定义、声明和调用的完整示例。

【例 5-1】 简单函数的定义和调用。

```
#include <iostream>
using namespace std;
int add(int a, int b);                          //函数声明
int main()
{
    int result = add(3, 4);                     //函数调用
    cout << "The result is: " << result << endl;
    return 0;
}
//以下是函数定义
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

在例 5-1 中，也可以将函数的定义部分写到 main()函数的前面，此时可以不需要函数声明，由此可以看出，函数定义的同时就具有声明的作用。代码较少时，这样定义显得相对简洁，但是当程序需要大量函数时，这样写就不利于程序代码的分析。通常会将函数的声明和定义分离开，这样便于程序开发人员直接从 main()函数开始分析代码和调试程序。

【例 5-2】 函数的声明和定义分开。

```
#include <iostream>
using namespace std;
//以下是函数定义
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
int main()
{
    int result = add(3, 4);                    //函数调用
    cout << "The result is: " << result << endl;
    return 0;
}
```

在实际编程中，通常会将函数声明放在头文件中，并在源文件中包含相应的头文件。这样可以确保在其他源文件中使用该函数时，编译器能够正确地检查参数类型和返回类型。同时，将函数定义放在源文件中可以避免重复定义的问题。

函数调用语句是表达式语句，表达式的类型是函数返回值的类型，表达式的值是函数的返回值。既然函数调用是表达式语句，所有表达式语句可以出现的形式，函数调用语句都可以出现。

函数调用时要求实参和形参一一对应，在函数调用时，如果实参的数据类型和形参的数据类型不同，编译器先试图使用自动（隐式）类型转换将实参类型转换为形参类型，转换成功后再按照转换后的类型调用函数；如果转换不成功或者无法进行自动（隐式）类型转换，编译器将报错。

2. 有默认参数的函数调用

函数定义时的默认形参，有以下两种调用方式（参见例 5-3）。

（1）传递了实参，则使用实参调用函数。

（2）没有传递实参，则使用形参的默认值调用函数。

【例 5-3】 有默认参数的函数的定义和调用。

```
#include <iostream>
using namespace std;
int add(int a=0, int b=0);      //函数声明，形参 a 和 b 有默认值
int main()
{
    int result_1 = add(3, 4);   //函数调用，以实参调用
    int result_2 = add(3);      //等价于 add(3,0)，第二个实参没有传递，使用默认值 0
    int result_3 = add();       //等价于 add(0,0)，第一个和第二个实参没有传递，均使用
                                //默认值 0
    cout << "The result is: " << result_1 << endl;
    return 0;
}
//以下是函数的定义
int add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```

3. 函数调用的执行过程

函数调用的执行过程涉及参数传递、栈帧的创建与销毁、函数体的执行等过程。

1）准备参数

调用函数前，首先需要准备传递给函数的实参值，如计算表达式的值、取变量的值等，具体根据参数传递方式的不同而不同。

2）创建栈帧

系统会在栈上为被调用函数的运行创建一个新的栈帧(stack frame)。栈帧是在栈上分配的一块独立的内存区域，用于存储其运行内容。

- 局部变量。函数内部声明的局部变量。
- 函数参数。传递给函数的参数，函数参数是向被调用函数传递数据的唯一途径。
- 返回地址。调用者代码中下一条指令的地址，以便函数返回后继续执行。
- 保存的寄存器。某些寄存器的值可能需要保存，以便在函数返回后恢复。

3）传递参数

参数传递的方式取决于函数的调用约定，常见的调用约定包括 cdecl(参数从右到左入栈，调用者负责清理栈)、stdcall(参数从右到左入栈，被调用者负责清理栈)、fastcall(部分参数通过寄存器传递，减少栈操作，提高效率) 等。参数的传递方式有值传递、指针传递和引用传递 3 种。

4）调用函数

调用指令(如 call 汇编指令)将控制权转移到函数的入口点。此时，程序计数器(PC)指向函数的第一条指令。

5）执行函数体

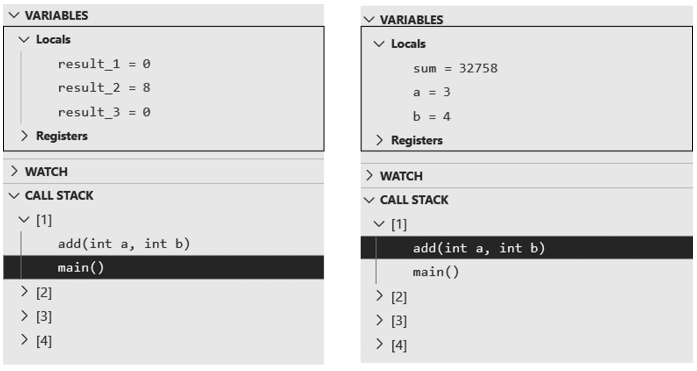
进入函数后，开始执行函数体中的代码，函数代码在独立的栈帧中执行。包括以下过程。

(1) 初始化局部变量。分配并初始化局部变量。

(2) 执行函数逻辑。执行函数中的各种语句，如赋值、条件判断、循环等。

(3) 调用其他函数。如果函数内部调用了其他函数，上述过程会重复进行。

例 5-3 程序的执行过程如图 5-1 所示，程序中有 main()和 add()两个函数，每个函数运行时的栈帧不同，main()函数的栈帧为图 5-1(a)，add()函数的栈帧为图 5-1(b)。



(a) (b)

图 5-1 例 5-3 程序的执行过程

6）返回值

如果函数有返回值，会在适当的位置（如寄存器或栈）设置返回值。

7）销毁栈帧

函数执行完毕后，需要销毁栈帧，恢复调用者的上下文。包括以下过程。

（1）弹出栈上的参数。如果参数是通过栈传递的，需要从栈中移除。

（2）恢复寄存器。恢复之前保存的寄存器值。

（3）恢复栈指针。将栈指针恢复到调用前的位置。

分配的函数栈帧销毁后，其执行过程中的局部变量失效。

8）返回调用者

控制权返回到调用函数的下一条语句，继续执行后续代码。

5.2 参数传递

参数传递是指在调用函数时，将实际参数（实参）传递给形式参数（形参）的过程。参数传递是函数间通信的一种方式，它使得函数能够根据提供的输入执行特定的操作，并可能返回结果。根据函数定义给出参数的形式，可以有值传递、引用传递、指针传递三种参数传递方式。

当程序代码执行到函数调用语句时，会首先为该函数的执行分配一段独立的栈帧空间，函数中定义的变量（包括形式参数）均在这段独立的栈帧空间中，函数代码也在这段独立的栈帧空间中执行。调用函数（在 main()函数代码中调用 add()函数，则 main()函数称为调用函数，add()函数称为被调用函数）和被调用函数之间通过参数传递和函数返回值进行通信。

5.2.1 值传递

值传递（Pass by Value）是一种最常见的参数传递方式，它在函数调用时将实际参数的值复制到一个形式参数中。由于形式参数和实际参数都有独立的存储空间，所以在函数内部对形式参数的任何修改都不会影响到实际参数的值，值传递是实际参数向形式参数的单向传递过程。以下是一个值传递的例子。

【例 5-4】 值传递。

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
int main() {
    int a = 10, b = 20;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
```

在上述例子中，swap()函数接收两个整数参数 x 和 y，当在 main()函数中调用 swap(a, b)函数时，a 的值被复制到 x 中，b 的值被复制到 y 中。因此，在 swap()函数内部对 x 和 y 的修改不会影响到 a 和 b 的值。输出结果显示 a 和 b 的值仍然是 10 和 20。

值传递的优点是简单易懂，且不会影响实际参数的值。然而对于占用存储空间较大的数据类型(如数组、结构体、对象)，由于值传递需要将实际参数的值复制到形式参数中，可能会出现性能问题。在这种情况下可以考虑使用引用传递或指针传递。

5.2.2 引用传递

引用传递是指在调用函数时，将实际参数的内存地址传递给形式参数，而不是参数的值本身。因此函数中对参数的任何修改都会直接反映到实际参数中，是实际参数和形式参数之间的双向传递过程。

引用传递不需要创建参数的副本，特别适合传递大型数据结构，如数组和对象，以避免复制数据的开销。同时引用传递允许函数修改传递给它们的变量，这在需要修改外部变量时非常有用，但如果函数意外修改了参数，也可能会导致不可预见的结果，尤其是当参数是全局变量或共享变量时，降低了代码的安全性。下面是一个通过引用传递交换两个变量值的代码。

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
int main() {
    int a = 10, b = 20;
    swap(a, b);
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
```

在上述代码中定义了一个 swap()函数，该函数使用两个整数的引用作为参数，并交换它们的值。在 main()函数中，传递了 a 和 b 两个整型变量，因此在 swap()函数中交换它们的值后，实际参数 a 和 b 的值也被交换了。

5.2.3 指针传递

指针传递是指在调用函数时将实际参数的地址传递给形式参数，这样函数就可以通过指针变量来访问和修改实际参数的值。例 5-5 是一个指针传递的例子。

【例 5-5】 指针参数的使用。

```
#include <iostream>
using namespace std;
void increment(int *ptr)            //函数 increment()的参数的为指针参数
{
    (*ptr)++;                       //通过指针变量、解引用运算符访问并修改实际参数
}
int main()
{
    int num = 10;                   //定义整型变量 num
    cout<<num<<endl;
```

```
    increment(&num);                    //调用函数 increment(),实际参数为变量 num 的地址
    cout<<num<<endl;                    //输出结果
    return 0;
}
```

在例 5-5 中，increment()函数的形式参数是指向 int 类型的指针，在程序中可以通过指针变量和解引用运算符来修改实际参数的值。

5.2.4 数组作为函数的形参

在程序中，数组作为函数的形式参数是经常遇到的任务，如按照统一规则修改数组中元素的值、输出数组元素等。如何将数组作为参数传递给函数访问呢？

1. 数组作为函数形参

在 C++ 中数组作为函数形式参数时，实际上传递的是数组的首地址。函数中接收到的是指向数组第一个元素的指针，这种传递方式避免了复制整个数组的内容，比较高效。

以下代码是打印一维数组中的元素：

```
#include <iostream>
void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

在上述代码中，int arr[] 实际上在函数内部被当作 int * arr 来处理，函数中的参数 size 必不可少，因为在函数内部仅通过指针 arr 无法确定数组的大小。所以函数的形式参数是一维数组时，数组参数实际上是指针参数，在函数中无法知道数组的大小，需要通过另外一个参数来传递数组大小。参数传递时可以传递不同长度的一维数组。

二维数组作为函数参数时，第二维的大小必须明确指定，编译器需要根据第二维的大小来正确地计算数组元素的偏移量。打印二维整数数组的元素的代码如下：

```
void print2DArray(int arr[][3], int rows) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << arr[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

在上述代码中，int arr[][3] 实际上在函数内部被当作 int (* arr)[3]来处理，如图 5-2 所示，可以看出 arr 实际上是一个指向数组的指针，参数传递时形参数组和实参数组第二维的大小必须相同。

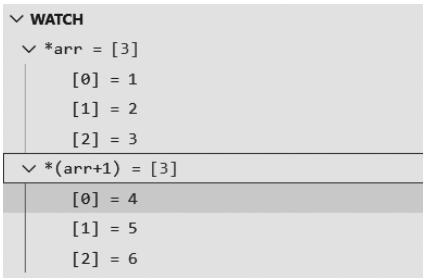


图 5-2 二维数组作为函数参数

除了传递数组指针，C++ 还允许传递数组的引用，这样会使代码更加清晰。例如：

```
void printArrayRef(int (&arr)[5]) {
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << " ";
```

```
    }
    std::cout << std::endl;
}
```

在上述代码中，int (&arr)[5]表示 arr 是一个对 5 个整数数组的引用，这种方式的优点是数组的大小是引用的一部分，不需要额外的参数来指定大小。缺点是函数只能接收指定大小的数组，灵活性不足。

2. std::span 作为函数形参来表示数组

std::span 是 C++ 20 标准引入的视图(view)类型，它提供了一种轻量级的方式来引用一段连续的内存，而不复制数据。当使用 std::span 作为函数形参时，可以接收常规数组类型(不需要传递数组的大小)、C++ 容器类中的 vector 和 array 类作为实参而不需要复制数据。在使用 std::span 作为函数形参时，需要使用<数据类型>指定一段连续内存空间中存储数据的数据类型。

```
void printArray(std::span<int> arr) {  //使用 std::span 表示数组，指定存储的元素类
                                       //型为 int
    for (auto el:arr) {                //遍历数组，类型用 auto 自动推断
        std::cout << el << " ";        //访问数组元素
        el = el * 2;                   //修改数组元素
    }
    std::cout << std::endl;
}
```

5.3 返回值

5.3.1 返回单个值

函数返回值是指函数执行完毕后，返回给调用函数的结果，它可以是函数执行过程中计算得到的结果，也可以是函数内部变量的值。返回值可以是任何数据类型，包括基本数据类型(如整数、浮点数、字符)和复杂数据类型(如数组、对象、结构体)，此外还能以地址、引用等形式返回。

```
//返回引用
int &get(int *arr, int index){
    return arr[index];
}
int main(){
    int arr[5] = {1, 2, 3, 4, 5};
    get(arr, 3) = 10;
    cout << arr[3] << endl;
    return 0;
}
//返回输出结果:10
```

上述代码定义了一个 get()函数，该函数返回一个整型变量的引用。在 main()函数中，使用 get()函数获得 arr[3]的引用，并将其设置为 10。由于 get()函数返回的是变量的实际引用，因此数组 arr 中的对应元素也被更新为 10。

函数也可以返回指向变量的指针。在这种情况下，函数返回的是指向变量的地址，而不