第五章 函数

目录

学习目标

1. 掌握函数的定义，以及常用的参数传递方式和值返回方式；
2. 理解函数调用机制及对象生命期的概念；
3. 能够根据需要编写具有一定实际用途的函数；
4. 掌握递归程序设计方法和多文件结构的使用；

5.1 认识函数

函数

具有名字的语句块。

- 通过调用函数的名字可以执行相应的代码块；
- 模块化程序设计的基础。

函数四要素

- 返回值类型 (return type)；
- 函数名 (function name)；
- 参数列表 (parameter list)；
- 函数体 (function body)。

5.1 认识函数 — 定义函数

函数四要素

- 返回值类型 (return type)；
- 函数名 (function name)；
- 参数列表 (parameter list)；
- 函数体 (function body)。

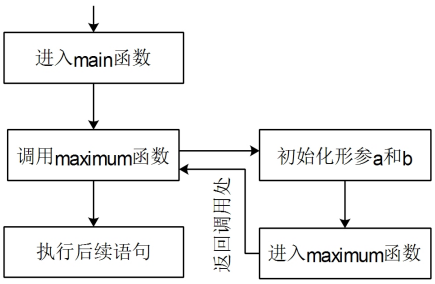
maximum 函数: 返回两个整数中较大的数

```
int maximum(int a, int b) { //a和b为两个int类型形式参数(简称形参)
    int c;                  //用来保存结果
    c=a > b ? a : b;
    return c;               //返回结果
}
```

调用 maximum 函数

```
int maximum(int a, int b) { //被调函数
    int c;
    c=a > b ? a : b;
    return c;
}
int main() {          //主调函数
   int x, y, z;
   cin >> x >> y;
                      //调用函数maximum
   z = maximum(x, y); //x和y为实际参数(简称实参)
   cout << "The maximum value is " << z << endl;
   return 0;
}
```

函数调用过程（如下）：



调用规则

- 调用处的函数名要和被调函数的函数名一致；

调用规则

- 调用处的函数名要和被调函数的函数名一致；
- 实参和形参存在一一对应的关系，类型要兼容；

调用规则

- 调用处的函数名要和被调函数的函数名一致；
- 实参和形参存在一一对应的关系，类型要兼容；
- 实参可以是左值或右值，但形参必须是左值（why?）。

调用规则

- 调用处的函数名要和被调函数的函数名一致；
- 实参和形参存在一一对应的关系，类型要兼容；
- 实参可以是左值或右值，但形参必须是左值（why?）。

maximum 函数

```
int maximum(int a, int b)
{
    int c;
    c = a > b ? a : b;
    return c
}
```

是否正确调用 maximum 函数。

```
maximum(1);           //错误：实参数目不足
maximum("c++", "max"); //错误：类型不匹配
maxi(1, 2);           //错误：函数名和被调函数名不一致
maximum(1, 2, 3);     //错误：实参个数太多
maximum(2.3, 4 + 1);  //正确：第一个实参将被转换为int类型
```

问题：

函数的形参和返回值的作用是什么？

问题：

函数的形参和返回值的作用是什么？

函数的形参和返回值是主调函数和被调函数间信息传递的主要方式。

5.1 认识函数 — 无参列表和 void 返回类型

问题:

函数的形参和返回值的作用是什么?

函数的形参和返回值是主调函数和被调函数间信息传递的主要方式。

无参列表和 void 返回类型例子

```
void fun(){/**/}     // 隐式定义空形参列表，返回值类型为~void

void fun(void){/**/} // 显式定义空形参列表，返回值类型为~void
```

练习

找出下面函数的错误。

练习 1

```
int f(){
    string s;
    cin>>s;
    return s;
}
```

练习

找出下面函数的错误。

练习 1

```
int f(){
    string s;
    cin>>s;
    return s;
}
```

练习 2

```
//定义一个无返回值的函数
f2(int i){
    cout<<i<<endl;
}
```

练习

找出下面函数的错误。

练习 1

```
int f(){
    string s;
    cin>>s;
    return s;
}
```

练习 2

```
//定义一个无返回值的函数
f2(int i){
    cout<<i<<endl;
}
```

练习 3

```
int f(int v1,int v2)
    int x=v1+v2;
}
```

练习

找出下面函数的错误。

练习 1

```
int f(){
    string s;
    cin>>s;
    return s;
}
```

练习 2

```
//定义一个无返回值的函数
f2(int i){
    cout<<i<<endl;
}
```

练习 3

```
int f(int v1,int v2)
    int x=v1+v2;
}
```

练习 4

```
//定义函数square
double square(double x)
return x*x;
```

引入函数声明的原因

语法上对程序文件中函数的排列次序要求满足先定义后使用。

引入函数声明的原因

语法上对程序文件中函数的排列次序要求满足先定义后使用。

函数声明（函数原型）

- 返回值类型
- 名字
- 形参列表（可以省略形参名）

引入函数声明的原因

语法上对程序文件中函数的排列次序要求满足先定义后使用。

函数声明（函数原型）

- 返回值类型
- 名字
- 形参列表（可以省略形参名）

函数声明的示例

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

引入函数声明的原因

语法上对程序文件中函数的排列次序要求满足先定义后使用。

函数声明（函数原型）

- 返回值类型
- 名字
- 形参列表（可以省略形参名）

函数声明的示例

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

```
int maximum(int a, int b); //函数声明

int main() {
    int x, y, z;
    cin >> x >> y;
    z = maximum(x, y);
    cout << "z=" << z << endl;
}
int maximum(int a, int b) {
     return c=a > b ? a : b;
}
```

对象的生命期和作用域

一个对象的生命期取决于其存储周期类型，可访问性取决于作用域和链接性

对象的生命期和作用域

一个对象的生命期取决于其存储周期类型，可访问性取决于作用域和链接性

存储周期

存储周期（storage duration）表明了对象可以在内存里面存在的时间。

对象的生命期和作用域

一个对象的生命期取决于其存储周期类型，可访问性取决于作用域和链接性

存储周期

存储周期（storage duration）表明了对象可以在内存里面存在的时间。

C++ 支持四种类型的存储周期

- 自动存储周期（automatic storage duration）；
- 静态存储周期（static storage duration）；
- 动态存储周期（dynamic storage duration）；
- 线程存储周期（thread storage duration）。

自动存储周期

定义在函数体或语句块内部的对象（包括函数的形参）。在程序执行到其定义的位置时创建，离开其作用域时被释放，在栈（stack）区分配存储空间。

自动存储周期

定义在函数体或语句块内部的对象（包括函数的形参）。在程序执行到其定义的位置时创建，离开其作用域时被释放，在栈（stack）区分配存储空间。

静态存储周期

定义在函数外面或使用 static 关键字声明的对象具有静态存储周期，即在程序运行期间，始终存在，直到程序结束，在全局数据区分配存储空间。

自动存储周期

定义在函数体或语句块内部的对象（包括函数的形参）。在程序执行到其定义的位置时创建，离开其作用域时被释放，在栈（stack）区分配存储空间。

静态存储周期

定义在函数外面或使用 static 关键字声明的对象具有静态存储周期，即在程序运行期间，始终存在，直到程序结束，在全局数据区分配存储空间。

动态存储周期

利用运算符 new 生成的对象具有动态存储周期，可利用运算符 delete 释放其内存空间，存储周期从 new 操作开始，到 delete 操作结束，在堆（heap）区分配存储空间。

自动存储周期

定义在函数体或语句块内部的对象（包括函数的形参）。在程序执行到其定义的位置时创建，离开其作用域时被释放，在栈（stack）区分配存储空间。

静态存储周期

定义在函数外面或使用 static 关键字声明的对象具有静态存储周期，即在程序运行期间，始终存在，直到程序结束，在全局数据区分配存储空间。

动态存储周期

利用运算符 new 生成的对象具有动态存储周期，可利用运算符 delete 释放其内存空间，存储周期从 new 操作开始，到 delete 操作结束，在堆（heap）区分配存储空间。

线程存储周期

为了支持并行程序设计，C++11 引入了 thread_local 关键字。存储周期在其所在的线程创建时开始，线程结束时结束。

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

局部对象（local object）：在函数内部定义的对象，包括函数的形参

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

局部对象（local object）：在函数内部定义的对象，包括函数的形参

- 仅在相应的语句块内部可见；

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

局部对象（local object）：在函数内部定义的对象，包括函数的形参

- 仅在相应的语句块内部可见；
- 屏蔽外层作用域中的同名对象；

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

局部对象（local object）：在函数内部定义的对象，包括函数的形参

- 仅在相应的语句块内部可见；
- 屏蔽外层作用域中的同名对象；
- 生命期取决于存储类型；

按照对象定义的位置

- 局部对象：函数内部
- 全局对象：函数外部

局部对象（local object）：在函数内部定义的对象，包括函数的形参

- 仅在相应的语句块内部可见；
- 屏蔽外层作用域中的同名对象；
- 生命期取决于存储类型；
- 局部自动对象和局部静态对象。

局部自动对象

- 自动存储周期；
- 初始化方式：（1）初始值；（2）默认初始化（内置类型除外）。

局部自动对象

- 自动存储周期；
- 初始化方式：（1）初始值；（2）默认初始化（内置类型除外）。

使用自动对象的例子

```
void fun(float x) { //x和t均为局部自动对象
    int t = x + 5;
    cout << "t=" << t << endl;
}
int main() {
    float t = 3.5;
    cout << "t=" << t << endl;
    fun(t);
}
```

局部静态对象

函数体内部定义的局部对象需要保存上一次调用之后的计算结果

- 局部作用域；

局部静态对象

函数体内部定义的局部对象需要保存上一次调用之后的计算结果

- 局部作用域；
- 静态生命周期；

局部静态对象

函数体内部定义的局部对象需要保存上一次调用之后的计算结果

- 局部作用域；
- 静态生命周期；
- 如果内置类型的局部静态对象没有提供初始值，则初始化为 0

局部静态对象

函数体内部定义的局部对象需要保存上一次调用之后的计算结果

- 局部作用域；
- 静态生命周期；
- 如果内置类型的局部静态对象没有提供初始值，则初始化为 0

请问程序输出结果?

```
int fun() {
    int a = 0;          //a为局部自动对象
    static int b = 0;   //b为局部静态对象
    return ++b + ++a;
}
int main() {
    for (int i = 0; i < 3; ++i)
        cout << fun() << endl;
}
```

局部静态对象

函数体内部定义的局部对象需要保存上一次调用之后的计算结果

- 局部作用域；
- 静态生命周期；
- 如果内置类型的局部静态对象没有提供初始值，则初始化为 0

请问程序输出结果?

```
int fun() {
    int a = 0;          //a为局部自动对象
    static int b = 0;   //b为局部静态对象
    return ++b + ++a;
}
int main() {
    for (int i = 0; i < 3; ++i)
        cout << fun() << endl;
}
```

结果为：2 3 4

全局对象

在函数外面定义的对象称为全局对象。

- 静态存储周期；
- 全局作用域（文件域）；
- 如果内置类型的全局对象没有提供初始值，则初始化为 0。

全局对象

在函数外面定义的对象称为全局对象。

- 静态存储周期；
- 全局作用域（文件域）；
- 如果内置类型的全局对象没有提供初始值，则初始化为 0。

全局对象使用示例

```
int sum = 10;                          //定义全局对象
int main() {
    int sum = 1;                       //定义局部对象
    std::cout << sum << std::endl;     //访问局部对象sum，打印输出1
    std::cout << ::sum << std::endl;   //访问全局对象sum，打印输出10
    return 0;
}
```

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

- 局部对象没有链接性，不能共享；

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

- 局部对象没有链接性，不能共享；
- 全局对象具有外部链接性(external linkage），可在文件间共享；

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

- 局部对象没有链接性，不能共享；
- 全局对象具有外部链接性(external linkage），可在文件间共享；
- static 修饰的全局对象具有内部链接性（internal linkage），只能由同一个文件中的函数共享。

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

- 局部对象没有链接性，不能共享；
- 全局对象具有外部链接性(external linkage），可在文件间共享；
- static 修饰的全局对象具有内部链接性（internal linkage），只能由同一个文件中的函数共享。

fun.cpp

```
//g_val~具有外部链接性
int g_val = 10;
```

main.cpp

```
extern int g_val;
//gs_val内部链接性
static int gs_val = 20;
int main() {
    cout << g_val + gs_val;
    return 0;
}
```

5.2.3 认识函数 — 全局对象

链接性（linkage）：在作用域外部的可见性（共享性）

- 局部对象没有链接性，不能共享；
- 全局对象具有外部链接性(external linkage），可在文件间共享；
- static 修饰的全局对象具有内部链接性（internal linkage），只能由同一个文件中的函数共享。

fun.cpp

```
//g_val~具有外部链接性
int g_val = 10;
```

main.cpp

```
extern int g_val;
//gs_val内部链接性
static int gs_val = 20;
int main() {
    cout << g_val + gs_val;
    return 0;
}
```

提示

尽量不要使用全局对象。

5.3 参数传递

参数传递

主调函数和被调函数间信息传递的主要方式，通过实参和形参来实现信息传递

参数传递

主调函数和被调函数间信息传递的主要方式，通过实参和形参来实现信息传递

实参和形参的交互方式

- 单向的值传递（passed by value）方式；
- 双向的引用传递（passed by reference）方式。

5.3.1 参数传递 — 值传递

普通值传递

将实参的值拷贝给形参。

- 非引用类型；
- 改变形参的值，不会影响到实参。

5.3.1 参数传递 — 值传递

普通值传递

将实参的值拷贝给形参。

- 非引用类型；
- 改变形参的值，不会影响到实参。

普通值传递的例子

```
void Swap(int x, int y) {
    int z(x);
    x = y;
    y = z;
}
int main() {
   int i(4), j(5);
   Swap(i, j);      //调用Swap函数，实参i和j分别初始化Swap函数的形参x和y
   cout << "i=" << i << ",j=" << j << endl;//输出i=4,j=5
}
```

5.3.1 参数传递 — 值传递

例 5.1

找出 10 到 1000 之内的所有回文数。

5.3.1 参数传递 — 值传递

例 5.1

找出 10 到 1000 之内的所有回文数。

分析

回文数指左右对称的数，如 11、121、14341 等。判断一个数是否是回文数，可以把该数字的每一位数计算出来，然后再按照回文数定义判断。

5.3.1 参数传递 — 值传递

代码清单 5.1，例 5.1

```
...
bool is_palindrome(int x);          //函数声明

int main() {
    for (int i = 10; i <= 1000; ++i) {
        if (is_palindrome(i))
            cout << i << endl;
    }
}

bool is_palindrome(int x) {
    vector<int> digit;              //存放x的每一位数字
    while (x != 0) {
        digit.push_back(x % 10);    //获取当前x的个位数，并将其尾插到digit
        x /= 10;                    //去掉x的个位数
    }
    for (int i = 0, j = digit.size() - 1; i < j; ++i, --j) {
        if (digit[i] != digit[j]) return false;
    }
    return true;
}
```

5.3.1 参数传递 — 值传递

下面程序输出的结果是什么?

```
int f(int a, int b){
    int c;
    c = a + b;
    return c;
}
int main(){
    int x = 6, y = 7, z = 8, r;
    r = f((x--, y++, x + y), z--);
    cout << r << endl;
    return 0;
}
```

5.3.1 参数传递 — 值传递

下面程序输出的结果是什么?

```
int f(int a, int b){
    int c;
    c = a + b;
    return c;
}
int main(){
    int x = 6, y = 7, z = 8, r;
    r = f((x--, y++, x + y), z--);
    cout << r << endl;
    return 0;
}
```

结果

21

5.3.1 参数传递 — 值传递

地址传递

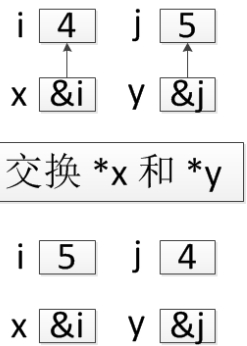
将实参的地址传递给形参，也就是说形参的类型为指针类型，通过形参改变实参的值。

地址传递

将实参的地址传递给形参，也就是说形参的类型为指针类型，通过形参改变实参的值。

示例 1

```
void Swap(int *x, int *y) {
    int z(*x);
    *x = *y;
    *y = z;
}
int main() {
    int i(4), j(5);
    Swap(&i, &j);
    cout << "i=" << i << ",j=" << j << endl;
    return 0;
}
```



地址传递

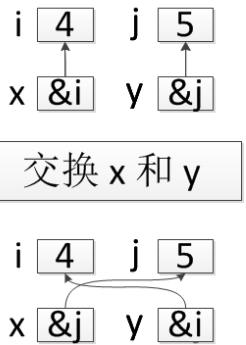
将实参的地址传递给形参，也就是说形参的类型为指针类型，通过形参间接访问实参。

地址传递

将实参的地址传递给形参，也就是说形参的类型为指针类型，通过形参间接访问实参。

示例 2

```
void Swap(int *x, int *y) {
    int *z(x);
    x = y;
    y = z;
}
int main() {
    int i(4), j(5);
    Swap(&i, &j);
    cout << "i=" << i << ",j=" << j << endl;
    return 0;
}
```



引用传递

形参是实参的引用

- 引用类型；
- 通过形参改变实参的值。

引用传递

形参是实参的引用

- 引用类型;
- 通过形参改变实参的值。

引用传递的例子

```
void Swap(int &x, int &y) {           //x和y分别是实参i和j的别名
    int z(x);
    x = y;
    y = z;
}                                     //交换x和y所绑定的对象的值
int main() {
    int i(4), j(5);
    Swap(i, j);
    cout << "i=" << i << ",j=" << j << endl;//输出i=5,j=4
    return 0;
}
```

5.3.3 参数传递 — const 形参

兼顾效率与安全

- 避免内容的拷贝：高效性；
- 不能通过形参改变实参：安全性。

5.3.3 参数传递 — const 形参

兼顾效率与安全

- 避免内容的拷贝：高效性；
- 不能通过形参改变实参：安全性。

const 形参

const 修饰的形参名字和 const 修饰的对象名字的含义相同。

5.3.3 参数传递 — const 形参

兼顾效率与安全

- 避免内容的拷贝：高效性；
- 不能通过形参改变实参：安全性。

const 形参

const 修饰的形参名字和 const 修饰的对象名字的含义相同。

请问以下 const 修饰的形参名字的含义

```
void f_cval(const int i);
void f_cptr(const int *i);
void f_cref(const int &i);
```

5.3.3 参数传递 — const 形参

兼顾效率与安全

- 避免内容的拷贝：高效性；
- 不能通过形参改变实参：安全性。

const 形参

const 修饰的形参名字和 const 修饰的对象名字的含义相同。

请问以下 const 修饰的形参名字的含义

```
void f_cval(const int i); //i 为 const 对象
void f_cptr(const int *i); //i 指向 const 类型的实参
void f_cref(const int &i); //i 为 const 类型实参的引用
```

5.3.3 参数传递 — const 形参

非 const 引用形参

- 非 const 引用形参不能接受字面值常量、表达式的求值结果、需要转换的对象或者 const 对象。

```
void f_ref(int &i); //左值引用形参

const int cx = 1;
int x = 1;
f_ref(41);          //错误：左值引用不能绑定字面值常量
f_ref(cx);          //错误：左值引用不能绑定常量
f_ref(x+1);         //错误：左值引用不能绑定右值表达式
```

5.3.3 参数传递 — const 形参

非 const 引用形参

- 非 const 引用形参不能接受字面值常量、表达式的求值结果、需要转换的对象或者 const 对象。

```
void f_ref(int &i); //左值引用形参

const int cx = 1;
int x = 1;
f_ref(41);          //错误：左值引用不能绑定字面值常量
f_ref(cx);          //错误：左值引用不能绑定常量
f_ref(x+1);         //错误：左值引用不能绑定右值表达式
```

建议

尽量使用引用形参，需要的时候使用指针形参

数组的特点

- 不可复制；
- 指针化。

数组的特点

- 不可复制；
- 指针化。

数组形参

通常使用指针的方式来传递首元素的地址。

5.3 参数传递 — 数组形参

数组的特点

- 不可复制;
- 指针化。

数组形参

通常使用指针的方式来传递首元素的地址。

示例

```
void fun(int *p);
void fun2(int p[]);//等价于上式，数组的方式
```

5.3 参数传递 — 数组形参

数组的特点

- 不可复制;
- 指针化。

数组形参

通常使用指针的方式来传递首元素的地址。

示例

```
void fun(int *p);
void fun2(int p[]);//等价于上式，数组的方式
```

调用方式

```
int arr[5] = {1, 2};
fun(arr);            //正确：数组名转化为首元素的地址
fun(&arr[0]);        //正确：显式传递首元素的地址
```

5.3 参数传递 — 数组形参

数组的特点

- 不可复制;
- 指针化。

数组形参

通常使用指针的方式来传递首元素的地址。

示例

```
void fun(int *p);
void fun2(int p[]);//等价于上式，数组的方式
```

调用方式

```
int arr[5] = {1, 2};
fun(arr);           //正确：数组名转化为首元素的地址
fun(&arr[0]);       //正确：显式传递首元素的地址
```

传递的是数组首元素的地址，编译器不会检查数组的大小

```
void fun(int p[5]); //“指明”数组的长度，无用
```

5.3.3 参数传递 — 数组形参

显式传递数组的长度

```
void print(char *str, unsigned size) {
    for (unsigned i = 0; i < size; ++i)
        cout << str[i];
}
```

5.3.3 参数传递 — 数组形参

显式传递数组的长度

```
void print(char *str, unsigned size) {
    for (unsigned i = 0; i < size; ++i)
        cout << str[i];
}
```

使用标记位来识别数组长度

```
void print(char *str) {
    if (str) //如果str不是一个空指针
        while (*str) //当前指针指向非空字符
            cout << *str++; //输出当前指针指向的字符
             并指向下一个字符
}
    char arr[] = "Hello C++";
    print(arr);
```

常处理 C 风格的字符数组，对其他数据类型的数组可能没有效果。

5.3.3 参数传递 — 数组形参

显式传递数组的长度

```
void print(char *str, unsigned size) {
    for (unsigned i = 0; i < size; ++i)
        cout << str[i];
}
```

使用标记位来识别数组长度

```
void print(char *str) {
    if (str) //如果str不是一个空指针
        while (*str) //当前指针指向非空字符
            cout << *str++; //输出当前指针指向的字符
             并指向下一个字符
}
    char arr[] = "Hello C++";
    print(arr);
```

常处理 C 风格的字符数组，对其他数据类型的数组可能没有效果。

利用指针标明访问范围

```
void print(char *beg, char * end) {
    //输出beg和end之间的元素（包含beg但不包含end指
     向的元素）
    while (beg != end)
        cout << *beg++; //输出当前指针指向的字符并
         指向下一个字符
}
    char arr[] = "Hello C++";
    print(begin(arr), end(arr));
```

begin 和 end 函数可以获取数组的首元素和尾后元素的地址。

5.3.3 参数传递 — 数组形参

使用 const 形参

```
void print(const char *str);
void print(const char *str, unsigned size);
void print(const char *beg, const char * end);
```

使用 const 形参

```
void print(const char *str);
void print(const char *str, unsigned size);
void print(const char *beg, const char * end);
```

使用 const 形参提高安全性

只有当对数组进行写操作时，数组形参才使用非 const 类型，否则一律要使用 const 修饰，保证程序的安全性。

5.3.3 参数传递 — 数组形参

使用 const 形参

```
void print(const char *str);
void print(const char *str, unsigned size);
void print(const char *beg, const char * end);
```

使用 const 形参提高安全性

只有当对数组进行写操作时，数组形参才使用非 const 类型，否则一律要使用 const 修饰，保证程序的安全性。

课下思考

如何传递一个数组而非首元素地址？并分析以这种参数传递方式编写的函数是否可以处理任意长度的数组。

5.3.3 参数传递 — 数组形参

多维数组特点

- 传递的是数组的首元素地址；
- 编译器只忽略第一维的长度。

5.3.3 参数传递 — 数组形参

多维数组特点

- 传递的是数组的首元素地址；
- 编译器只忽略第一维的长度。

二维数组形参示例

```
void fun(int (*a2d)[5]); //a2d 指向一个含有5个元素的一维实参数组
void fun(int a2d[][5]); //与上式等价
int matrix[4][5] = {};
fun(matrix);            //传递 matrix 首元素地址，即一个具有5个元素的一维数组
```

5.4 返回值类型

返回值类型

- 有值返回;
- 无值返回（返回一个 void 类型)。

5.4 返回值类型 — 无值返回

无值返回示例

```
void Swap(int &x, int &y) {
    if (x == y)
        return;
    int z(x);
    x = y;
    y = z;
}
```

问题

return 的作用是什么?

无值返回示例

```
void Swap(int &x, int &y) {
    if (x == y)
        return;//显式返回主调函数
    int z(x);
    x = y;
    y = z;
    //隐式返回主调函数，无需return语句
}
```

回答

使用 return 语句来控制程序执行的流向。

有值返回

- 值返回（return by value）；
- 引用返回（return by reference）；
- 指针返回（return by pointer）。

5.4.2 返回值类型 — 有值返回

值返回

通过拷贝返回值的方式将结果传递给主调函数。

- 简单、安全、效率低。

5.4.2 返回值类型 — 有值返回

值返回

通过拷贝返回值的方式将结果传递给主调函数。

- 简单、安全、效率低。

值返回示例

```
int maximum(int a, int b) {
   return a > b ? a : b;
}
```

5.4.2 返回值类型 — 有值返回

值返回

通过拷贝返回值的方式将结果传递给主调函数。

- 简单、安全、效率低。

值返回示例

```
int maximum(int a, int b) {
   return a > b ? a : b;
}
```

问题

为什么值返回效率低?

5.4.2 返回值类型 — 有值返回

值返回

通过拷贝返回值的方式将结果传递给主调函数。

- 简单、安全、效率低。

值返回示例

```
int maximum(int a, int b) {
   return a > b ? a : b;
}
```

问题

为什么值返回效率低?

```
int a = 10, b = 5;
int c = maximum(a, b);
```

maximum 函数返回值存放在一个临时对象里，用来初始化对象 c。

5.4.2 返回值类型 — 有值返回

引用返回

返回对象的一个别名，与返回对象指向同一个存储空间，不会产生临时对象。

5.4.2 返回值类型 — 有值返回

引用返回

返回对象的一个别名，与返回对象指向同一个存储空间，不会产生临时对象。

引用返回示例

```
const int & maximum(const int &a, const int &b) {
    return a > b ? a : b; //返回对象a或对象b的引用
}
```

调用 maximum 函数

```
int x,y;
cin >> x >> y;
int z = maxinum(x,y);
```

5.4.2 返回值类型 — 有值返回

引用返回

返回对象的一个别名，与返回对象指向同一个存储空间，不会产生临时对象。

引用返回示例

```
const int & maximum(const int &a, const int &b) {
    return a > b ? a : b; //返回对象a或对象b的引用
}
```

调用 maximum 函数

```
int x,y;
cin >> x >> y;
int z = maxinum(x,y);
```

定义 x 或 y 的引用

```
const int &ref = maximum(x,y);
```

5.4.2 返回值类型 — 有值返回

返回一个非 const 左值引用

```
int& setMaximum(int &a, int &b) {
   //返回对象a 或对象b 的引用
    return a > b ? a : b;
}
int main(){
   int x = 0, y = 1;
   //把整数10赋值给 x 和 y 中较大者
   setMaximum(x, y) = 10;
   return 0;
}
```

5.4.2 返回值类型 — 有值返回

返回一个非 const 左值引用

```
int& setMaximum(int &a, int &b) {
    //返回对象a 或对象b 的引用
    return a > b ? a : b;
}
int main(){
   int x = 0, y = 1;
   //把整数10赋值给 x 和 y 中较大者
   setMaximum(x, y) = 10;
   return 0;
}
```

返回实参对象的地址

```
int* setMaximum(int &a, int &b) {
//返回引用a或b所绑定的实参对象的地址
    return a > b ? &a : &b;
}
int main(){
   int x = 0, y = 1;
   //通过返回指针把10赋值给 x和 y中较大者
   *setMaximum(x, y) = 10;
}
```

5.4.2 返回值类型 — 有值返回

问题: 下面程序是否有问题，为什么？如果有问题，该如何修改?

```
int& maximum(int a, int b) {
    return a > b ? a : b;
}
```

5.4.2 返回值类型 — 有值返回

问题: 下面程序是否有问题，为什么？如果有问题，该如何修改?

```
int& maximum(int a, int b) {
    return a > b ? a : b;
}
```

回答：切忌返回局部对象的地址或引用

形参对象 a 和 b 都是局部对象， maximum 函数终止时将会从内存中消亡，因此返回一个已经不存在的对象的引用是无效的引用。

```
int& maximum(int a, int b) {
    static int c;
    c = a > b ? a : b;
    return c; //正确：返回静态局部对象c的引用
}
```

值返回方法比较

- 值返回方式可以返回局部对象的值，但需要借助于一个额外的临时对象完成值的返回；

值返回方法比较

- 值返回方式可以返回局部对象的值，但需要借助于一个额外的临时对象完成值的返回；
- 引用和指针返回不需要借助于临时对象，但不能返回局部对象的引用或地址；

值返回方法比较

- 值返回方式可以返回局部对象的值，但需要借助于一个额外的临时对象完成值的返回；
- 引用和指针返回不需要借助于临时对象，但不能返回局部对象的引用或地址；
- 指针返回还经常用于返回一个具有动态存储周期的对象的地址；

5.4.2 返回值类型 — 有值返回

值返回方法比较

- 值返回方式可以返回局部对象的值，但需要借助于一个额外的临时对象完成值的返回；
- 引用和指针返回不需要借助于临时对象，但不能返回局部对象的引用或地址；
- 指针返回还经常用于返回一个具有动态存储周期的对象的地址；
- 根据实际需要并结合安全性和效率来进行选择

重载

把同一作用域下具有相同名字但不同形参列表的一组函数称为重载（overloaded）函数，这些函数执行相似的操作。

重载

把同一作用域下具有相同名字但不同形参列表的一组函数称为重载（overloaded）函数，这些函数执行相似的操作。

示例

```
const int& getMax(const int &a, const int &b)
 {
   return a > b ? a : b;
}
const int& getMax(const int &a, const int &b,
 const int &c) {
   return a > b ? (a > c ? a : c) : (b > c ?
    b : c);
}
const string& getMax(const string &a, const
 string &b) {
   return a > b ? a : b;
}
```

重载

把同一作用域下具有相同名字但不同形参列表的一组函数称为重载（overloaded）函数，这些函数执行相似的操作。

示例

```
const int& getMax(const int &a, const int &b)
 {
   return a > b ? a : b;
}
const int& getMax(const int &a, const int &b,
 const int &c) {
   return a > b ? (a > c ? a : c) : (b > c ?
    b : c);
}
const string& getMax(const string &a, const
 string &b) {
   return a > b ? a : b;
}
```

最佳匹配的原则

- 如果有精确匹配的函数，则调用此函数；否则选择实参与形参类型最接近的转换函数；
- 如果有一个以上无法区分的匹配，则会出现二义性调用（ambiguous call）错误；
- 如果找不到任何一个与实参相匹配的函数，则会出现无匹配错误。

示例

```
const int& getMax(const int &a, const int &b) {
    return a > b ? a : b;
}
const int& getMax(const int &a, const int &b, const int &c) {
    return a > b ? (a > c ? a : c) : (b > c ? b : c);
}
const string& getMax(const string &a, const string &b) {
    return a > b ? a : b;
}
```

示例

```
const int& getMax(const int &a, const int &b) {
    return a > b ? a : b;
}
const int& getMax(const int &a, const int &b, const int &c) {
    return a > b ? (a > c ? a : c) : (b > c ? b : c);
}
const string& getMax(const string &a, const string &b) {
    return a > b ? a : b;
}
```

问题: 以下函数调用第几个重载函数?

```
getMax(7, 8);
getMax("C++", "Programming");
```

默认参数

函数的某些形参总是接受一个默认的实参值（default argument）。

默认参数

函数的某些形参总是接受一个默认的实参值（default argument）。

示例

```
void turnoff(int time = 21);
```

默认参数

函数的某些形参总是接受一个默认的实参值（default argument）。

示例

```
void turnoff(int time = 21);
```

使用

```
turnoff(); //省略实参，使用默认值
turnoff(22); //提供实参，接受实参值
```

默认参数

函数的某些形参总是接受一个默认的实参值（default argument）。

示例

```
void turnoff(int time = 21);
```

使用

```
turnoff(); //省略实参，使用默认值
turnoff(22); //提供实参，接受实参值
```

形参的默认值可以是任何可以转换成形参类型的表达式

```
void turnoff(int time = getTime());
```

具有多个默认值

一个函数可以有多个默认值，但所有具有默认值的参数必须靠右侧放置。

5.5.2 函数重载和特殊用途的函数 — 默认参数

具有多个默认值

一个函数可以有多个默认值，但所有具有默认值的参数必须靠右侧放置。

定义是否正确?

```
void print(int a, int b, int c = 3);
void print(int a, int b = 2, int c);
void print(int a = 1, int b = 2, int c = 3);
```

具有多个默认值

一个函数可以有多个默认值，但所有具有默认值的参数必须靠右侧放置。

定义是否正确?

```
void print(int a, int b, int c = 3);
void print(int a, int b = 2, int c);
void print(int a = 1, int b = 2, int c = 3);
```

答案

```
void print(int a, int b, int c = 3); //正确：部分参数具有默认值
void print(int a, int b = 2, int c); //错误：最右侧参数没有默认值
void print(int a = 1, int b = 2, int c = 3); //正确：所有参数具有默认值
```

函数优缺点

- 优点：函数的使用体现了模块化程序设计思想，降低了程序设计的复杂性；
- 不足：函数调用，比如执行流的转移操作、参数传递、返回值处理等，需要时间和空间的开销。

5.5 函数重载和特殊用途的函数 — 内联函数

函数优缺点

- 优点：函数的使用体现了模块化程序设计思想，降低了程序设计的复杂性；
- 不足：函数调用，比如执行流的转移操作、参数传递、返回值处理等，需要时间和空间的开销。

示例

C++ 提供了一种即保证函数化的形式又兼顾执行效率的方法：内联函数。

```
inline void Swap(int &x, int &y){ /*...*/ }
```

5.5 函数重载和特殊用途的函数 — 内联函数

函数优缺点

- 优点：函数的使用体现了模块化程序设计思想，降低了程序设计的复杂性；
- 不足：函数调用，比如执行流的转移操作、参数传递、返回值处理等，需要时间和空间的开销。

示例

C++ 提供了一种即保证函数化的形式又兼顾执行效率的方法：内联函数。

```
inline void Swap(int &x, int &y){ /*...*/ }
```

减少开销

编译器将在调用处嵌入内联函数的代码，不会发生函数调用，因而也不会产生函数调用的开销了。

函数优缺点

- 优点：函数的使用体现了模块化程序设计思想，降低了程序设计的复杂性；
- 不足：函数调用，比如执行流的转移操作、参数传递、返回值处理等，需要时间和空间的开销。

示例

C++ 提供了一种即保证函数化的形式又兼顾执行效率的方法：内联函数。

```
inline void Swap(int &x, int &y){ /*...*/ }
```

减少开销

编译器将在调用处嵌入内联函数的代码，不会发生函数调用，因而也不会产生函数调用的开销了。

注意

对于编译器来说，inline 关键字只是一个建议。通常为使用频率高的简单函数。另外，内联函数放在头文件中。

例 5.2

利用冒泡排序法将容器中的学生成绩排序。

例 5.2

利用冒泡排序法将容器中的学生成绩排序。

利用冒泡排序法将容器中的学生成绩排序

```
inline void Swap(int &x, int &y) {
    int z(x);
    x = y;
    y = z;
}

int main() {
    srand(0);
    vector<int> score(10);
    for (auto &i : score) i = rand() % 100;

    for (int i = score.size()-1; i >=0; --i) {
        for (int j = 0; j < i;++j) {          //每一轮冒泡过程将最大的数浮出来（右边）
            if (score[j+1] < score[j])        //相邻的两个数比较
                Swap(score[j+1], score[j]);
        }
    }

    for (auto &i : score)    cout << i << endl;

}
```

问题

以下程序是否正确，为什么，该如何修改?

```
const int getNumber() { return 10; }
const int numStudent = getNumber();

int arr[numStudent];
```

此时编译器会报错，提示无法在编译时计算 numStudent 的值。这是因为只能在运行期间调用函数 getNumber 后才能计算 numStudent 的值。

5.5 函数重载和特殊用途的函数 — constexpr 函数 *

constexpr 函数

constexpr 函数指的是能用于常量表达式的函数。要求：

- 函数体中有且仅有一条 return 语句；
- 函数返回值类型不能为 void；
- return 语句中的表达式必须是编译时的常量表达式。

示例

```
constexpr int f() { return 10; }
constexpr int getNumber(int i) { return i; }
int stu1[getNumber(10)]; //正确：getNumber(10)是常量表达式
int num = 10;
int stu2[getNumber(num)]; //错误：运行时才能确定num的值
```

提示

constexpr 函数会隐式转化为内联函数。

函数指针

- 函数名对应于函数的执行代码的入口地址；
- 指针可以指向一个函数；
- 函数的类型由返回值类型和形参列表决定。

函数指针

- 函数名对应于函数的执行代码的入口地址；
- 指针可以指向一个函数；
- 函数的类型由返回值类型和形参列表决定。

示例

```
//声明一个~compareInt~函数：比较两个整数大小
bool compareInt(int, int);
bool(*pf)(int, int);//定义一个指向此类型函数的指针
```

函数指针

- 函数名对应于函数的执行代码的入口地址；
- 指针可以指向一个函数；
- 函数的类型由返回值类型和形参列表决定。

示例

```
//声明一个~compareInt~函数：比较两个整数大小
bool compareInt(int, int);
bool(*pf)(int, int);//定义一个指向此类型函数的指针
```

问题：请说明下面指针的含义？

```
bool* pf(int, int);
```

函数指针

- 函数名对应于函数的执行代码的入口地址；
- 指针可以指向一个函数；
- 函数的类型由返回值类型和形参列表决定。

示例

```
//声明一个~compareInt~函数：比较两个整数大小
bool compareInt(int, int);
bool(*pf)(int, int);//定义一个指向此类型函数的指针
```

问题：请说明下面指针的含义？

```
bool* pf(int, int);
```

函数名为 pf，返回值类型为 bool* 的函数声明。

简化函数指针定义

利用类型别名关键字 typedef 或 using 来简化函数指针的定义。

简化函数指针定义

利用类型别名关键字 typedef 或 using 来简化函数指针的定义。

示例

```
typedef bool(*pFun)(int, int);
using pFun = bool(*)(int, int);//与上式等价

pFun pf; //和定义普通指针一样的方式来定义一个函数指针
```

函数指针初始化

和普通指针类型一样，应该在定义函数指针时初始化

函数指针初始化

和普通指针类型一样，应该在定义函数指针时初始化

示例

```
pFun pf1 = compareInt;  //隐式初始化，pf1指向compareInt函数
pFun pf2 = &compareInt; //显式初始化，pf2指向compareInt函数
pFun pf3 = nullptr;     //pf3不指向任何函数
```

函数指针初始化

和普通指针类型一样，应该在定义函数指针时初始化

示例

```
pFun pf1 = compareInt;  //隐式初始化，pf1指向compareInt函数
pFun pf2 = &compareInt; //显式初始化，pf2指向compareInt函数
pFun pf3 = nullptr;     //pf3不指向任何函数
```

示例

当一个指针指向了一个具体的函数后，可以通过该指针来调用其指向的函数

```
bool b1 = pf1(1, 2);
bool b2 = (*pf2)(1, 2);
```

例 5.3

利用梯形法设计一个求数值积分的通用函数。

例 5.3

利用梯形法设计一个求数值积分的通用函数。

提示

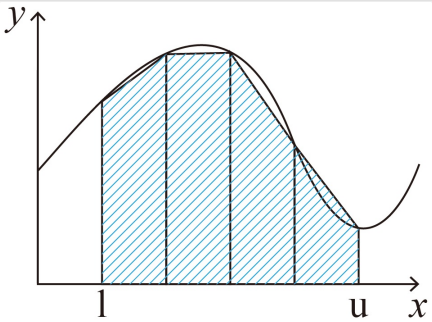
想要实现一个求积分的通用函数，可以将待求积分函数以参数的形式传递到这个通用函数里面，然后利用梯形法求解被积函数的积分。

例 5.3

利用梯形法设计一个求数值积分的通用函数。

提示

想要实现一个求积分的通用函数，可以将待求积分函数以参数的形式传递到这个通用函数里面，然后利用梯形法求解被积函数的积分。



代码清单 5.3，例 5.3

```
using pFun = double(*)(double);

double f_sphere(double x) { return x*x;  }
double f_default(double x) {   return 0; }
double f_sin(double x) { return sin(x);  }

double integrate(double l, double u, pFun pf = f_default, int n = 1000) {
    double sum = 0.0;
    double gap = (u - l) / n;              //每个间隔的长度
    for (int i = 0; i < n; i++)
        sum += (gap / 2.0) * (pf(l + i*gap) + pf(l + (i + 1)*gap));
    return sum;
}

int main() {
    cout<<"默认函数在区间[0:1]上的积分为: "<<integrate(0, 1) << endl;
    cout<<"Sphere函数在区间[0:1]上的积分为: "<<integrate(0, 1, f_sphere)<<endl;
    cout<<"sin函数在区间[0:1]上的积分为: "<<integrate(0, 1, f_sin)<<endl;
}
```

识别指针的原则

- 取出标识符;
- 由里向外;
- 由右向左。

识别指针的原则

- 取出标识符;
- 由里向外;
- 由右向左。

简单例子

```
int(*p1)[5];
int *p2[5];
int *&p3 = p2[0];
```

识别指针的原则

- 取出标识符;
- 由里向外;
- 由右向左。

简单例子

```
int(*p1)[5];
int *p2[5];
int *&p3 = p2[0];
```

识别指针

```
void(*a[5])(int);
void(*(*b)[5])(int);
void(*c(int, void(*fp)(int)))(int);
```

识别指针的原则

- 取出标识符;
- 由里向外;
- 由右向左。

简单例子

```
int(*p1)[5];
int *p2[5];
int *&p3 = p2[0];
```

识别指针

```
void(*a[5])(int);
void(*(*b)[5])(int);
void(*c(int, void(*fp)(int)))(int);
```

利用 using 声明简化语句

```
using PF = void(*)(int);
PF a[5];
PF(*b)[5];
PF c(int, PF);
```

lambda 表达式

lambda 表达式

- 只使用一次;

lambda 表达式

- 只使用一次；
- 临时的匿名函数，表示一个可以调用的代码单元；

lambda 表达式

- 只使用一次；
- 临时的匿名函数，表示一个可以调用的代码单元；
- 与函数类似，lambda 表达式具有返回值、形参列表和函数体；

lambda 表达式

- 只使用一次；
- 临时的匿名函数，表示一个可以调用的代码单元；
- 与函数类似，lambda 表达式具有返回值、形参列表和函数体；
- 可定义在一个函数内部。

lambda 表达式

- 只使用一次；
- 临时的匿名函数，表示一个可以调用的代码单元；
- 与函数类似，lambda 表达式具有返回值、形参列表和函数体；
- 可定义在一个函数内部。

1. 定义 lambda 表达式

[captures](parameters) -> return type {statements}

- captures：指定在同一作用域下 lambda 主体捕获（访问）哪些对象以及如何捕获这些对象；
- parameters：形参列表；
- return type：返回值类型；
- statements：函数体。

auto 关键字

```
auto fun = [](int i) {cout << i << endl; };
fun(17);              //输出17
```

- auto 关键字把一个 lambda 表达式存放到一个对象
- lambda 可以根据函数主体来推断返回类型。

auto 关键字

```
auto fun = [](int i) {cout << i << endl; };
fun(17);              //输出17
```

- auto 关键字把一个 lambda 表达式存放到一个对象
- lambda 可以根据函数主体来推断返回类型。

显式指明 return type

```
[](int i)->int { return i*i; }
```

5.6.2 函数指针和 lambda 表达式 — 捕获对象

捕获对象

lambda 可以捕获外围作用域内的局部对象，而且还可以指定捕获的方式

捕获对象

lambda 可以捕获外围作用域内的局部对象，而且还可以指定捕获的方式

按值捕获特定的外围对象

```
int divisor = 5;
vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20,
 25, 35, 45, 50 };
for_each(numbers.begin(), numbers.end(), [divisor
 ](int y) { //divisor为外围divisor的拷贝
    if (y % divisor == 0)
        cout << y << endl;//输出被divisor整除的元素
});
```

divisor 作为副本在函数体内部使用

捕获对象

lambda 可以捕获外围作用域内的局部对象，而且还可以指定捕获的方式

按值捕获特定的外围对象

```
int divisor = 5;
vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20,
 25, 35, 45, 50 };
for_each(numbers.begin(), numbers.end(), [divisor
 ](int y) { //divisor为外围divisor的拷贝
    if (y % divisor == 0)
        cout << y << endl;//输出被divisor整除的元素
});
```

divisor 作为副本在函数体内部使用

按引用捕获特定的外围对象

```
int sum = 0;

for_each(numbers.begin(), numbers.end(), [divisor
 , &sum](int y) { //sum为外围sum 的引用
    if (y % divisor == 0)
        sum += y;
    //累加被divisor整除的元素，结果存放到外围对象
     sum中
});
```

sum 作为引用在函数体内部使用

捕获对象

lambda 可以捕获外围作用域内的局部对象，而且还可以指定捕获的方式

按值捕获特定的外围对象

```
int divisor = 5;
vector<int> numbers{ 1, 2, 3, 4, 5, 10, 15, 20,
 25, 35, 45, 50 };
for_each(numbers.begin(), numbers.end(), [divisor
 ](int y) { //divisor为外围divisor的拷贝
    if (y % divisor == 0)
        cout << y << endl;//输出被divisor整除的元素
});
```

divisor 作为副本在函数体内部使用

按引用捕获特定的外围对象

```
int sum = 0;

for_each(numbers.begin(), numbers.end(), [divisor
 , &sum](int y) { //sum为外围sum 的引用
    if (y % divisor == 0)
        sum += y;
    //累加被divisor整除的元素，结果存放到外围对象
     sum中
});
```

sum 作为引用在函数体内部使用

按引用捕获特定的外围对象

[=]和 [&] 分别以值捕获和引用捕获方式捕获外围所有对象

5.7 递归调用

递归调用：一种程序设计方法

一个函数直接或间接地调用自己，这个函数称为递归函数（recursive function）

- 易实现；
- 可读性强。

例 5.4

求解以下数列的第 n 项。

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

例 5.4

```
#include<iostream>
int sumTo(int i) {
    if (i == 0) return 0;
    return sumTo(i - 1) + i;
}
int main() {
    std::cout << sumTo(3);
    return 0;
}
```

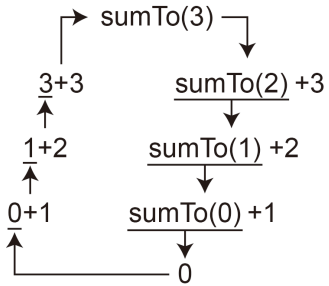


图: 函数调用 sumTo(3) 的执行过程

例 5.4

```
#include<iostream>
int sumTo(int i) {
    if (i == 0) return 0;
    return sumTo(i - 1) + i;
}
int main() {
    std::cout << sumTo(3);
    return 0;
}
```

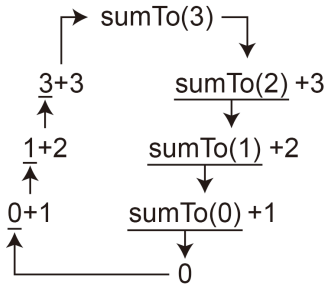


图: 函数调用 sumTo(3) 的执行过程

递归调用三要素

例 5.4

```
#include<iostream>
int sumTo(int i) {
    if (i == 0) return 0;
    return sumTo(i - 1) + i;
}
int main() {
    std::cout << sumTo(3);
    return 0;
}
```

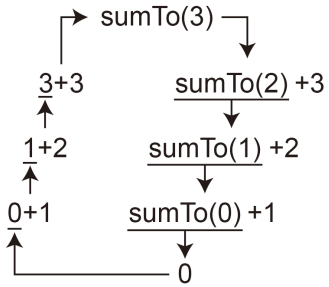


图: 函数调用 sumTo(3) 的执行过程

递归调用三要素

- 递推;

5.7 递归调用 — 递推和回归

例 5.4

```
#include<iostream>
int sumTo(int i) {
    if (i == 0) return 0;
    return sumTo(i - 1) + i;
}
int main() {
    std::cout << sumTo(3);
    return 0;
}
```

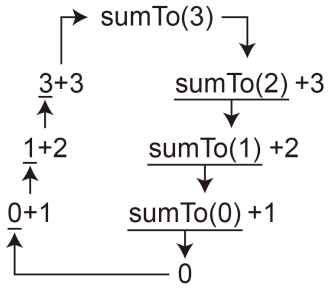


图: 函数调用 sumTo(3) 的执行过程

递归调用三要素

- 递推;
- 终止条件;

例 5.4

```
#include<iostream>
int sumTo(int i) {
    if (i == 0) return 0;
    return sumTo(i - 1) + i;
}
int main() {
    std::cout << sumTo(3);
    return 0;
}
```

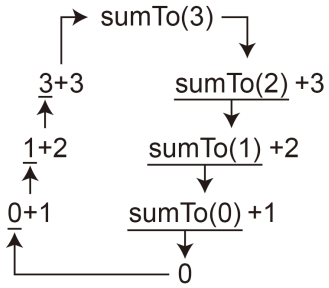


图: 函数调用 sumTo(3) 的执行过程

递归调用三要素

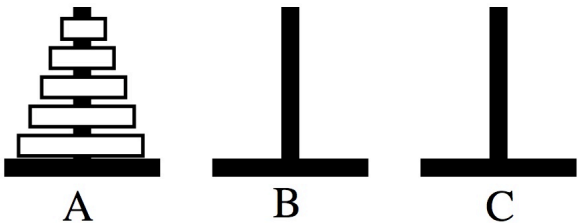
- 递推;
- 终止条件;
- 回归。

例 5.5

汉诺塔问题（Tower of Hanoi）。如图所示，有 A、B、C 三个柱子，A 柱上有 n 个大小不同的盘子，大盘在下、小盘在上依次存放。要求将 A 柱上的盘子全部移动到 C 柱上，每一次只能移动一个盘子，可以借助任何一个柱子，但必须要保证在任意时刻大盘在下、小盘在上。

例 5.5

汉诺塔问题（Tower of Hanoi）。如图所示，有 A、B、C 三个柱子，A 柱上有 n 个大小不同的盘子，大盘在下、小盘在上依次存放。要求将 A 柱上的盘子全部移动到 C 柱上，每一次只能移动一个盘子，可以借助任何一个柱子，但必须要保证在任意时刻大盘在下、小盘在上。



分析

要想将 n 个盘子从 A 柱移到 C 柱，有三个步骤必须要完成：

- 先把 A 柱上 n-1 个盘子移到 B 柱上；
- 将第 n 个盘子移到 C 柱上；
- 将 n-1 个盘子从 B 柱移动 C 柱上。

分析

要想将 n 个盘子从 A 柱移到 C 柱，有三个步骤必须要完成：

- 先把 A 柱上 n-1 个盘子移到 B 柱上；
- 将第 n 个盘子移到 C 柱上；
- 将 n-1 个盘子从 B 柱移动 C 柱上。

目标

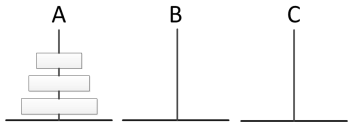
设计递归函数，其功能是将 n 个盘子从初始柱 src 上借助中间柱 mid 移动到目标柱 tar 上。

汉诺塔问题

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

void hanoi(int n, char src, char mid, char tar) {
    if (n == 1)
        cout << src << "->" << tar << '\t';
    else {
        hanoi(n - 1, src, tar, mid);  //将n-1个盘子移到中间柱
        cout<<src<<"->"<< tar << '\t'; //将最后一个盘子移到目标柱
        hanoi(n - 1, mid, src, tar);  //将n-1个盘子从中间柱移到目标柱
    }
}

int main() {
    int n;
    cin >> n;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```



汉诺塔问题

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

void hanoi(int n, char src, char mid, char tar) {
    if (n == 1)
        cout << src << "->" << tar << '\t';
    else {
        hanoi(n - 1, src, tar, mid);  //将n-1个盘子移到中间柱
        cout<<src<<"->"<< tar << '\t'; //将最后一个盘子移到目标柱
        hanoi(n - 1, mid, src, tar);  //将n-1个盘子从中间柱移到目标柱
    }
}

int main() {
    int n;
    cin >> n;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

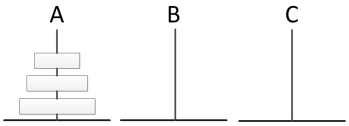


图: 移动过程

递归和循环

- 递归程序的递推和回归过程可以用循环结构来实现；
- 和循环结构相比，递归函数实现简单并且思路清晰，但缺点是需要消耗内存和其它的函数调用开销

例 5.6

利用递归程序设计方法求解八皇后问题，要求找出所有可能方案。

例 5.6

利用递归程序设计方法求解八皇后问题，要求找出所有可能方案。

分析

对于八皇后问题，每一次递推在新一行安排一个皇后。有两种情形需要回归：

- 当前行没有可行位置，则程序需要回归，即进行回溯操作；
- 当所有的皇后都已经成功摆放，需要回归寻找下一个可行的方案。

代码清单 5.6，例 5.6

```
bool isSafe(int i, const vector<int> &que) {
    for (int k = 0; k < i; ++k)
        if (que[k] == que[i] || (abs(que[i] - que[k]) == abs(i - k))) return false;
    return true;
}
void queen(int i, vector<int> &que, int &cnt) {
    if (i == 8) {
        cout << "方案" << ++cnt << ":";
        for (int k = 0; k < que.size(); ++k) cout << que[k];
        return;
    }
    for (int k = 0; k < que.size(); ++k) {
        que[i] = k;
        if (isSafe(i, que)) queen(i + 1, que, cnt); //如果安全，安排下一行皇后
    }
}
int main() {
    int sz(8), cnt(0);
    vector<int> que(sz);
    queen(0, que, cnt);
}
```

5.8 编译预处理和多文件结构

预处理指令

- 由 # 符号开头；
- 每条指令占一行；
- 通常放在文件的开始部分。

例如

- 文件包含指令 #include ；
- 宏定义；
- 条件编译指令。

宏定义

- 宏定义指令为 #define；
- 功能是定义一个标识符来代替一串字符，该标识符称为宏名；
- 分为带参和不带参两种。

宏定义

- 宏定义指令为 #define；
- 功能是定义一个标识符来代替一串字符，该标识符称为宏名；
- 分为带参和不带参两种。

1. 不带参宏定义

#define 宏名　字符串常量

```
#define PI 3.14159
void testPI() {
    cout << 2 * PI << endl;
}
//用来编译希望执行的代码
#define DEBUG
```

宏定义

- 宏定义指令为 #define；
- 功能是定义一个标识符来代替一串字符，该标识符称为宏名；
- 分为带参和不带参两种。

1. 不带参宏定义

#define 宏名　字符串常量

```
#define PI 3.14159
void testPI() {
    cout << 2 * PI << endl;
}
//用来编译希望执行的代码
#define DEBUG
```

2. 带参宏定义

#define　宏名（参数表）　字符串常量

```
#define fun2(a,b) (a)*(b)
#define fun3(a,b) a*b
void testFun() {
    cout << fun2(1 + 2, 3 - 2) << endl;
    cout << fun3(1 + 2, 3 - 2) << endl;
}//参数必须用小括号括起来
```

宏定义

- 宏定义指令为 #define；
- 功能是定义一个标识符来代替一串字符，该标识符称为宏名；
- 分为带参和不带参两种。

1. 不带参宏定义

#define 宏名　字符串常量

```
#define PI 3.14159
void testPI() {
    cout << 2 * PI << endl;
}
//用来编译希望执行的代码
#define DEBUG
```

2. 带参宏定义

#define　宏名（参数表）　字符串常量

```
#define fun2(a,b) (a)*(b)
#define fun3(a,b) a*b
void testFun() {
    cout << fun2(1 + 2, 3 - 2) << endl;
    cout << fun3(1 + 2, 3 - 2) << endl;
}//参数必须用小括号括起来
```

建议：尽量使用 const 对象和内联函数

特殊操作符

可以使用一些特殊的操作符来实现特殊的功能

特殊操作符

可以使用一些特殊的操作符来实现特殊的功能

字符串化操作符

把语言符号转化成字符串，即将后面的宏参数进行字符串化操作。

```
#define str(a) #a
cout << str(test) << endl;
```

等价于

```
cout << "test" << endl;
```

特殊操作符

可以使用一些特殊的操作符来实现特殊的功能

字符串化操作符

把语言符号转化成字符串，即将后面的宏参数进行字符串化操作。

```
#define str(a) #a
cout << str(test) << endl;
```

等价于

```
cout << "test" << endl;
```

连接符

将两个语言符号连接为一个语言符号

```
#define glue(a,b) a##b
glue(c, out) << "test" << endl;
int glue(x, 1) = 1;
```

等价于

```
cout << "test" << endl;
int x1 = 1;
```

条件编译

- 程序中的代码在满足一定条件下才会被编译；
- 编译的条件有两类：宏名和常量表达式；
- 条件编译指令包括：#ifdef、#ifndef、#undef、#if、#else、#endif 等。

条件编译

- 程序中的代码在满足一定条件下才会被编译；
- 编译的条件有两类：宏名和常量表达式；
- 条件编译指令包括：#ifdef、#ifndef、#undef、#if、#else、#endif 等。

例子 1

```
#ifdef DEBUG
    cout << "x=" << x << endl;
#endif
```

如果定义了 DEBUG 宏，则上述 cout 语句将被编译，否则将被忽略。

条件编译

- 程序中的代码在满足一定条件下才会被编译；
- 编译的条件有两类：宏名和常量表达式；
- 条件编译指令包括：#ifdef、#ifndef、#undef、#if、#else、#endif 等。

例子 1

```
#ifdef DEBUG
    cout << "x=" << x << endl;
#endif
```

如果定义了 DEBUG 宏，则上述 cout 语句将被编译，否则将被忽略。

例子 2

```
#if 0
    cout << "此行代码永远也不会被编译" << endl;
#endif
```

常量表达式的值为 false ，上面的 cout 语句永远也不会被编译。

多文件结构

编写一个较大的程序时，通常根据代码之间的耦合关系将它们放到不同的文件里面，进行单独的编写与编译，最终完成一个完整的程序。

- 标识符的可见性：把标识符的声明放到头文件里面，实现名字共享；
- 头文件保护： #ifndef、#define 、 #endif 预编译指令或者 #pragma once 宏；
- 头文件包含
  - 函数的声明
  - const 对象
  - 全局对象的声明
  - 内联函数
  - constexpr 函数
  - 类和模版的定义

myHeader.h

```
#ifndef MYHEADER_H
#define MYHEADER_H
const double pi = 3.1415926; //const对象
int add(int, int);//函数声明
extern int g_sum; //全局对象声明
inline bool isNumber(char ch) {//内联函数
    return ch >= '0'&&ch <= '9' ? 1 : 0;
}
constexpr int scale() { //constexpr 函数
    return 10;
}
class myClass { }; // 类定义
template<typename T> // 函数模板
const T& getMax(const T &a, const T &b) {
    return a > b ? a : b;
}
#endif // !MYHEADER_H
```

add.cpp

```
int g_sum = 10;
int add(int a, int b) {
    return a + b;
}
```

multi_file.cpp

```
#include "myHeader.h"
#include <iostream>
using namespace std;
int main() {
    g_sum = add(4, 5);
    cout << g_sum << endl;
    return 0;
}
```

课后作业

作业本

1. 习题 5.6、5.9 和 5.17

上机练习

1. 实验指导书：第五章

本章结束