C++语言程序设计：MOOC版

清华大学出版社（ISBN 978-7-302-42104-7）

第5章 结构化程序设计之一

中国農業大學 阚道宏

第5章 结构化程序设计之一

- 大量数据：数组
- 复杂算法
  - 结构化程序设计方法：模块的分解和组装
  - C++语言：函数的定义和调用

- 数据管理
  - 分散管理
  - 集中管理

第5章 结构化程序设计之一

- 本章内容
  - 5.1 结构化程序设计方法
  - 5.2 函数的定义和调用
  - 5.3 数据的管理策略
  - 5.4 程序代码和变量的存储原理
  - 5.5 函数间参数传递的3种方式
  - 5.6 在函数间传递数组

5.1 结构化程序设计方法

- **结构化程序设计方法**就是：

  将一个求解复杂问题的过程划分为若干个**子过程**，每个子过程完成一个独立的、相对简单的功能；用算法描述各个过程的操作步骤，每个算法称为一个**模块**；采用“**自顶向下，逐步细化**”的方法逐步分解和设计算法模块，再通过**调用关系**将各个模块组织起来，最终形成一个完整的数据处理算法

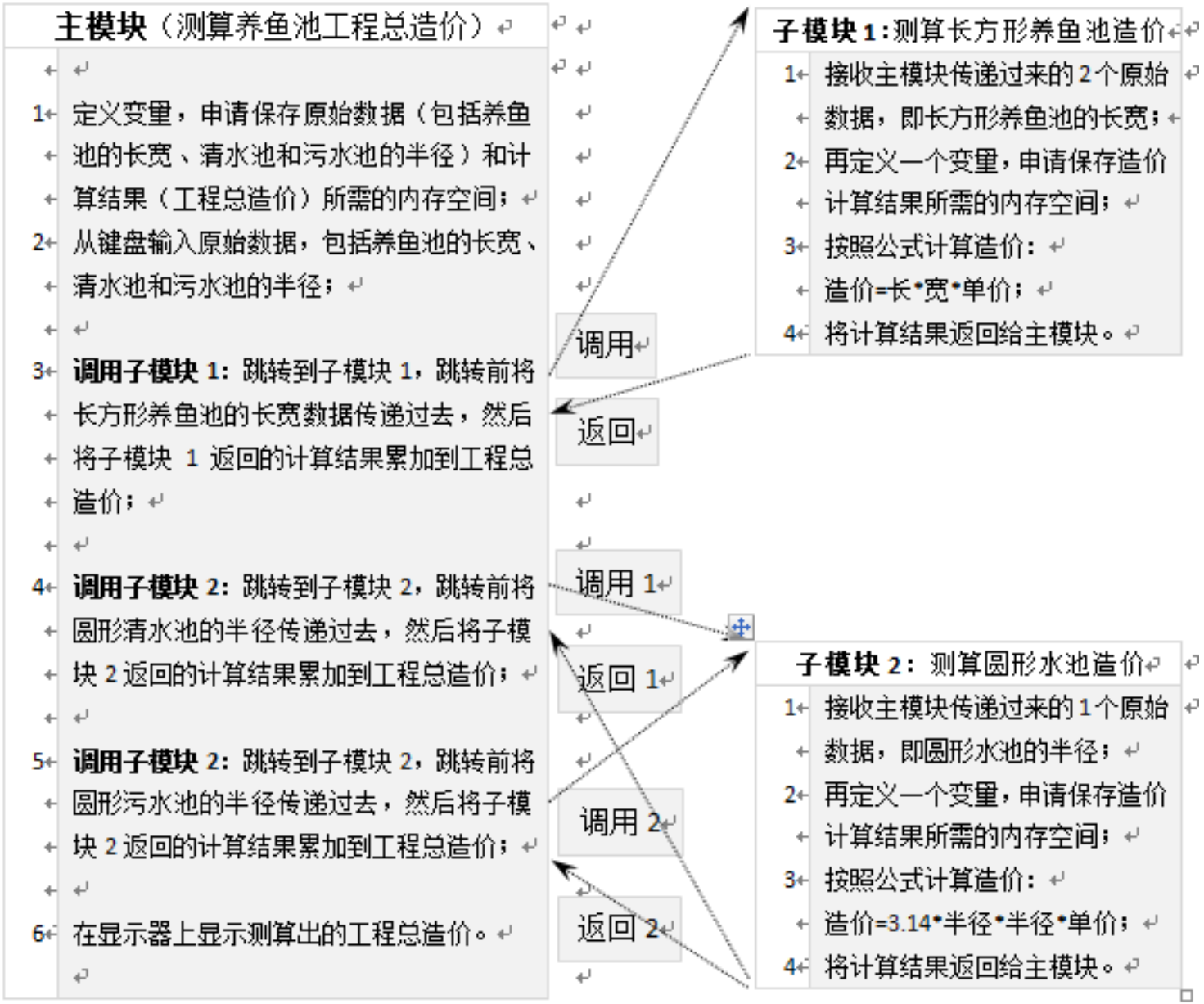
5.1 结构化程序设计方法

- **设计任务**：公园计划修建1个长方形观赏鱼池，另外配套修建一大一小2个圆形蓄水池，分别存放清水和污水，如图5-1所示。养鱼池和蓄水池的造价均为10元/m^2。请设计一个测算养鱼池工程总造价的算法。

5.1 结构化程序设计方法

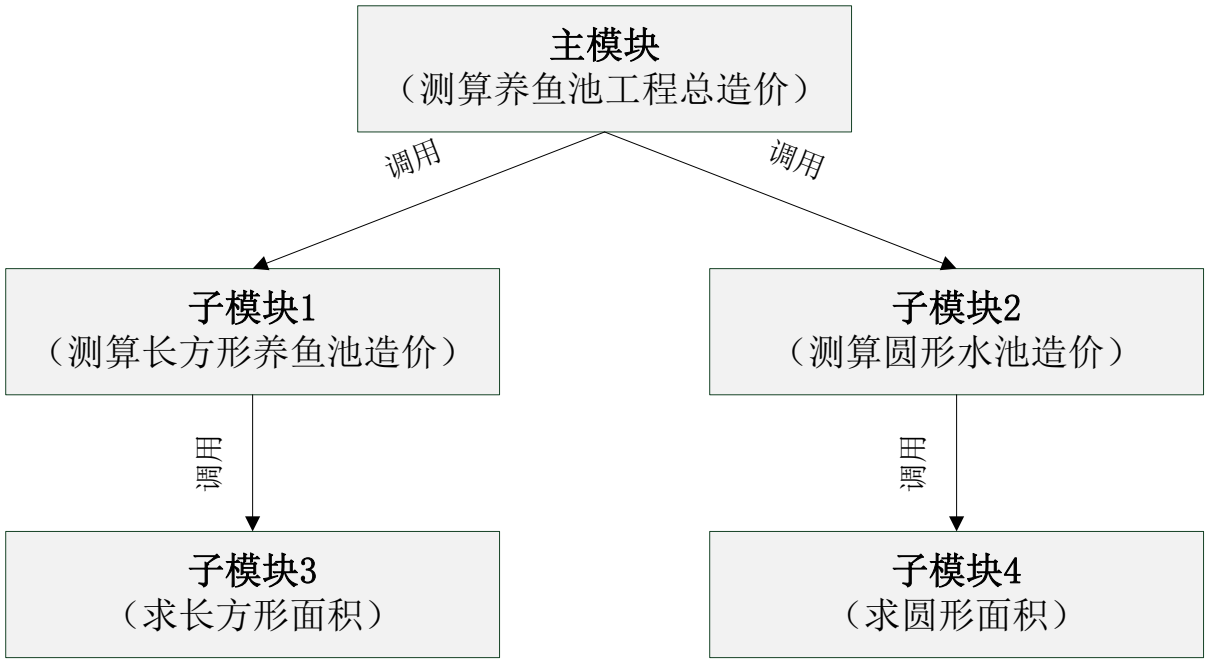
例5-1 测算养鱼池工程总造价的算法（概要设计）

1 定义变量，申请保存原始数据（包括养鱼池的长宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间；
2 从键盘输入原始数据，包括养鱼池的长宽、清水池和污水池的半径；
3 计算长方形养鱼池的造价，累加到工程总造价；
4 计算圆形清水池的造价，累加到工程总造价；
5 计算圆形污水池的造价，累加到工程总造价；
6 在显示器上显示测算出的工程总造价。


主模块（测算养鱼池工程总造价）
1 定义变量，申请保存原始数据（包括养鱼池的长宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间；
2 从键盘输入原始数据，包括养鱼池的长宽、清水池和污水池的半径；
3 调用子模块 1：跳转到子模块 1，跳转前将长方形养鱼池的长宽数据传递过去，然后将子模块 1 返回的计算结果累加到工程总造价；
4 调用子模块 2：跳转到子模块 2，跳转前将圆形清水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价；
5 调用子模块 2：跳转到子模块 2，跳转前将圆形污水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价；
6 在显示器上显示测算出的工程总造价。
调用
返回
调用 1
返回 1
调用 2
返回 2
子模块 1：测算长方形养鱼池造价
1 接收主模块传递过来的 2 个原始数据，即长方形养鱼池的长宽；
2 再定义一个变量，申请保存造价计算结果所需的内存空间；
3 按照公式计算造价：
造价=长*宽*单价；
4 将计算结果返回给主模块。
子模块 2：测算圆形水池造价
1 接收主模块传递过来的 1 个原始数据，即圆形水池的半径；
2 再定义一个变量，申请保存造价计算结果所需的内存空间；
3 按照公式计算造价：
造价=3.14*半径*半径*单价；
4 将计算结果返回给主模块。

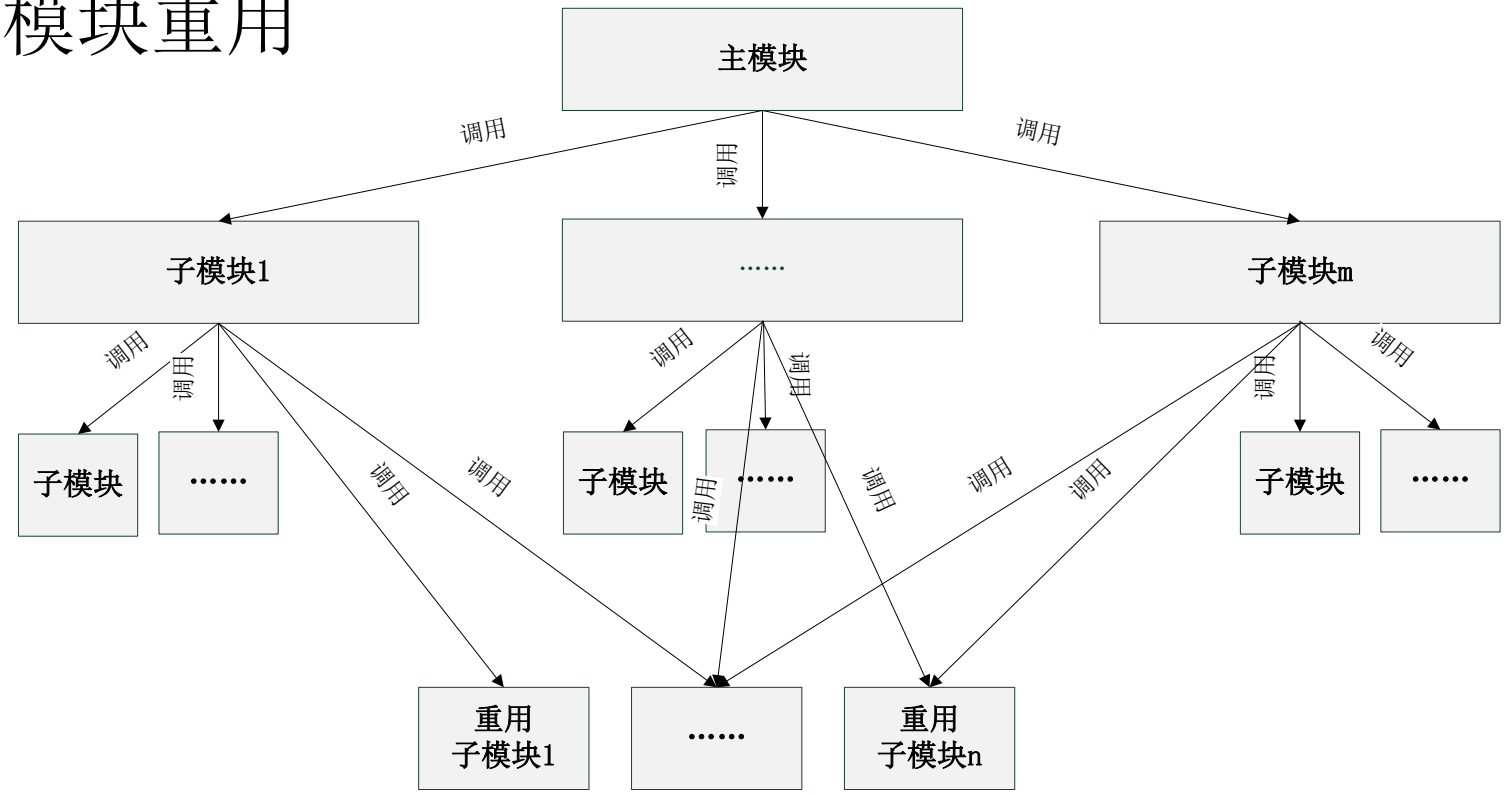
5.1 结构化程序设计方法

- 基于模块的团队分工协作开发
  - 自顶向下，逐步细化



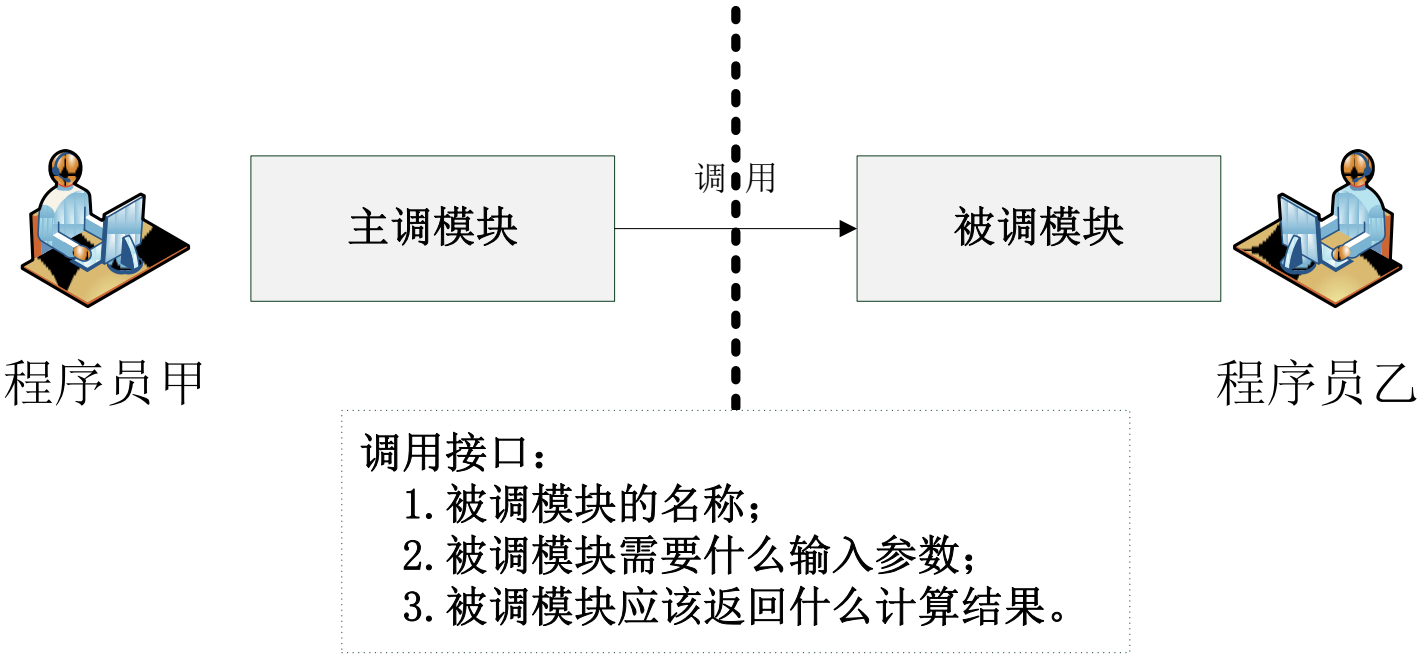
5.1 结构化程序设计方法

- 常见的模块调用关系图
  - 模块重用



5.1 结构化程序设计方法

- 主调模块与被调模块



5.1 结构化程序设计方法

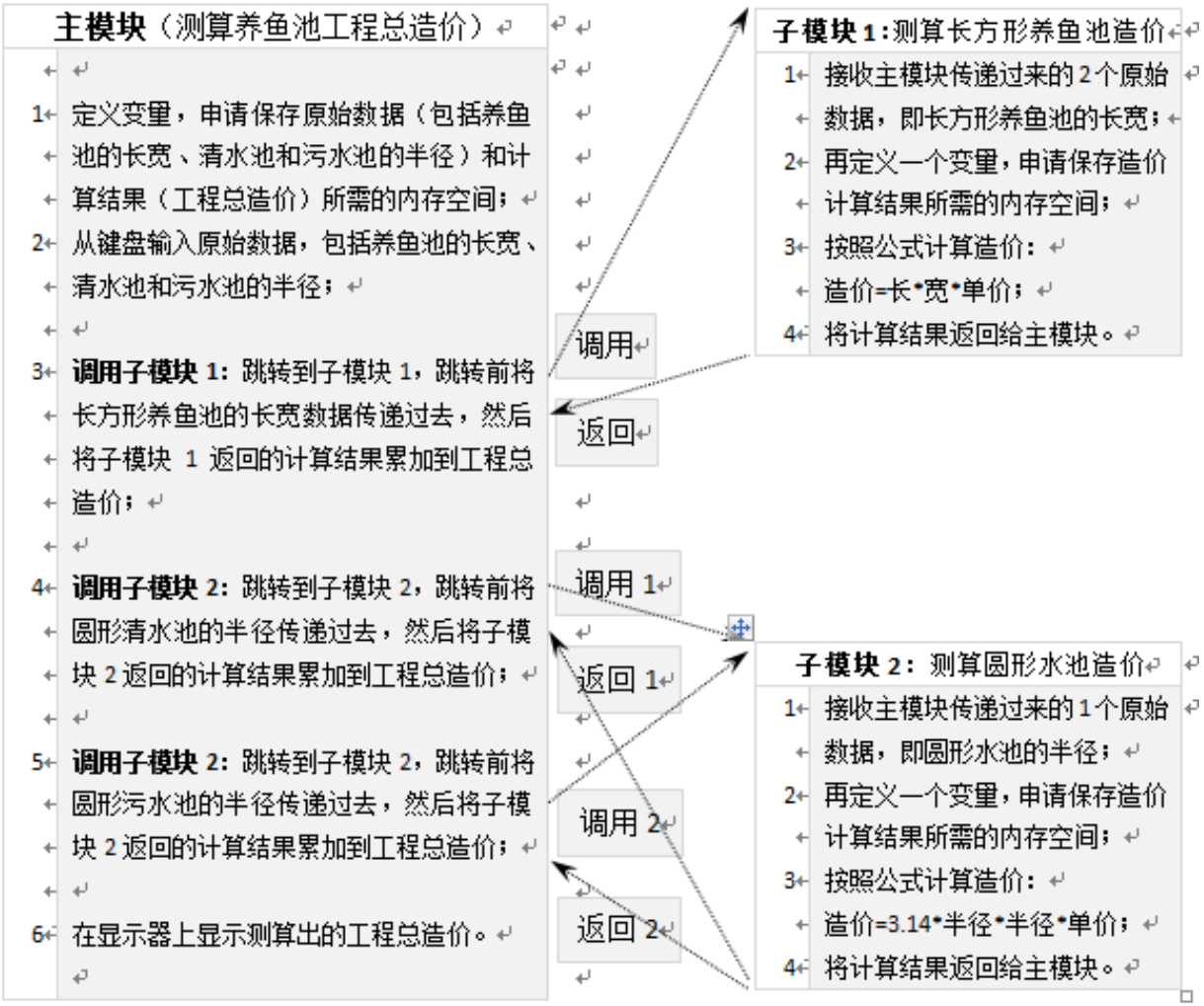
- 结构化程序设计方法
  - 模块化是团队分工的基础
  - 模块接口是团队协作的基础
  - 模块重用影响大型软件开发的组织与管理方式

- 代码重用
  - 现在的软件开发项目可以重用以前项目所开发的代码，即跨时间段重用
  - 一个软件项目可以重用另一个项目中的代码，即跨项目重用
  - 可以重用本单位已有的代码，也可以购买外单位已有的代码，或委托外单位开发所需的子模块，即跨组织机构的重用

5.1 结构化程序设计方法

- 模块的4大要素
  - 模块名称、输入参数、返回值、算法
- 模块设计者
- 模块调用者
  - *f(x)*
- 主模块与子模块
- C++语言支持结构化程序设计方法，以函数的语法形式来描述和组装模块，即函数的定义和调用

5.2 函数的定义和调用


主模块（测算养鱼池工程总造价）
1 定义变量，申请保存原始数据（包括养鱼池的长宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间；
2 从键盘输入原始数据，包括养鱼池的长宽、清水池和污水池的半径；
3 调用子模块 1：跳转到子模块 1，跳转前将长方形养鱼池的长宽数据传递过去，然后将子模块 1 返回的计算结果累加到工程总造价；
4 调用子模块 2：跳转到子模块 2，跳转前将圆形清水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价；
5 调用子模块 2：跳转到子模块 2，跳转前将圆形污水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价；
6 在显示器上显示测算出的工程总造价。
调用
返回
调用 1
返回 1
调用 2
返回 2
子模块 1：测算长方形养鱼池造价
1 接收主模块传递过来的 2 个原始数据，即长方形养鱼池的长宽；
2 再定义一个变量，申请保存造价计算结果所需的内存空间；
3 按照公式计算造价：
造价=长*宽*单价；
4 将计算结果返回给主模块。
子模块 2：测算圆形水池造价
1 接收主模块传递过来的 1 个原始数据，即圆形水池的半径；
2 再定义一个变量，申请保存造价计算结果所需的内存空间；
3 按照公式计算造价：
造价=3.14*半径*半径*单价；
4 将计算结果返回给主模块。

5.2 函数的定义和调用

- 函数的定义

C++语法：**定义函数**

```
函数类型 函数名(形式参数列表)
{
  函数体
}
```

语法说明：

- **函数类型**定义的是函数返回值（即函数值）的数据类型。函数类型由函数功能决定，可以是除数组之外的任何数据类型，省略时默认为int型。某些函数可能只是完成某种功能，但没有返回值，此时函数类型应定义为void；
- **函数名**定义函数的名称，由程序员命名，需符合标识符的命名规则。通常函数之间不能重名；
- **形式参数列表**定义了函数接收输入参数所需的变量，这些变量称为**形式参数**，简称为**形参**。可以有多个形参，每个形参以“**数据类型 变量名**”的形式定义，形参之间用“,”隔开。某些函数可能不需要输入参数，此时形式参数列表省略为空；
- **函数体**是描述数据处理算法的C++语句序列，用大括号“{}”括起来。函数体中可以定义专供本函数使用的变量。如果函数有返回值，则应使用return语句返回，返回值的数据类型应与函数类型一致；
- “**函数类型 函数名(形式参数列表)**”也称为**函数头**，它定义了函数的调用接口，即函数名、输入参数和返回值类型。

5.2 函数的定义和调用

例5-3 测算长方形养鱼池造价模块的函数定义

子模块1：测算长方形养鱼池造价	**子函数1**：RectCost
1 接收主模块传递过来的2个原始数据，即长方形养鱼池的长宽； 2 再定义一个变量，申请保存造价计算结果所需的内存空间； 3 4	double RectCost(double a, double b) {

主模块如何调用这2个子模块？

例5-4 测算圆形水池造价模块的函数定义

子模块2：测算圆形水池造价	**子函数2**：CircleCost

1 接收主模块传递过来的1个原始数据，即圆形水池的半径；
2 再定义一个变量，申请保存造价计算结果所需的内存空间；
3 按照公式计算造价：
造价=3.14*半径*半径*单价；
4 将计算结果返回给主模块。
	double CircleCost(double r)
	{
	double cost ;
	cost = 3.14 * r * r * 10 ;
	return cost ;
	}

5.2 函数的定义和调用

- 函数的调用

C++语法：**调用函数**

函数名(实际参数列表)

语法说明：

- **函数名**指定被调用函数的名称；
- **实际参数列表**指定函数所需要的输入参数。调用函数时应按被调用函数的要求给定具体的输入参数值，这些参数值称为**实际参数**，简称为**实参**。实际参数可以是常量、变量或表达式，参数之间用“,”隔开。调用时，首先将实参值按位置顺序一一赋值给对应的形参变量，这称为函数调用时的**参数传递**。实参与形参应当个数一致，类型一致；
- “**函数名(实际参数列表)**”是调用某个函数。有返回值的函数调用可作为操作数参与表达式运算，该操作数等于函数返回值。某些函数可能只是完成某种功能，但没有返回值。无返回值的函数调用加分号“;”即构成一条函数调用语句；
- 一个函数调用另一个函数，调用别人的函数称为**主调函数**，被调用的函数称为**被调函数**。

5.2 函数的定义和调用

例5-5 测算养鱼池工程总造价主模块的函数定义

主模块：测算养鱼池工程总造价		**主函数**：main

1	定义变量，申请保存原始数据（包括养鱼池的长宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间；	int main( )
		{
		double length, width;
		double r1, r2;
		double totalCost = 0;

2	从键盘输入原始数据，包括养鱼池的长宽、清水池和污水池的半径；	cout << “请输入长方形的长宽：”; cin >> length >> width; cout << “请输入清水池和污水池的半径：”; cin >> r1 >> r2;
3	**调用子模块1**：跳转到子模块1，跳转前将长方形养鱼池的长宽数据传递过去，然后将子模块1返回的计算结果累加到工程总造价；	totalCost += RectCost(length, width);
4	**调用子模块2**：跳转到子模块2，跳转前将圆形清水池的半径传递过去，然后将子模块2返回的计算结果累加到工程总造价；	totalCost += CircleCost(r1);
5	**调用子模块2**：跳转到子模块2，跳转前将圆形污水池的半径传递过去，然后将子模块2返回的计算结果累加到工程总造价；	totalCost += CircleCost(r2);
6	在显示器上显示测算出的工程总造价。	cout << “工程总造价为” << totalCost << endl; return 0; }

5.2 函数的定义和调用

- 数据参数化

带参数的计算圆面积函数	**不带参数**的计算圆面积函数

// 计算半径为r的圆面积
double CircleArea1(double r)
{
  double area;
  area = 3.14 * r * r;
  return area;
}
	// 计算半径为5的圆面积
	double CircleArea2( )
	{
	  double area;
	  area = 3.14 * 5 * 5;
	  return area;
	}

```
cout << CircleArea1( 5 );  // 得到半径为5的圆面积
cout << CircleArea1( 10 );  // 得到半径为10的圆面积
```

5.2 函数的定义和调用

- 函数的执行

例 5-6 测算养鱼池工程总造价的 C++程序

```
#include <iostream>   // 导入外部程序：标准输入输出流
using namespace std;

double RectCost(double a, double b)   // 计算长方形养鱼池造价的函数定义
{
    double cost ;
    cost = a * b * 10 ;
    return cost ;
}

double CircleCost(double r)   // 计算圆形水池造价的函数定义
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
}

int main( )   // 主函数定义
{
    double length, width;   // 定义变量：分别保存长方形养鱼池的长宽
    double r1, r2;   // 定义变量：分别保存圆形清水池和污水池的半径
    double totalCost = 0;   // 定义变量：保存最终的计算结果，即总造价

    cout << "请输入长方形的长宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;

    totalCost += RectCost( length, width );   // 调用函数 RectCost 计算长方形养鱼池造价
    totalCost += CircleCost( r1 );   // 调用函数 CircleCost 计算圆形清水池造价
    totalCost += CircleCost( r2 );   // 再次调用函数 CircleCost 计算圆形污水池造价

    cout << "工程总造价为 " << totalCost << endl;
    return 0;
}
```

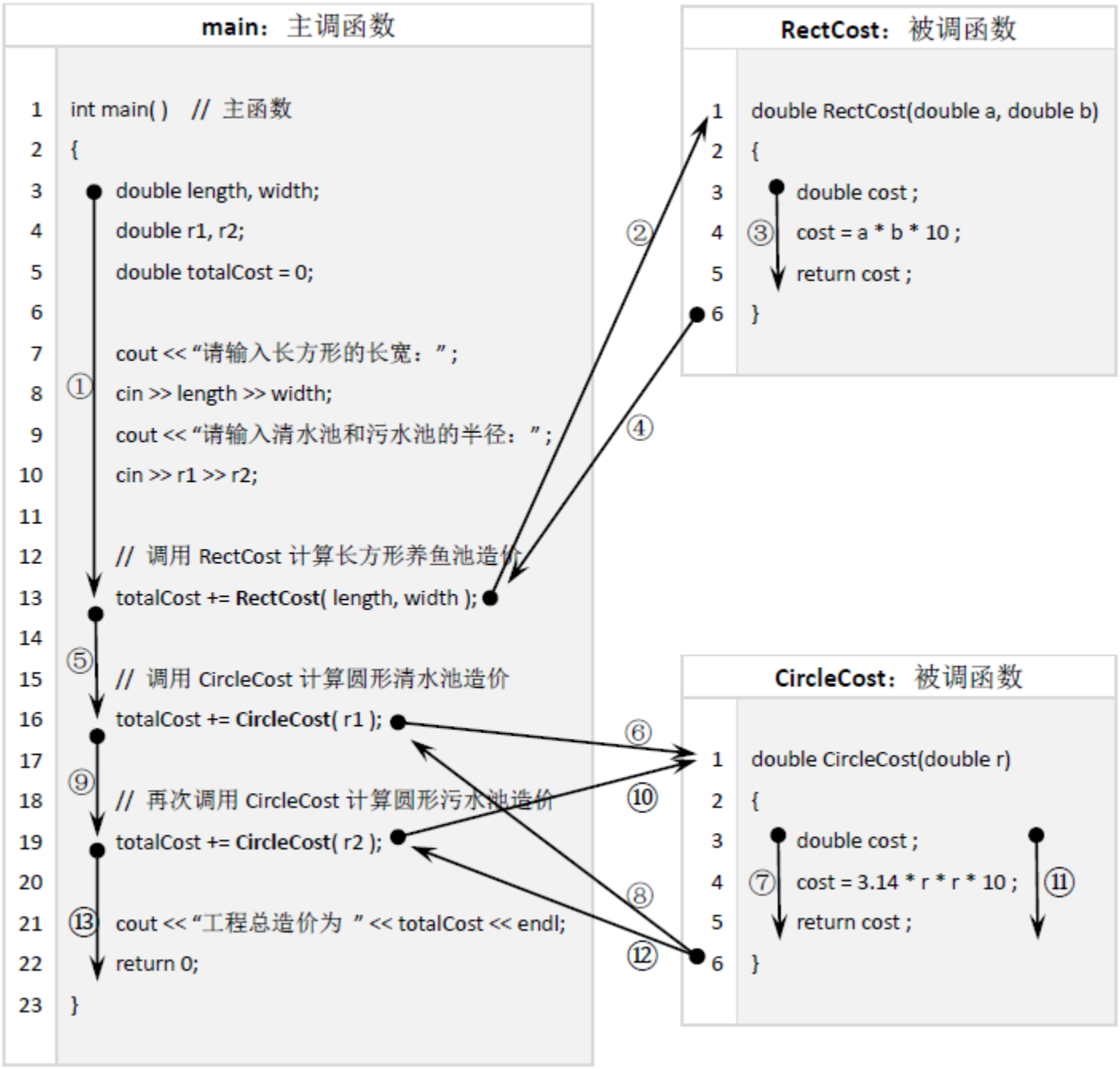


图 5-5 例 5-6 程序中函数的执行过程

5.2 函数的定义和调用

- 主调函数与被调函数间的2次数据传递
  - 主调函数传给被调函数：实参 → 形参
  - 被调函数传给主调函数：返回值

C++语法：**return语句**

```
    return (表达式) ;
或
    return ;
```

语法说明：

- 如果函数有返回值，则应当使用“**return (表达式)** ;”语句结束函数执行，返回主调函数。**表达式**指定返回时的返回值，其数据类型应当与函数头中的函数类型一致。常量或变量可认为是一个最简单的表达式。**小括号**“()”可以省略；
- 如果函数的类型为void，即没有返回值，则可以使用“**return** ;”语句结束函数执行，返回主调函数。如果省略“return ;”语句，则在执行完函数体中最后一条语句后，自动返回主调函数。

5.2 函数的定义和调用

例5-7 在return语句中使用表达式简化函数代码

函数RectCost： 简化前

```
1  double RectCost(double a, double b)
2  {
3      double cost ;
4      cost = a * b * 10 ;
5      return cost ;
6  }
```

函数RectCost： 简化后

```
double RectCost(double a, double b)
{
    return (a * b * 10) ;
}
```

函数CircleCost： 简化前

```
1  double CircleCost(double r)
2  {
3      double cost ;
4      cost = 3.14 * r * r * 10 ;
5      return cost ;
6  }
```

函数CircleCost： 简化后

```
double RectCost(double r)
{
    return (3.14 * r * r * 10) ;
}
```

5.2 函数的定义和调用

- 函数的声明

C++语法：**声明函数原型**

函数类型 函数名(形式参数列表);

语法说明：

- 一个函数的原型声明语句可简单认为是由该函数定义中的**函数头**+分号“;”组成；
- 形式参数列表中的数据类型是函数原型声明中必须包含的信息，它除了指明形参的数据类型之外，还暗含了形参的个数，而形参变量名不重要，可以省略。形参名的作用是为了便于调用该函数的程序员理解其实际含义；
- C++源程序中，被调函数的原型声明语句可放在主调函数定义之前、或整个源程序文件的前面，也可以放在主调函数的函数体里面，但必须在该函数的调用语句之前；
- 如果被调函数与主调函数定义在同一个源程序文件中，并且被调函数定义在主调函数之前，则被调函数的原型声明语句可以省略；
- 声明函数原型的目的是将被调函数的调用接口预先告知编译器程序，这样编译器程序就可以按照该调用接口检查接下来的函数调用语句是否正确。

举例：例5-6中函数RectCost的原型声明

```
double RectCost(double a, double b) ; // 复制RectCost函数定义中的函数头，再加“;”
```

或

```
double RectCost(double, double) ; // 省略上述原型声明中的形参变量名
```

5.2 函数的定义和调用

例5-8 测算养鱼池工程总造价的C++程序（声明函数原型）

```
1  #include <iostream>
2  using namespace std;
3
4  double RectCost(double a, double b);  // 声明子函数RectCost的原型
5  double CircleCost(double r);  // 声明子函数CircleCost的原型
6
7  int main( ) // 主函数定义（略）
8  {
9      ......
10     totalCost += RectCost( length, width );  // 调用函数RectCost计算长方形养鱼池造价
11     totalCost += CircleCost( r1 );  // 调用函数CircleCost计算圆形清水池造价
12     totalCost += CircleCost( r2 );  // 再次调用函数CircleCost计算圆形污水池造价
13     ......
14 }
15
16 double RectCost(double a, double b)  // 计算长方形养鱼池造价的函数定义（省略）
17 { ..... }
18
19 double CircleCost(double r)  // 计算圆形水池造价的函数定义（省略）
20 { ..... }
```

5.2 函数的定义和调用

- 源程序中定义的函数一定会被执行吗？
  - 源程序中定义的函数可能执行，也可能不执行。只有被主函数直接或间接调用的函数才会被执行，否则就不会被执行
  - 源程序中定义的函数可能被执行多次。函数被调用一次就执行一次，调用多次则执行多次

5.2 函数的定义和调用

- 程序员与函数的关系
  - 编写被调函数的程序员：被调函数应能够完成特定的程序功能。程序员编写被调函数就是为了自己多次使用，或提供给别的程序员使用
  - 编写主调函数的程序员：主调函数调用被调函数实际上是重用其代码，以实现相应的程序功能。编写主调函数的程序员只会关心如何调用函数，比如被调函数的函数名是什么、需传递什么参数、返回值是什么类型，不会关心函数体中的算法是怎么实现的

5.3 数据的管理策略

- 函数间需共享数据
  - 数据分散管理：将数据分散交由各个函数管理，函数各自定义变量申请自己所需的内存空间，其它函数不能直接访问其中的数据，需要时可通过数据传递来实现共享。采用分散管理策略时，程序员应当将定义变量语句放在函数的函数体中，这样所定义的变量称为局部变量。局部变量属本函数所有，其它函数不能直接访问

  - 数据集中管理：将数据集中管理，统一定义公共的变量来存放共享数据，所有函数都可以访问。采用集中管理策略时，程序员应当将定义变量语句放在函数外面（不在任何函数的函数体中），这样所定义的变量称为全局变量。全局变量不属于任何函数，是公共的，所有函数都可以访问

5.3 数

- 分散管理，按需传递
  - 形实结合
  - 返回值

例5-6 测算养鱼池工程总造价的C++程序

```
#include <iostream> // 导入外部程序：标准输入输出流
using namespace std;

double RectCost(double a, double b) // 计算长方形养鱼池造价的函数定义
{
    double cost ;
    cost = a * b * 10 ;
    return cost ;
}

double CircleCost(double r) // 计算圆形水池造价的函数定义
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
}

int main( ) // 主函数定义
{
    double length, width; // 定义变量：分别保存长方形养鱼池的长宽
    double r1, r2; // 定义变量：分别保存圆形清水池和污水池的半径
    double totalCost = 0; // 定义变量：保存最终的计算结果，即总造价

    cout << "请输入长方形的长宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;

    totalCost += RectCost( length, width ); // 调用函数RectCost计算长方形养鱼池造价
    totalCost += CircleCost( r1 ); // 调用函数CircleCost计算圆形清水池造价
    totalCost += CircleCost( r2 ); // 再次调用函数CircleCost计算圆形污水池造价

    cout << "工程总造价为 " << totalCost << endl;
    return 0;
}
```

5.3 数据的管理策略

- 数据集中管理，全局共享

  数据集中管理策略就是将数据集中管理，统一定义公共的变量来存放共享数据，所有函数都可以访问，这样可以减少函数间的数据传递。采用集中管理策略时，程序员应当将定义变量语句放在函数外面（不在任何函数的函数体中），这样所定义的变量称为全局变量。全局变量不属于任何函数，是公共的，所有函数都可以访问

例5-9 测算养鱼池工程总造价的C++程序（数据集中管理策略）

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

// 下列变量是全局变量，是公有的，所有函数都可以访问
double length, width; // 定义变量：分别保存长方形养鱼池的长宽
double r1, r2; // 定义变量：分别保存圆形清水池和污水池的半径
double totalCost = 0; // 定义变量：保存最终的计算结果，即总造价

void RectCost( ) // 计算长方形养鱼池造价：改用全局变量后的函数定义
{
    double cost ;
    cost = length * width * 10 ; // 直接读取全局变量length和width中的长宽数据
    totalCost += cost ; // 将计算结果直接累加到全局变量totalCost中
    return; // 该语句可以省略
}

double CircleCost(double r) // 计算圆形水池造价：改用全局变量后，函数定义不变
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
}

int main( ) // 主函数定义
{
  // 将例5-6在此定义的局部变量全部移到函数外面定义（第5 ~7行），变成全局变量
  // 下列语句将键盘输入的原始数据直接存放到对应的全局变量中
    cout << "请输入长方形的长宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;

    RectCost( ); // 调用函数RectCost计算长方形养鱼池造价
    totalCost += CircleCost( r1 ); // 调用函数CircleCost计算圆形清水池造价
    totalCost += CircleCost( r2 ); // 再次调用函数CircleCost计算圆形污水池造价

    cout << "工程总造价为" << totalCost << endl;
    return 0;
}
```

5.3 数据的管理策略

- 变量的作用域
  - 在函数的函数体中定义的变量是局部变量，只能被本函数访问。定义在函数外面（不在任何函数的函数体中）的变量是全局变量，可以被所有函数访问。不同类型的变量具有不同的访问范围，这就是变量作用域的概念
  - C++源程序中的变量需要遵循“先定义，后访问”的原则，即变量在定义之后，其后续的语句才能访问该变量。变量的**作用域**（Scope）指的是C++源程序中可以访问该变量的代码区域
  - C++语言根据定义位置将变量分为局部变量、全局变量和函数形参等3种类型，它们具有不同的作用域。作用域也分为3种，分别是块作用域、文件作用域和函数原型作用域。所有变量只能在其作用域范围内访问

5.3 数据的管理策略

- 变量的作用域
  - 局部变量

    用一对大括号括起来的源程序代码称为一个代码块，例如函数的函数体是一个代码块，一条复合语句也是一个代码块。块作用域是从变量定义位置开始，一直到其所在代码块的右大括号为止。局部变量具有块作用域，只能被作用域内的语句访问

5.3 数据的管理策略

- 变量的作用域
  - 全局变量

    文件作用域是从变量定义位置开始，一直到其所在源程序文件结束为止。全局变量具有文件作用域，其作用域内的函数都可以访问

5.3 数据的管理策略

- 变量的作用域
  - 形式参数
    - 函数定义中的形参具有块作用域，这里的代码块指的是该函数的函数体。函数定义中的形参只能被本函数体内的语句访问
    - 函数声明中的形参不能也不需被访问，其作用域为空，称为函数原型作用域。声明函数时，其形参列表可以只声明形参个数和类型，而形参名可以省略。函数声明中形参名的作用仅仅是为了便于调用该函数的程序员理解其实际含义，没有其它语法作用

$$x = y^2 + z^2$$

例 5-10 演示不同类型变量及其作用域的 C++程序

```
1  #include <iostream>
2  using namespace std;
3
4  int fun1(int p1, int p2);   // fun1 函数声明：计算 p1^2+p2^2。形参 p1 和 p2 具有函数原型作用域
5  int fun2(int p);    // fun2 函数声明：计算 p^2。形参 p 具有函数原型作用域
6
7  int x = 0;   // 全局变量 x：保存最终的计算结果
8
9  int main( )   // 主函数定义
10 {
11     cout << "Function main." ;   // 执行主函数时显示该信息
12
13     int y = 5, z = 10;   // 局部变量 y 和 z
14
15     x = fun1(y, z);   // 调用函数 fun1 计算 y^2+z^2
16
17     cout << "计算结果为：" << x << endl;
18     return 0;
19 }
20
21 int fun1(int p1, int p2)   // fun1 函数定义：计算 p1^2+p2^2
22 {
23     cout << "Function fun1." ;   // 执行函数 fun1 时显示该信息
24
25     int result;   // 局部变量 result
26     result = fun2( p1 );   // 调用函数 fun2 计算 p1^2
27     result += fun2(p2);   // 调用函数 fun2 计算 p2^2
28     return result;
29 }
30
31 int fun2(int p)   // fun2 函数定义：计算 p^2
32 {
33     cout << "Function fun2." ;   // 执行函数 fun2 时显示该信息
34
35     int result;   // 局部变量 result
36     result = p * p;   // 计算 p^2
37     return result;
38 }
```

全局变量 x 具有**文件作用域**

变量 y 和 z 具有**块作用域**

形参 p1 和 p2 具有**块作用域**

变量 result 具有**块作用域**

形参 p 具有**块作用域**

变量 result 具有**块作用域**

5.3 数据的管理策略

- 全局变量的“先声明，后访问”

程序1：将全局变量定义在源程序的开头	**程序2**：将全局变量定义在源程序的末尾

#include <iostream>
using namespace std;
 	
int r; // 将全局变量r定义在源程序的开头
 	
int main( )
{
  cin >> r;
  cout << (3.14 * r * r);
  return 0;
}
	#include <iostream>
	using namespace std;
	
	// 使用外部声明语句延伸全局变量r的作用域
	extern int r;
	
	int main( )
	{
	  cin >> r;
	  cout << (3.14 * r * r);
	  return 0;
	}
	
	int r; // 将全局变量r定义在源程序的末尾

5.3 数据的管理策略

- 变量C++名

例5-11 演示重名变量的C++示例程序

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

int x = 10;  // 定义int型全局变量x，初始化为10
double x = 1.5;  // 定义double型全局变量x，初始化为1.5。语法错误：不能重名

int main( )
{
    cout << x << endl;  // 显示第4行变量x的值：10

    int x = 20;  // 定义int型局部变量x，初始化为20
    double x = 2.5;  // 定义double型局部变量x，初始化为2.5。语法错误：不能重名

    cout << x << endl;  // 显示第11行变量x的值：20

    for (int n = 1; n <= 5; n++)
    {
        int x = 30;  // 定义int型局部变量x，初始化为30
        double x = 3.5;  // 定义double型局部变量x，初始化为3.5。语法错误：不能重名

        cout << x << endl;  // 显示第18行变量x的值：30
    }

    cout << x << endl;  // 显示第11行变量x的值：20
    return 0;
}
```

访问重名变量时：局部优先

5.4 程序代码和变量的存储原理

加载后的程序副本

操作系统 及已运行的应用程序	
int main( ) { ..... }	代码区
int fun1(int p1, int p2) { ..... }	
int fun2(int p) { ..... }	
“Function main.” “Function fun1.” “Function fun2.”	常量区
x: 0	静态存储区
栈剩余内存	自动存储区（栈）
系统空闲内存	堆

(a) 加载后的程序副本

$$x = y^2 + z^2$$

```
int x = 0;

int main()
{
   cout << “Function main.” ;
   int y=5, z=10;
   x = fun1(y, z);
   cout << x;
   return 0;
}

int fun1(int p1, int p2)
{
   cout << “Function fun1.” ;
   int result;
   result = fun2( p1 );
   result += fun2(p2);
   return result;
}

int fun2(int p)
{
   cout << “Function fun12” ;
   int result;
   result = p*p;
   return result;
}
```

5.4 程序代码和变量的存储原理

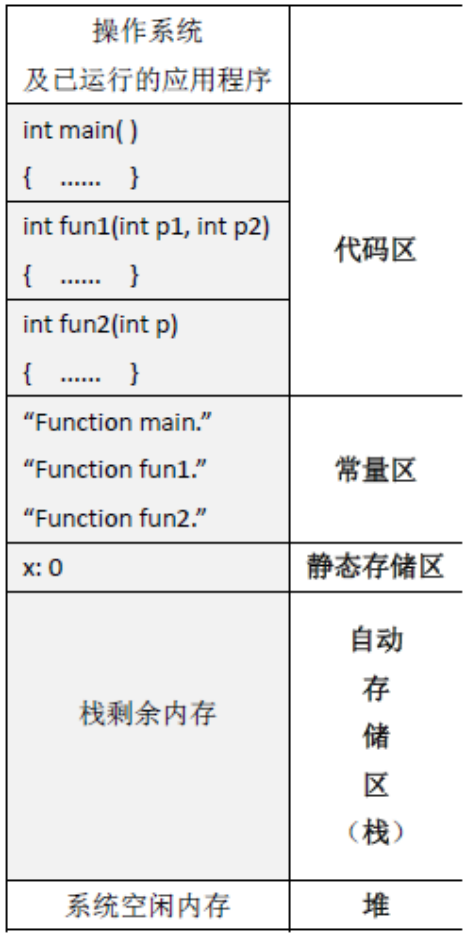
- 程序加载后立即为其中的全局变量分配内存。全局变量将一直占用所分配的内存，直到程序执行结束退出时才被释放，这种内存分配方法称为**静态分配**

- 局部变量是在计算机执行到其定义语句时才分配内存，到其所在代码块执行结束即被释放，这种内存分配方法称为**自动分配**
  - 一个函数或复合语句可能被执行多次，随着这些函数或复合语句的执行，其中所定义的局部变量将不断重复内存的分配-释放过程。这个过程是自动完成的，不需要程序员干预
  - 函数定义中的形参也是自动分配内存的，当执行到该函数的调用语句时为形参分配内存，并将传递来的实参值写入该内存，然后执行函数体，当函数体执行结束后即释放内存

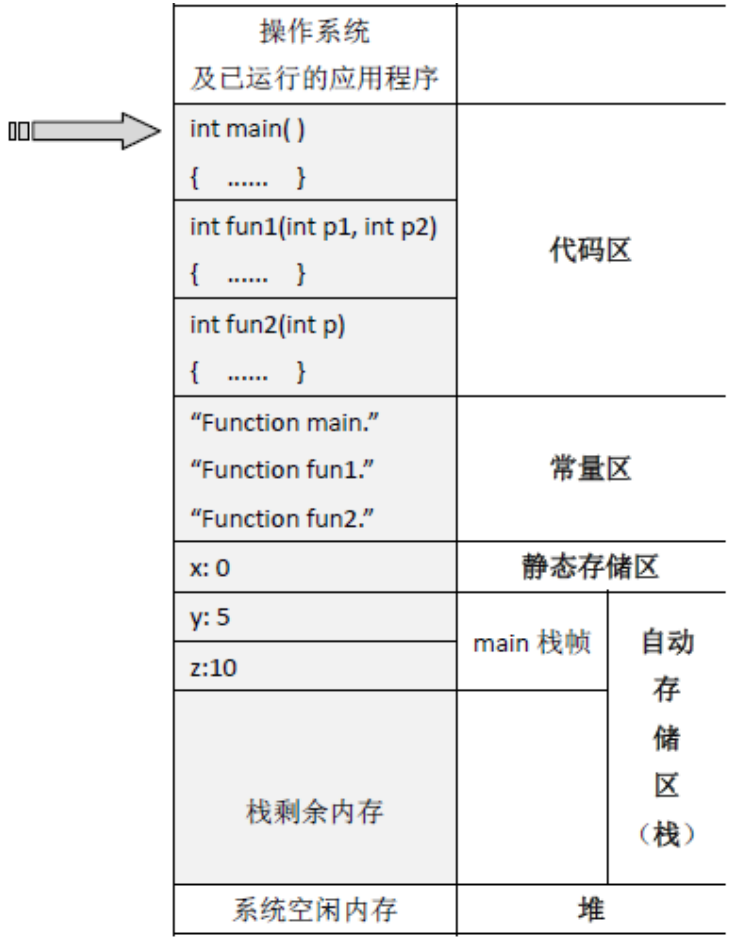
- 生存期
  - 从加载到执行结束退出这个时间段是一个程序在内存中的**生存期**
  - 从内存分配到释放这个时间段就是一个变量在内存中的**生存期**

5.4 程序代码和变量的存储原理

```
int x = 0;

int main()
{
   cout << “Function main.” ;
   int y=5, z=10;
   x = fun1(y, z);
   cout << x;
   return 0;
}

int fun1(int p1, int p2)
{
   cout << “Function fun1.” ;
   int result;
   result = fun2( p1 );
   result += fun2(p2);
   return result;
}

int fun2(int p)
{
   cout << “Function fun12” ;
   int result;
   result = p*p;
   return result;
}
```



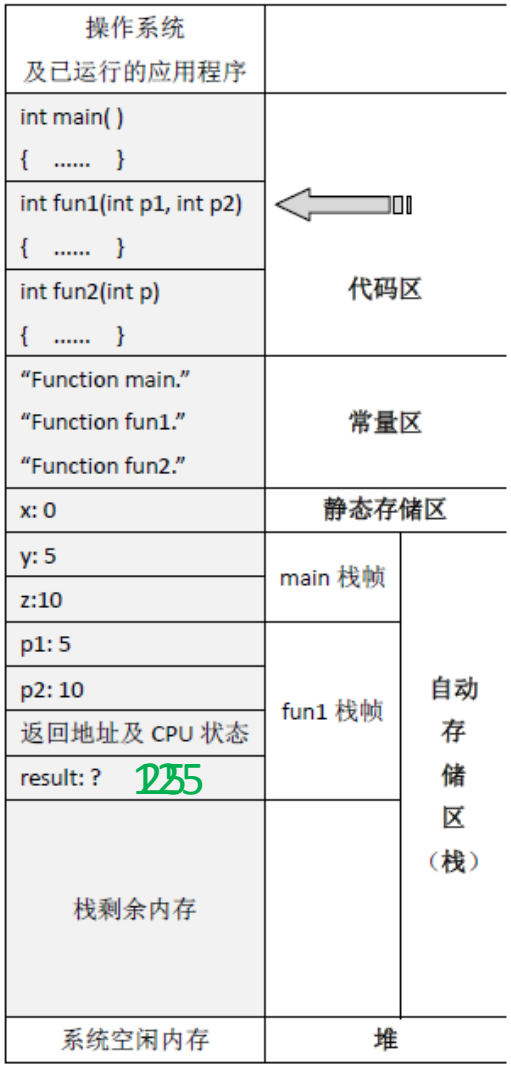
(a) 加载后的程序副本



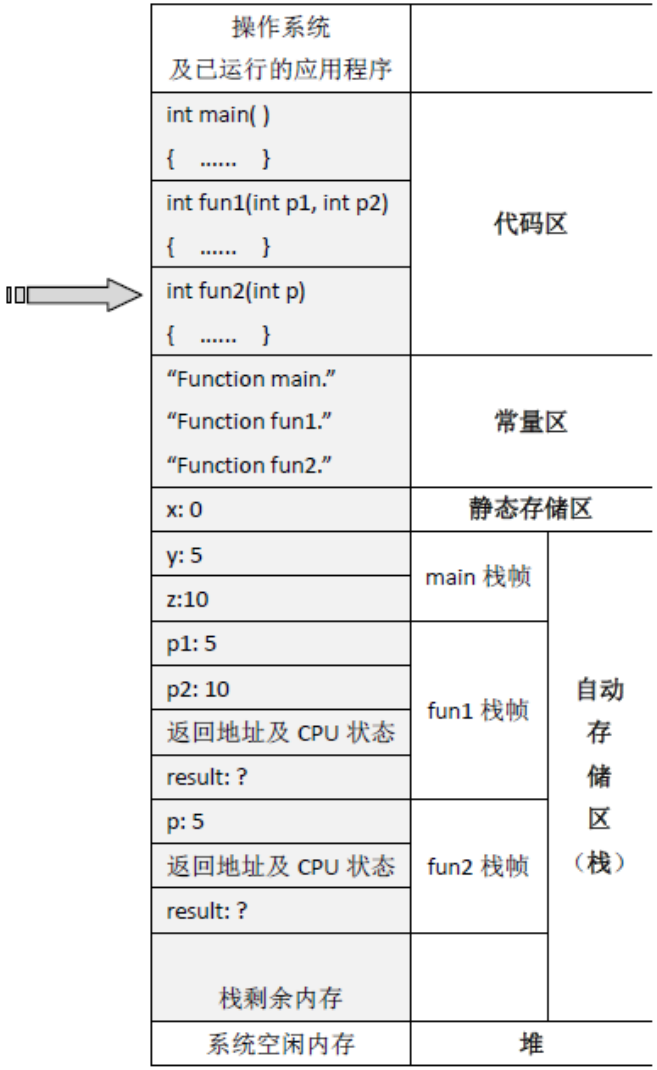
(b) 主函数执行时的程序副本

5.4 程序代码和变量的存储原理

```
int x = 0;

int main()
{
   cout << “Function main.” ;
   int y=5, z=10;
   x = fun1(y, z);
   cout << x;
   return 0;
}

int fun1(int p1, int p2)
{
   cout << “Function fun1.” ;
   int result;
   result = fun2( p1 );
   result += fun2(p2);
   return result;
}

int fun2(int p)
{
   cout << “Function fun12” ;
   int result;
   result = p*p;
   return result;
}
```



(c) 调用函数 fun1 时的程序副本



(d) 调用函数 fun2 时的程序副本

5.4 程序代码和变量的存储原理

- 计算机执行函数调用语句的具体过程：
  - 执行到函数调用语句时，暂停主调函数的执行，跳转去执行被调函数；
  - 为被调函数定义的形参分配好内存，计算调用语句中的实参表达式，将表达式结果按位置顺序一一赋值给对应的形参，即形实结合；
  - 保存好返回地址和当前CPU状态信息，即保存主调函数调用前的现场；
  - 执行被调函数的函数体；
  - 执行完被调函数的函数体或执行到其中的return语句，停止被调函数的执行。如果有返回值，将返回值传回主调函数；
  - 恢复主调函数调用前的现场，调用结束；
  - 按照返回地址继续执行主调函数中剩余的指令。

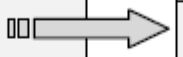
5.4 程序代码和变量的存储原理

- 动态内存分配

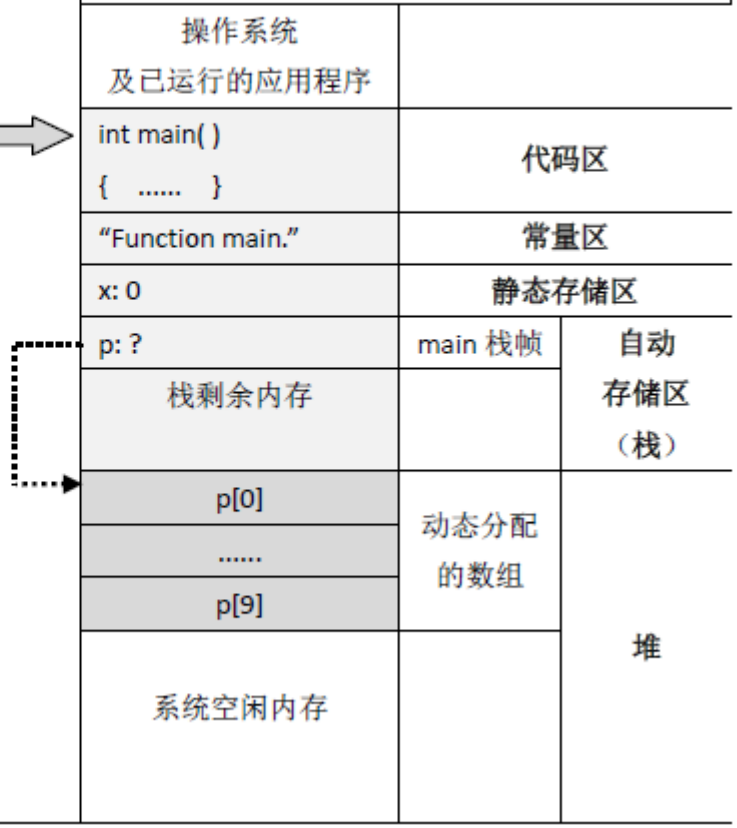
例 5-12 一个运用动态内存分配方法的 C++演示程序

源程序

```
1  #include <iostream>
2  using namespace std;
3
4  int x = 0;   // 定义一个全局变量 x
5
6  int main( )
7  {
8        cout << "Function main.";
9
10       int *p;   // 定义一个指针变量 p
11       p = new int[10];   // 动态分配数组
12       ..... // 访问数组
13       delete []p;   // 释放数组占用的内存
14       return 0;
15 }
```



执行时的内存示意图



5.4 程序代码和变量的存储原理

- 函数指针
  - 可以通过内存地址访问变量，也可以通过内存地址调用函数
  - 计算机程序在执行时被读入内存，在内存中建立一个程序副本，其中包括各函数的代码。也就是说，执行时程序中各函数的代码是存放在内存中的
  - 调用函数一般是通过函数名来调用，也可以函数代码的首地址来调用
  - 通过地址调用函数需分为3步，依次是定义函数型指针变量、将函数首地址赋值给该指针变量、通过指针变量间接调用函数

5.4 程序代码和变量的存储原理

- 函数指针

C++语法：**定义函数型指针变量**

函数类型 (*指针变量名)(形式参数列表) ;

语法说明：

- 函数型指针变量需要与函数匹配。“**匹配**”的含义是：在定义函数型指针变量时，除了在变量名前加指针说明符“*”之外，同时还需要指定函数的形式参数和返回值类型；
- **形式参数列表**和**函数类型**分别指定了函数的形参和返回值类型，其中的形参名可以省略。指针变量将只能指向其定义语句中所规定形参和返回值类型的函数；
- **指针变量名**需符合标识符的命名规则。

5.4 程序代码和变量的存储原理

- 函数指针

```
double fun1(double x, int y) { return (x+y); }
cout << fun1(3.5, 2);

double (*p)(double, i

p = fun1;
cout << (*p)(3.5, 2);
cout << p(3.5, 2);   // 调用形式更加简单
```

•函数名实际上可以理解成是指向函数代码的指针，只不过它是一种指针常量，固定指向某个函数

•一个函数型指针变量可以指向多个函数。这些函数都需要与指针变量匹配，即它们具有相同的形参和返回值类型

5.4 程序代码和变量的存储原理

例5-13 应用函数型指针变量的C++程序

```
1  #include <iostream>
2  using namespace std;
3
4  // 下列4个函数具有相同的形参和函数类型
5  double fun1(double x, int y)  // 函数fun1实现简单的加法功能
6  {   return (x+y);   }
7  double fun2(double x, int y)  // 函数fun2实现简单的减法功能
8  {   return (x-y);   }
9  double fun3(double x, int y)  // 函数fun3实现简单的乘法功能
10 {   return (x*y);   }
11 double fun4(double x, int y)  // 函数fun4实现简单的除法功能
12 {   return (x/y);   }
13
14 int main( )  // 主函数定义
15 {
16     double (*p)(double, int);
17     p = fun1;  cout << (*p)(3.5, 2) << endl;  // 间接调用fun1实现加法，显示结果：5.5
18     p = fun2;  cout << (*p)(3.5, 2) << endl;  // 间接调用fun2实现减法，显示结果：1.5
19     p = fun3;  cout << p(3.5, 2) << endl;     // 间接调用fun3实现乘法，显示结果：7.0
20     p = fun4;  cout << p(3.5, 2) << endl;     // 间接调用fun4实现除法，显示结果：1.75
21     return 0;
22 }
```

5.5 函数间参数传递的3种方式

- 参数传递
  - 值传递
  - 引用传递
  - 指针传递

- 编写程序

  定义2个变量： int x = 5, y = 10; 编写一个函数swap来交换这2个变量的数值。交换后，x的值应为10，y的值应为5

5.5 函数间参数传递的3种方式

- 值传递

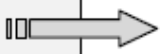
值传递是
调函数的
保存一份
参中的副

例 5-14 一个交换变量值的 swap 函数（值传递）

源程序

```
1  #include <iostream>
2  using namespace std;
3
4  void swap(int a, int b)
5  {
6      int t;
7      // 下列 3 条语句可交换 a 和 b 的值
8      t = a;   a = b;   b = t;
9  }
10
11 int main( )
12 {
13     cout << "Exchange x and y." << endl;
14     int x = 5, y = 10;
15     cout << x << "," << y << endl;
16
17     swap(x, y);   // 调用函数来交换变量值
18     cout << x << "," << y << endl;
19     return 0;
20 }
```

执行时的内存示意图



操作系统及已运行的应用程序		
int main( ) { ..... }	代码区	
void swap(int a, int b) { ..... }		
"Exchange x and y."	常量区	
x: 5	main 栈帧	自动存储区（栈）
y: 10		
a: 10	swap 栈帧	
b: 5		
t: 5		
返回地址和 CPU 状态		
栈剩余内存		
系统空闲内存		堆

5, 10

5, 10

5.5 函数间参数传递的3种方式

- 值传递的特点
  - 值传递只是将主调函数中实参的值传递给被调函数的形参，无论被调函数如何修改形参都不会影响到实参。
  - 值传递只能将数据从主调函数传给被调函数，这是一种单向数据传递机制。
  - 值传递时，实参可以是常量、变量或表达式。
  - 值传递的好处是：如果被调函数错误修改了形参，它不会影响到主调函数中的实参。

5.5 函数间参数传递的3种方式

- 引用传递
  - 函数中定义的局部变量不能被其它函数直接访问，但能够被间接访问。**引用传递**就是传递实参变量的引用，被调函数通过该引用间接访问主调函数中的变量，从而达到传递数据的目的
  - 采用引用传递时，被调函数的形参需定义成引用变量

5.5 函数间参数传递的3种方式

int &a = x;
int &b = y;

5, 10

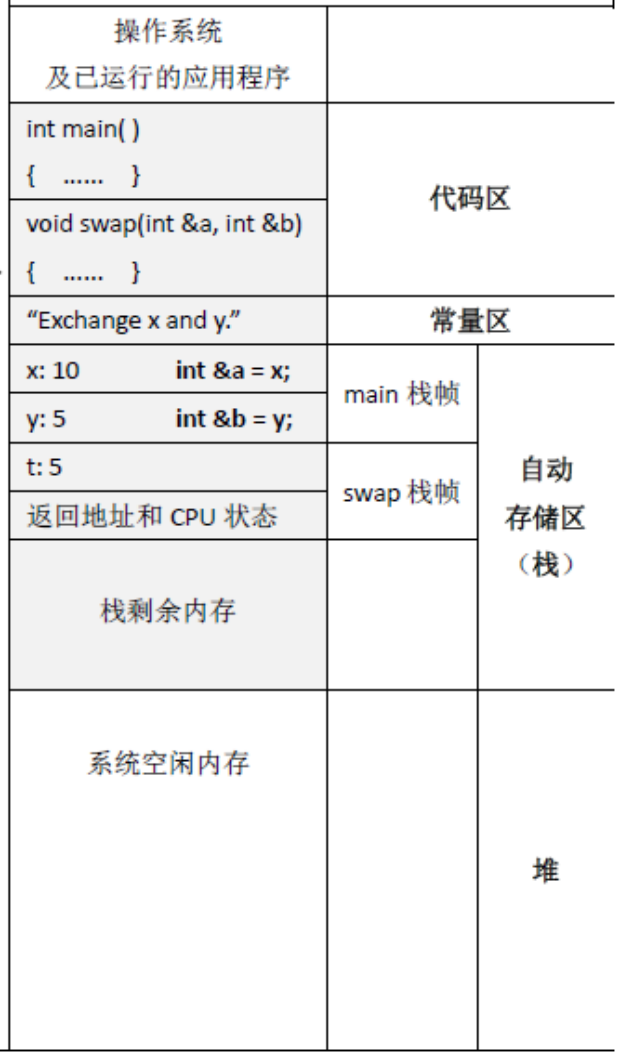
10, 5

例 5-15 一个交换变量值的 swap 函数（引用传递）

源程序

```
1  #include <iostream>
2  using namespace std;
3
4  // 引用传递：将形参定义成引用变量
5  void swap(int &a, int &b)
6  {
7      int t;
8      // 下列 3 条语句可交换 a 和 b 的值
9      t = a;   a = b;   b = t;
10 }
11
12 int main( )
13 {
14     cout << "Exchange x and y." << endl;
15     int x = 5, y = 10;
16     cout << x << "," << y << endl;
17
18     swap(x, y);   // 调用格式不变
19     cout << x << "," << y << endl;
20     return 0;
21 }
```

执行时的内存示意图



5.5 函数间参数传递的3种方式

- 引用传递的特点
  - 引用传递将被调函数的形参定义成主调函数中实参变量的引用，被调函数通过该引用间接访问主调函数中的变量。
  - 被调函数修改形参实际上修改的是对应的实参。换句话说，主调函数可以通过引用将数据传给被调函数，被调函数也可以通过该引用修改实参的值将数据传回主调函数。引用传递是一种双向数据传递机制。
  - 引用传递时，实参必须是变量。
  - 引用传递的好处：一是形参直接引用实参，不需分配内存、传递数值，这样可以提高函数调用速度，降低内存占用；二是可以通过引用实现双向数据传递。

5.5 函数间参数传递的3种方式

- 指针传递
  - **指针传递**就是传递实参变量的内存地址，被调函数通过该地址间接访问主调函数中的变量，从而达到传递数据的目的
  - 采用指针传递时，被调函数的形参需定义成指针变量，以接收实参变量的内存地址

5.5 函数间参数传递的3种方式

int *a = &x;
int *b = &y;

例 5-16 一个交换变量值的 swap 函数（指针传递）

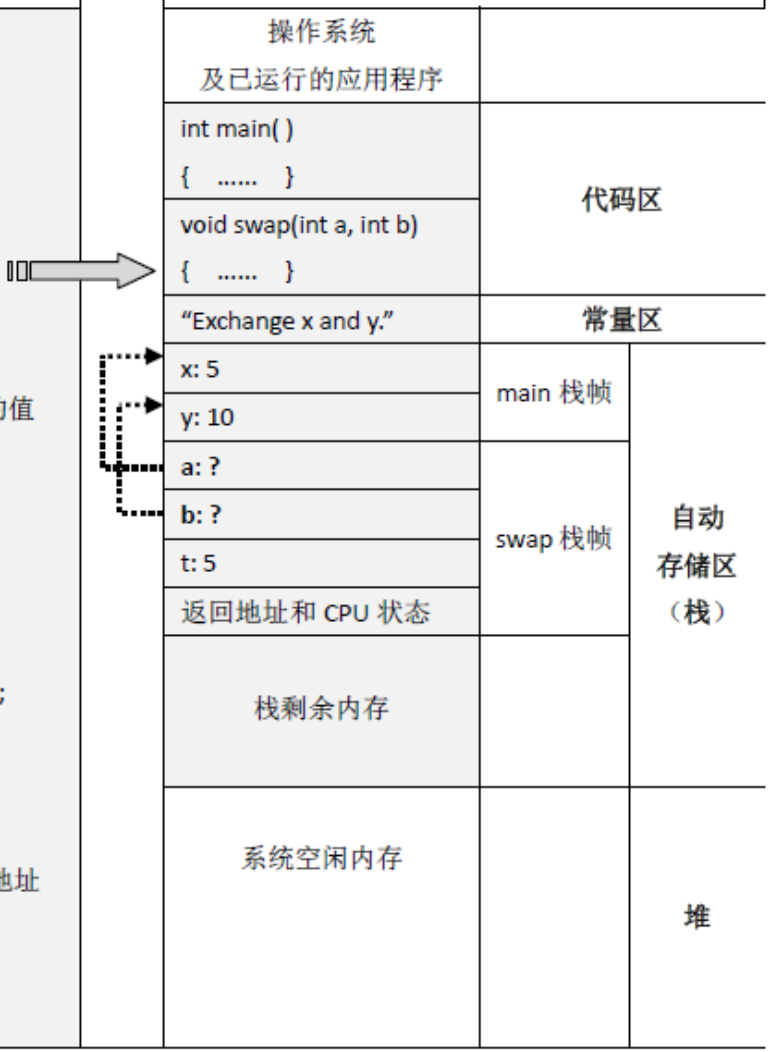
源程序

```
1  #include <iostream>
2  using namespace std;
3
4  // 指针传递：将形参定义成指针变量
5  void swap(int *a, int *b)
6  {
7      int t;
8      // 下列 3 条语句可交换*a 和*b 的值
9      t = *a;   *a = *b;   *b = t;
10 }
11
12 int main( )
13 {
14     cout << "Exchange x and y." << endl;
15     int x = 5, y = 10;
16     cout << x << "," << y << endl;
17
18     swap(&x, &y);   // 实参改为内存地址
19     cout << x << "," << y << endl;
20     return 0;
21 }
```

5, 10

10, 5

执行时的内存示意图



5.5 函数间参数传递的3种方式

- 指针传递的特点
  - 指针传递将主调函数中实参变量的内存地址传给被调函数的指针形参，被调函数通过该内存地址间接访问主调函数中的变量。
  - 被调函数可通过内存地址修改对应的实参。换句话说，主调函数可以通过内存地址将数据传给被调函数，被调函数也可以通过该地址修改实参的值将数据传回主调函数。指针传递也是一种双向数据传递机制。
  - 指针传递时，实参必须是指针变量或指针表达式。
  - 指针传递的好处：一是可以通过内存地址实现双向数据传递，二是在传递批量数据（例如数组）时只需要传递一个内存首地址，这样可以提高函数调用速度，降低内存占用。

5.6 在函数间传递数组

- 函数在函数体中定义的数组是局部变量，其它函数不能直接访问其中的数据，需要时可通过传递来实现对数组的共享

- 传递数组时，被调函数需定义数组形式的形参，同时还需传递表明数组大小的参数，例如数组的元素个数

5.6 在函数间传递数组

例5-17 在函数间传递一维数组的C++演示程序（**数组形式**的形参）

```
1  #include <iostream>
2  using namespace std;
3
4  int Max( int array[ ], int length )  // 求数组array中元素的最大值，length表示该数组的长度
5  {
6      int value = array[0], n;
7      for (n = 1; n < length; n++)
8      {
9          if (array[n] > value)    value = array[n];
10     }
11     return value;
12 }
13
14 int main( )
15 {
16     int a[5] = { 2, 8, 4, 6, 10 };  // 定义一维数组a，定义时初始化
17     cout << Max( a, 5 ) << endl;  // 调用函数Max求数组a中元素的最大值，显示结果：10
18     return 0;
19 }
```

5.6 在函数间传递数组

- 计算机内部对数组的管理和访问是通过指针（即内存地址）来实现的

- 在函数间传递数组，所传递的实际上是数组的首地址。换句话说，函数间数组传递采用的是指针传递，而不是值传递

5.6 在函数间传递数组

例5-18 在函数间传递一维数组的C++演示程序（**指针形式**的形参）

```
1  #include <iostream>
2  using namespace std;
3
4  int Max( int *pArray, int length )  // 求指针pArray所指向数组中元素的最大值
5  {
6      int value = pArray[0], n;
7      for (n = 1; n < length; n++)
8      {
9          if (pArray[n] > value)    value = pArray[n];  // pArray[n]与*(pArray+n)等价
10     }
11     return value;
12 }
13
14 int main( )
15 {
16     int a[5] = { 2, 8, 4, 6, 10 };  // 定义一维数组a，定义时初始化
17     cout << Max( a, 5 ) << endl;  // 调用函数Max求数组a中元素的最大值，显示结果：10
18     return 0;
19 }
```

5.6 在函数间传递数组

例5-18 在函数间传递二维数组的C++演示程序（数组形式的形参）

```
1  #include <iostream>
2  using namespace std;
3
4  int Max( int array[ ][3], int row ) // 求数组array中元素的最大值，row表示行数，3是列数
5  {
6      int value = array[0][0], m, n;
7      for (m = 0; m < row; m++)
8          for (n = 0; n < 3; n++)
9          {
10           if (array[m][n] > value)    value = array[m][n];
11         }
12     return value;
13 }
14
15 int main( ) // 主函数定义
16 {
17     int a[2][3] = { 2, 8, 4, 6, 10, 12 }; // 定义二维数组a，定义时初始化
18     cout << Max( a, 2 ) << endl; // 调用函数Max求数组a中元素的最大值，显示结果：12
19     return 0;
20 }
```

本章要点

- 函数是在被调用时才会执行，执行时会按需传递数据
- 深入计算机内部，了解程序执行时其代码和变量在内存中的存储原理，这样可以更容易理解变量存储类型、作用域等抽象的概念
- 要准确把握函数间传递数据的3种方式
- 要从2个不同的角度，即定义函数的程序员和调用函数的程序员，这样才能更容易地理解函数相关的各种语法知识