Ch-04

函 数

主要内容：

- 函数的定义
- 函数的调用
- 函数的嵌套调用
- 函数的递归调用
- 内置函数
- 函数重载
- 函数模板
- 有默认参数的函数
- 变量的作用域
- 变量的存储类别
- 变量的声明和定义
- 内部函数和外部函数
- 头文件

4.1 概述

- 函数是基本操作的实现，是C/C++程序的重要组成部分

  在C中，函数是程序的主体，程序功能的实现依赖于函数之间的相互调用；而在C++中，函数（除了主函数）都是被封装在类中，程序功能的实现依赖于类对象对函数的调用。

- 一个C/C++程序可以有多个源文件，每个源文件是一个编译单元

- 一个C/C++程序必须有且只能有一个名为main的主函数，程序的执行总是从main函数开始，在main中结束

□ 函数分类

- **从用户角度**
  - ➢ **系统函数（库函数）：由编译系统提供；**
  - ➢ **用户自定义函数。**
- **从函数形式**
  - ➢ **无参函数：主调函数与被调函数间没有参数的传递，一般用来指定一组操作，可以带回或不带回函数值，但一般不带回函数值。**
  - ➢ **有参函数：主调函数通过参数向被调函数传递数值，一般情况下，被调函数会给主调函数返回一个值供其使用。**

4.2 函数的定义

□ 无参函数的定义

```
类型标识符 函数名( )
{
    函数体
}
```

或

```
类型标识符 函数名(void)
{
    函数体
}
```

□ 有参函数的定义

```
类型标识符   函数名(形式参数表列)
{
        函数体
}
```

- 说明：类型标识符用于指定函数返回值的类型，若省略则默认为int型；若函数没有返回值，则类型标识符可以声明为void（空）类型。

4.3 函数的调用

4.3.1 函数调用的形式

函数名(实参表列)

- **说明：**
  - **实参表列：有确定值的数据或表达式；**
  - **实参与形参个数相等，类型一致，按顺序一一对应，多个实参用“,”分隔；**
  - **实参表列的求值顺序，由编译系统决定（VC++6.0 自右向左）；**
  - **调用无参函数时，实参表列为空，但( )不能省。**

例4.1 参数的求值顺序

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

int main()
{
    int f(int a, int b);
    int i=2, p;

    p = f(i, ++i);    //参数表列从右到左进行求值

    cout << "p = " << p << endl;

    return 0;
}

int f(int a, int b)
{
    int c;

    cout << "a = " << a << " " << "b = " << b << endl;

    if(a > b)  c = 1;
    else if(a == b) c = 0;
    else c = -1;

    return(c);
}
```

```
a = 3 b = 3
p = 0
```

若参数表列自左到右求值，则函数调用等于**f(2,3)**，因此程序运行结果等于 **-1**。

4.3.2 函数调用的使用方式

- **函数调用语句**：函数调用单独作为语句，不要求有返回值，仅完成一定的操作。

  例如：printstar();

- **表达式的操作数**：函数调用是表达式的一个操作数，用函数返回值参与运算。

  例如：m=max(a,b)*2;

- **函数参数**：函数调用作为另一个函数调用时的实际参数。

  例如：m=max(a, max(b,c)); /*三数比大小*/

4.3.3 函数参数与参数传递

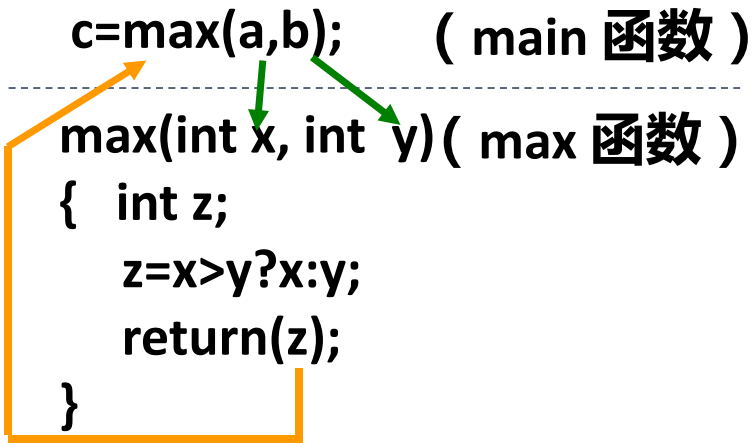
口 形参与实参

- **形式参数（形参）：定义函数时函数名后面括号中的变量。**
- **实际参数（实参）：调用函数时函数名后面括号中的常量、变量或表达式。**

说明：

- **实参必须有确定的值；形参必须指定类型，而且不能是常量和表达式。**
- **形参与实参个数相同，对应位置上的类型相同或赋值兼容；若类型不相同，自动按形参类型转换———函数调用转换。**
- **形参在函数被调用前（即函数定义时）不占内存；函数调用时为形参分配内存；调用结束，形参内存单元释放。**
- **实参对形参的参数传递只与位置有关，与参数名字无关。**

□ 函数参数传递



- 说明：
  - 实参对形参的参数传递只与位置有关，而与参数名字无关。
  - 参数传递根据实参的属性可以分为值传递和地址传递。

1、值传递

- **传递方式**

  函数调用时，为形参分配存储单元，并将实参的值传递（复制）给形参；调用结束时，形参存储单元被释放，实参单元仍保留并维持原值。

- **特点：**

  - **形参与实参具有相同的值，但使用不同的存储单元进行存储；**
  - **单向传递：实参向形参传递数值，形参被赋值后参与运算，当形参的值被改变后，不会改变实参的值。**

2、地址传递

- **方式**

  函数调用时，将实参存储单元的地址传递（复制）给形参。

- **特点：**

  - **形参与实参使用不同的存储单元，但具有相同的值指向同一个的内存存储单元；**
  - **“双向”传递：实参向形参传递其存储单元的地址，形参被赋值后参与运算，若形参指向的存储单元中的值被改变后，就会改变实参的值；**
  - **实参是地址常量，而形参必须是地址变量。**

4.3.4 函数的返回值

函数的返回值是通过函数中的return语句获得的。

□ return语句

- 使用形式

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

- 功能：使程序控制从被调函数返回到主调函数中，同时把返回值带给主调函数。

- **说明：**
  - **函数的返回值必须用 return 语句带回，并且return语句只能把一个返回值传递给主调函数。**
  - **函数中可有多个return语句，执行哪一个由程序实现逻辑决定。**

```
if(a>b)   return(a);
else      return(b);
```

  - **一般情况下，return 语句中的表达式类型要与定义的函数类型相同；若不同，则按函数类型进行类型自动转换。**
  - **若无return语句，函数执行完毕后自动返回主调函数，此时的返回值无意义 。**
  - **若函数无返回值，在定义时应该用void类型进行声明（如void printStar(); ），则其他函数中使用该函数的返回值参与运算则产生编译错误。**

4.3.5 对被调用函数的声明和函数原型

口 对被调用函数要求：

- **必须是已存在的函数；**
- **库函数: #include <库名>**
- **用户自定义函数：如果被调函数的定义出现在主调函数之后，则必须在主调函数中对被调函数进行声明。**

□ 函数原型

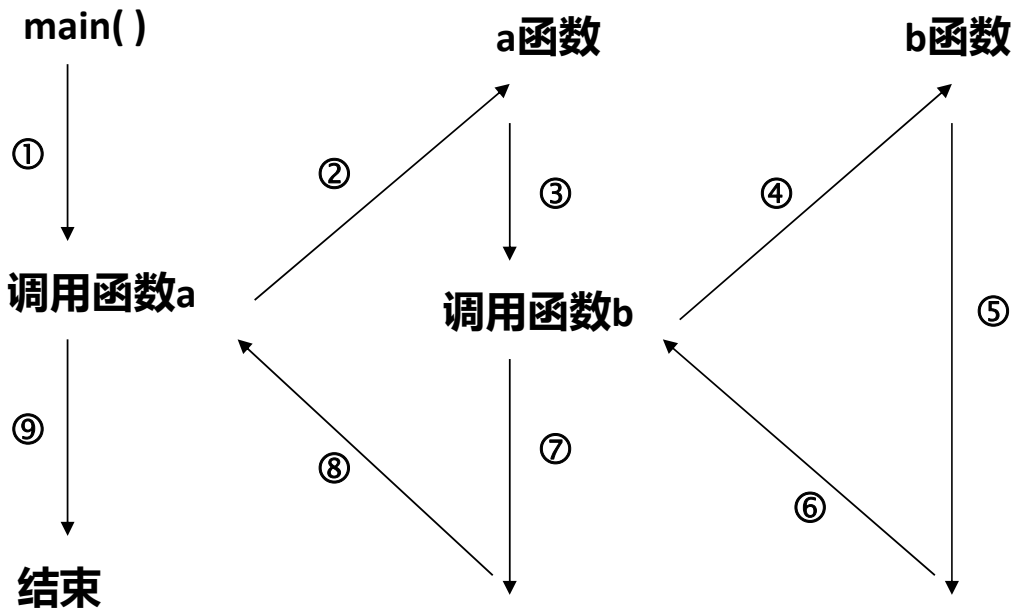
- **一般形式：**

```
函数类型  函数名(参数类型1  参数名1, 参数类型2  参数名2, … );
函数类型  函数名(参数类型1, 参数类型2, … );
```

- **说明：**
  - **作用：编译系统在编译阶段对调用函数的合法性（函数类型、参数个数及类型）行检查；**
  - **C/C++使用函数原型进行函数声明。**
  - **函数定义与函数声明不同：声明只包含函数定义的首部，而且声明可以只写参数类型，不写参数名。**
  - **函数声明的位置：必须置于函数调用位置之前，但可以在函数体内或者在所有函数定义之外，通常放在函数体或源文件的最前面；如果被调用数的定义放置在主调函数定义之前，则在主调函数体中不需要进行被调函数声明。**

4.4 函数的嵌套调用

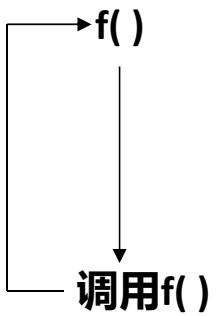
C/C++规定：函数定义不可嵌套，但函数调用可以嵌套。



4.5 函数的递归调用

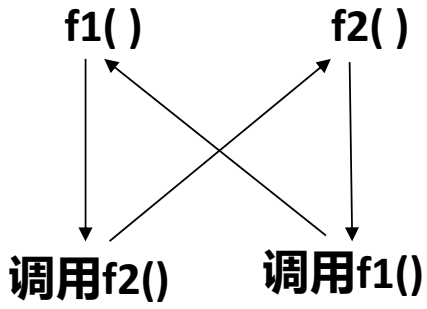
- 定义：在函数调用过程中，直接或间接的调用自身。
- 递归调用的两种形式：
  - 直接递归调用：在函数体内又调用自身；
  - 间接递归调用：函数1调用函数2，而函数2执行过程中又调用函数1。

```
int  f(int x)
{    int y, z;
     …
     z = f(y);
     …
     return(2*z);
}
```




```
int  f1(int x)
{    int y,z;
      ……
     z=f2(y);
     …….
     return(2*z);
}
```

```
int  f2(int t)
{    int a,c;
      ……
     c=f1(a);
     …….
     return(3+c);
}
```



□ 说明

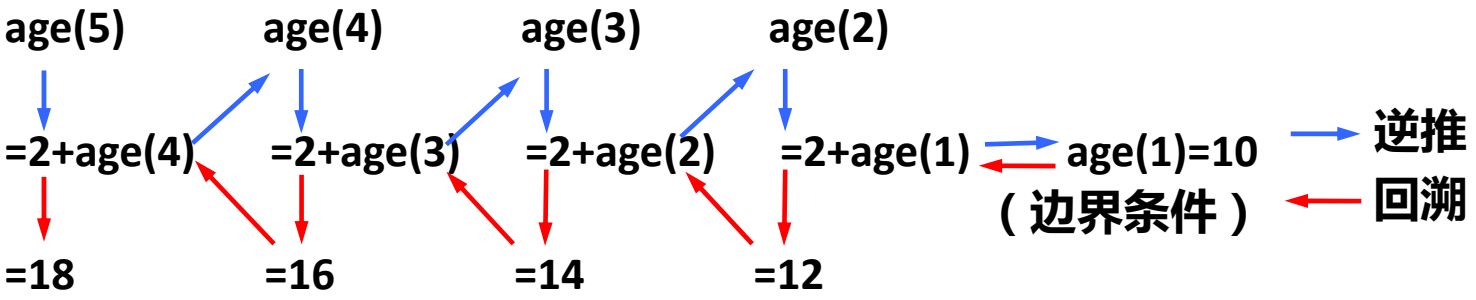
- **递归包含递推（逆推）和回溯两个子过程**

  逆推是将大规模问题不断分解成规模更小的相同问题，直到获得边界条件为止；
  回溯是利用边界条件沿着逆推的逆过程，不断求解规模更大的相同问题，从而求解最终问题。

- **C/C++编译系统对递归函数的递归次数没有限制;**
- **每次调用函数，都要在内存堆栈区分配新空间，用于存放函数变量、返回值等信息，因此若递归次数过多，可能引起堆栈溢出；**
- **必须有能够终止递归的边界条件。**

例4.2 递归

有5个人，第5个人比第4个人大2岁，第4个人比第3个人大2岁，……，第2个人比第1个人大2岁，第1个人10岁，问第5个人多大？



递归实现

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

int age(int n)
{
    int  c;

    if(n==1) c = 10;
    else c = 2+age(n-1);

    return(c);
}

int main()
{
    cout << "age(5) = " << age(5) << endl;

    return 0;
}
```

```
age(5) = 18
```

递推实现

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

int main()
{
    int i, c=10;

    for(i=0; i<4; i++) {
        c += 2;
    }

    cout << "age(5) = " << c << endl;

    return 0;
}
```

*4.6 内联函数

□ 引入内联函数的目的

为了解决程序中一些函数体代码不是很大，但又频繁地被调用的函数的函数调用的效率问题。

□ 解决方法

以目标代码的增加为代价来换取时间上的节省；即在编译时将函数体中的代码替换到程序中，增加目标程序代码量，进而增加空间开销，从而在时间开销上不像函数调用时那么大。

口 内联函数的定义方式

在函数定义的前面加上关键字inline

例如：inline int add(int x,int y,int z) { return x+y+z; }

- 说明：

1、C++中，除在函数体内含有循环，switch分支和复杂嵌套的if语句外，所有的函数均可定义为内联函数。

2、内联函数也要定义在前，调用在后。形参与实参之间的关系与一般的函数相同。

3、对于用户指定的内联函数，编译器是否作为内联函数来处理由编译器自行决定。说明内联函数时，只是请求编译器当出现这种函数调用时，作为内联函数的扩展来实现，而不是命令编译器要这样去做。

4、可以同时在声明和定义时加inline，也可以只在声明时加inline。

*4.7 函数的重载

- 定义

  同一个函数名可以对应着多个函数实现

- 要求：编译器能够唯一确定调用函数时应执行哪个函数代码

- 重载条件：（1）参数个数不同；（2）参数类型不同。

  但是函数返回值类型不能作为重载的条件。

例4.3.1 重载函数—参数类型不同

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

//两个add函数的参数类型不同
int add(int, int);
double add(double, double);

int main()
{
    cout << add(5,10) << endl;
    cout << add(5.2, 10.3) << endl;

    return 0;
}

int add(int x, int y)
{
    cout << "int add(int, int)" << endl;
    return x+y;
}

double add(double x, double y)
{
    cout << "double add(double, double)" << endl;
    return x+y;
}
```

```
int add(int, int)
15
double add(double, double)
15.5
```

例4.3.2 重载函数—参数个数不同

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

int min(int a, int b);
int min(int a, int b, int c);
int min(int a, int b, int c, int d);

int main()
{
    cout<<min(13, 5, 4, 9)<<endl;
    cout<<min(-2, 8, 0)<<endl;

    return 0;
}
```

```cpp

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

int min(int a, int b, int c)
{
    int t = min(a, b);
    return min(t, c);
}

int min(int a, int b, int c, int d)
{
    int t1 = min(a, b);
    int t2 = min(c, d);
    return min(t1, t2);
}
```

*4.8 函数模板

函数模板给出了一种通用的函数描述。

利用函数模板可以创建一个通用函数，在这个函数中用到的数据类型可以不具体指定，而是用一个虚拟类型来代替，当发生函数调用时在根据传入的实参来逆推出真正的类型。

□ 定义的一般形式

template <typename T> 函数定义

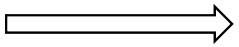
template <class T> 函数定义

- class与typename都是表示“类型名”，二者可互换。
- 虚拟类型可以用在函数定义的各个位置，包括返回值、形参列表和函数体。
- 类型参数可以有多个，可根据需要确定个数，例如template <class T1, class T 2>
- 只适用于函数物参数个数相同而类型不同，且函数体相同的情况，如参数个数不同，则不能用函数模板。

例4.4 函数模板的使用

函数重载

```
35  //交换 int 变量的值
36  void Swap(int *a, int *b){
37      int temp = *a;
38      *a = *b;
39      *b = temp;
40  }
41  //交换 float 变量的值
42  void Swap(float *a, float *b){
43      float temp = *a;
44      *a = *b;
45      *b = temp;
46  }
47  //交换 char 变量的值
48  void Swap(char *a, char *b){
49      char temp = *a;
50      *a = *b;
51      *b = temp;
52  }
53  //交换 bool 变量的值
54  void Swap(bool *a, bool *b){
55      char temp = *a;
56      *a = *b;
57      *b = temp;
58  }
```



函数模板

```
1   #include <iostream>
2   using namespace std;
3
4   template <typename T>
5   void Swap(T *a, T *b)
6   {
7       T temp = *a;
8       *a = *b;
9       *b = temp;
10  }
11  int main()
12  {
13      //交换int变量的值
14      int n1 = 100, n2 = 200;
15      Swap(&n1, &n2);
16      cout<<n1<<", "<<n2<<endl;
17
18      //交换float变量的值
19      float f1 = 12.5, f2 = 56.93;
20      Swap(&f1, &f2);
21      cout<<f1<<", "<<f2<<endl;
22
23      //交换char变量的值
24      char c1 = 'A', c2 = 'B';
25      Swap(&c1, &c2);
26      cout<<c1<<", "<<c2<<endl;
27
28      //交换bool变量的值
29      bool b1 = false, b2 = true;
30      Swap(&b1, &b2);
31      cout<<b1<<", "<<b2<<endl;
32      return 0;
33  }
```

*4.9 设置函数参数的默认值

- C++中，在函数声明或定义时可以为一个或多个参数指定默认值

```
int add(int x, int y=10);
```

- 进行函数调用时，若未指定足够的实参，则编译器将按从左向右的顺序用函数声明或定义中的默认值来补足所缺少的实参

```
add(15);   等价于   add(15, 10);
```

  - 因此在一个指定了默认值的参数的右边，不能出现没有指定默认值的参数，例如 void f(int x,int y=1,int z);是错误的，因为f(2, 4)等价于f(x=2, y=4, z)，z未赋值。

- 在给某个参数指定默认值时，既可以是一个数值，也可以是任意复杂的表达式

```
int f();
……
void delay(int k, int time=f());
```

例4.5 分析下列程序的输出结果

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

void fun(int a=1, int b=3, int c=5)
{
    cout << "a = " << a << ", "
         << "b = " << b <<
         ", "
         << "c = " << c << endl;
}

int main()
{
    fun();
    fun(7);
    fun(7,9);
    fun(7,9,11);
    cout << "OK! " << endl;
    return 0;
}
```

```
a = 1, b = 3, c = 5
a = 7, b = 3, c = 5
a = 7, b = 9, c = 5
a = 7, b = 9, c = 11
OK!
```

4.10 变量的作用域

变量的作用域就是变量的有效范围。

根据作用域进行分类，可将变量分为局部变量和外部变量两种。

□ 局部变量

- 定义：在一个函数（或复合语句）内部定义的变量。
- 作用域：本函数（或复合语句）内部。
- 说明：
  - 不同函数（或复合语句）中的同名变量属于不同对象，占用不同的内存单元。
  - 形参属于局部变量。

举例：局部变量的作用域

```
float  f1(int a)
{  int m,n;
   .......
}

char f2(int x,int y)
{   int i,j;
   ......
}

main()
{  int m,n;
   .......
}
```

a,m,n有效

x,y,i,j有效

m,n有效

注意：main()中定义的m、n与f1()中定义的m、n是不同的变量。

```
main()
{  int a, b;
    ┆
   {
     int c;
     c=a+b;
     ┆
   }
   ┆
}
```

c有效

a,b有效

□ 全局变量

- 定义：在所有函数外部定义的变量。
- 作用域：从定义的位置开始到本源文件结束，以及包含该变量的extern声明的其它所有源文件。

```
int  p=1,q=5;
float  f1()
{  int b,c;
    .......
}
char c1,c2;
main()
{  int m,n;
   .......
}
```

p,q的作用范围

c1,c2的作用范围

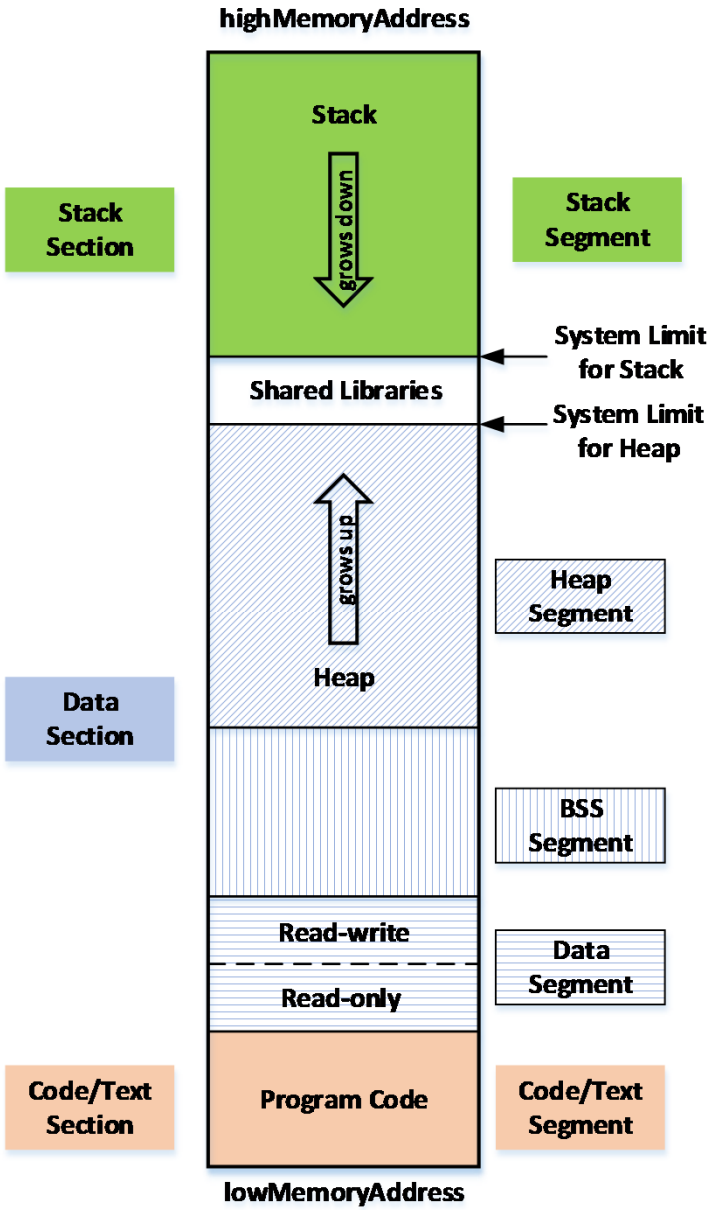
口 说明：

- **若全局变量与局部变量同名，则在局部变量的有效范围内将屏蔽外部变量；**
- **全局变量的使用增加了函数之间传递数据的途径。若一个函数需要返回两个或两个以上的计算结果时，可以使用全局变量传递数据。**
- **设置全局变量是双刃剑，建议非必要时不要使用全局变量，原因：**
  - **全局变量在程序全部执行过程中占用存储单元，而不是需要时再开辟；**
  - **降低了函数的通用性、可靠性，可移植性；**
  - **若使用过多，很难判断出全局变量的当前值， 因此降低程序清晰性，容易出错。**

4.11 变量的存储类别

C/C++中，变量有2种存储方式：静态存储和动态存储。不同存储方式的变量具有不同的生存期。

- 静态存储指在程序编译时由系统分配存储空间，当程序运行完毕时由系统释放，在整个程序运行时间内占用固定的存储空间；
- 动态存储是在程序运行过程中根据需要（由系统或程序员）动态地分配和释放（临时）存储空间。


highMemoryAddress
Stack
grows down
Stack Section
Stack Segment
System Limit for Stack
Shared Libraries
System Limit for Heap
grows up
Heap
Heap Segment
Data Section
BSS Segment
Read-write
Read-only
Data Segment
Code/Text Section
Program Code
Code/Text Segment
lowMemoryAddress

Area	Description
Code/text segment	Often referred to as the text segment, this is the area in which the executable instructions reside. For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible. Only one copy of the instructions for the same program resides in memory at any time. The portion of the executable file containing the text segment is the text section.
Initialized data – data segment	Statically allocated and global data that are initialized with nonzero values live in the data segment. Each process running the same program has its own data segment. The portion of the executable file containing the data segment is the data section.
Uninitialized data – bss segment	BSS stands for '**Block Started by Symbol**'. Global and statically allocated data that initialized to zero by default are kept in what is called the BSS area of the process. Each process running the same program has its own BSS area. When running, the BSS data are placed in the data segment. In the executable file, they are stored in the BSS section. For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable's disk file.
Heap	The heap is where dynamic memory (obtained by malloc(), calloc(), realloc() and new – C++) comes from. Everything on a heap is **anonymous**, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process's address space grows. Although it is possible to give memory back to the system and shrink a process's address space, this is almost never done because it will be allocated to other process again. Freed memory (free() and delete) goes back to the heap, creating what is called **holes**. It is typical for the heap to grow upward. This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items. It is also typical for the heap to start immediately after the BSS area of the data segment. The end of the heap is marked by a pointer known as the **break**. You cannot reference past the break. You can, however, move the break pointer (via brk() and sbrk() system calls) to a new position to increase the amount of heap memory available.
Stack	The stack segment is where local (automatic) variables are allocated. In C program, local variables are all variables declared inside the opening left curly brace of a function body including the main() or other left curly brace that aren't defined as static. The data is popped up or pushed into the stack following the **Last In First Out** (LIFO) rule. The stack holds local variables, temporary information/data, function parameters, return address and the like. When a function is called, a stack frame (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (return address), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler. When a function returns, the stack frame is POPped from the stack. Typically the stack grows downward, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap.

源：http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1c.html

□ 变量的存储类别

自动的	寄存器的	静态的	外部的
auto	register	static	extern
动态存储		静态存储	

- **说明：**
  - **auto、 register和static只能用于修饰变量定义，而不能独立定义变量。**
  - **带存储类别的变量定义格式:**

[存储类别] 数据类型 变量名;

4.11.2 auto变量

- 局部变量的默认存储类型，可缺省；
- 保存在动态存储区，如果没有初始化，则初值不确定；
- 作用域：函数内部或复合语句内部；
- 生存期：函数调用开始到函数调用结束，或复合语句中变量定义开始到复合语句结束。

4.11.3 static变量

- static变量存放在内存的静态存储区，在整个程序运行期间占用固定的内存单元。
- 系统在编译时为static变量分配空间并赋初值，因此只能赋一次初值且只能在定义时进行。
  - 对未赋值的局部static数值型变量，系统自动给它赋值为0；
  - 对未赋值的局部static字符型变量，自动赋值为空字符('\0')。
- 由于变量占用的存储单元不消失，再次调用static局部变量时，static局部变量的值为上次调用结束时的值。
- 注意：static局部变量的生存期是整个程序运行期间，但作用域仍然是定义该变量的函数体（或复合语句）内部。

  即静态局部变量在函数调用结束后仍然存在，但其他函数不能引用它。
- 若用static声明外部变量，则该外部变量称为“静态外部变量”，静态外部变量的作用域限定在本文件中。

例4.6 局部静态变量值具有可继承性

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

int main()
{
    void increment(void);

    increment();
    increment();
    increment();

    return 0;
}

void increment(void)
{
    int x = 0;
    x++;
    cout << "x = " << x << endl;
}
```

```
x = 1
x = 1
x = 1
```

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

int main()
{
    void increment(void);

    increment();
    increment();
    increment();

    return 0;
}

void increment(void)
{
    static int x = 0;
    x++;
    cout << "x = " << x << endl;
}
```

```
x = 1
x = 2
x = 3
```

4.11.4 register变量

- 只有auto变量与形式参数可以作为register变量；
- register变量和auto变量不同之处在于register变量存放在CPU的寄存器中，因此比auto变量存取速度快。通常将频繁使用的变量放在寄存器中，以提高程序执行速度；
- 不能对寄存器变量使用取地址运算符&；
- 计算机中寄存器的数量是有限的，而且寄存器的数据长度也是有限的。因此register变量不能定义太多，也不能是数据类型太大的变量(long、float、double型)；
- 目前register变量定义通常是不必要的，优化的编译系统能够识别使用频繁的变量，并将其放到寄存器中。

4.11.5 extern外部变量

用extern声明外部变量，以扩展外部变量的作用域。

- 在一个文件内扩展外部变量的作用域

外部变量没在源文件开头定义，其作用域为定义处到文件结束。若处于定义位置之前的函数要使用该外部变量，则需要在引用前用extern作“外部变量声明”。

例4.7

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

int max(int x, int y)
{
    int z;
    z = x>y?x:y;
    return(z);
}

int main()
{
    extern int A, B;     //(1) extern A, B;
                         //(2)全局变量声明，扩展全局变量的作用域
    cout << max(A,B) << endl;
    return 0;
}

int A = 13, B = -8; //全局变量定义
```

- **将外部变量的作用域扩展到其他文件**

如果一个程序由多个文件组成，且多个文件要使用同一个外部变量，此时，应该在任一文件中进行外部变量定义，而在其它文件中用extern进行外部变量声明。

说明：静态外部变量不能使用extern扩展作用域，否则将在链接时产生错误。

例4.8

```
/*file1.c*/
#include <iostream>
using namespace std;

static int A;
```

ERROR

```
C:\Users\vc\Desktop\C++\程序\CH04\4.8.2.o    4.8.2.cpp:(.rdata$.refptr.A[.refptr.A]+0x0): undefined reference to `A'
```

```
{
    int power(int);
    int m, d;

    cout << "Enter the number a and its power m: " << endl;
    cin >> A >> m;

    d = power(m);
    cout << A << " ^ " << m << " = " << d << endl;

    return 0;
}
```

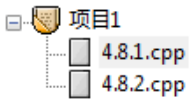
```
/*file2.c*/

extern int A;

int power(int n)
{
    int i, y=1;

    for(i=1; i<=n; i++)
        y *= A;

    return(y);
}
```




```
Enter the number a and its power m:
2 3
2 ^ 3 = 8
```

□ 变量存储类别小结

	局部变量			全局变量	
存储类别	auto（默认）	register	局部static	全局static	extern
存储方式	动态存储		静态存储		
存储区	动态存储区	寄存器	静态存储区		
生存期	变量定义所在的函数体或复合语句内			本文件	其他文件
初始化	每次函数调用时		编译时初始化，只初始化一次		
未初始化	变量值不确定		系统自动初始化0或'\0'		

- 局部变量默认为auto型；
- register型变量个数受限，且不能为long, double, float型；
- 局部static变量具有全局生存性和局部有效性；
- 局部static变量具有可继承性；
- extern不是变量定义，可扩展全局变量作用域。

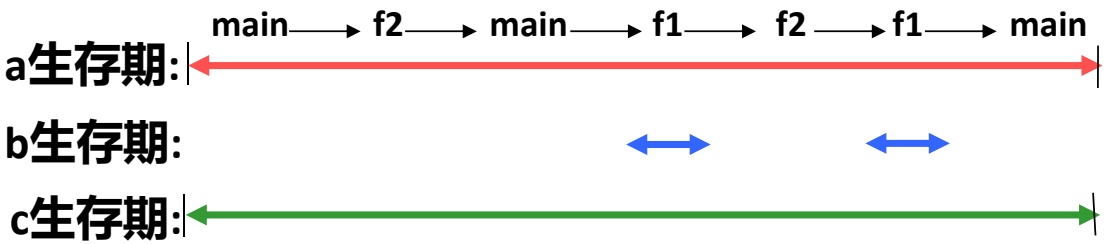
例如：

```
int a;
main( )
{   .....
    .....
      f2;
    .....
      f1;
    .....
}
f1( )
{   auto int b;
    .....
     f2;
    .....
}
f2( )
{   static int c;
    .....
}
```

a作用域

b作用域

c作用域



4.12 关于变量的声明和定义

- 广义地讲，变量的声明有两种情况：
  - 定义性声明：建立存储空间，如int a;
  - 引用性声明：不建立存储空间，如extern A;
- 约定:
  - 建立存储空间的声明称“定义”；
  - 不建立存储空间的声明称“声明”。

说明：广义上声明包括定义，但并非所有的声明都是定义。

如：int A; 既包含声明又包含定义；

extern A; 只是声明，而无定义。

4.13 内部函数和外部函数

根据函数能否被其它源文件调用，将函数分为内部函数和外部函数。

□ 内部函数

只能被本文件中其它函数所调用。

- 定义形式：

```
static 类型标识符  函数名([形参列表])
{ 函数体 }
```

- 说明：内部函数的作用域仅限于函数定义所在的文件。在其它的文件中可以有相同的函数名，它们相互之间互不干扰。

口 外部函数（默认类型）

可以被其它文件中的函数所调用

- 定义形式：

```
extern 类型标识符 函数名([形参列表])
{ 函数体 }
```

- 说明：
  - 外部函数的定义和声明都可以省略extern；
  - 调用此函数的文件中要对该函数进行extern声明。

例4.9 有一个字符串，内有若干个字符，今输入一个字符，程序将字符串中该字符删去，要求用外部函数实现。

```cpp
/*File1.c*/
#include <iostream>
using namespace std;

int main()
{
    extern  enter_string(char str[80]);
    extern delete_string(char str[],char ch );
    extern print_string(char str[] );

    char c;
    char str[80];

    cout << "Input a string: " << endl;
    enter_string(str);
    cout << "Choose a letter to delete: " << endl;
    cin >> c;
    delete_string(str, c);
    cout << "New string: " << endl;
    print_string(str);

    return 0;
}
```

```cpp
/*File2.c*/
#include  <cstdio>
using namespace std;

void enter_string(char str[80])
{
    gets(str);
}
```

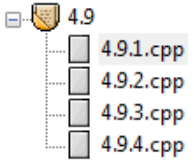
```cpp
/*File4.c*/
#include <iostream>
using namespace std;

void print_string(char str[])
{
    cout << str << endl;
}
```

```cpp
/*File3.c*/
void delete_string(char str[], char ch)
{
    int i, j;

    for(i=j=0; str[i]!='\0'; i++)
        if(str[i] != ch)
            str[j++] = str[i];

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




```
Input a string:
I am a teacher
Choose a letter to delete:
a
New string:
I m  techer
```

4.14 头文件

	非模板类型(none-template)	模板类型(template)
头文件(.h)	• 全局变量申明（带extern限定符） • 全局函数的申明 • 带inline限定符的全局函数的定义	• 带inline限定符的全局模板函数的申明和定义
	• 类的定义 • 类函数成员和数据成员的申明（在类内部） • 类定义内的函数定义（相当于inline） • 带static const限定符的数据成员在**类内部**的初始化 • 带inline限定符的类定义外的函数定义	• 模板类的定义 • 模板类成员的申明和定义（定义可以放在类内或者类外，类外不需要写inline）
实现文件(.cpp)	• 全局变量的定义（及初始化） • 全局函数的定义	(无)
	• 类函数成员的定义 • 类带static限定符的数据成员的初始化	

*申明：declaration
*定义：definition

源：http://www.cnblogs.com/ider/archive/2011/06/30/what_is_in_cpp_header_and_implementation_file.html

□ 标准头文件格式

头文件的所有内容，都应该包含在

```
#ifndef  Filename
#define  Filename
//Contents of head file
#endif
```

这样才能保证该头文件被多个其他文件引用(include)时，内部的数据不会出现重复定义的错误

例如

myheader.h

```
1  #ifndef MYHEADER_H
2  #define MYHEADER_H
3
4  //头文件内容
5
6  #endif
```

口 引用头文件的三种方式

- #include <headerName.h>

  C标准的头文件引用，从标准库目录开始查找该头文件，例如#include <math.h>.

- #include <headerName>

  C++标准的头文件引用，从标准库目录开始查找该头文件，例如#include <string> 。为了兼容C程序，因此C++中保留了C的标准头文件，但是要以c开头，且结尾没有.h后缀，例如#include <cmath>。

- #include “myHeaderName.h”

  自定义头文件的引用方式，从当前目录开始查找该头文件，若找不到再去标准库目录查找。

CH04 End