

## § 4. 利用函数实现指定的功能

### 4.0. 为什么要引入函数

★ 目前为止的所讲的内容及作业都是只有一个main函数，负责完成一个程序的所有功能

例：输入两数求max

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,m;
    cin >> a >> b;
    m = a > b ? a : b;
    cout << "max=" << m;
    return 0;
}
```

**//例1.3**

```
#include <iostream>
using namespace std;
int max(int x, int y)
{
    int z;
    if (x>y) z=x;
    else z=y;
    return (z);
}
int main()
{
    int a,b,m;
    cin >> a >> b;
    m=max(a,b);
    cout << "max=" << m;
    return 0;
}
```

例：人民币转大写

- ① 输入一个浮点数
- ② 分解各位
- ③ 每位依次转为汉字  
(大量重复，仅小部分不同)

```
switch(shiyi) {
    case 9:
        cout << "玖拾";
        break;
    ...
    case 1:
        cout << "壹拾";
        break;
    case 0:
        break;
}
switch(yi) {
    case 9:
        cout << "玖亿";
        break;
    ...
    case 1:
        cout << "壹亿";
        break;
    case 0:
        if (shiyi>0)
            cout << "亿";
        break;
}
```

```
void daxie(int num, int flag)
{
    switch(num) {
        case 0:
            if (flag)
                cout << "零";
            break;
        case 1:
            cout << "壹";
            break;
        ...
        case 9:
            cout << "玖";
            break;
    }
}
```

**在main函数中:**

```
daxie(shiyi, **);
... //可能需要的其它语句
daxie(yi, **);
... //可能需要的其它语句
```

- 能否不同功能分开，使程序逻辑更明确？
- 重复的代码能否只写一遍？  
(如何体现差别部分)

## § 4. 利用函数实现指定的功能

### 4. 1. 概述

- ★ C/C++程序的基本组成单位
- ★ 一个函数实现一个特定的功能
- ★ 有且仅有一个main函数，程序执行从main开始
- ★ 函数平行定义，嵌套调用
- ★ 一个源程序文件由多个函数组成，  
一个程序可由多个源程序文件组成

a1. cpp	a2. cpp	a3. cpp
fun1( )	fun5( )	fun4( )
{	{	{
}	}	}
fun2( )		main( )
{		{
}		}
fun6( )		
{		
}		

一个程序由3个源程序文件组成  
共6个函数，有且仅有一个main

### ★ 函数的分类

用户使用角度 {	标准函数（库函数）	由系统提供
	自定义函数	用户自己编写

- 在使用上无任何的区别

函数形式 {	无参	调用与被调用函数间无数据传递
	有参	调用与被调用函数间有数据传递

## § 4. 利用函数实现指定的功能

### 4.2. 函数的定义

#### 4.2.1. 无参函数的定义

函数返回类型 函数名 ([void])

{

函数体 {  
声明语句  
执行语句

}

{ ()  
(void)

```
int fun(void) long fun2()  
{  
    ...  
}
```

```
int fun()  
{  
    cout << "***" << endl;  
    return 0;  
}  
int fun(void)  
{  
    cout << "***" << endl;  
    return 0;  
}
```

★ 函数名的命名规则同变量

★ 返回类型与数据类型相同

★ 返回类型可以是void, 表示不需要返回类型

★ C的缺省返回类型为int, C++不支持默认int, 必写

★ ANSI C++要求main函数的返回值只能是int并且不能缺省不写, 否则编译会报错;

但部分编译器可缺省不写; VS系列还允许void等其它类型(建议唯一int)

```
void fun3()  
{  
}
```

```
fun3(...)  
{  
} //C++编译报错
```

```
main()  
{  
    return 0;  
}
```

//VS2017报错  
//其余三编译器正确

```
void main()  
{  
    return;  
}
```

//VS2017正确  
//其余三编译器报错

```
long main()  
{  
    return 0L;  
}
```

//VS2017正确  
//其余三编译器报错

## § 4. 利用函数实现指定的功能

### 4.2. 函数的定义

#### 4.2.2. 有参函数的定义

函数返回类型 函数名 (形式参数表)

```
{  
    函数体 {  
        声明语句  
        执行语句  
    }  
}
```

```
int max(int x, int y)  
{  
    int z;          /* 声明语句 */  
    if (x>y)  
        z=x;  
    else  
        z=y;  
    return z;  
}
```

- ★ 函数名的命名规则同变量
- ★ 返回类型与数据类型相同
- ★ 返回类型可以是void, 表示不需要返回类型
- ★ 缺省的返回类型是int(不建议缺省, int也写)

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

形式参数：在被调用函数中出现的参数

实际参数：在调用函数中出现的参数

<pre>int main() {     int i=15, j=10, m;     m = max(i, j);     cout&lt;&lt; "max=" &lt;&lt;m;     return 0; }</pre> <p><b>i, j为实参</b></p>	<pre>int max(int x, int y) {     int z;     z = x&gt;y ? x : y;     return z; }</pre> <p><b>x, y为形参</b></p>	<pre>int main() {     int x=15, y=10, m;     m = max(x, y);     cout&lt;&lt; "max=" &lt;&lt;m;     return 0; }</pre> <p><b>x, y为实参</b></p>	<pre>int max(int x, int y) {     int z;     z = x&gt;y ? x : y;     return z; }</pre> <p><b>x, y为形参</b></p>
--	---	--	---

★ 实参与形参分别占用不同的内存空间，实形参名称既可以相同，也可以不同

★ 参数的传递方式是“单向传值”，即将实参的值复制一份到形参中（**理解为 形参=实参 的形式**）

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

- ★ 实参与形参分别占用不同的内存空间
- ★ 参数的传递方式是“单向传值”，即将实参的值复制一份到形参中 (理解为 形参=实参 的形式)
- ★ 执行后，形参的变化不影响实参值

- ★ 实参可以是常量、变量、表达式，形参只能是变量

```
int main()
{
    int k=10;
    fun(2+k*3);
    return 0;
}
```

fun(int x)

x = 2+k\*3

```
#include <iostream>
using namespace std;
void fun(int x)
{
    cout << "x1=" << x << endl;
    x=5;
    cout << "x2=" << x << endl;
}
int main( )
{
    int k=15;
    cout << "k1=" << k << endl;
    fun(k);
    cout << "k2=" << k << endl;
    return 0;
}
```

k1=15  
x1=15  
x2=5  
k2=15

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

★ 形参在使用时分配空间，函数运行结束后释放空间

```
int main()      void f1(int x)      void f2(int y)
{  f1(10);      {  ...      {  ...
    f2(15);      }
    ...
}
```

x和y可能共用4个字节的空间

```
int main()      void f1(int x)
{  ...          {  ...
    f1(..);      }
    ...
    f1(..);
    ...
    f1(..);
    ...
}
```

1、假设main中调用10000次f1()，则x的分配释放会重复10000次  
2、每次x分配的4字节不保证是同一个空间

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

★ 实参、形参类型必须一致，否则结果可能不正确

```
#include <iostream>
using namespace std;
int fun(short x)
{
    cout << "x=" << x << endl;      x=4464
    return 0;
}
int main()
{
    long k=70000;
    fun(k); //编译有警告
    cout << "k=" << k << endl;      k=70000
    return 0;
}
```

实形参类型不一致时，  
转换规则同第2章



## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

#### 4.3.2. 函数的值（函数的返回值）

★ 通过return语句获得，若return类型与返回类型定义不一致，以返回类型为准进行数据转换

```
... f(...)
{
    int k;
    ...
    k = fun(...);
    ...
}

int fun(...)
{
    int s;
    ...
    return s;
}
```

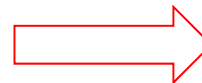
若s=10  
则k=10

理解为  
调用函数中值=return后值的形式

```
long fun2()
{
    long a;
    ...
    return a;
} 正确
```

```
long fun2()
{
    short a;
    ...
    return a;
} 正确
```

```
short fun3()
{
    long a;
    ...
    return a;
} 可能不正确
```



//问1: 运行结果（d的值是多少？）

//问2: 哪句会有warning错？

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

short fun3()
{
    long a = 70000;
    return a;
}

int main()
{
    long d;
    d = fun3();
    cout << d << endl;

    return 0;
}
```

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

#### 4.3.2. 函数的值（函数的返回值）

★ return后可以是变量、常量、表达式，有两种形式（带括号、不带括号）

```
return a;          return k*2;
return (a);        return (k*2);
```

★ 若函数不要求有返回值，则指定返回类型为void

<pre>void fun1() {     ... <del>return;</del> }</pre>	<pre>int main() {     ...     return 0; }</pre>	<pre>int fun() {     ... <del>return 0;</del> }</pre>
---	---	---

无return语句  
空return语句

return int 型

return int型

返回类型非void的函数，如果不带return语句，不同编译器表现不同(error/warning/不报错)  
VS2017: main无return不报错，其余函数报error

=> 推论：① 返回类型为void的函数不能出现在除逗号表达式外的任何表达式中  
② 若逗号表达式要参与其它运算，则不能做为最后一个表达式出现

```
#include <iostream>
using namespace std;
void f()
{
    int x=10;
}
int main()
{
    int k=10;
    k=k+f(); //编译错
    k, f();  //可编译通过，无意义
    cout << (k, f()) << endl; //编译错
    cout << (k, f(), k+2) << endl; //可编译通过
    return 0;
}
```

## § 4. 利用函数实现指定的功能

### 4.3. 函数参数与函数的值

#### 4.3.1. 形式参数与实际参数

#### 4.3.2. 函数的值（函数的返回值）

★ 一个return只能带回一个返回值

★ 函数中可以有多个return语句，但只能根据条件执行其中的一个，执行return后，函数调用结束（return后的语句不会被执行到）

```
int fun(void)
{
    if (...)
        return ...;
    else
        return ...;
    ....; //无法被执行到
}
```

★ 如果函数中有分支语句，但return未覆盖全部分支，则VS2017会报warning错（不会判断条件是否覆盖!）

warning C4715: “f”: 不是所有的控件路径都返回值

```
int fun(int x)
{
    if (x>1) {
        if (x>10)
            return 1;
    }
    else
        return 2;
} //报warning
```

```
int fun(int x)
{
    if (x>1)
        return 1;
    else if (x<=1)
        return 2;
} //仍会报warning
```

## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

函数的编写方法:

通过第2-3章的基本知识, 定义不同数据类型的变量, 采用顺序、分支、循环等基本结构, 按照函数的预期功能来编写每个函数

## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

#### 4. 4. 1. 基本形式

函数名 ( ) : 适用于无参函数

函数名 (实参表列) : 适用于有参函数, 用 , 分开

与形参表的个数、顺序、类型一致

★ 若同一变量同时出现在一个函数的多个参数中, 且有自增、赋值、复合赋值等改变变量值的操作, 则不同编译器处理的方式可能不同

fun(i, ++i)

{ 从左至右: fun(3, 4)

{ 从右至左: fun(4, 4)

书P. 89错

4. 4. 2前一段 fun(3, 3)

不再讨论

也不建议深入

注意: fun(i++, --j) 这种不同变量是必须讨论的

printf/scanf等函数有参数个数、类型不等的情况出现, 称为可变参数方式, 本课程不讨论

```
printf("%d\n", a);           //2个参数
printf("%d %d\n", a, b);     //3个参数
scanf("%d", &a);             //2个参数
scanf("%d %d", &a, &b);      //3个参数
```

## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

#### 4. 4. 2. 调用方式

函数语句: 函数调用+;

```
printf("Hello. \n");  
putchar('A');
```

函数表达式: 出现在某个表达式中

```
c=max(a, b)+4;  
k=sqrt(m);
```

函数参数: 作为另一个函数的参数

```
printf("max=%d", max(a, b));  
putchar( getchar() );  
sqrt( fabs(x) );
```

函数返回类型  
不能是void

问题: 其它函数的返回值  
可由调用函数使用,  
main的返回值给谁?

★ 函数调用时, 不能写返回类型

定义及实现时:

```
long f1()  
{ ...  
}  
int max(int x, int y)  
{ ...  
}
```

调用时:

```
k = f1();           ✓  
k = long f1();      ✗  
  
k = max(i, j);      ✓  
k = int max(i, j);  ✗
```

★ 无参函数调用时, 参数位置不能写void

定义及实现时:

```
int fun()           //空  
{ ...  
}  
int fun(void)       //写void  
{ ...  
}
```

调用时:

```
k = fun();          ✓  
k = fun(void);      ✗
```

★ 有参函数调用时, 实参不能写类型

定义及实现时:

```
int max(int x, int y)  
{ ...  
}
```

调用时:

```
int i=10, j=15;  
k=max(i, j);        ✓  
k=max(int i, int j); ✗
```

## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

#### 4. 4. 1. 基本形式

#### 4. 4. 2. 调用方式

#### 4. 4. 3. 对被调用函数的说明

#### ★ 对库函数，加相应的头文件说明

`#include <stdio>` 输入输出函数

`#include <math>` 数学运算函数

`#include <string>` 字符串运算函数

注意：<stdio>和<math>这两个头文件在VS2017中缺省可以不加，其它编译器一般需要加

#### ★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前

两种方法：

返回类型 函数名(形参类型)；

返回类型 函数名(形参类型 形参表)；

```
int max(int, int);
int main()
{
    k=max(i, j);
}
int max(int x, int y)
{
    ...
}
```

```
int max(int x, int y);
int main()
{
    k=max(i, j);
}
int max(int x, int y)
{
    ...
}
```

```
int max(int p, int q);
//pq不要求与实现中xy一致
int main()
{
    k=max(i, j);
}
int max(int x, int y)
{
    ...
}
```

## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

#### 4. 4. 1. 基本形式

#### 4. 4. 2. 调用方式

#### 4. 4. 3. 对被调用函数的说明

- ★ 对库函数，加相应的头文件说明
- ★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前
- ★ 若被调用函数出现在调用函数之前，可以不加说明 (有些编译器可能必须加)

//可以没有说明

```
float fun()
{
    ...
}
int main()
{
    float k;
    k=fun();
    return 0;
}
```

float fun(); //必须有说明

```
int main()
{
    float k;
    k=fun();
    return 0;
}
float fun()
{
    ...
}
```



## § 4. 利用函数实现指定的功能

### 4. 4. 函数的调用

#### 4. 4. 3. 对被调用函数的说明

★ 调用说明可以在函数外，针对后面所有函数均适用；也可在函数内部，只对本函数有效

```
int max(int x, int y);  
int main()  
{ ..max(...); ✓  
}  
int f1()  
{ ..max(...); ✓  
}  
int max(int x, int y)  
{ ....  
}
```

```
int main()  
{ int max(int, int);  
  ..max(...); ✓  
}  
int f1()  
{ ..max(...); ✗  
}  
int max(int x, int y)  
{ ....  
}
```

```
int main()  
{ ..max(...); ?  
}  
int max(int x, int y);  
int f1()  
{ ..max(...); ?  
}  
int max(int x, int y)  
{ ....  
}
```

```
int main()  
{ int max(int, int);  
  ..max(...); ?  
}  
int max(int x, int y)  
{ ....  
}  
int f1()  
{ ..max(...); ?  
}
```

## § 4. 利用函数实现指定的功能

### 4.5. 函数的嵌套调用

#### 4.5.1. C++程序的执行过程 (P. 93 9步)

- (1) 执行main函数的开头部分
- (2) 遇到调用a函数的语句，流程转去a函数
- (3) 执行a函数的开头部分
- (4) 遇到调用b函数的语句，流程转去b函数
- (5) 执行b函数，如果再无其他嵌套的调用，则完成b函数的全部操作
- (6) 返回原来调用b函数的位置，即返回a函数
- (7) 继续执行a函数中尚未执行的部分，直到a函数结束
- (8) 返回main中调用a函数的位置
- (9) 继续执行main函数的剩余部分直到结束

例：程序如下

```
void b()
{
    ...
}
void a()
{
    ...
    b();
    ...
}
int main()
{
    ...
    a();
    ...
    return 0;
}
```

如何返回？

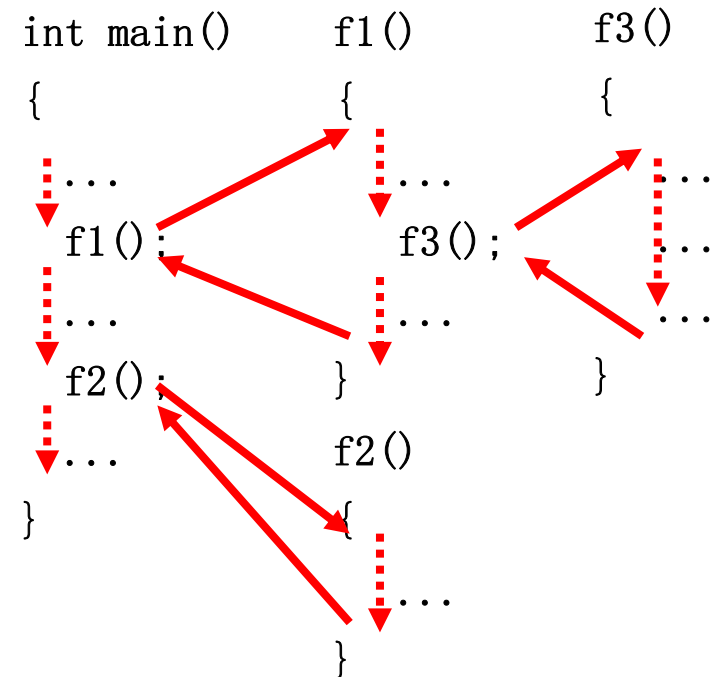


## § 4. 利用函数实现指定的功能

### 4.5. 函数的嵌套调用

#### 4.5.1. C++程序的执行过程 (通用描述)

- (1) 从main函数的第一个执行语句开始依次执行
- (2) 若执行到函数调用语句, 则保存调用函数当前的一些系统信息 (保存现场)
- (3) 转到被调用函数的第一个执行语句开始依次执行
- (4) 被调用函数执行完成后返回到调用函数的调用处, 恢复调用前保存的系统信息 (恢复现场)
- (5) 若被调用函数中仍有调用其它函数的语句, 则嵌套执行步骤 (2) - (4)
- (6) 所有被调用函数执行完后, 顺序执行main函数的后续部分直到结束



#### 4.5.2. 特点

- ★ 嵌套的层次、位置不限
- ★ 遵循后进先出的原则 (栈)
- ★ 调用函数时, 被调用函数与其所调用的函数的关系是透明的, 适用于大程序的分工组织

## § 4. 利用函数实现指定的功能

### 4. 5. 函数的嵌套调用

#### 4. 5. 3. 实例

例1: P. 93-94 例4. 4 (和书上的函数名有所不同)

```
int main()
{
    int a, b, c, d, m;

    ... 输入a/b/c/d四个数字
    m = max4(a, b, c, d);
    ... 输出最大值

    return 0;
}
```

```
int max4(int a, int b, int c, int d)
{
    int m;
    m = max2(a, b);
    m = max2(m, c);
    m = max2(m, d);
    return m;
}
```

```
int max2(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
int main()
{
    int a, b, c, d, m;

    ... 输入a/b/c/d四个数字
    m = max4(a, b, c, d);
    ... 输出最大值

    return 0;
}
```

```
int max4(int a, int b,
          int c, int d)
{
    int m1, m2, m;
    m1 = max2(a, b);
    m2 = max2(c, d);
    m = max2(m1, m2);
    return m;
} //改进
```

```
int max2(int a, int b)
{
    return (a > b ? a : b);
}
//改进
```

```
int main()
{
    ...
    m = max2( max2( max2(a, b), c), d );
    ...
}
```

一个函数的返回值做为  
另一个函数的参数  
(本例中函数名相同)

```
int main()
{
    ...
    m = max2( max2(a, b), max2(c, d) );
    ...
}
```

## § 4. 利用函数实现指定的功能

### 4.5. 函数的嵌套调用

#### 4.5.3. 实例

例2：写一个函数，判断某正整数是否素数 (P. 122习题3)

```
#include <iostream>
#include <cmath>
using namespace std;
int prime(int n)
{
    int i;
    int k = sqrt(n);
    for(i=2; i<=k; i++)
        if (n%i == 0) //两个=
            break;
    return i<=k ? 0 : 1;
}
int main()
{
    int n;
    cin >> n; //为简化讨论，此处假设输入正确
    cout << n << (prime(n) ? "是":"不是") << "素数" << endl;
    return 0;
}
```

参考 P. 78 例3.14

循环的结束有两个可能性：  
1、表达式2 (i<=k) 不成立  
2、因为 break 而结束

## § 4. 利用函数实现指定的功能

### 4.5. 函数的嵌套调用

#### 4.5.3. 实例

例2：写一个函数，判断某正整数是否素数 (P. 122习题3)

例3.14：找出100-200间的全部素数

```
for(m=101; m<=200; m+=2) { //偶数没必要判断
    prime=true;           //对每个数，先认为是素数
    k=int(sqrt(m)); // k=sqrt(m) 也可(有警告)
    for(i=2; i<=k; i++)
        if (m%i==0) {
            prime=false;
            break;
        }
    if (prime) {
        cout << setw(5) << m;
        n=n+1;           //计数器，只为了加输出换行
        if (n%10==0)      //每10个数输出一行，已改
            cout<<endl;
    }
} //本程序中prime是bool型变量
```

改写为用prime函数

```
int prime(int n)
{
    int i;
    int k = sqrt(n);
    for(i=2; i<=k; i++)
        if (n%i == 0)
            break;
    return i<=k ? 0 : 1;
}

int main()
{
    int m, ret;
    for(m=101; m<=200; m+=2) {
        if (prime(m)) {
            ...//打印
        }
    }
    return 0;
}

//本程序中，prime是函数名
```

## § 4. 利用函数实现指定的功能

### 4. 5. 函数的嵌套调用

#### 4. 5. 3. 实例

例3: 验证哥德巴赫猜想 (P. 122习题7)

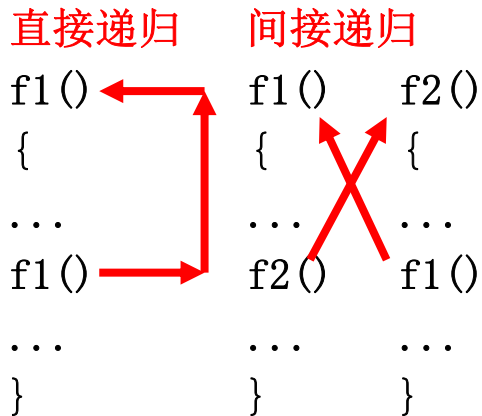
```
#include <iostream>
#include <cmath>
using namespace std;
int prime(int n)
{
    int i;
    int k = sqrt(n);
    for(i=2; i<k; i++)
        if (n%i == 0) //两个=      一道题目的解可用于另一题中
            break;              强调过程的积累、经验的积累
    return i<k ? 0 : 1;
}
void gotbaha(int even)
{
    int x;
    for (x=3; x<=even/2; x+=2)
        if (prime(x)+prime(even-x)==2) {
            cout<<x<<"+"<<even-x<<"="<<even<<endl;
            break; //不要break则求出全部组合
        }
}
int main()
{
    int n;
    cin >> n; //为简化讨论, 此处假设输入正确
    gotbaha(n);
    return 0;
}
```

## § 4. 利用函数实现指定的功能

### 4.6. 函数的递归调用

#### 4.6.1. 含义

函数直接或间接地调用本身



必然有条件判断是否进行下次递归调用!!!

★函数的返回值做本函数的参数，是嵌套，不是递归

```
int main()
{
    ...
    m = max2( max2(a, b), max2(c, d));
    ...
}
```

#### 4.6.2. 递归的求解过程

回推：到一个确定值为止 (递归不再调用)

递推：根据回推得到的确定值求出要求的解

P. 95 例4.5

回溯

```
age(5) = age(4) + 2;
age(4) = age(3) + 2;
age(3) = age(2) + 2;
age(2) = age(1) + 2;
age(1) = 10;
```

P. 96 图4.8

递推



## § 4. 利用函数实现指定的功能

### 4. 6. 函数的递归调用

#### 4. 6. 3. 如何写递归函数

★ 确定递归何时终止

★ 假设第 $n-1$ 次调用已求得确定值，确定第 $n$ 次调用和第 $n-1$ 次调用之间存在的逻辑关系

=> 不要全面考虑 $1..n$ 之间的变换关系，而应理解为只有 $n$ 和 $n-1$ 两层，且第 $n-1$ 层数据已求得

例1：P. 95 例4. 5（求解5个学生的年龄）

```
int age(int n)
{
    if (n==1)
        return 10;
    else
        return age(n-1)+2;
}
```

```
int main()
{
    cout << age(5) << endl;
    return 0;
}
```

$\text{age}(5) = \text{age}(4) + 2;$

$\text{age}(4) = \text{age}(3) + 2;$

$\text{age}(3) = \text{age}(2) + 2;$

$\text{age}(2) = \text{age}(1) + 2;$

$\text{age}(1) = 10;$

## § 4. 利用函数实现指定的功能

### 4.6. 函数的递归调用

#### 4.6.3. 如何写递归函数

例2: P. 97 例4.6 (采用非递归法和递归法两种方式求解 $n!$ )

##### 非递归法:

全面考虑1-n的关系,

可得出下列公式:

$$n! = 1*2*\dots*n;$$

```
int fac(int n)
{
    int s=1, i;
    for(i=1; i<=n; i++)
        s = s*i;
    return s;
}
```

##### 递归法:

不全面考虑1-n的关系,

仅考虑 $n$ 和 $n-1$ 两层,

且假设 $n-1$ 层已知

$$n! = n * (n-1)!$$

$$(n-1)! = (n-1) * (n-2)!$$

...

$$1! = 1$$

$$0! = 1;$$

```
int main() //也可以由键盘输入n值, 此处略
{
    int n = 5;
    cout << n << "!=" << fac(5) << endl;
}
```

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

## § 4. 利用函数实现指定的功能

### 4. 6. 函数的递归调用

#### 4. 6. 3. 如何写递归函数

#### 4. 6. 4. 如何读递归函数

- ★ 每次递归调用时，借助**栈**来记录调用的层次
- ★ 栈初始为空，每次递归函数被调用时在栈中增加一项，递归函数运行结束后栈中减少一项
- ★ 本次调用结束后，返回上次的调用位置，继续执行后续的语句
- ★ 重复操作至栈空为止

例1： 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

例1: 写出程序的运行结果及程序的功能

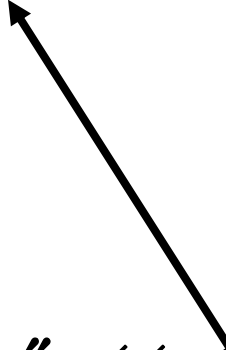
```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5)
    return 0;
}
```

例1： 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```



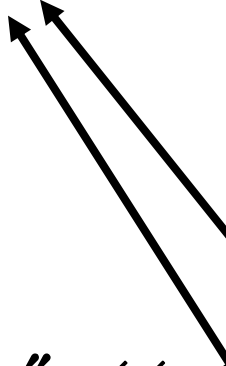
fac(4)	5
fac(5)	

例1: 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```

fac(3)	4
fac(4)	5
fac(5)	

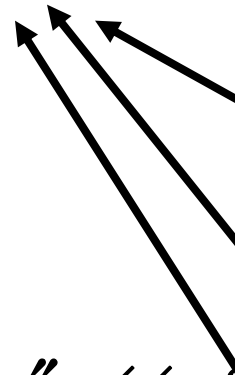


例1: 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```

fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	



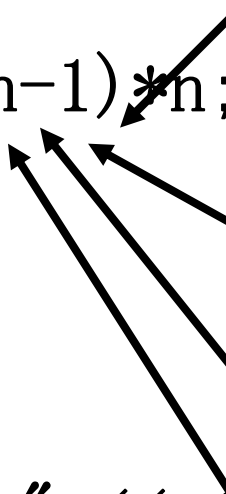


例1：写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```

fac(1)	2
fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	



例1: 写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0 || n==1)
```

```
    return 1;
```

```
    else
```

```
    return fac(n-1)*n;
```

```
}
```

```
int main()
```

```
{
```

```
    cout << "fac(5)=" << fac(5);
```

```
    return 0;
```

```
}
```

fac(1)	2	1
fac(2)	3	
fac(3)	4	
fac(4)	5	
fac(5)		

例1: 写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0 || n==1)
        return 1;
```

```
    else
```

```
        return fac(n-1)*n;
```

```
}
```

```
int main()
```

```
{
```

```
    cout << "fac(5)=" << fac(5);
```

```
    return 0;
```

```
}
```

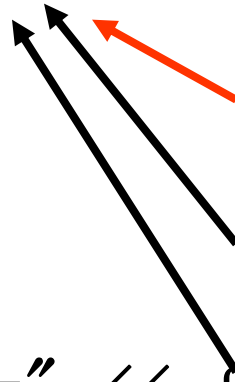
fac(2)	3	2
fac(3)	4	
fac(4)	5	
fac(5)		

例1: 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```

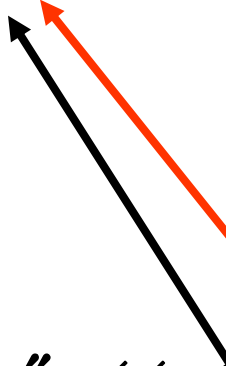
fac(3)	4	6
fac(4)	5	
fac(5)		



例1: 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

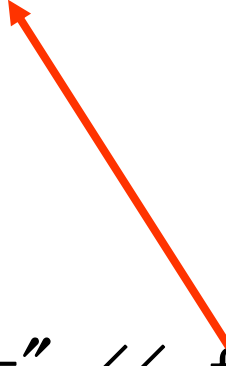


fac(4)	5	24
fac(5)		

例1: 写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```



fac(5)		120
--------	--	-----

例1：写出程序的运行结果及程序的功能

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

int main()
{
    cout << "fac(5)=" << fac(5) << endl;
    return 0;
}
```

**fac(5)=120**

fac(1)	2	1
fac(2)	3	2
fac(3)	4	6
fac(4)	5	24
fac(5)		120

例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k'); //VS2017中main无return不报错
}
```



例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

7, k

例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

5, k

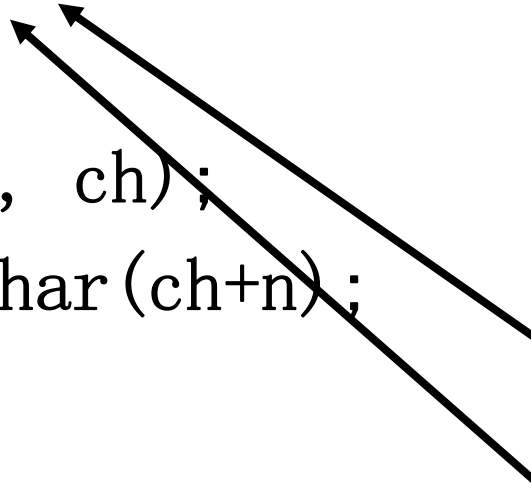
7, k

例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

3, k
5, k
7, k

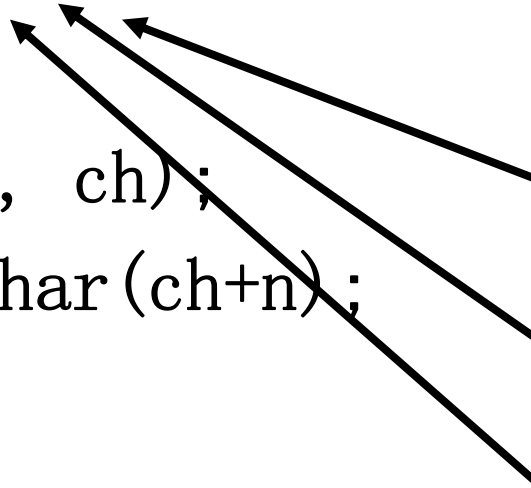


例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

1, k
3, k
5, k
7, k

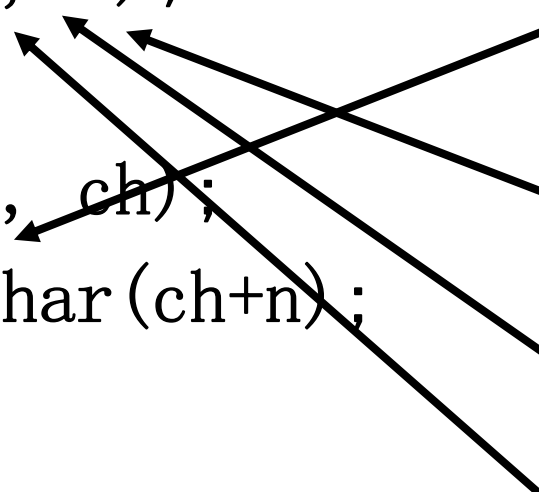


例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

2, k
1, k
3, k
5, k
7, k



例2：写出程序的运行结果

```
void f(int n, char ch)
```

```
{  if (n==0)
```

```
    return;
```

```
    if (n>1)
```

```
        f(n-2, ch);
```

```
    else
```

```
        f(n+1, ch);
```

```
    cout << char(ch+n);
```

```
}
```

```
int main()
```

```
{  f(7, 'k');
```

```
}
```

0, k

2, k

1, k

3, k

5, k

7, k

例2：写出程序的运行结果

```
void f(int n, char ch)
```

```
{   if (n==0)
```

```
    return;
```

```
    if (n>1)
```

```
        f(n-2, ch);
```

```
    else
```

```
        f(n+1, ch);
```

```
    cout << char(ch+n);
```

```
}
```

```
int main()
```

```
{   f(7, 'k');
```

```
}
```

0, k

2, k

1, k

3, k

5, k

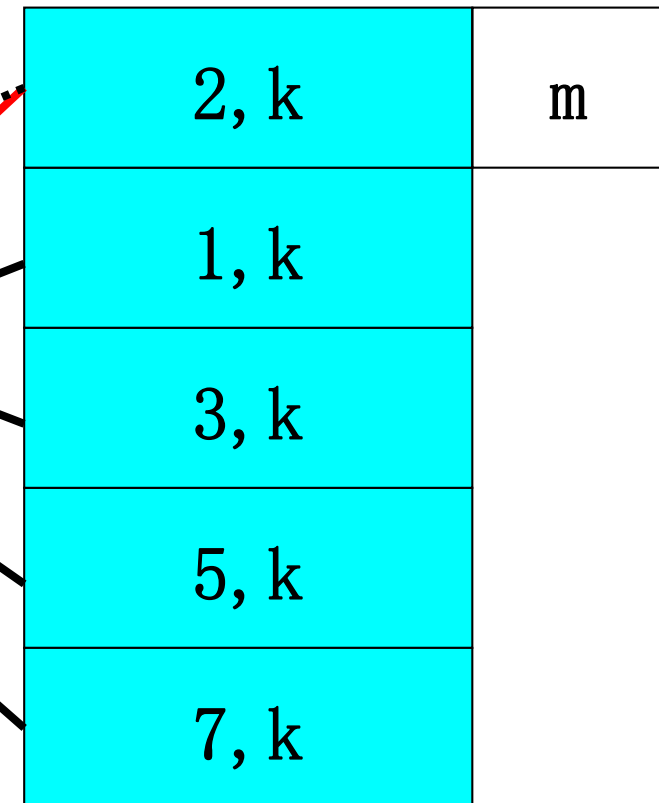
7, k

例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

黑虚:上次保存现场位置  
红实:本次恢复现场位置



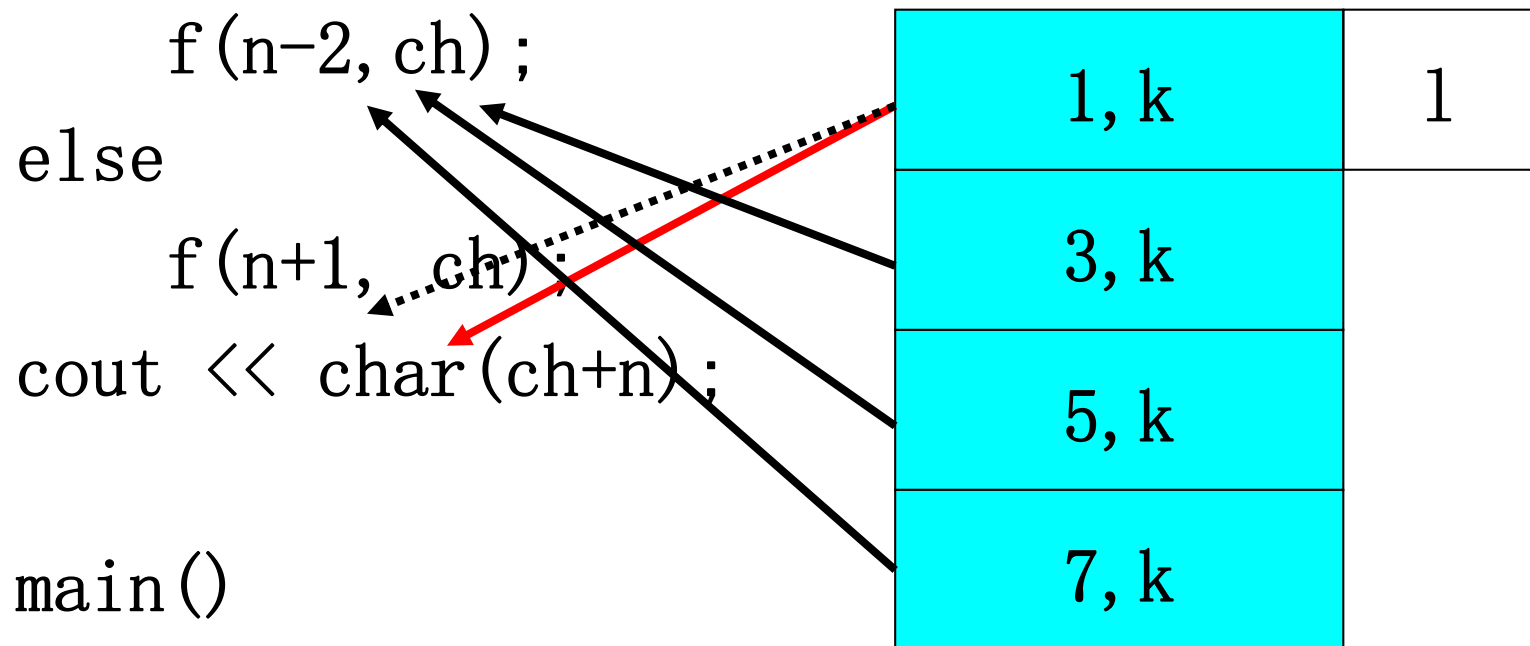


例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

黑虚:上次保存现场位置  
红实:本次恢复现场位置

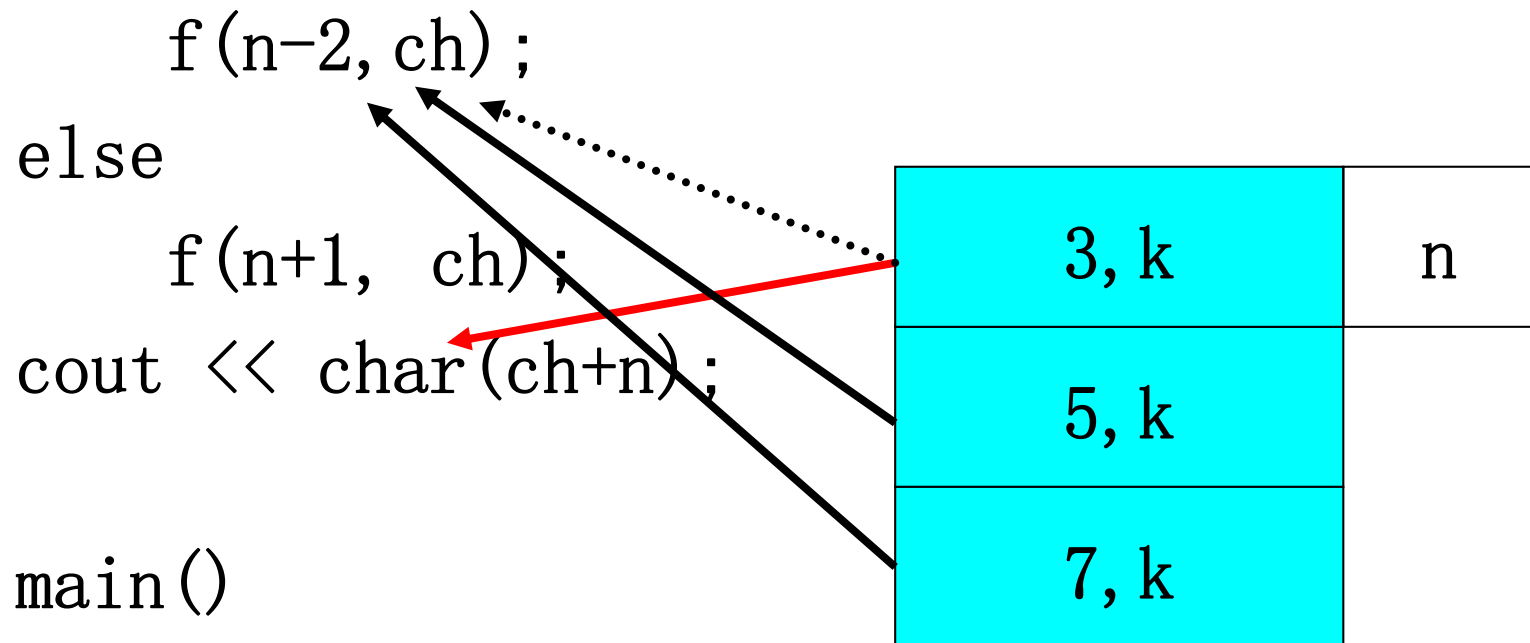


例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

黑虚:上次保存现场位置  
红实:本次恢复现场位置

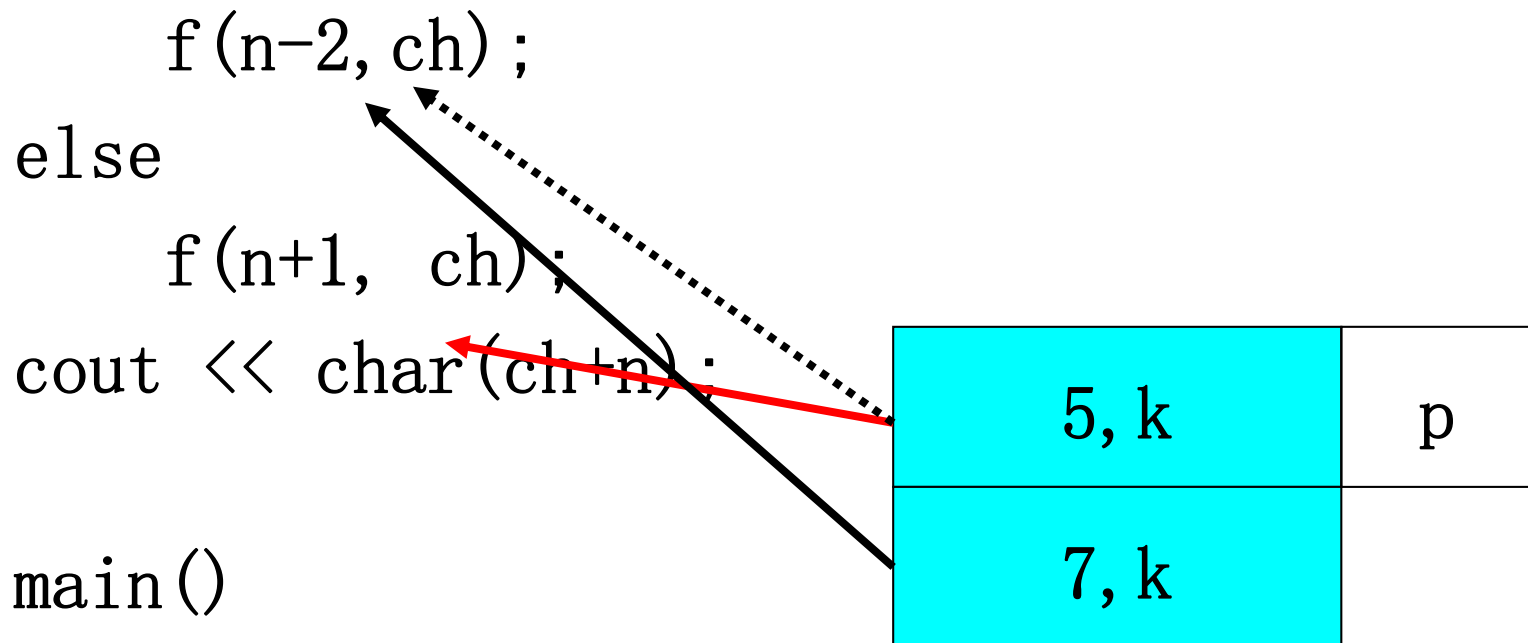


例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

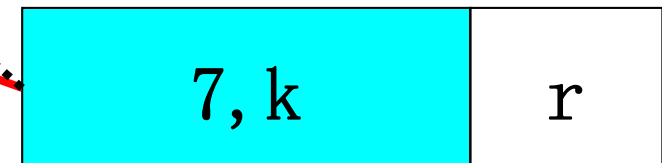
黑虚:上次保存现场位置  
红实:本次恢复现场位置



例2: 写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```



例2：写出程序的运行结果

```
void f(int n, char ch)
{
    if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}

int main()
{
    f(7, 'k');
}
```

mlnpr

0, k
2, k
1, k
3, k
5, k
7, k

### 例3：写出程序的运行结果及功能

```
void f(int n, int k)
{
    if (n>=k)
        f(n/k, k);
    cout << n%k;
}

int main()
{
    f(14, 2); 1110
    cout << endl;
    f(65, 8); 101
    return 0;
}
```

请用栈的方式  
自行画图理解

## § 4. 利用函数实现指定的功能

### 4. 6. 函数的递归调用

#### 4. 6. 5. 不设定终止条件的递归函数 (错误的用法)

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4. 11中详述
void fun()
{   num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

四编译器分别运行, 观察结果

1、为什么崩溃?

答:

2、不定义变量、定义10个int、10个double的情况下崩溃时打印的num值不同, 为什么?

答:

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4. 11中详述
void fun()
{   int a,b,c,d,e,f,g,h,i,j;
    a=b=c=d=e=f=g=h=i=j=10;
    num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

四编译器分别运行, 观察结果

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4. 11中详述
void fun()
{   double a,b,c,d,e,f,g,h,i,j;
    a=b=c=d=e=f=g=h=i=j=10;
    num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

四编译器分别运行, 观察结果

## § 4. 利用函数实现指定的功能

### 4.7. 内置函数(C++特有)

形式:

`inline` 返回类型 函数名 (形式参数表)

```
{  
    函数体  
}
```

`inline int max(int x, int y)`

```
{  
    return x>y?x:y;  
}
```



## § 4. 利用函数实现指定的功能

### 4.7. 内置函数(C++特有)

使用:

- ★ 不单独编为一段代码, 而是直接插入每个调用处, 调用时不按函数调用过程执行, 而是直接将该函数的代码放在调用处顺序执行

```
int main()          void f(void)
{
  ...
  f();
  ...
}
```

在可执行代码中有f()  
及main()的存在, 按正  
常函数调用方式进行

---

```
int main()          inline void f(void)
{
  ...
  ***
  ***
  ...
}
```

可执行代码中无f(), 仅  
有main(), main()代码  
长度增加, 但没有保存  
及恢复现场的消耗

## § 4. 利用函数实现指定的功能

### 4.7. 内置函数(C++特有)

```
int main()      inline void f(void)
{
    ...
    f();
    ...
    f();
    ...
    f();
    ...
}
```

假设f()被调用了10000次

```
int main()
{
    ...
    cout ...
    ...
    cout ...
    ...
    cout ...
    ...
}
```

相当于

```
inline void f(void)
{
    cout ...;
}
```

执行代码中f()  
已不存在

可执行代码中无f(), 仅有main(),  
main()代码中包含了10000份f()  
的代码, 长度增加, 但没有保存  
及恢复现场的消耗

以空间的增加换取时间的加快

问: 为什么不去掉f(), 直接在  
main()中写10000次cout?

答: 和前面定义符号常量一样  
#define pi 3.14159  
便于源程序的修改和维护

## § 4. 利用函数实现指定的功能

### 4.7. 内置函数(C++特有)

使用:

- ★ 不单独编为一段代码，而是直接插入每个调用处，调用时不按函数调用过程执行，而是直接将该函数的代码放在调用处顺序执行
- ★ 可执行程序的代码长度增加，但执行速度加快，适用于函数体短小且调用频繁的情况（1-5行）  
(保存/恢复现场的代价超过函数体自身代价的情况)
- ★ 不能包含分支、循环等复杂的控制语句
- ★ 系统编译时会自动判断是否需要真正采用内置方式  
(写了inline，最终也不一定真正成为内置函数)
- ★ 递归函数不能内置(递归必须要保存/恢复现场)
- ★ 允许只在函数声明或函数定义中加inline，也可以同时加

不同的编译器，三种情况可能都正确/部分正确(VS2017下都正确)

```
inline void fun();  
int main()  
{  
    ...  
}  
  
inline void fun()  
{  
    ...  
}
```

```
inline void fun();  
int main()  
{  
    ...  
}  
  
inline void fun()  
{  
    ...  
}
```

```
inline void fun();  
int main()  
{  
    ...  
}  
  
inline void fun()  
{  
    ...  
}
```

## § 4. 利用函数实现指定的功能

### 4.7. 内置函数(C++特有)

使用:

- ★ inline函数及调用函数必须在同一个源程序文件中, 否则编译出错  
(普通函数可以放在不同源程序文件中)

```
//ex1. cpp
inline void fun();
int main()
{
    ...
}
```

```
//ex2. cpp
inline void fun()
{
    ...
}
```

假设ex1. cpp和ex2. cpp共同构成一个可执行文件, 则编译**出错**

```
//ex1. cpp
void fun();
int main()
{
    ...
}
```

```
//ex2. cpp
void fun()
{
    ...
}
```

假设ex1. cpp和ex2. cpp共同构成一个可执行文件, 则编译**正确**

## § 4. 利用函数实现指定的功能

### 4.8. 函数的重载(C++特有)

重载：同一作用域中多个函数使用相同的名称

引入：对同一类功能的实现，仅参数的个数或类型不同，希望采用相同的函数名

C不允许  
C++允许

```
int    imax(int x,    int y);
float  fmax(float x,  float y);
long   lmax(long x,   long y);
====> 希望 imax/fmax/lmax 都叫 max ?
int    max(int x,    int y);
float  max(float x,  float y);
long   max(long x,   long y);
```

```
int max2(int x, int y);
int max3(int x, int y, int z);
int max4(int x, int y, int z, int w);
====> 希望 max2/max3/max4 都叫 max ?
int max(int x, int y);
int max(int x, int y, int z);
int max(int x, int y, int z, int w);
```

例：分别求两个int和double型数的最大值

```
int max(int x, int y)
{   cout << sizeof(x) << endl;
    return (x > y ? x : y);
}

double max(double x, double y)
{   cout << sizeof(x) << endl;
    return (x > y ? x : y);
}

int main()
{   cout << max(10,    15)    << endl;
    cout << max(10.2, 15.3) << endl;
}
```

?

例：分别求两个/三个int数的最大值

```
int max(int x, int y)
{   cout << 2 << ' ';
    return (x > y ? x : y);
}

int max(int x, int y, int z)
{   cout << 3 << ' ';
    int t = (x > y ? x : y);
    return (t > z ? t : z);
}

int main()
{   cout << max(10, 17)    << endl;
    cout << max(23, 15, 8) << endl;
}
```

?

## § 4. 利用函数实现指定的功能

### 4.8. 函数的重载(C++特有)

重载：同一作用域中多个函数使用相同的名称

引入：对同一类功能的实现，仅参数的个数或类型不同，希望采用相同的函数名

重载函数调用时的匹配查找顺序：

- (1) 寻找参数个数、类型完全一致的定义(严格匹配)
- (2) 通过系统定义的转换寻找匹配函数
- (3) 通过用户定义的转换寻找匹配函数

★ 若某一步匹配成功，则不再进行下一顺序的匹配

★ 若某一步中发现两个以上的匹配则出错

例：分别求两个int和double型数的最大值

```
#include <iostream>
using namespace std;
int max(int x, int y)
{   cout << sizeof(x) << ' ';
    return (x > y ? x : y);
}
double max(double x, double y)
{   cout << sizeof(x) << ' ';
    return (x > y ? x : y);
}
int main()
{   cout << max(10, 15) << endl; //int, int
    cout << max(10.2, 15.3) << endl; //double, double
    cout << max(10, int(15.3)) << endl; //int, double
    cout << max(5+4i, 15.3) << endl; //复数, double
    return 0;
}
```

哪句语句编译会错？  
其它正确语句的输出是什么？

复数形式目前编译会错，  
如何定义复数以及定义复数  
向double的转换，具体见  
第10章相关内容

严格匹配1  
严格匹配2  
系统转换1  
需自定义转换

## § 4. 利用函数实现指定的功能

### 4.8. 函数的重载(C++特有)

使用:

★ 要求同名函数的参数个数、参数类型不能完全相同

void fun(int x, int y);	正确
void fun(int x, int y, int z);	参数个数不同, 类型同
void fun(int x, int y);	正确
void fun(long x, long y);	参数个数同, 类型不同
void fun(int x, int y);	正确
void fun(long x, long y, long z);	个数类型均不同
void fun(int x, int y);	错误
void fun(int x, int y);	个数类型均相同

★ 返回类型及参数名不做检查(仅这两个不同认为错)

int max(int x, int y);	错误, 仅返回类型不同
long max(int x, int y);	参数类型个数完全相同
int max(int x, int y);	错误, 仅参数名不同
int max(int p, int q);	参数类型个数完全相同

★ 若参数类型是由typedef定义的不同名称的相同类型, 则会产生二义性

typedef INTEGER int;	相当于给int起个别名叫INTERGER, 具体见第7章
int fun(int a);	错误
INTEGER fun(INTEGER a);	

★ 尽量使同名函数完成相同或相似的功能, 否则可能导致概念混淆

## § 4. 利用函数实现指定的功能

### 4.9. 函数模板 (C++特有)

函数重载的不足：对于参数个数**相同**，类型**不同**，而实现过程**完全相同**的函数，仍要分别给出各个函数的实现

问题：两段一样的代码  
能否合并为一段？

```
int max(int x, int y)
{
    return x>y?x:y;
}
double max(double x, double y)
{
    return x>y?x:y;
}
```

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为**函数模板**，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

一段代码, 两个功能  
1、两个int型求max  
2、两个double型求max

```
#include <iostream>
using namespace std;
template <typename T>
T max(T x, T y)
{
    cout << sizeof(x) << ' ';
    return x>y?x:y;
}
int main()
{
    int    a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << max(a, b) << endl;
    cout << max(f1, f2) << endl;
    return 0;
}
```

4 15  
8 23.45



## § 4. 利用函数实现指定的功能

### 4.9. 函数模板 (C++特有)

函数重载的不足：对于参数个数**相同**，类型**不同**，而实现过程**完全相同**的函数，仍要分别给出各个函数的实现

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为**函数模板**，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

使用：

★ 仅适用于参数个数**相同**、类型**不同**，实现过程**完全相同**的情况

★ typename可用class替代

★ 类型定义允许多个

```
template <typename T1, typename t2>
```

```
template <class T1, class t2>
```

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
char max(T1 x, T2 y)
{
    cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}
int main()
{
    int    a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << max(a, f1) << endl;
    cout << max(f2, b) << endl;
    return 0;
}
```

?

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

引入：假设已经定义了某个函数，并进行了大量的应用后来随着要求的增加，需要扩充函数的功能并且增加相应的参数来满足扩充的功能

例：定义 `circle(int x, int y)` 用于画圆心在  $(x, y)$  处半径为10的圆，并已被调用1000次

```
int main()
{
    ...
    circle(...);
    ...
    circle(...);
    ...
    circle(...);
    ...
    circle(...);
    ...
} //有1000次调用

void circle(int x, int y)
{
    //具体实现过程
}
```

例：增加要求，要求半径可变，前面已调用的1000次中900次维持半径为10不变，100次改为不同值，又新增调用1000次

```
int main()
{
    ...
    circle(.旧.);
    ...
    circle(.旧.);
    ...
    circle(.新.);
    ...
    circle(.新.);
    ...
}

void circle(int x, int y, int r)
{
    //具体实现过程
}
```

首先：修改`circle`的定义及实现

其次：修改旧的1000次调用语句，从两参改为三参

最后：新增1000次三参调用

经过不断的测试，程序已稳定运行

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

例：一个程序要求的不断演变

- 1、定义 `circle(int x, int y)` 用于画圆心在  $(x, y)$  处半径为10的圆，并已被调用1000次
- 2、增加要求，要求半径可变，前面已调用的1000次中900次维持半径为10不变，100次改为不同值，又新增调用1000次
- 3、增加要求，要求指定不同的颜色，前面已调用过的2000次中1800次保持白色，200次改为其它颜色，又新增调用1000次
- 4、新增要求，要求指定空心还是实心，前面已调用过的3000次中2700次都是空心，300次改为实心，又新增调用1000次

- 1、使程序稳定运行所需要的测试工作工作量很大
- 2、一旦修改了程序，原来稳定运行的部分也可能出现各种问题，需要重新测试
- 3、新功能的增加是必须的

问题：能否在功能增加的同时使程序的改动尽可能少？

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

引入：假设已经定义了某个函数，并进行了大量的应用后来随着要求的增加，需要扩充函数的功能并且增加相应的参数来满足扩充的功能

含义：对函数的某一形参，大部分情况下都对应同一个实参值时，可以采用默认参数  
(默认值为常量)

形式：

返回类型 函数名(无默认参数形参, 有默认参数形参)

```
{  
    函数体  
}
```

```
void circle(int x, int y, int r=10)  
{  
    ...  
}
```

调用：circle(0,0); ⇔ circle(0,0,10);  
circle(5,8,12);

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

使用:

★ 便于函数功能的扩充, 减少代码维护, 修改的数量

针对刚才的例子:

=> 1、两个参数的原始程序完成, 调用1000次

```
void circle(int x, int y)
```

=> 2、加半径参数, 不变900处, 改100处, 增1000处

```
void circle(int x, int y, int r=10)
```

=> 3、加颜色参数, 不变1800处, 改200处, 增1000处

```
void circle(int x, int y, int r=10, int color=WHITE)
```

=> 4、加填充参数, 不变2700处, 改300处, 增1000处

```
void circle(int x, int y, int r=10, int color=WHITE, int filled=NO)
```

有效地减少了修改次数, 减少了工作量

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

使用:

★ 便于函数功能的扩充, 减少代码维护, 修改的数量

★ 允许有多个默认参数, 但必须是连续的最后几个

`void circle(int x=0, int y, int r=5)` (错)

`void circle(int y, int x=0, int r=5)` (对)

★ 若有多个默认参数, 调用时, 前面使用缺省值, 后面不使用缺省值, 则前面也要加上

`void circle(int x, int y, int r=5, int c=WHITE)`

`circle(10, 15);`

`circle(10, 15, 10);`

`circle(10, 15, 12, BLUE);`

`circle(10, 15, 5, BLUE);`

虽然是缺省, 也要加

## § 4. 利用函数实现指定的功能

### 4. 10. 有默认参数的函数(C++特有)

使用:

★ 若函数定义在调用函数之后, 则声明时必须给出默认值, 定义时不再给出

```
void circle(int x, int y, int r=10);  
int main()  
{ ...  
}  
void circle(int x, int y, int r)  
{ ...  
}
```

正确

```
void circle(int x, int y, int r=10);  
int main()  
{ ...  
}  
void circle(int x, int y, int r=5)  
{ ...  
}
```

错误

```
void circle(int x, int y, int r=10);  
int main()  
{ ...  
}  
void circle(int x, int y, int r=10)  
{ ...  
}
```

错误, 即使相同

★ 重载与带默认参数的函数一起使用时, 可能会产生二义性

```
int fun(int a, int b=10);
```

```
int fun(int a);
```

若调用为: fun(10, 20) 正确

fun(50) 二义性

```
void circle(int x, int y, int r);  
int main()  
{ ...  
}  
void circle(int x, int y, int r=10)  
{ ...  
}
```

错误

★ 默认参数的默认值必须是常量

```
int fun(int a, int b=10); //正确
```

```
int fun(int a, int b=3+7); //正确
```

```
int fun(int a, int b=a-1); //错误
```

## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第2章中：变量不能同名，不够准确)

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
...	...	...	...
}	a=15;	a=70000;	a=23;
	f2();	f3();	...
	}	}	}

- 在f3()执行时，三个a占用不同的内存空间，互不干扰
- 在f2()执行时，f1()/f2()的两个a占用不同内存空间，f3()的a未分配或已释放

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
f2();	...	...	...
f3();	a=15;	a=70000;	a=23;
}	...	...	...
	}	}	}

- f1()/f2()/f3()在不同时刻占用不同/相同(不保证)的内存空间，互不干扰

★ 形参是局部变量

int f1(int x)	int f2(long x)	int f3(int x)
{	{	{
...	...	...
}	}	}



## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量使用：

★ 复合语句内的变量，只在复合语句中有效(包括循环)

允许多层嵌套下各自定义  
属于自己作用范围的变量

```
void fun()
{   int i,a;
    a=15;
    for(i=0;i<10;i++) {
        int y;
        y=11; ✓
        a=16; ✓
    }
    y=12; ✗(超出复合
           语句的范围)
    a=17; ✓
}
```

```
void fun()
{   int i,a;
    a=15;
    {
        int y;
        y=11; ✓
        a=16; ✓
    }
    y=12; ✗(超出复合
           语句的范围)
    a=17; ✓
}
```

```
void fun()
{   int i,a=15;
    {
        int y;
        y=11; ✓
        a=16; ✓
        {
            int w=10;
            y=12; ✓
            a=13; ✓
            w=14; ✓
        }
        w=15; ✗(超出复合
               语句的范围)
    }
    y=12; ✗(超出复合
           语句的范围)
    a=17; ✓
}
```

## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量  
使用：

- ★ 不同函数内的局部变量可以同名
- ★ 形参是局部变量
- ★ 复合语句内的变量，只在复合语句中有效(包括循环)
- ★ 在该函数的被调用函数内也无效(不可访问)

```
void f1()
{
    a=14; ✗
}

int main()
{
    int a;
    a=15;
    f1();
    a=16;
}
```

```
void f1()
{
    int a;
    a=14; ✓
}

int main()
{
    int a;
    a=15;
    f1();
    a=16;
}
```

两个a都是局部变量，  
分占不同的内存空间

## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

#### 4. 11. 2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量  
使用：

★ 从定义点到源文件结束之间的所有函数均可使用

```
int f1()  
{ a=15; ✗  
}  
  
int a;  
  
int main()  
{ a=16; ✓  
}  
  
int f2()  
{ a=17; ✓  
}
```

```
int a;  
  
int f1()  
{ a=15; ✓  
}  
  
int main()  
{ a=16; ✓  
}  
  
int f2()  
{ a=17; ✓  
}
```

## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

#### 4. 11. 2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

★ 从定义点到源文件结束之间的所有函数均可使用

★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果

<pre>int a; void f1() { a=15; } int main() {     a=10;     cout &lt;&lt; "a=" &lt;&lt; a &lt;&lt; endl; a=10     f1();     cout &lt;&lt; "a=" &lt;&lt; a &lt;&lt; endl; a=15 }</pre>	<pre>void f1(int a) { a=15; }  int main() { int a =10;   cout &lt;&lt; "a=" &lt;&lt; a; a=10   f1(a);   cout &lt;&lt; "a=" &lt;&lt; a; a=10 }</pre>	<div>实参向形参单向传值的例子不一样!!!</div>	<pre>int f1(int a) { a=15;   return a; }  int main() { int a =10;   cout &lt;&lt; "a=" &lt;&lt; a; a=10   a = f1(a);   cout &lt;&lt; "a=" &lt;&lt; a; a=15 }</pre>	<div>用函数返回值改变实参的值不一样!!!</div>
<div>main()和f()访问的是同一个a，内存空间相同</div>	<div>main()和f()访问的是不同的a，占用不同的内存空间，单向传值</div>		<div>main()和f()访问的是不同的a，实参a是因为赋值语句而改变的</div>	

## § 4. 利用函数实现指定的功能

### 4. 11. 局部变量和全局变量

#### 4. 11. 1. 局部变量

#### 4. 11. 2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

- ★ 从定义点到源文件结束之间的所有函数均可使用
- ★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果
- ★ 在使用全局变量时应加以限制，提高程序的通用性和可靠性 (别处的无意修改会导致结果变化)
- ★ 若全局变量与局部变量同名，按“低层屏蔽高层”的原则处理 (应尽量避免，以免理解错误)

```
#include <iostream>
using namespace std;
int a=10;
void f1()
{
    cout << "a=" << a;  a=10;
    int a=5;
    cout << "a=" << a;  a=5
}
void f2()
{
    cout << "a=" << a;  a=10;
}
int main()
{
    f1();
    f2();
}
```

全局变量和局部变量分别占用不同的内存空间

能否在f1()中访问全局变量a？

C：不能

C++：可以(第9章)

## § 4. 利用函数实现指定的功能

### 4.11. 局部变量和全局变量

使用:

★ 若全局变量与局部变量同名, 按“**低层屏蔽高层**”的原则处理 (应尽量避免, 以免理解错误)

=> “**低层屏蔽高层**”的规则同样适用于局部变量和复合语句内的局部变量同名

=> 在多层次嵌套的情况下允许不同层次的变量同名, 遵循的基本规则是“**低层屏蔽高层**”

```
void f1()
{
    int a=5, i;
    for(i=0;i<10;i++) {
        int a=10;
        cout << "a=" << a;    a=10
    }
    cout << "a=" << a;        a=5
}
```

```
inline int f()
{
    int a=5;
    cout << "fa=" << a << endl;
}
int main()
{
    int a=10;
    f();
    cout << "ma=" << a << endl;
}
```

问: inline应该这么理解吗?

```
int a=15; ←
void f1()
{
    int a=5, i; ←
    for(i=0;i<10;i++) {
        int a=10; ←
        if (i==5) {
            long a=20; ←
            cout << "a=" << a;    a=20
        }
        cout << "a=" << a;        a=10
    }
    cout << "a=" << a;            a=5
}
```

```
int main()
{
    int a=10;
    int a=5;
    cout << "fa=" << a << endl;
    cout << "ma=" << a << endl;
}
```

## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 1. 应用程序执行时的内存分布

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中, 变量占固定的存储空间
动态存储区	程序执行中, 变量根据需要分配不同位置的存储空间

### 4. 12. 2. 局部变量的存储

#### 4. 12. 2. 1. 分类

自动变量: 函数进入后, 分配空间, 函数运行结束后, 释放空间 (重复进行)

- 1、假设main()中调用10000次f1(), 则x, a的分配释放会重复10000次
- 2、不保证每次x/a的空间与上次相同

- 1、假设main()中调用10000次f1(), 则a的分配释放只有1次(x仍为10000次)
- 2、每次进入f1中, a都保持上次的值不变

静态局部变量: 变量所占存储单元在程序的执行过程中均不释放 (无论函数体内外)

```
int main()      void f1(int x)
{ ...          {
    f1(..);      int a;
    ...          ...
    f1(..);      }
    ...          }
} //假设调用10000次f1()
```

```
int main()      void f1(int x)
{ ...          {
    f1(..);      static int a;
    ...          ...
    f1(..);      }
    ...          }
} //假设调用10000次f1()
```

## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 2. 局部变量的存储

##### 4. 12. 2. 1. 分类

自动变量：函数进入后，分配空间，函数运行结束后，释放空间（重复进行）

#### ★ 关于自动变量(auto)的新旧标准

- C++新标准中，缺省不写就是**自动变量**，而auto用来表示**自动存储类型**的变量(VS2017)  
=>新标准中，自动变量/auto变量是**不同的变量**
- C++旧标准中，缺省不写就是自动变量，也可以加auto来表示(其余三编译器)  
=>旧标准中，自动变量/auto变量是**相同的变量**

```
#include <iostream>
using namespace std;
int main()
{
    auto int a;
    int b=10;
    auto char c=2.1;

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

三编译器编译  
VS2017编译

```
#include <iostream>
using namespace std;
int main()
{
    auto a = 1;
    auto b = 'A';
    auto c=2.1;

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

VS2017编译  
三编译器编译

```
int main()
{
    auto int a;        //int 型自动变量
    int b=10;          //int 型自动变量(未加auto)
    auto char c=2.1;    //char型自动变量
} //三者相同
```

旧标准

//auto变量不允许跟类型，定义时必须初始化，根据初始化值决定类型

新标准

```
int main()
{
    auto int x; //错误
    auto a=1;   //int型
    auto b='A'; //char型
    auto f=1.0; //double型
}
```

静态局部变量：变量所占存储单元在程序的执行过程中均不释放（无论函数体内外）



## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 2. 局部变量的存储

##### 4. 12. 2. 2. 使用

★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量

★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均**重复赋初值**;

静态局部变量在第一次调用时执行, 以后每次调用**不再赋初值**, 保留上次调用结束时的值

```
#include <iostream>
using namespace std;
void f1()
{
    int a=1; //VS2017不能加auto
    a++;
    cout << "a=" << a << endl;
}
int main()
{
    f1();           a=2
    f1();           a=2
    f1();           a=2
}
```

- 1、a的分配/释放重复了3次
- 2、3次的a不保证分配同一空间

```
#include <iostream>
using namespace std;
void f1()
{
    static int a=1;
    a++;
    cout << "a=" << a << endl;
}
int main()
{
    f1();           a=2
    f1();           a=3
    f1();           a=4
}
```

- 1、a在编译时已分配了空间, 在3次调用中未进行过分配/释放
- 2、每次进入, a都是同一空间
- 3、在f1()内部a可被访问, 在f1()外a不能访问(但存在)

## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别


#### 4. 12. 2. 局部变量的存储

##### 4. 12. 2. 2. 使用


★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量

★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均重复赋初值;  
静态局部变量在第一次调用时执行, 以后每次调用不再赋初值, 保留上次调用结束时的值

```
#include <iostream>
using namespace std;
int f(int n)
{
    int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n", i, f(i));
    return 0;
}
```



```
#include <iostream>
using namespace std;
int f(int n)
{
    static int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n", i, f(i));
    return 0;
}
```



## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 2. 局部变量的存储

##### 4. 12. 2. 2. 使用

- ★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量
- ★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均**重复赋初值**;  
静态局部变量在第一次调用时执行, 以后每次调用**不再赋初值**, 保留上次调用结束时的值
- ★ 若定义时不赋初值, 则自动变量的值不确定, 静态局部变量的值为0 (' \0' )

```
#include <iostream>
using namespace std;
int main()
{
    short a;
    static short b;
    static char c;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "c=" << (int)c << endl;
    return 0;
}
```

a=VS2017下编译报错  
其它系统: 不可预知值  
不同系统各不相同

b=0  
c=0

- ★ 函数的形参同自动变量

## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 3. 寄存器变量

含义：对一些频繁使用的变量，可放入CPU的寄存器中，提高访问速度

(CPU访问寄存器比内存快一个数量级)

```
register int a;
```

★ 仅对自动变量和形参有效

★ 编译系统会自动判断(即使定义了register，最终是否放入寄存器中，仍需要编译系统决定)

#### 4. 12. 4. 用extern扩展全局变量的使用范围

原因：全局变量从定义点到源文件结束之间的所有函数均可使用，为了能在其它部分使用变量，需要进行使用范围的扩展

方法：在定义范围外使用全局变量时，应加上extern的说明，**extern不分配存储空间**，只说明对应关系

分配4字节空间

```
int f1()
{ a=15; ✗
}
int a;
main()
{ a=16; ✓
}
int f2()
{ a=17; ✓
}
```

```
extern int a;
int f1()
{ a=15; ✓
}
int a;
main()
{ a=16; ✓
}
int f2()
{ a=17; ✓
}
```

不分配空间  
说明对应关系

源程序文件ex1.cpp、ex2.cpp共同构成一个程序

ex1.cpp

```
int a;
```

```
main()
```

```
{ a=16; ✓
```

```
}
```

```
int f2()
```

```
{ a=17; ✓
```

```
}
```

ex2.cpp

```
int f1()
```

```
{ a=18; ✗
```

```
}
```

```
int f3()
```

```
{ a=19; ✗
```

```
}
```

ex1.cpp

```
int a;
```

```
main()
```

```
{ a=16; ✓
```

```
}
```

```
int f2()
```

```
{ a=17; ✓
```

```
}
```

ex2.cpp

```
extern int a;
```

```
int f1()
```

```
{ a=18; ✓
```

```
}
```

```
int f3()
```

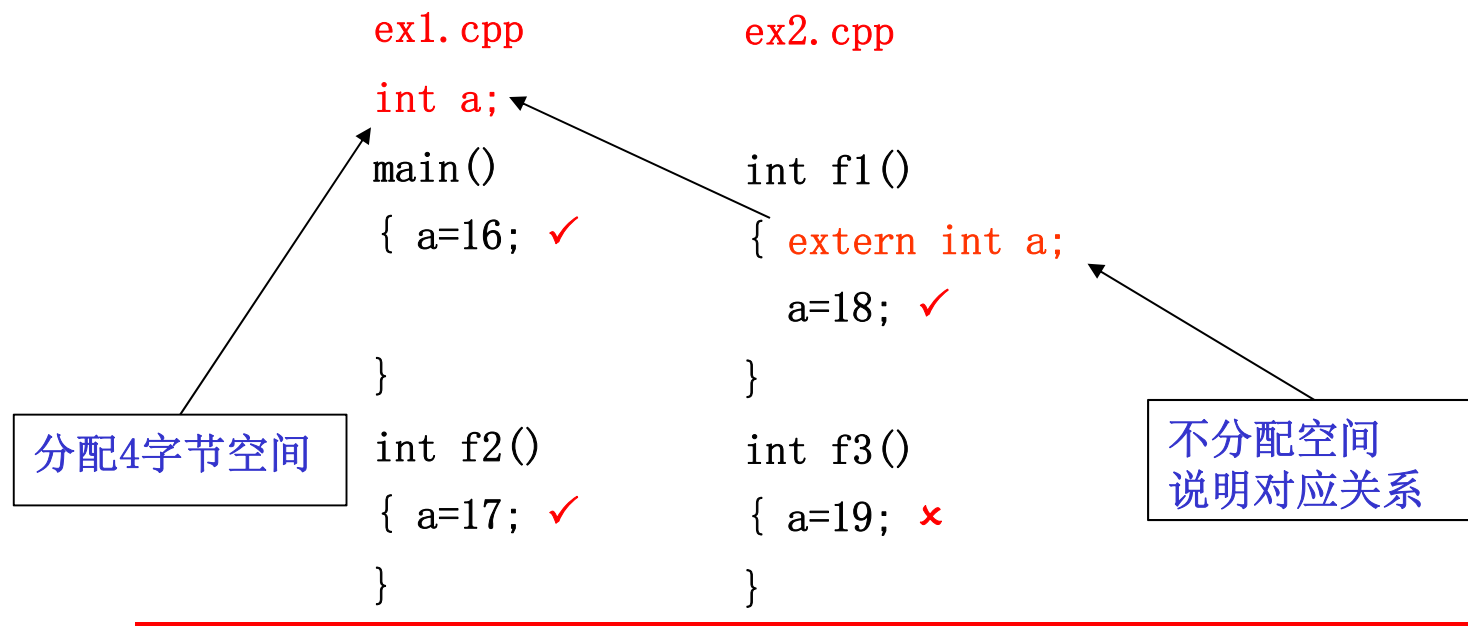
```
{ a=19; ✓
```

```
}
```

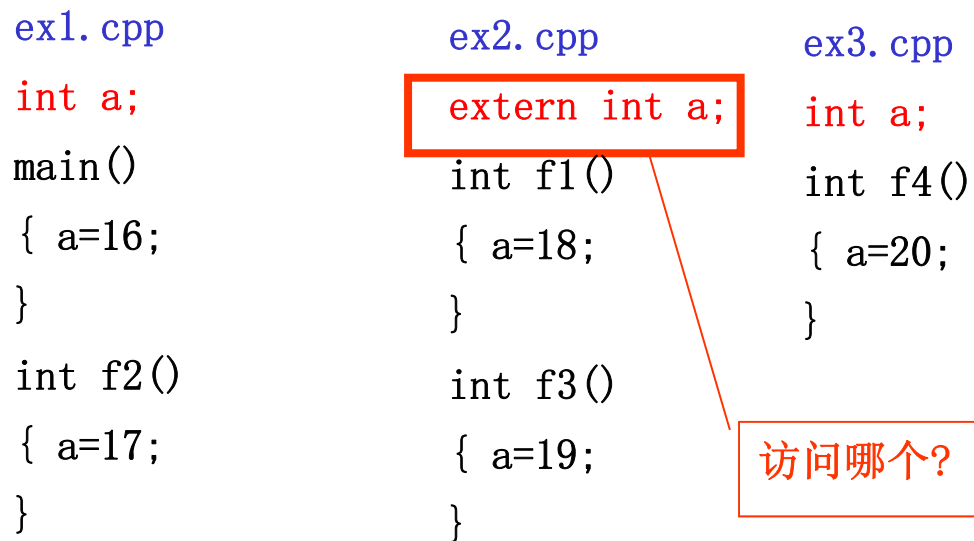
分配4字节空间

不分配空间  
说明对应关系

例：源程序文件ex1.cpp、ex2.cpp共同构成一个程序



例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序



## § 4. 利用函数实现指定的功能

### 4. 12. 变量的存储类别

#### 4. 12. 5. 全局变量的存储

外部全局变量：所有源程序文件中的函数均可使用

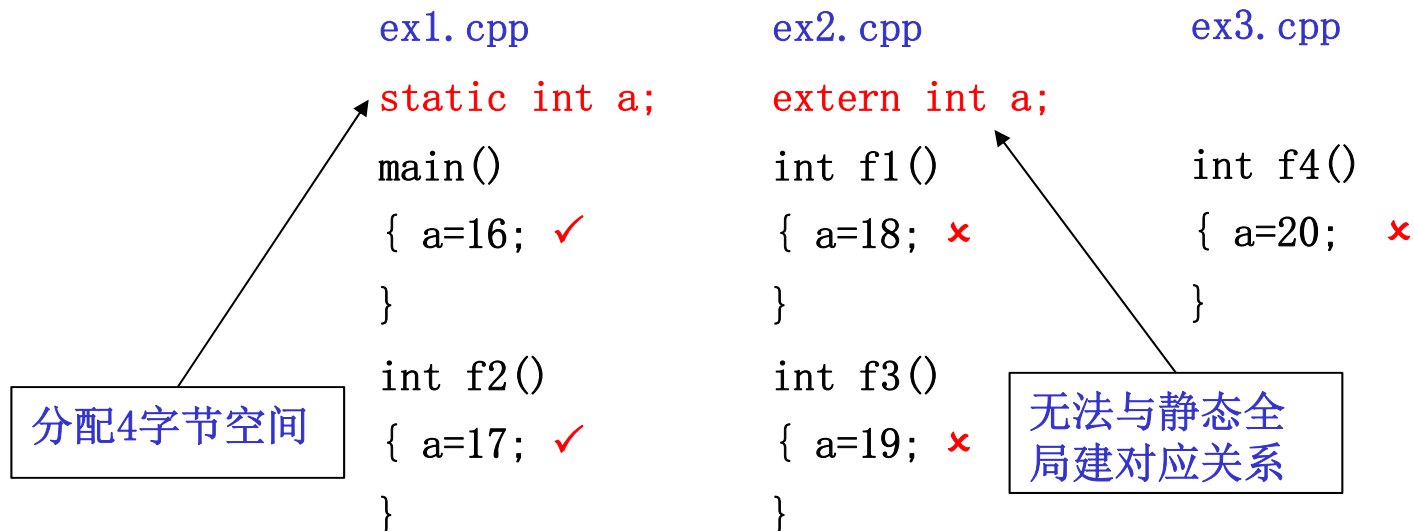
(其它源程序文件中加extern说明)

静态全局变量：只限本源程序文件的定义范围内使用

(static)

- ★ 两者均在静态数据区中分配，不赋初值则自动为0
- ★ 不同源程序文件中的静态全局变量允许同名
- ★ 静态全局变量可与其它源程序文件中的外部全局变量同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序



外部源程序文件  
无法访问静态全局  
变量

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;  
main()  
{ a=16; ✓  
}  
int f2()  
{ a=17; ✓  
}
```

ex2.cpp

```
static int a;  
int f1()  
{ a=18; ✓  
}  
int f3()  
{ a=19; ✓  
}
```

ex3.cpp

```
int f4()  
{ a=20; ✗  
}
```

不同源程序文件中的静态全局变量允许同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;  
main()  
{ a=16; ✓  
}  
int f2()  
{ a=17; ✓  
}
```

分配4字节空间

ex2.cpp

```
extern int a;  
int f1()  
{ a=18; ✓  
}  
int f3()  
{ a=19; ✓  
}
```

ex3.cpp

```
int a;  
int f4()  
{ a=20; ✓  
}
```

不分配空间  
说明对应关系

分配4字节空间

静态全局变量可与其它源程序文件中的外部全局变量同名



源程序ex1.cpp-ex4.cpp共同构成一个程序

ex1.cpp

static int a;

main()

```
{ a=16;  
}
```

int f2()

```
{ a=17;  
}
```

ex2.cpp

extern int a;

int f1()

```
{ a=18;  
}
```

int f3()

```
{ a=19;  
}
```

ex3.cpp

int a;

int f4()

```
{ a=20;  
}
```

ex4.cpp

int a;

int f5()

```
{ a=21;  
}
```

情况1: 正确/错误 ?

ex1.cpp

static int a;

main()

```
{ a=16;  
}
```

int f2()

```
{ a=17;  
}
```

ex2.cpp

extern int a;

int f1()

```
{ a=18;  
}
```

int f3()

```
{ a=19;  
}
```

ex3.cpp

int a;

int f4()

```
{ a=20;  
}
```

ex4.cpp

static int a;

int f5()

```
{ a=21;  
}
```

情况2: 正确/错误 ?

## § 4. 利用函数实现指定的功能

### 4. 13. 变量属性小结

#### 4. 13. 1. 变量的分类

按类型：字符型、整型、浮点型等

按作用域 {  
局部变量  
全局变量

按存储方式（生存期） {  
动态存储变量  
静态存储变量

按存储位置 {  
内存变量  
寄存器变量

#### 4. 13. 2. 变量的生存期与作用域

	生存期	作用域	存储区
自动变量	本函数	本函数	动态数据区
形参	本函数	本函数	动态数据区
寄存器	本函数	本函数	CPU的寄存器
静态局部	程序执行中	本函数	静态数据区
静态全局	程序执行中	本源程序文件	静态数据区
外部全局	程序执行中	全部源程序文件	静态数据区

## § 4. 利用函数实现指定的功能

### 4.14. 变量的声明与定义

定义：指定变量的类型，名称并分配存储空间

声明：指明变量的相互关系，不分配存储空间

`int a;`            定义

`extern int a;`    声明

## § 4. 利用函数实现指定的功能

### 4.15. 内部函数和外部函数

内部函数：仅能在本源程序中被调用的函数

`static` 返回类型 函数名（形参表）

★ 不同的源程序文件中可以同名

外部函数：可以在所有的源程序文件中被调用

★ 本源程序文件中直接使用

★ 其它源程序文件中加函数说明

（可以加`extern`，也可以不加）

例：源程序文件`ex1.cpp`、`ex2.cpp`共同构成一个程序

外部源程序文件  
无法访问内部函数

`ex1.cpp`

```
static float f2();
```

```
main()
```

```
{ f2();    ✓
```

```
}
```

```
static float f2()
```

```
{ ...
```

```
}
```

```
int f1()
```

```
{ f2();    ✓
```

```
}
```

`ex2.cpp`

```
int f3();
```

```
{ f2();    ✗
```

```
}
```

源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static float f2();  
main()  
{ f2(); ✓  
}  
  
static float f2()  
{ ...  
}  
  
int f1()  
{ f2(); ✓  
}
```

ex2.cpp

```
int f3();  
{ f2(); ✗  
}
```

ex3.cpp

```
static char f2();  
int f4()  
{ f2(); ✓  
}  
  
static char f2()  
{ ...  
}
```

不同的源程序文件  
中的内部函数可以  
同名

ex1.cpp

```
float f2();  
main()  
{ f2(); ✓  
}  
  
float f2()  
{ ...  
}  
  
int f1()  
{ f2(); ✓  
}
```

ex2.cpp

```
int f3();  
{ f2(); ✗  
}
```

ex3.cpp

```
extern float f2();  
int f4()  
{ f2(); ✓  
}
```

在其它源程序文件  
中加函数说明可以  
访问外部函数

extern可要可不要

源程序ex1.cpp-ex3.cpp共同构成一个程序

ex1.cpp

float f2();

main()

{ f2();

}

float f2()

{...}

ex2.cpp

static float f2()

int f3();

{ f2();

}

float f2()

{...}

ex3.cpp

extern float f2();

int f4()

{ f2();

}

情况1：正确/错误？

ex1.cpp

float f2();

main()

{ f2();

}

float f2()

{...}

ex2.cpp

float f2()

int f3();

{ f2();

}

float f2()

{...}

ex3.cpp

extern float f2();

int f4()

{ f2();

}

情况2：正确/错误？

## § 4. 利用函数实现指定的功能

### 4. 16. 头文件

#### 4. 16. 1. 头文件的内容及作用

头文件的内容:

- ★ 类型(struct-第7章)及类(class-第8章)的声明
  - ★ 函数的声明
  - ★ inline函数的定义与实现
  - ★ 符号常量的定义及常变量的定义
  - ★ 全局变量的extern声明
- P. 121 4. 16. 1中
- (5)全局变量定义 - 说法有错
- ★ 其它需要的头文件

例:程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//引用100个函数
#include <iostream>
using namespace std;

void f1();
...
void f100();

int main()
{
    f1();
    ...
    f100();
}
```

#include中  
<>和"的区别先忽略

```
//ex2.cpp
//共100个函数

void f1()
{
    ...
}
...
void f100()
{
    ...
}
```

```
//ex3.cpp
//引用100个函数
#include <iostream>
using namespace std;

void f1();
...
void f100();

void fun()
{
    f1();
    ...
    f100();
}
```

问题:

函数定义的声明被多处重复, 若修改了某个函数的定义, 则需要修改多处, 会造成不一致

```
//ex1.cpp
//引用100个函数
#include <iostream>
using namespace std;
#include "ex.h"

int main()
{
    f1();
    ...
    f100();
}
```

程序由ex1.cpp、  
ex2.cpp、ex3.cpp、  
ex.h (新增) 组成

```
//ex.h
//100个函数的声明
void f1();
...
void f100();
```

```
//ex3.cpp
//引用100个函数
#include <iostream>
using namespace std;
#include "ex.h"

void fun()
{
    f1();
    ...
    f100();
}
```

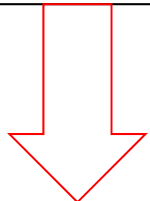
通过头文件使维护简单, 避免多处修改导致的不一致性



例:程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp
#include <iostream>
using namespace std;
inline void f1()
{
    ...
}

int main()
{
    f1();
    ...
    f1();
}
```



```
//ex1.cpp
#include <iostream>
using namespace std;

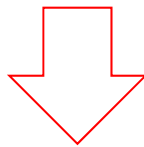
#include "ex.h"

int main()
{
    f1();
    ...
    f1();
}
```

程序由ex1.cpp、  
ex2.cpp、ex.h  
(新增) 组成

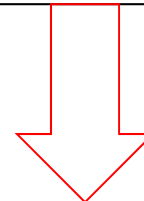
```
//ex.h

inline void f1()
{
    ...
}
```



```
//ex2.cpp
#include <iostream>
using namespace std;
inline void f1()
{
    ...
}

void fun()
{
    f1();
    ...
    f1();
}
```

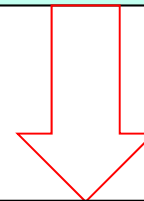


```
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
    f1();
    ...
    f1();
}
```

问题:  
因为inline函数  
必须和调用函数处在  
同一个源文件中,  
导致多处重复



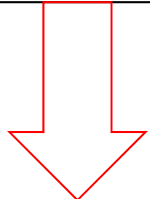
通过头文件使维护简单, 避免  
多处修改导致的不一致性

例:程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

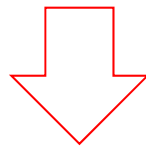
int main()
{
    ...pi...
    ...
    ...x...
}
```



```
//ex1.cpp
#include <iostream>
using namespace std;

#include "ex.h"

int main()
{
    ...pi...
    ...
    ...x...
}
```



程序由ex1.cpp、  
ex2.cpp、ex.h  
(新增) 组成

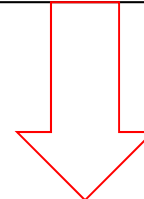
```
//ex.h

#define pi 3.14159
const int x=10;
```

```
//ex2.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

void fun()
{
    ...pi...
    ...
    ...x...
}
```

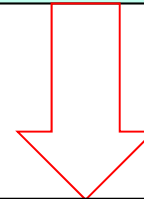


```
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
    ...pi...
    ...
    ...x...
}
```

问题:  
符号常量及常变  
量在多处定义, 导致  
重复定义以及维护困  
难



通过头文件使维护简单, 避  
免多处修改导致的不一致性

例:程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
    ...x...
}
```

```
//ex2.cpp
//引用全局变量

extern int x;
void f1()
{
    ...x...
}

void f2()
{
    ...x...
}
```

```
//ex3.cpp
//引用全局变量

extern int x;
void fun()
{
    ...x...
}
```

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
    ...x...
}
```

```
//ex2.cpp
//引用全局变量

#include "ex.h"
void f1()
{
    ...x...
}

void f2()
{
    ...x...
}
```

```
//ex3.cpp
//引用全局变量

#include "ex.h"
void fun()
{
    ...x...
}
```

程序由ex1.cpp、  
ex2.cpp、ex3.cpp、  
ex.h（新增）组成

```
//ex.h
//全局变量声明

extern int x;
```

例:程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
    ...x...
}
```



```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

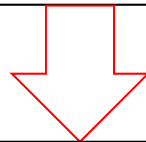
#include "ex.h"

int main()
{
    ...x...
}
```

```
//ex2.cpp
//引用全局变量

extern int x;
void f1()
{
    ...x...
}

void f2()
{
    ...x...
}
```



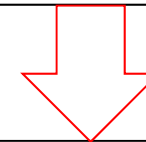
```
//ex2.cpp
//引用全局变量

#include "ex.h"
void f1()
{
    ...x...
}

void f2()
{
    ...x...
}
```

```
//ex3.cpp
//引用全局变量

extern int x;
void fun()
{
    ...x...
}
```



```
//ex3.cpp
//引用全局变量

#include "ex.h"
void fun()
{
    ...x...
}
```



程序由ex1.cpp、  
ex2.cpp、ex3.cpp、  
ex.h（新增）组成

```
//ex.h
//全局变量定义
int x; //错误
```

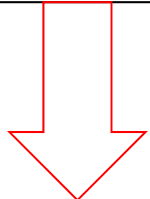
若头文件中包含全局  
变量定义，则被多个  
文件包含会导致重复  
定义

例:程序由ex1.cpp、ex2.cpp共同构成

//ex1.cpp

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

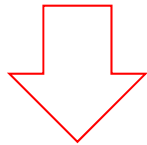
```
int main()
{
    ...
}
```



//ex1.cpp

```
#include "ex.h"
```

```
int main()
{
    ...
}
```



程序由ex1.cpp、  
ex2.cpp、ex.h  
(新增) 组成

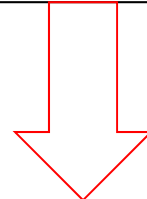
//ex.h

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

//ex2.cpp

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

```
void fun()
{
    ...
}
```



//ex2.cpp

```
#include "ex.h"
```

```
void fun()
{
    ...
}
```

## § 4. 利用函数实现指定的功能

### 4. 16. 头文件

#### 4. 16. 1. 头文件的内容及作用

头文件的作用：

- ★ 将编程者需要的在不同源程序文件传递的各种信息归集在一起，方便多次调用以及集中修改
- ★ 在一个源程序文件中包含头文件时，头文件的所有内容会被理解为包含到 `#include` 位置处，编译时（变量的定义及函数作用域等）均当作一个文件进行处理

头文件的包含方式：

`#include <文件名>`：直接到系统目录中寻找，找到则包含进来，找不到则报错

`#include "文件名"`：先在当前目录中寻找，找到则包含进来，

找不到则再到系统目录中寻找，找到则包含进来，找不到则报错

VS2017如果缺省安装，则头文件的目录为

32位Windows操作系统：

C:\Program Files\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.11.25503\include

64位Windows操作系统：

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.11.25503\include

## § 4. 利用函数实现指定的功能

例1:理解<>和""的差别

例: 在当前目录下有  
demo.h文件

内容:

```
int a=10;
```

源程序文件demo.c的内容

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

```
#include <demo.h>
int main()
{
    cout << a << endl;
    return 0;
}
```

编译报错, 因为<>不寻找  
当前目录中是否有demo.h

源程序文件demo.c的内容

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

```
#include "demo.h"
int main()
{
    cout << a << endl;
    return 0;
}
```

编译正确

例2:理解<>和""的差别

例: 在当前目录下有  
demo.h文件

内容:

```
int a=10;
```

例: 在系统目录下有  
demo.h文件

内容:

```
int b=10;
```

源程序文件demo.c的内容

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

```
#include <demo.h>
int main()
{
    cout << b << endl;
    return 0;
}
```

编译正确

源程序文件demo.c的内容

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

```
#include "demo.h"
int main()
{
    cout << b << endl;
    return 0;
}
```

编译报错, 因为""方式找到  
的是当前目录, 无b的定义

## § 4. 利用函数实现指定的功能

### 4. 16. 头文件

#### 4. 16. 1. 头文件的内容及作用

#### 4. 16. 2. C++的标准库及头文件

C++包含系统头文件的两种形式：

`#include <math.h>` : C形式

`#include <cmath>` : C++形式

两种方式都是指编译系统的include目录的math.h

VS2017如果缺省安装，则头文件的目录为

32位Windows操作系统：

C:\Program Files\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.11.25503\include

64位Windows操作系统：

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.11.25503\include