

c复习

董洪伟

<http://hwdong.com>

程序：数据 + 处理（运算）

- 数据： 变量和常量

`int i = 3;` → 初始化式

- 变量需要定义：

类型 变量名；

类型 变量名 = (变量或常量);

`double pi, r=3.45, area;`

`char var='A';`

类型 变量1, 变量2, 变量3; ←

同时定义多个变量，逗号隔开

运算：用运算符对数据进行处理

$$y = x * y - 2 / y;$$

算术：+ - * / % ++ --

关系：== != < > <= >=

逻辑：&& || !

位运算：& | ^

赋值：= *= %= <<= &= |=

其他：sizeof ?: (条件) & (取地址) * (取内容)

表达式：运算符+数据

- 表达式的结果也是数据：表达式嵌套

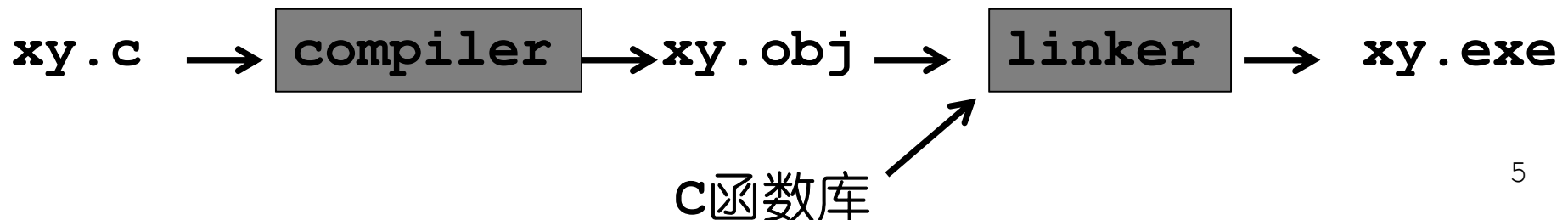
$$y = (x * y - 2 / y);$$

- 语句：以分号结尾的表达式
- 函数：以分号结尾的表达式

一个简单程序: $z=x+y$:

```
/* calculate z= x+y */  
#include<stdio.h>  
  
int main()  
{  
    int x,y =40;  
    int z = x+y;  
    printf("x+y=:%d",z);  
}
```

← 注释: 解释程序的功能
← 包含头文件: 函数的定义等
← 程序的主函数
← 两个输入变量 x, y
← 输出变量 z 等于表达式 $x+y$ 的值
← 函数调用表达式



编程环境

- **visual studio 2015**
- **Code blocks**
- **gcc/g++**

```
>gcc -o xy xy.c
```

```
>xy
```

```
>cl xy.c /out xy
```

```
>xy
```


$z = x + y$: $xy.c$

```
/* calculate  $z = x + y$  */  
#include <stdio.h>
```

```
int main() {
```

```
    int x, y = 40;
```

```
    int z = x + y;
```

```
    printf("x+y=: %d", z);
```

```
}
```

表达式: $y = 50$ $x + y$

$z = x + y$ `printf(...)`

常量

三个整型变量:

x, y, z 在内存中各有一块独立的空间 (4个字节)

x

?

y

40

z

?

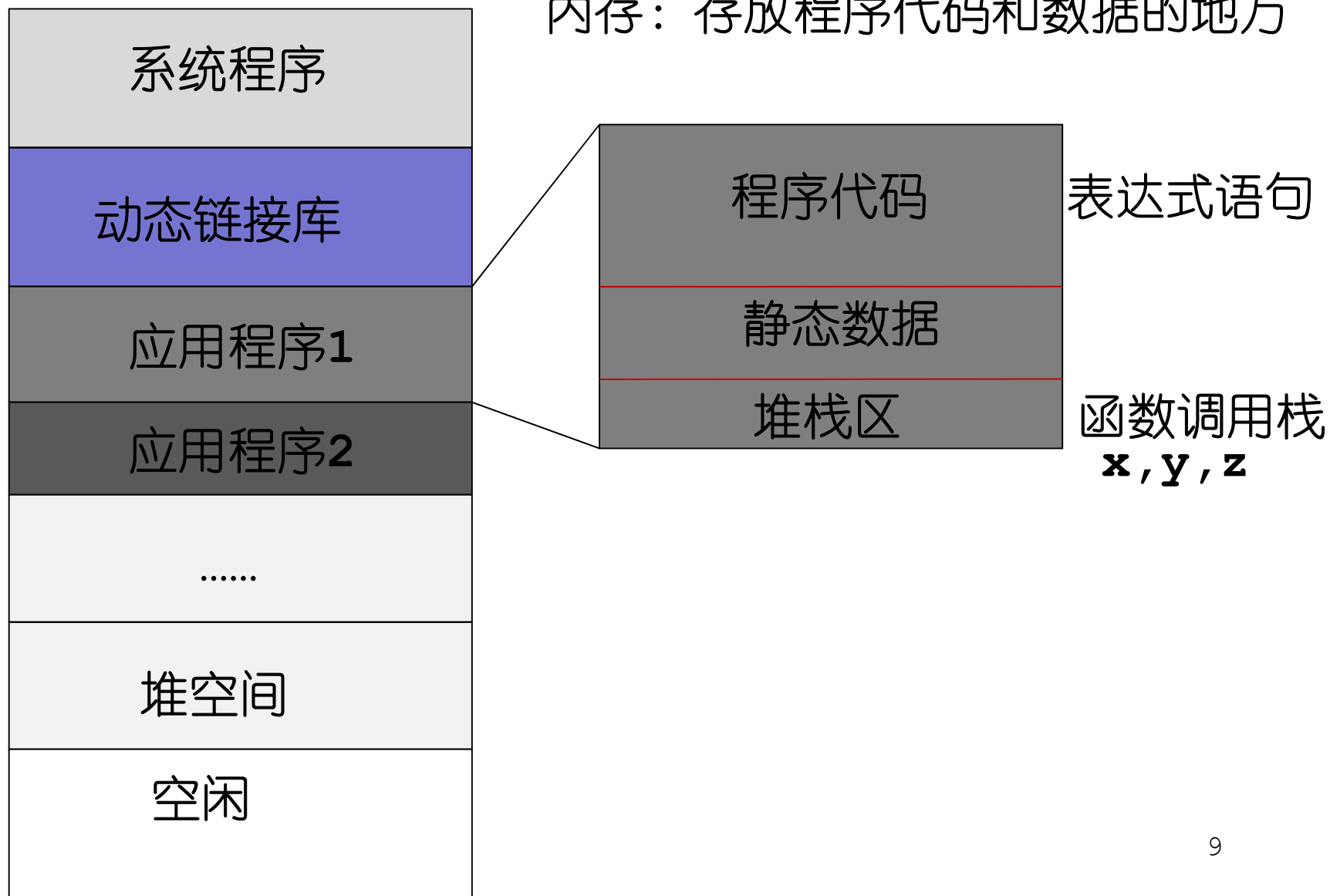
三个表达式语句

表达式: 变量、常数和运算符构成

语句: 后跟 ';' 的表达式

程序内存布局

内存：存放程序代码和数据的地方



程序错误

- **语法错误**：编译错误或链接错误
编译器和连接器会告诉我们错误信息！
- **逻辑错误**：运行的结果和预想的不一致！

```
int main() {  
    int x,y =40;  
    int z = x+y;  
    printf("x+y=:%d",z);  
}
```

该程序编译链接没有问题，但输出结果有问题-- 逻辑错误！
因为**x**没有初始化！

如何发现逻辑错误？

- **方法1：** 输出程序运行过程中的一些数据或信息。如**printf**

```
int main() {  
    int x,y =40;  
    int z = x+y;  
    printf("x+y=:%d",z);  
}
```

- **方法2：** 利用**IDE**开发环境提供的调试功能，如断点调试、单步调试、进入函数...

变量及其类型

- **类型**：规定了该类型变量占内存大小、值的取值范围、对这种类型变量能进行说明操作？
- **变量**：存储一个类型值的空间

int a = 3; ← 初始化

↑ ↙

整型 整型的一个变量

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 byte	1.2E-38 到 3.4E+38	6 位小数
double	8 byte	2.3E-308 到 1.7E+308	15 位小数
long double	10 byte	3.4E-4932 到 1.1E+4932	19 位小数

sizeof 运算符

```
main(){
    int a;
    printf("a或int类型占用内存大小: %d,%d",
           sizeof(a), sizeof(int));
}
```

类型规定了值和操作

- **bool**的值: **true** , **false**
- **bool**的操作 **&&**, **||**, **!**
- **推论**: 运算符对同类型 (或能转换为同类型) 的变量进行运算

自动类型转换

```
#include <stdio.h>
int main() {
    char ch = 'a'; //字符类型, 单引号括起的'a'表示字符a
    int a = 3, b;  //整型变量
    printf("%c\t%d\t\n", ch, ch);
    b = a + ch;
    ch = a;
    printf("%c\t%d\t\n", ch, b);
    return 0;
}
```

强制类型转换

- 形式： (类型名) 变量名

- 如：

```
double d1 = 34.56;
int i = (int)d1;
printf("%f\n", d1);
printf("%d\n", i);
```

- 输出：

```
34.560000
34
```


内在类型和用户定义类型

- 内在类型包含:

基本类型: `int, float, char, ...`

数组类型: `int A[10]`

指针类型: `int *p;`

- 用户定义类型: 枚举`enum`, 结构`struct`, ...

```
enum RGB{red, green, blue};
```

```
struct student{  
    char name[30];  
    float score;  
};
```

访问结构成员

```
struct student s;  
strcpy(s.name, "LiPin");  
s.score = 78.8;
```

变量指针与指针变量

- **变量指针**: 变量的地址, 用 **&** 运算符获取
- **指针变量**: 存放指针的变量. 用 ***** 可以获取指针变量指向的那个变量.

```
int i = 30;
```

```
int *j = &i; //j是存放整型变量指针的指针变量
```

```
int k = *j;    //即k=i=30
```

```
*j = 35;    //即i=35
```

双斜杠//表示的是单行注释

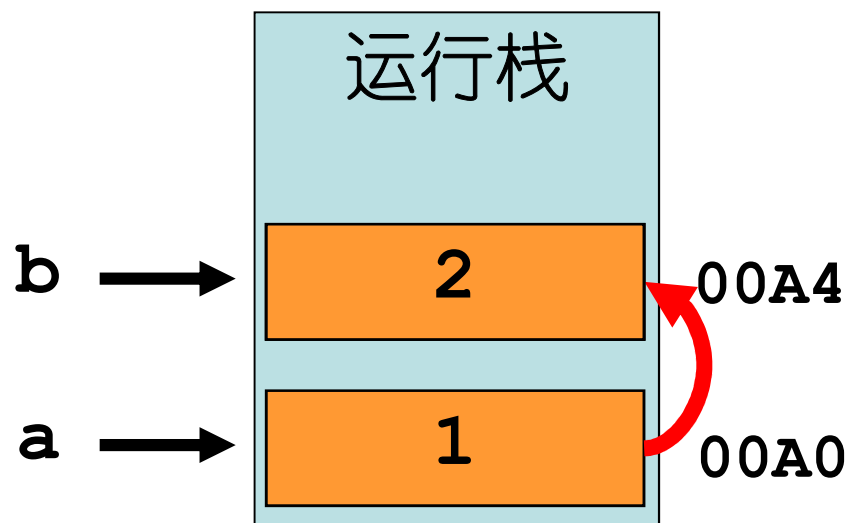
通过结构指针访问结构成员

```
struct student s;  
strcpy(s.name, "LiPin");  
s.score = 78.5;  
struct student *sp = &s;  
sp->score = 90.5;  
(*sp).score = 60;
```

值类型与引用类型

- C语言只有值类型

- 直接盛放自身数据
- 每个变量都有自身的值的一份拷贝
- 对一个值的修改不会影响另外一个值

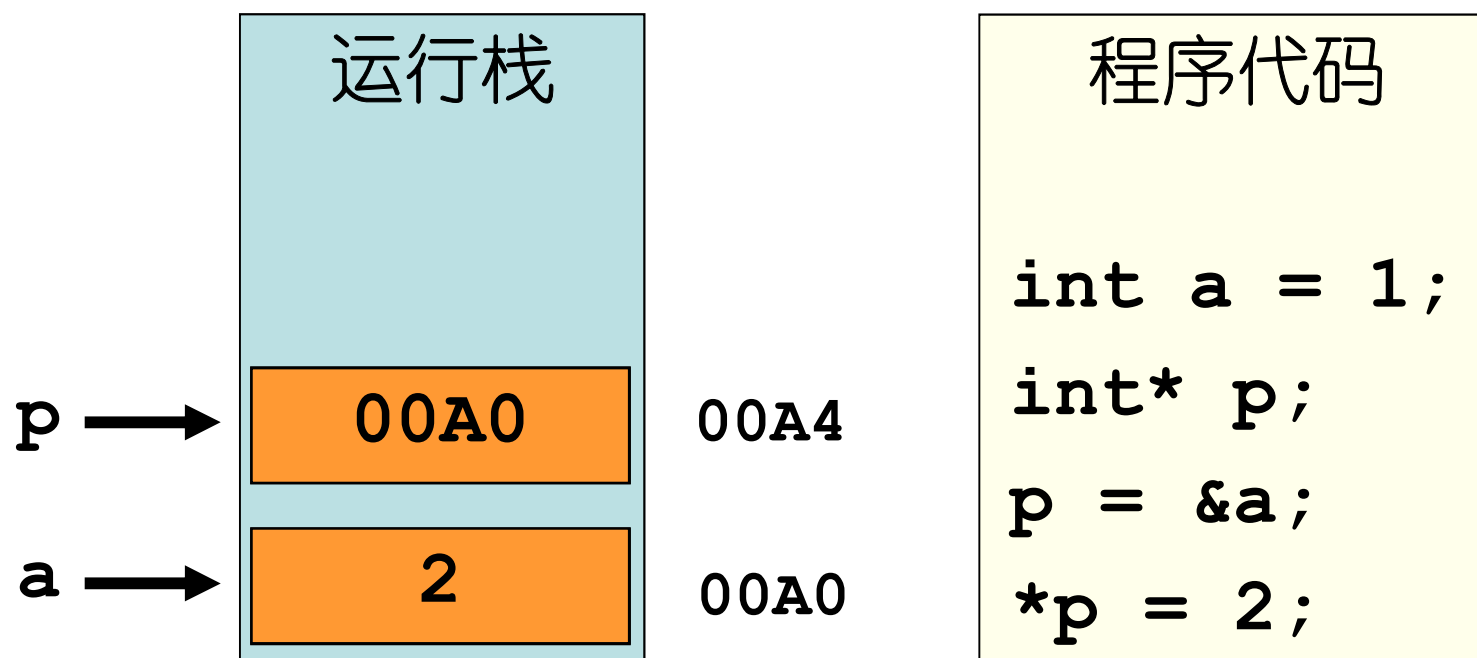


程序代码

```
int a = 1;  
int b;  
b = a;  
b = 2;
```

值类型与引用类型

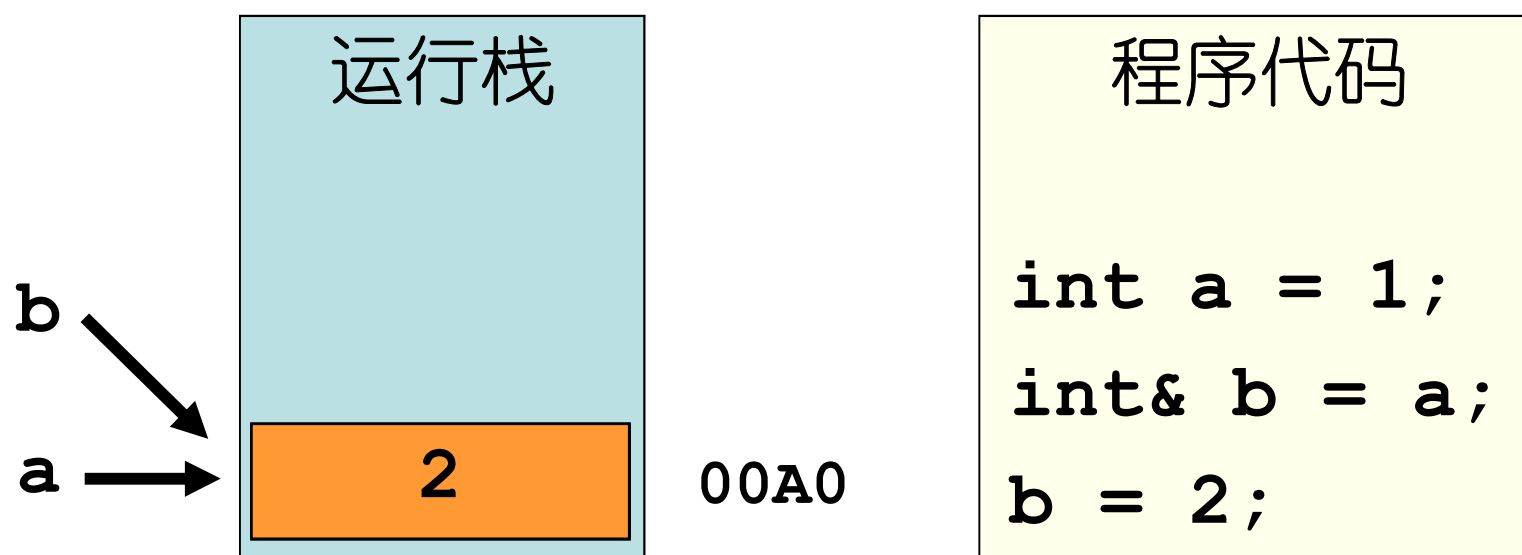
- 指针类型 (也属于值类型)
 - 保存的是另外一个变量的内存地址



值类型与引用类型

- C++的引用类型

– 简单理解：一个变量的别名



值类型与引用类型

- 引用变量：

- 1) 引用变量不过是已经存在变量的别名。
- 2) 既然是引用变量,定义时就必须初始化它
- 3) 一旦定义,就不能在修改引用别的变量

```
int a = 3;  
int &b = a;  
int &b = c;  
char &d = a;
```


表达式和语句

- **表达式**：由常量、变量和运算符构成。对数据进行加工
- **语句**：表达式后跟分号。除直接对数据进行运算的语句外，还有程序流程控制语句，如 **if**、**for**、**while**、**switch**等
- **程序块**：一个或多个语句构成，如**if**、**for**、**while**、**switch**或{ }等。函数就是一个命名的程序块

程序块

```
void main() {
```

```
    int x=3,y=4;
```

```
    {
```

```
        int t = x;
```

```
        x=y;
```

```
        y =t
```

```
    }
```

```
    t++;
```

```
}
```

t是{ }程序块内的局部变量

t是main程序块内的局部变量，但未定义！

函数：命名的程序块

- **函数**：函数名、参数列表、返回值
- **区分函数**：函数名、参数列表

函数名 (C) : 不允许同名函数

函数名+参数列表 (C++) : 允许同名函数，但参数列表必须不同！

```
void swap(int& x, int& y){  
    int t = x;  
    x=y;  
    y = t  
}
```

```
void swap(char& x, char& y){  
    char t = x;  
    x=y;  
    y = t  
}
```

函数：形式参数

- **形式参数**：函数定义中的参数列表中的参数称为形式参数。

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

函数：形式参数

- **形式参数**：函数定义中的参数列表中的参数称为形式参数。
- **实际参数**：调用函数时提供给该函数的参数称为实际参数。

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

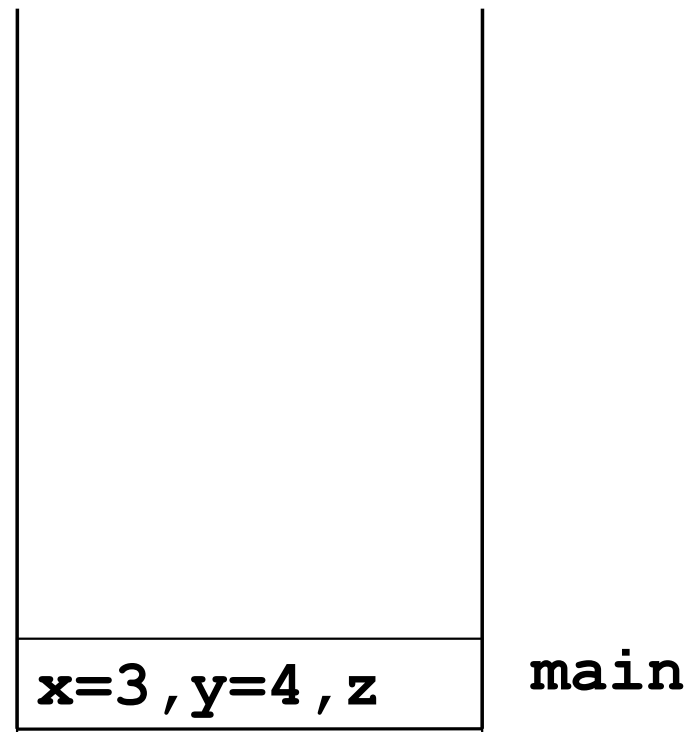
void main()
{ int x=3,y=4;
  int z = add(x,y);
}
```

函数：程序堆栈

- 每个程序有一个自己的堆栈区，用以维护函数之间的调用关系

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

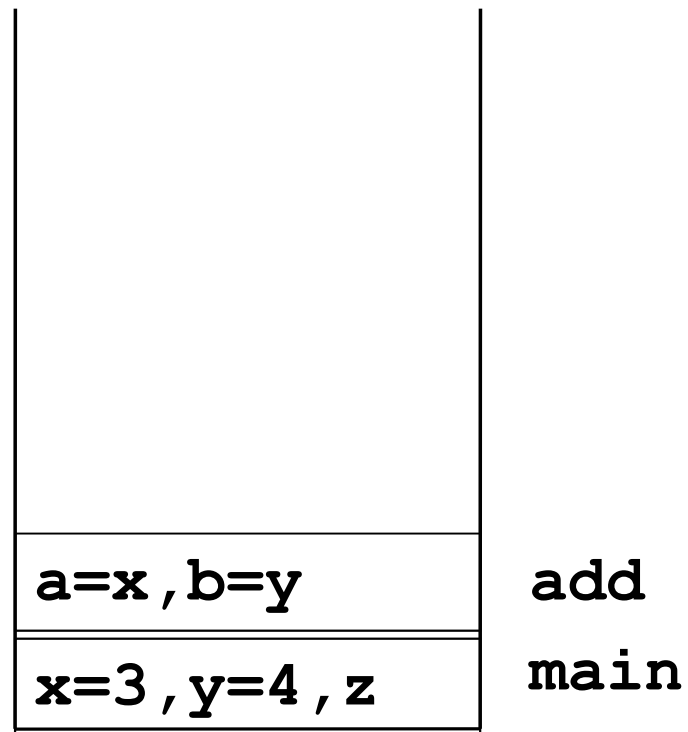
```
void main()
{ int x=3,y=4;  ←
  int z = add(x,y);
}
```



函数：程序堆栈

- 每个程序有一个自己的堆栈区，用以维护函数之间的调用关系

```
int add(int a,int b) ←  
{  
    return a+b;  
}  
  
void main()  
{ int x=3,y=4;  
  int z = add(x,y) ; ←  
}
```

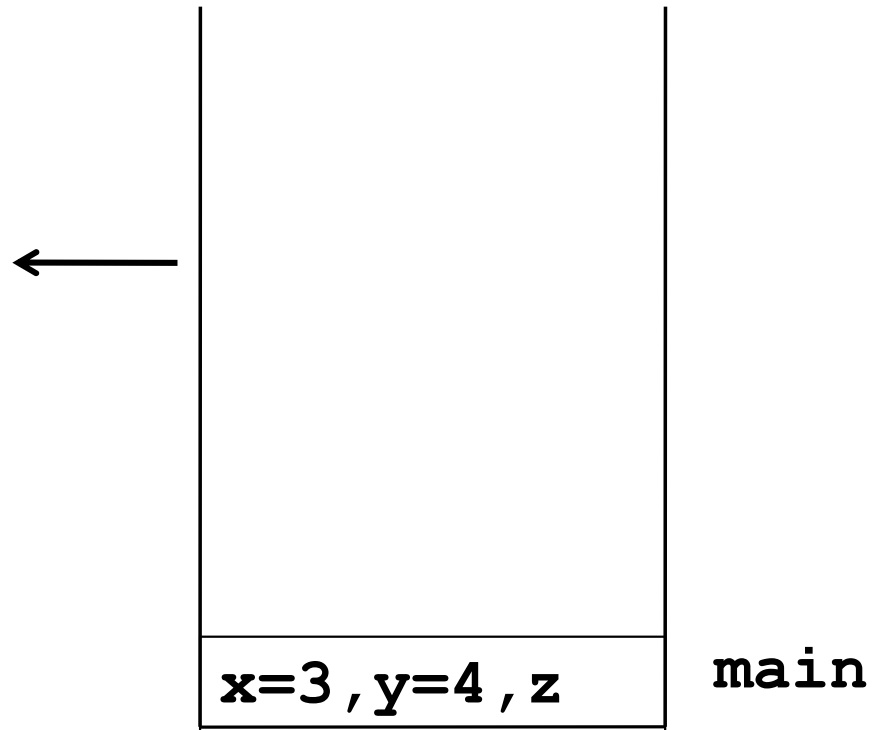


函数：程序堆栈

- 每个程序有一个自己的堆栈区，用以维护函数之间的调用关系

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

```
void main()
{ int x=3,y=4;
  int z = add(x,y);
}
```

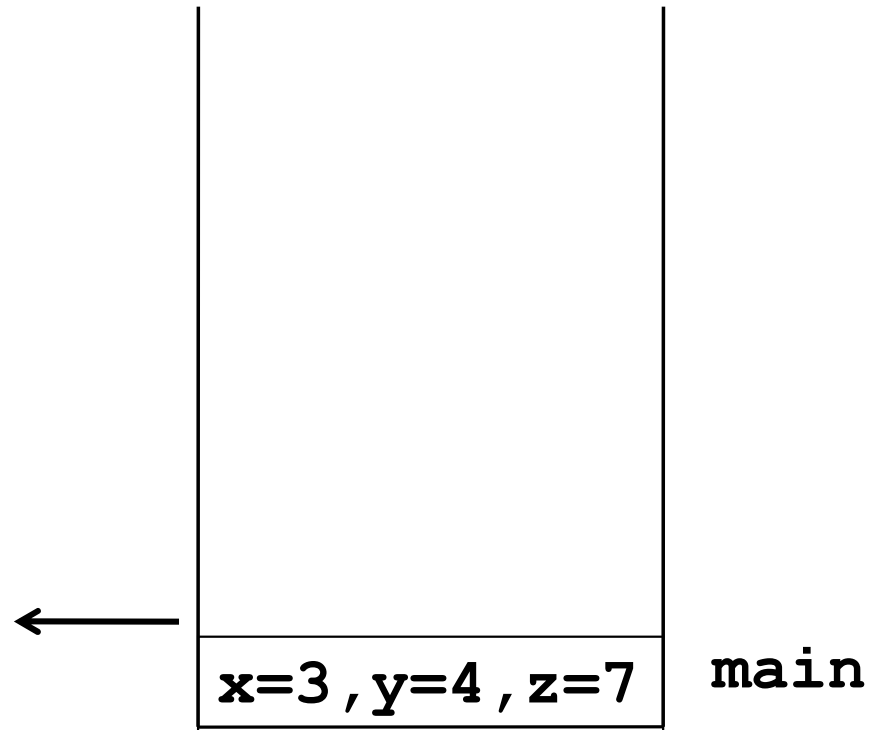


函数：程序堆栈

- 每个程序有一个自己的堆栈区，用以维护函数之间的调用关系

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

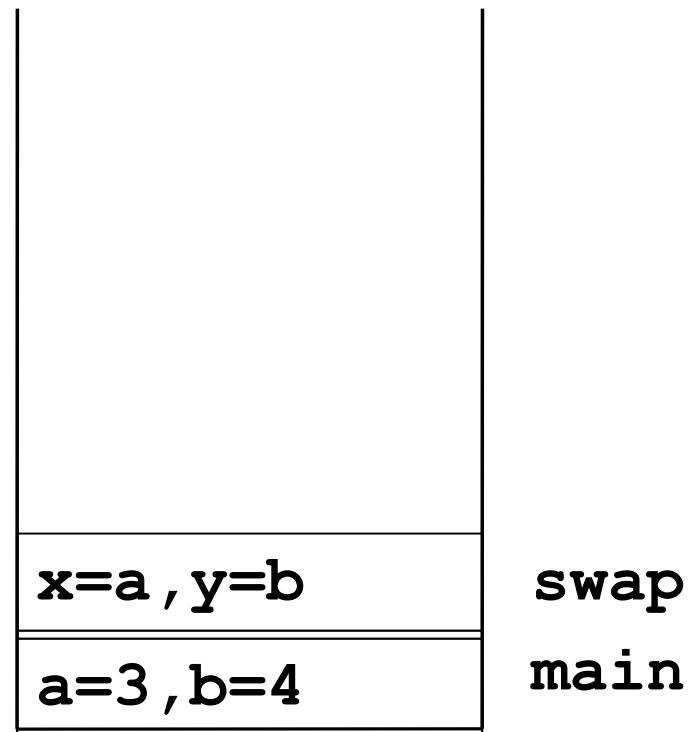
```
void main()
{ int x=3,y=4;
  int z = add(x,y);
}
```



函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

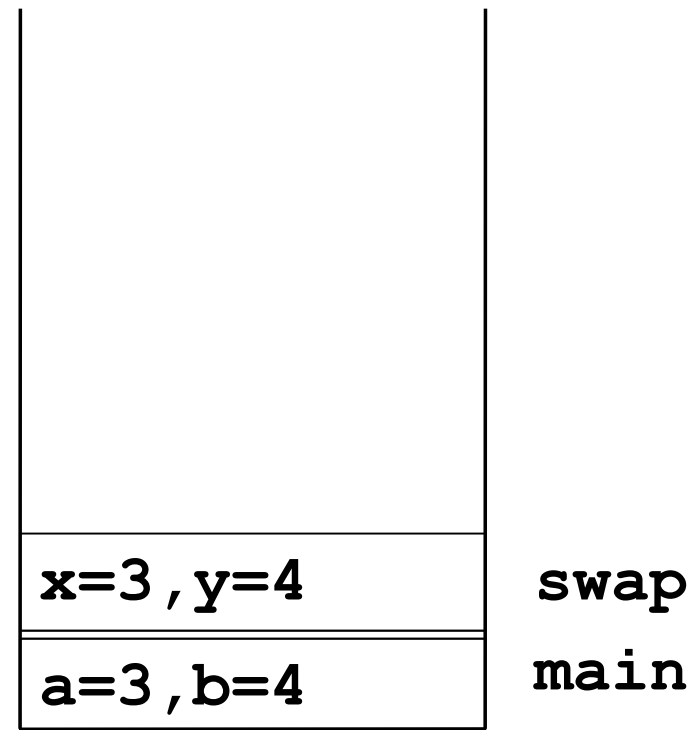
```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a::%d  b::%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```

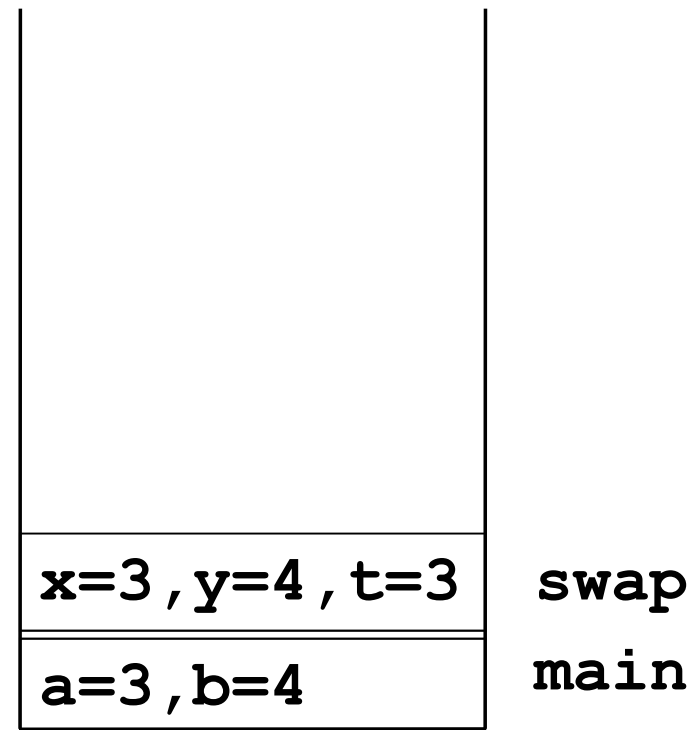


函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```



```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```



```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```

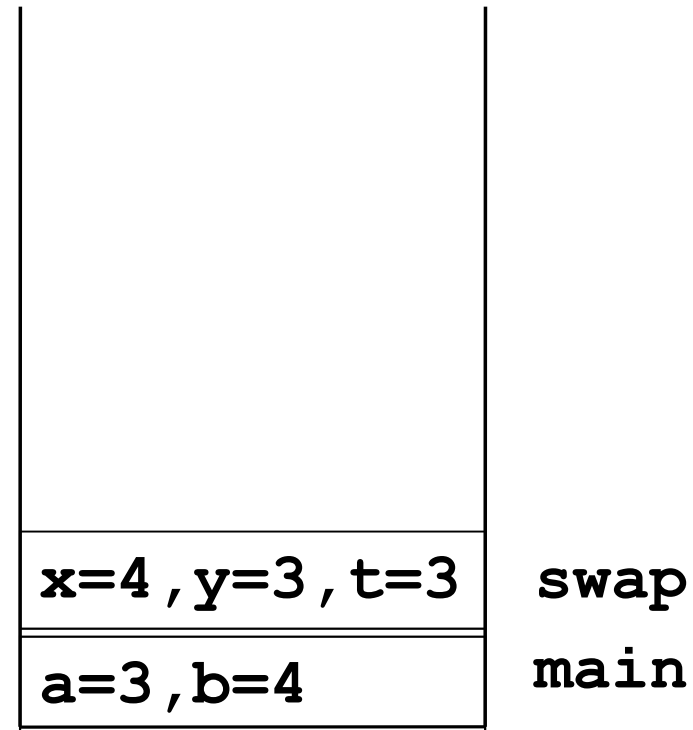
x=4 , y=4 , t=3	swap
a=3 , b=4	main

函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```



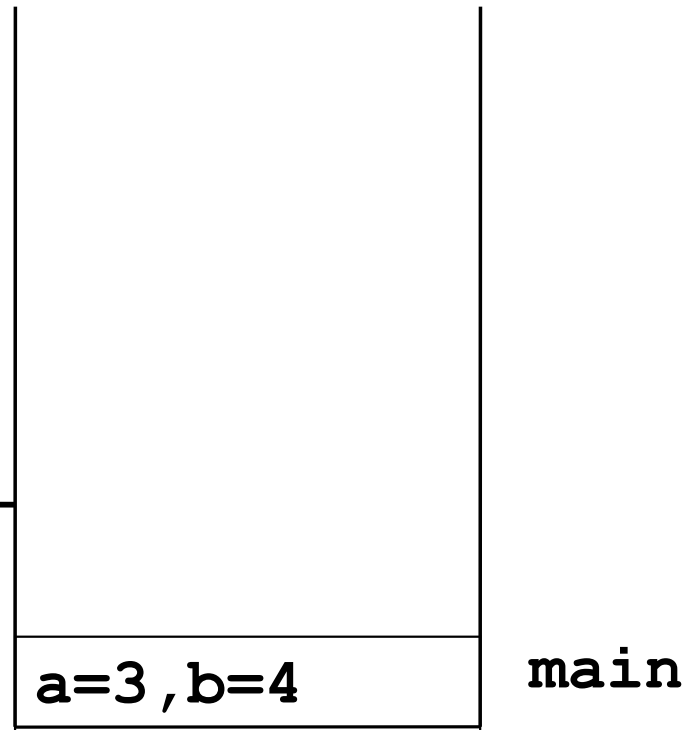
```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

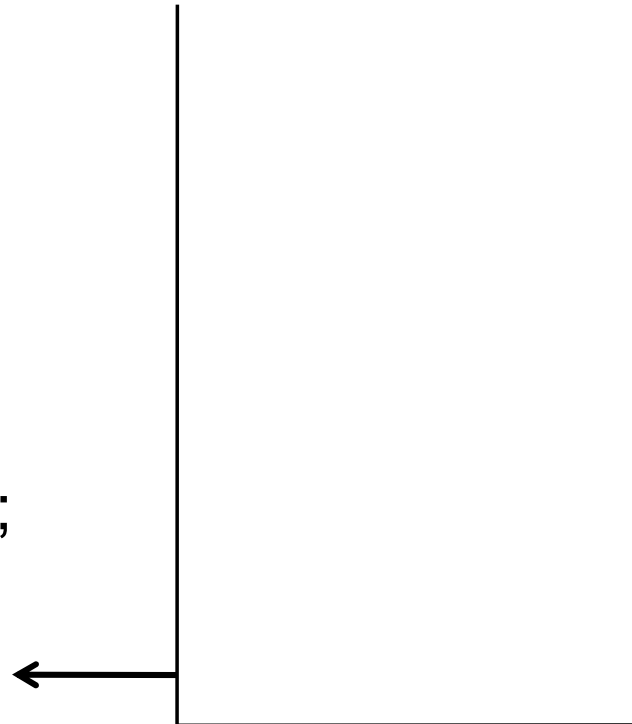
```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a::%d  b::%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int x,int y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

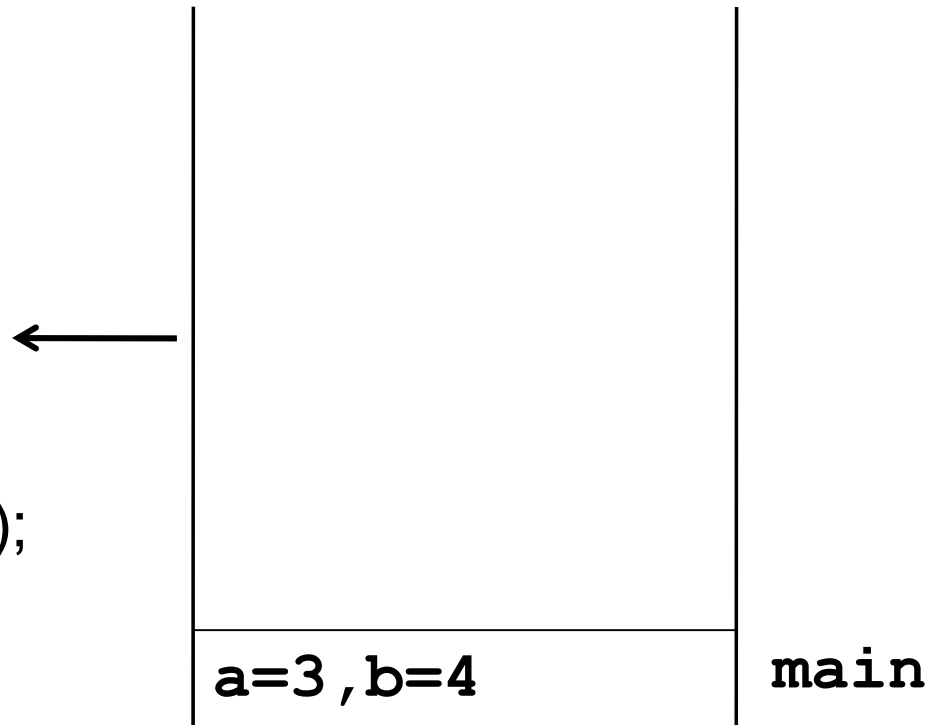
```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```

```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```

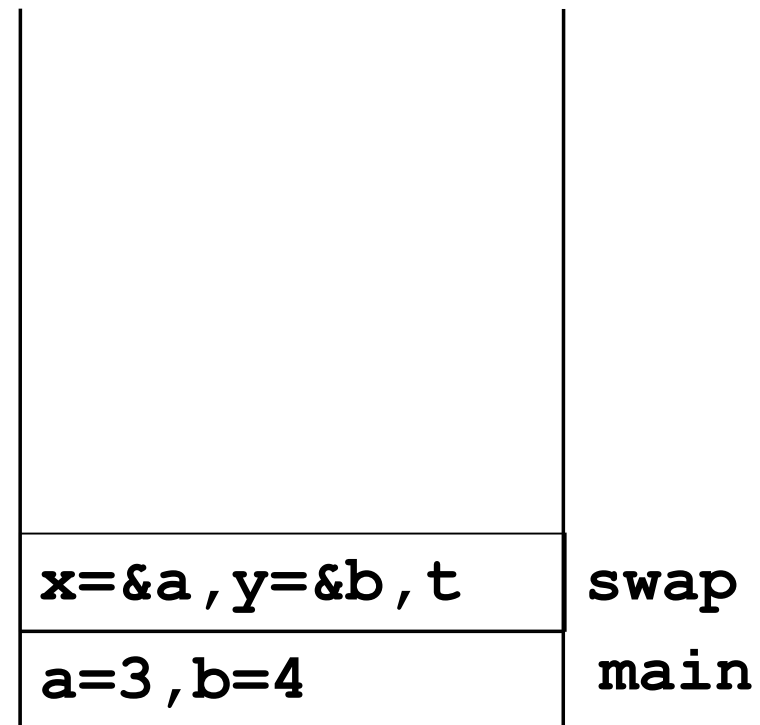


函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```



```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```



函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```



```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d  b=:%d\n",a,b);  
    return 0;  
}
```

x=&a , y=&b , t=3	swap
a=3 , b=4	main

函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```



```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

x=&a , y=&b , t=3	swap
a=4 , b=4	main

函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```



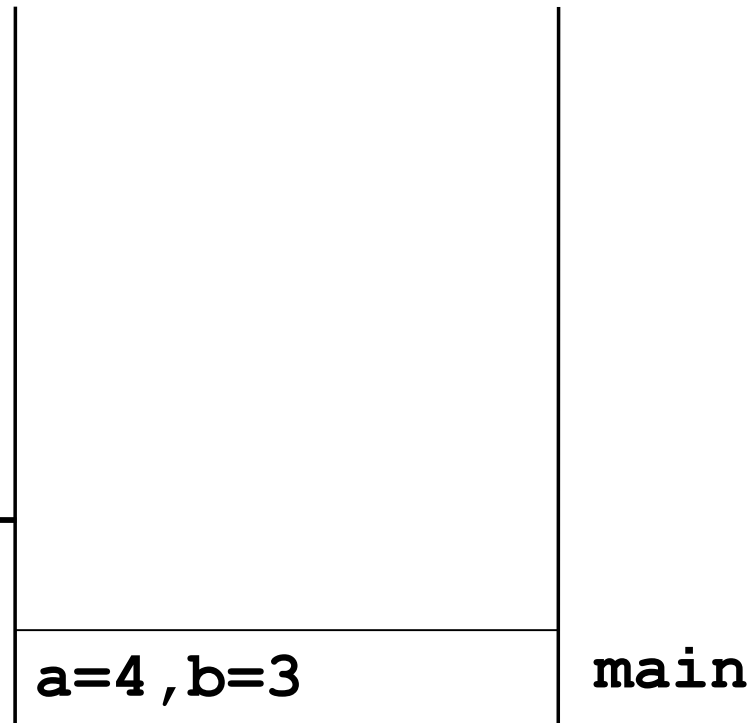
```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

x=&a , y=&b , t=3	swap
a=4 , b=3	main

函数调用：传值

```
void swap(int *x,int *y){  
    int t = *x;  
    *x = *y;  
    *y = t;  
}
```

```
int main(){  
    int a = 3,b= 4;  
    swap(&a,&b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

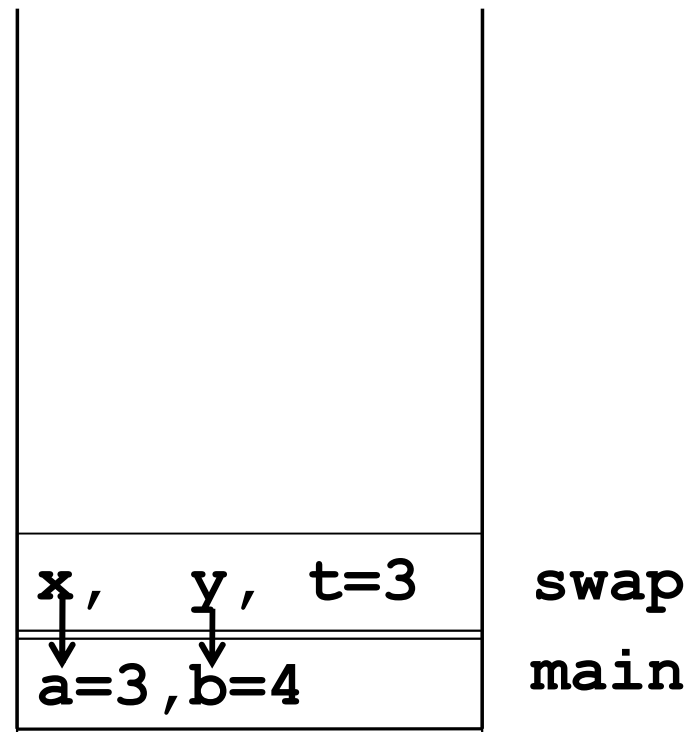


函数调用：传引用

```
void swap(int &x,int &y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

x就是a, y就是b

```
int main(){  
    int a = 3,b= 4;  
    swap(a, b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

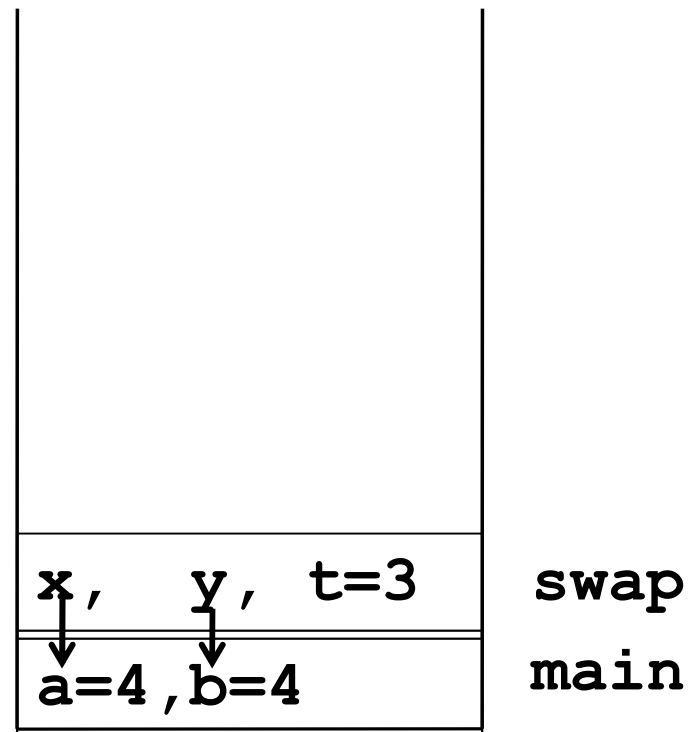


函数调用：传引用

```
void swap(int &x,int &y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

x就是a, y就是b

```
int main(){  
    int a = 3,b= 4;  
    swap(a, b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

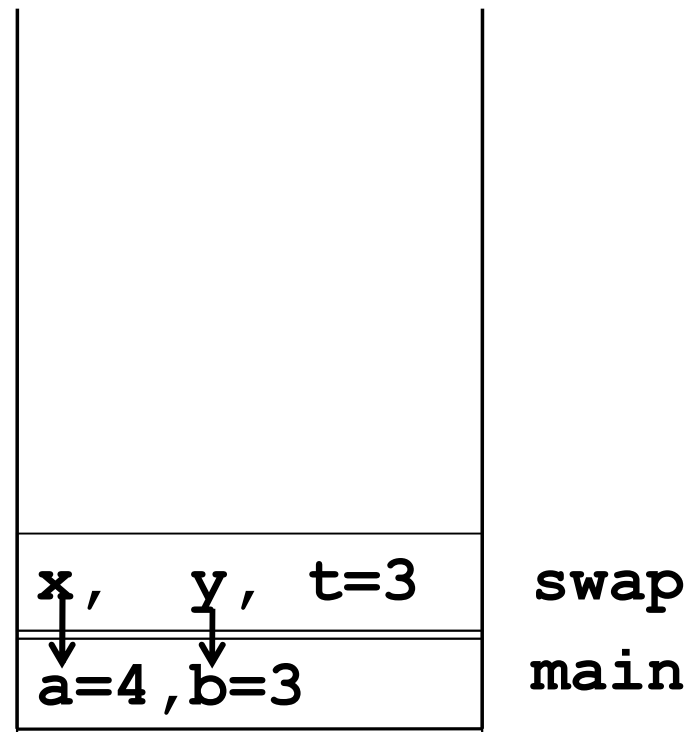


函数调用：传引用

```
void swap(int &x,int &y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

x就是a, y就是b

```
int main(){  
    int a = 3,b= 4;  
    swap(a, b);  
    printf("a=:%d b=:%d\n",a,b);  
    return 0;  
}
```

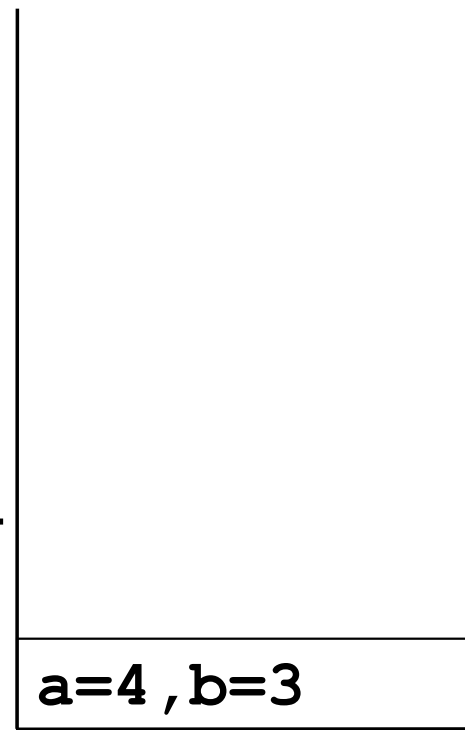


函数调用：传引用

```
void swap(int &x,int &y){  
    int t = x;  
    x = y;  
    y = t;  
}
```

x就是a, y就是b

```
int main(){  
    int a = 3,b= 4;  
    swap(a, b);  
    printf("a=:%d b=:%d\n",a,b); ←  
    return 0;  
}
```



main

函数调用：传引用

- 引用通常作为函数参数和返回值

```
void f(int val, int& ref) {  
    val++;  
    ref++;  
}
```

```
void main() {  
    int x=3, y = 9;  
    f (x, y);  
    printf ("%d   %d\n", x, y);  
}
```

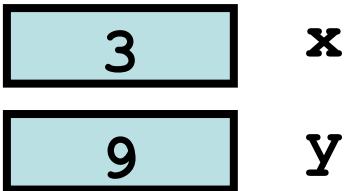


Diagram illustrating the state of variables x and y in memory. Variable x holds the value 3, and variable y holds the value 9. An arrow points from the 'y' parameter in the function call 'f(x, y)' to the 'y' box, indicating that the function is called with the reference to y.

函数调用：传引用

- 引用通常作为函数参数和返回值

```
void f(int val, int& ref) { ←
```

```
    val++;
```

```
    ref++;
```

```
}
```

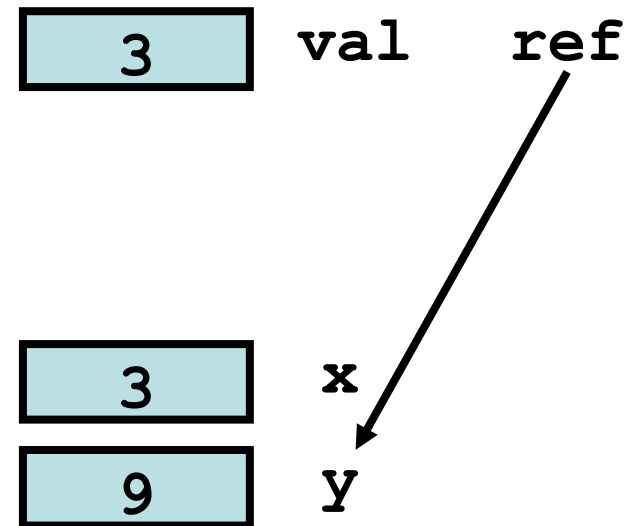
```
void main() {
```

```
    int x=3, y = 9;
```

```
    f (x, y);
```

```
    printf("%d  %d\n", x, y);
```

```
}
```

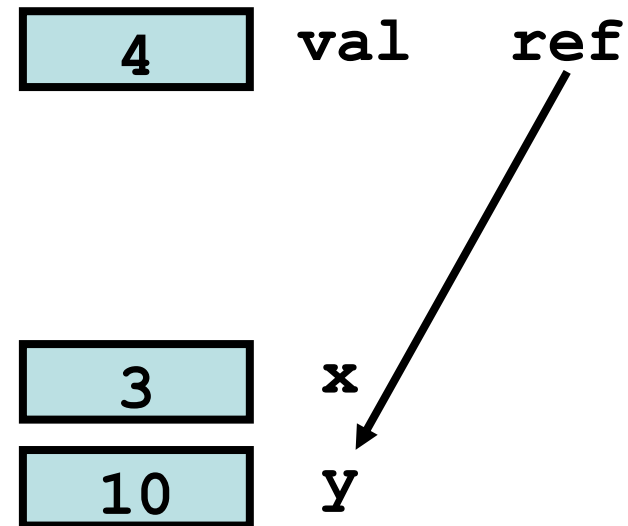


函数调用：传引用

- 引用通常作为函数参数和返回值

```
void f(int val, int& ref) {  
    val++;  
    ref++;  
}
```

```
void main() {  
    int x=3, y = 9;  
    f (x, y);  
    printf("%d  %d\n", x, y);  
}
```

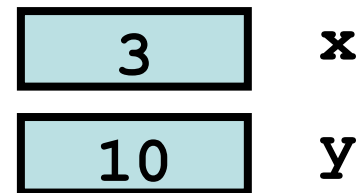


函数调用：传引用

- 引用通常作为函数参数和返回值

```
void f(int val, int& ref) {  
    val++;  
    ref++;  
}
```

```
void main() {  
    int x=3, y = 9;  
    f (x, y);  
    printf ("%d   %d\n", x, y);  
}
```



函数的传值参数和传引用参数

- 传值参数：实参复制到形参

```
void swap(int x,int y) ;
```

- 引用参数：形参是实参的别名

```
void swap(int &x,int &y) ;
```

值类型与引用类型

- 就象不能返回局部变量的指针一样,不能返回局部变量的引用.

```
X& fun(X& a) {  
    X b;  
  
    ...  
  
    return a;    // OK!  
    return b;    //bad!  
}
```


变量作用域(局部、全局、静态)

- 函数内部定义的变量(包括函数参数)称为**局部变量(内部变量)**，其作用域在函数内部。

局部变量随函数执行而产生，函数结束而销毁

```
void f(){  
    int x=0;  
    x++;  
    printf("%d",x);  
}  
  
main(){  
    f();  
}
```

变量作用域(局部、全局、静态)

- 函数外部定义变量称为**全局变量(外部变量)**，其作用域在整个程序。

程序开始执行就产生，程序结束才销毁

```
int y = 0;
void f(){
    int x=0;
    x++;
    y++;
    printf("%d,%d",x,y);
}

main(){
    f();
    f();
}
```

变量作用域(局部、全局、静态)

- 加**static**关键字的变量称为静态变量。

如果是外部变量，则只在其所在文件里有效；

如果是内部变量，则第一次初始化后就始终存在！

```
int y = 0;
void f(){
    static int x=0;
    x++;
    y++;
    printf("%d,%d",x,y);
}

main(){
    f();
    f();
}
```

变量的内存分配

- 内存分配的三种方式
 - 静态存储区分配
 - 栈上创建
 - 堆上分配
- 静态存储区分配 (固定座位)
 - 内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在
 - 例如：全局变量， **static** 变量

内存分配

- 栈上创建 (本部门的保留座位)

- 函数内部的局部变量都在栈上创建，函数执行结束时这些内存自动被释放
- 栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

```
void foo()  
{  
    int    a = 1;  
    float f = 1.0;  
}
```

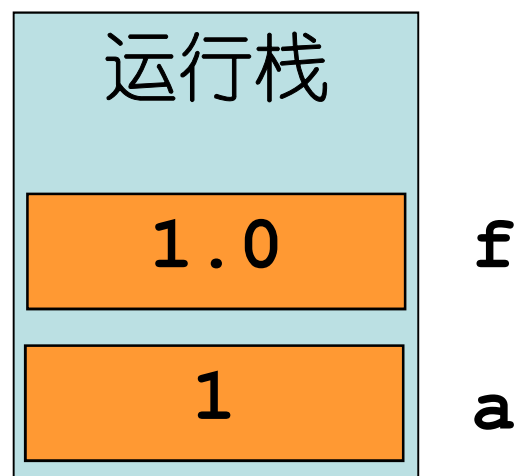
} 这两个变量的内存，
执行到这个函数时自动分配
离开这个函数时自动释放

内存分配

- 栈上创建

- 函数内部的局部变量都在栈上创建，函数执行结束时这些内存自动被释放
- 栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

```
void foo()  
{  
    int    a = 1;  
    float f = 1.0;  
}
```



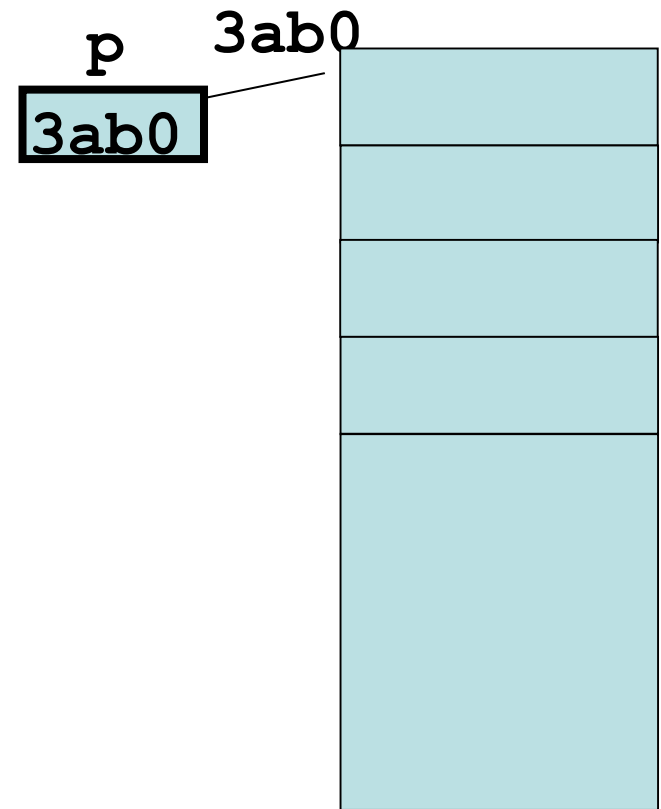
内存分配

- 堆上分配 (公共座位)
 - 亦称动态内存分配
 - 程序在运行的时候用**malloc**或**new**申请任意多少的内存
 - 程序员自己负责用**free**或**delete**释放内存 (否则就会出现内存泄露)
 - 动态内存的生存期由程序员决定, 使用非常灵活, 但问题也最多

内存分配

```
void * malloc(size_t size);
```

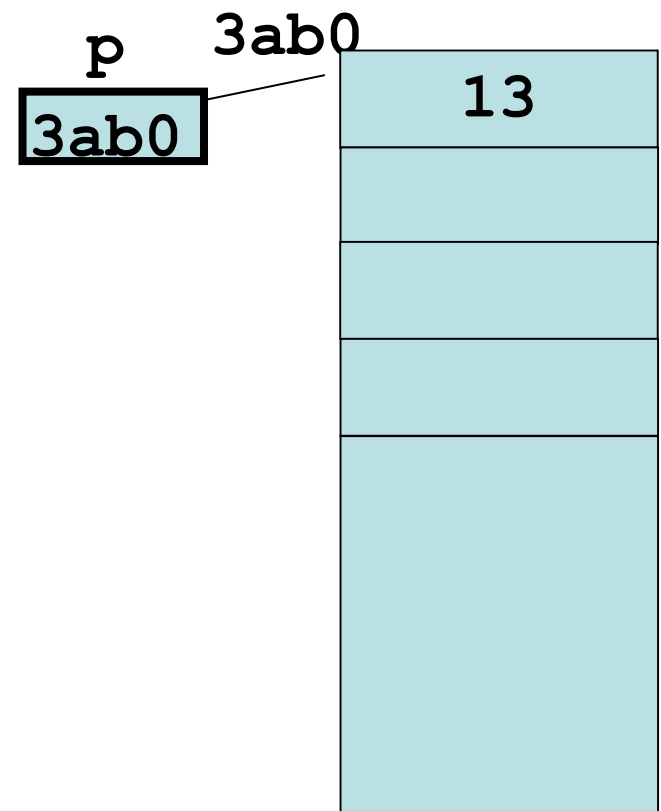
```
int *p = (int* )malloc(10*sizeof(int)) ;
```



内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;
```

```
p[0]=13;
```

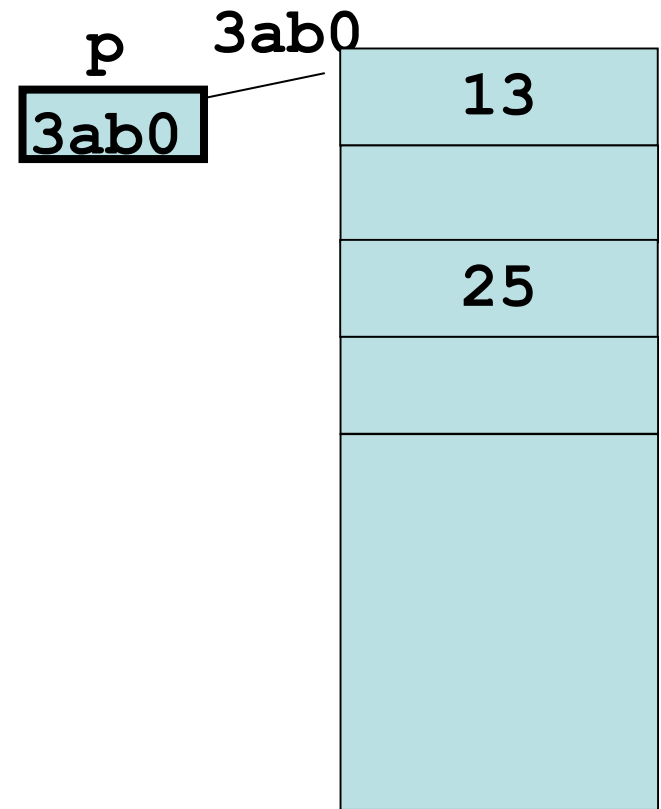


内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;
```

```
p[0]=13;
```

```
p[2]=25;
```



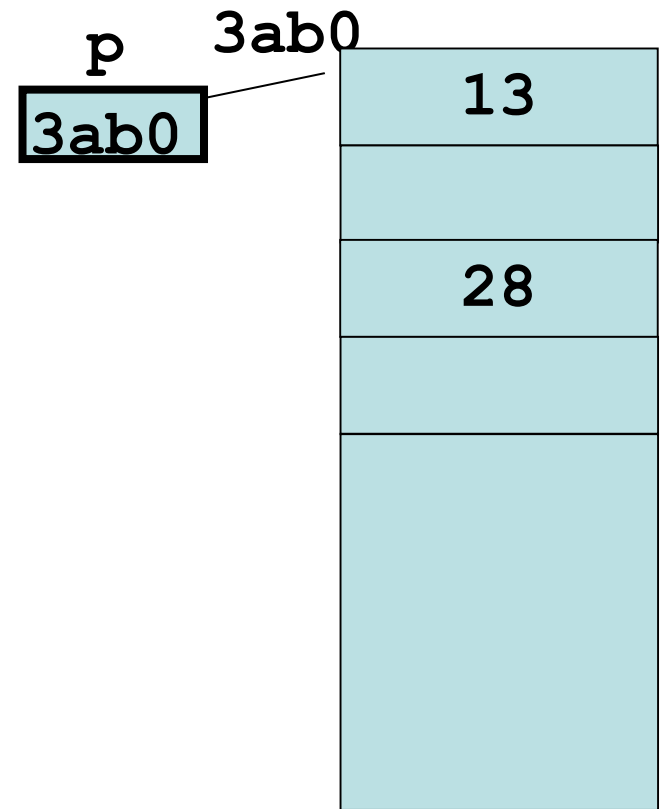
内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;
```

```
p[0]=13;
```

```
p[2]=25;
```

```
*(p+2) = 28;
```



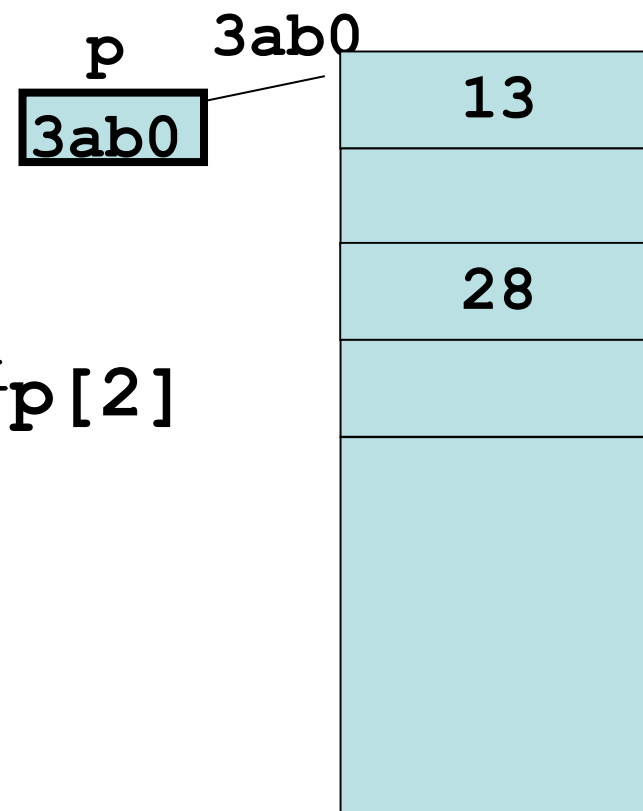
内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;
```

```
p[0]=13;
```

```
p[2]=25;
```

```
*(p+2) = 28; // *(p+2) 等于 p[2]
```



内存分配

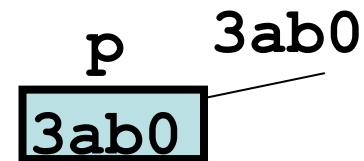
```
int *p = (int *)malloc(10*sizeof(int)) ;
```

```
p[0]=13;
```

```
p[2]=25;
```

```
*(p+2) = 28; // *(p+2) 等于 p[2]
```

```
free(p);
```



内存分配 new delete

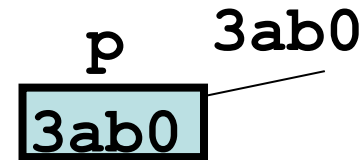
```
int *p = (int *)malloc(10*sizeof(int)) ;  
//int *p = new int[10];
```

```
p[0]=13;
```

```
p[2]=25;
```

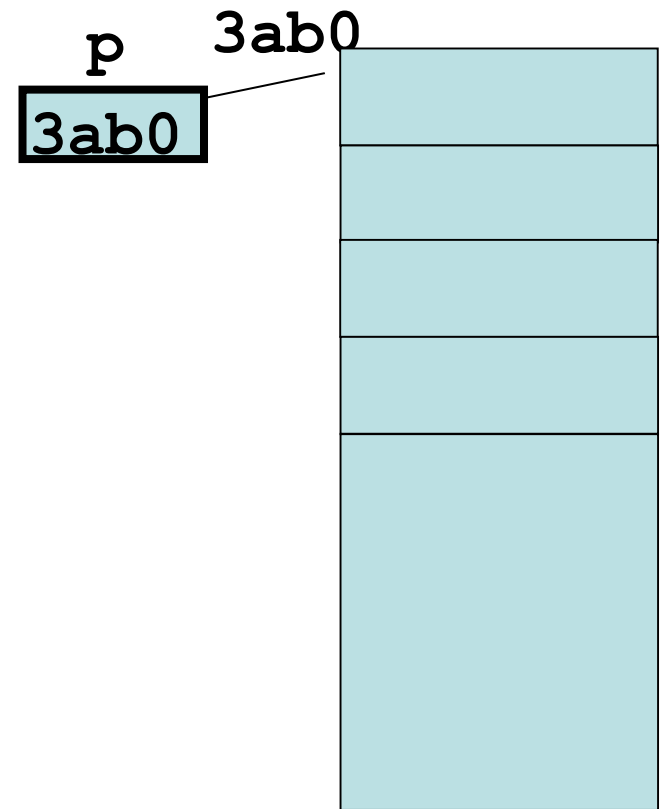
```
*(p+2) = 28; // *(p+2) 等于 p[2]
```

```
free(p); //delete[] p;
```



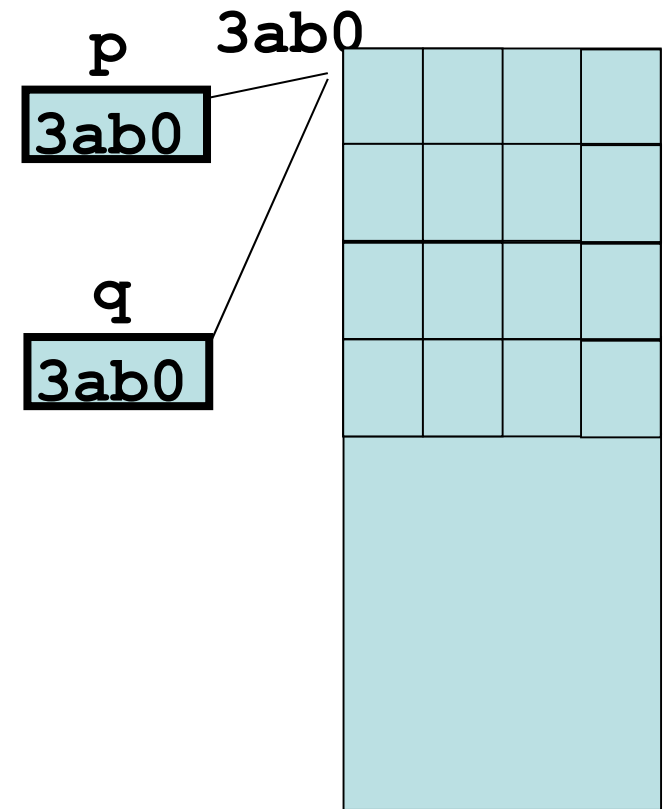
内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;
```



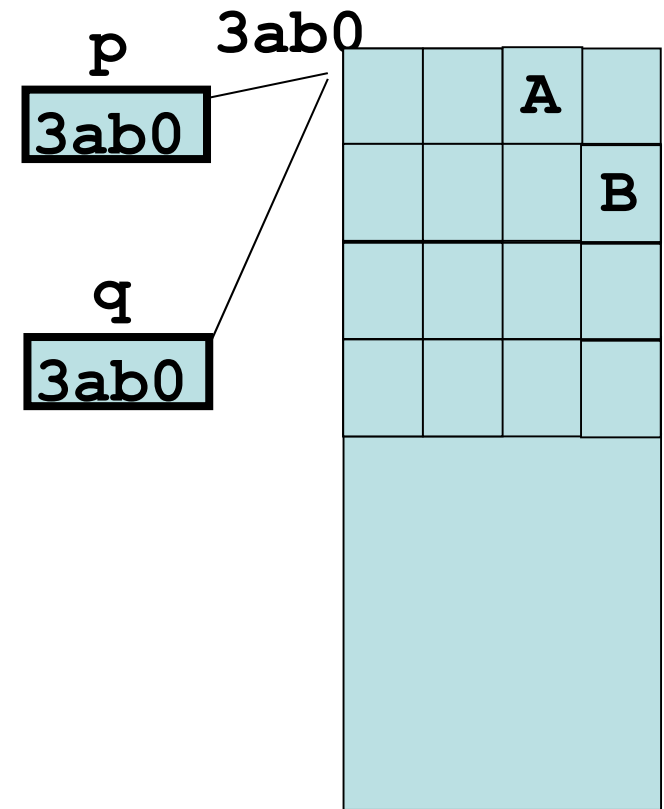
内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;  
char *q = (char *)p;
```



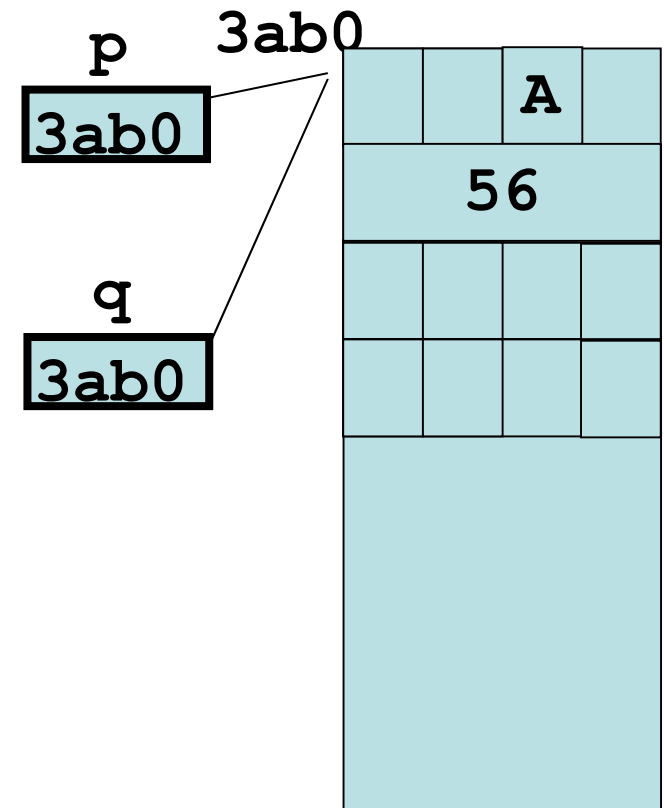
内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;  
char *q = (char *)p;  
q[2] = 'A' ;  
q[7] = 'B' ;
```



内存分配

```
int *p = (int* )malloc(10*sizeof(int)) ;  
char *q = (char *)p;  
q[2] = 'A' ;  
q[7] = 'B' ;  
P[1] = 56
```



指针、数组、字符串

```
main(){
    int a[10] , *p;
    for(int i = 0 ; i<10;i++)
        a[i] = 2*i+1;
    p = a;
    for(int i = 0 ; i<10;i++)
        p[i] = a[i] +3;

    for(int i = 0 ; i<10;i++)
        printf("%d,%d\n", *(p+i), *(a+i));
}
```

数组名就是指向数组第一个元素的指针

指针、数组、字符串

字符指针存储的是一个字符的内存地址

```
main(){
    char ch,*p;
    p = &ch;    //p存储的是ch的地址
    *p = 'A';    //*p就是ch
    printf("%c\t%c\n",*p,ch) ;
}
```

字符指针存储的是字符串第一个字符的内存地址

```
main(){
    /*p存储的"hello world!"第一个字符h的地址*/
    char *p = "hello world!";
    *p = 'A';    /*p就是"hello world!"第一个字符的内存地址*/
    printf("%s",*p) ;    // %s是表示打印p指向的字符串
}
```

字符串： 结束字符'\0'结尾的字符数组

```
#include <stdio.h>
#include <string.h>
main(){
    /*字符数组，但不是C语言规定的字符串*/
    char s[5] = {'h','e','l','l','o'};
    /* C语言规定的字符数组 */
    char t[5] = {'h','e','l','l','o','\0'};
    int len1,len2;
    printf("%s\n",s) ;
    printf("%s\n",t) ;
    len_s = strlen(s);
    len_t = strlen(t);
    printf("%d,%d\n",len_s,len_t);
}
```

字符串： 结束字符'\0'结尾的字符数组

```
#include <stdio.h>

int Strlen(char *s){
    char *p = s;
    while( *p!='\0' ) p++;
    return p - s;
}

main(){
    /*字符数组，但不是C语言规定的字符串*/
    char s[5] = {'h','e','l','l','o'};
    /* C语言规定的字符数组 */
    char t[5] = {'h','e','l','l','o','\0'};

    len_s = Strlen(s);
    len_t = Strlen(t);
    printf("%d,%d\n",len_s,len_t);
}
```

typedef

- 格式

- **typedef** [原类型] [新类型];
- 比如: **typedef int ElemType;**

- 作用

- 定义一个新的类型叫做[新类型], 就等价于[原类型]
- 上例中, 定义了一个**ElemType**类型, 就是**int**类型

typedef

- 如何理解

- 如:

```
typedef int A; //A就是int  
A a;           //相当于int a;
```


结构和typedef的结合使用

- 无名结构

- 定义结构的时候也可以不要结构名，而用这个无名结构直接来定义变量，比如

```
struct{  
    string name;  
    int     age;  
}LiMing;
```

- 这时候这个结构只有**LiMing**这一个变量，它不可能再去定义其它变量，因为它没有名字

结构和typedef的结合使用

- 结构和**typedef**的结合使用

– 例如课本P22有如下代码：

```
typedef struct{  
    ElemType *elem;  
    int      length;  
    int      listsize;  
} SqList;
```

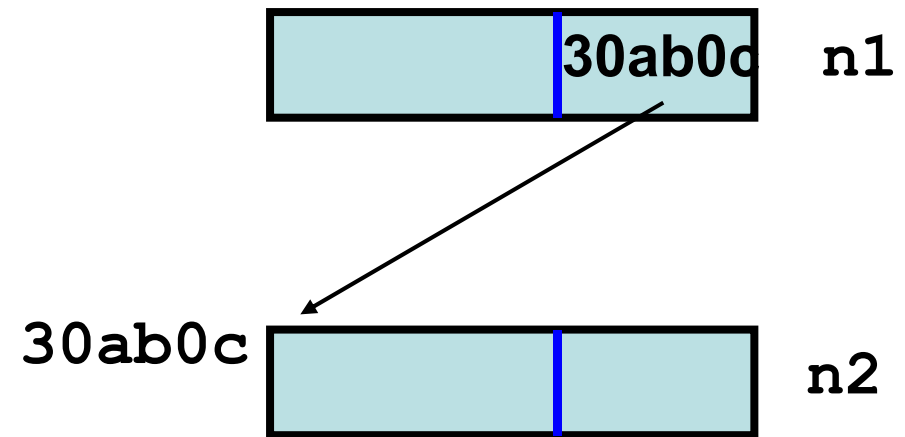
红色部分就是一个无名结构；
SqList就是这个无名结构的别名！

```
SqList L; //定义了SqList类型的一个变量L  
          //变量L有3个成员变量  
          //L.elem , L.length, L.listsize
```

结点和链表

```
typedef struct lnode{  
    double data;  
    struct lnode *next;  
} LNode;
```

```
LNode n1,n2;  
n1.next = &n2;
```



程序例子：读写学生成绩

- 输入：一组学生成绩（姓名、分数）
- 输出：这组学生成绩并统计及格人数
- 数据结构：
 定义学生类型，用数组存储学生成绩数据。
- 数据处理：
 键盘读入、存储、统计计算、输出

struct student

```
typedef struct{  
    char name[30];  
    float score;  
} student;
```

student: code

```
int main() {
    student stus[100];
    int i = 0, j = 0, k=0 ;
    do{ scanf("%s", stus[i].name) ;
        scanf("%f", &(stus[i].score)) ;
        if(stus[i].score>=60) j++;
    }while( ++i<100 && stus[i].score>=0) ;
    for(k=0; k<i; k++){
        printf("name:%s  score:%3.2f\n",
            stus[k].name, stus[k].score) ;
    }
    printf("num of passed:%d\n", j) ;
}
```

In_student, Out_student

```
void In_student (student &s) {  
    scanf ("%s", s.name) ;  
    scanf ("%f", &(s.score) ) ;  
}  
  
void Out_student(const student s) {  
    printf("name:%s    score:%3.2f\n",  
          s.name, s.score) ;  
}
```

student: code2

```
int main() {
    student stus[100];
    int i = 0, j = 0, k=0 ;
    do{
        In_student(stus[i]);
        if(stus[i].score>=60)    j++;
    }while(++i<100 && stus[i].score>=0) ;

    for(k=0; k<i; k++)
        Out_student(stus[k]) ;

    printf("num of passed:%d\n",j) ;
}
```


静态数组: 浪费空间和空间不够

```
student stus[100];
```

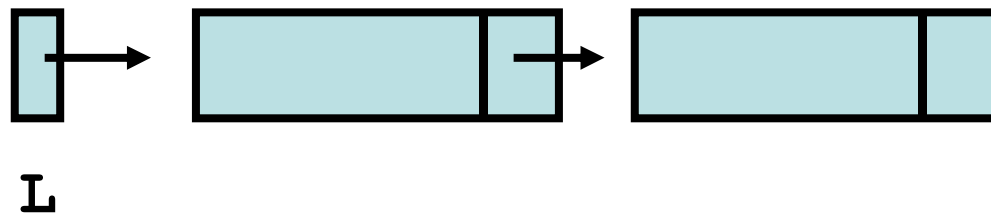
- 解决方法1:

动态分配数组空间

```
student *stus = (student *)  
                malloc(SIZE*sizeof(student));
```

- 解决方法2:

动态分配单个**student**, 并用链表串起来



动态分配数组空间

```
const int INITSIZE = 33;  
const int INC = 30;  
int SIZE = INITSIZE;  
student *stus = (student *)malloc(SIZE  
                                *sizeof(student));
```

当满时:

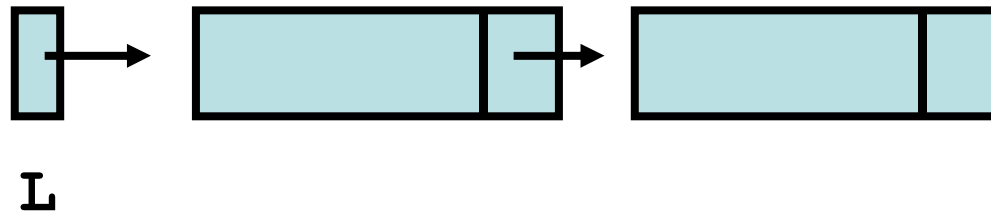
```
SIZE += INC;  
student * stusNew = (student *)  
    realloc(stus ,SIZE*sizeof(student));  
free(stus); //用完要释放空间  
stus = stusNew;
```

student: code3

```
int main() {
    int size = INITSIZE; int i = 0, j = 0, k=0 ;
    student *stus = (student *)malloc(size
                                         *sizeof(student)) ;
    do{
        if(i>=size){ student *stus_new =(student *)
                    realloc(stus, (size+INC)*sizeof(student)) ;
                    free(stus); stus = stus_new;
                    size += INC;      }
        In_student(stus[i]);
        if(stus[i].score>=60)    j++;
    }while(stus[i++].score>=0) ;
    for(k=0;k<i;k++)    Out_student(stus[k]);
    printf("num of passed:%d\n",j); free(stus);
}
```

链表存储

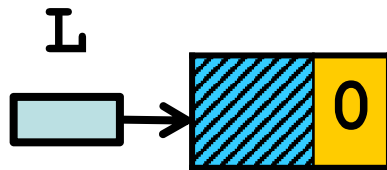
```
typedef struct lnode{  
    student data;  
    struct lnode *next;  
} LNode;  
LNode *L;
```



链表存储：复制student

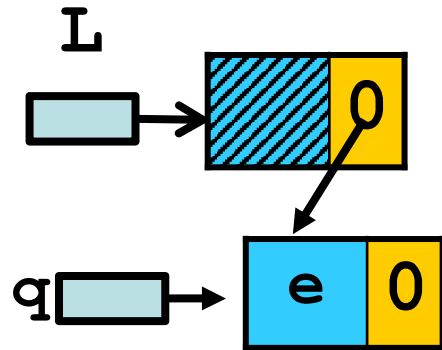
```
void copy_stu(student &d, const student &s){  
    strcpy((char *)d.name, (char *)s.name);  
    d.score = s.score;  
}
```

链表存储：初始化空的头结点



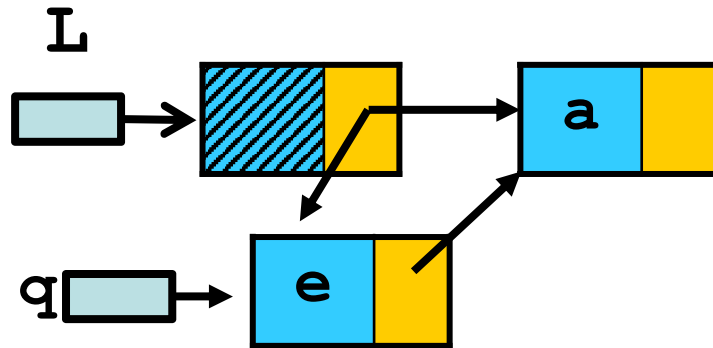
```
LNode *q = 0,  
LNode *L =(LNode *)malloc(sizeof(LNode));  
L->next = 0;
```

链表存储：头结点后插入新结点



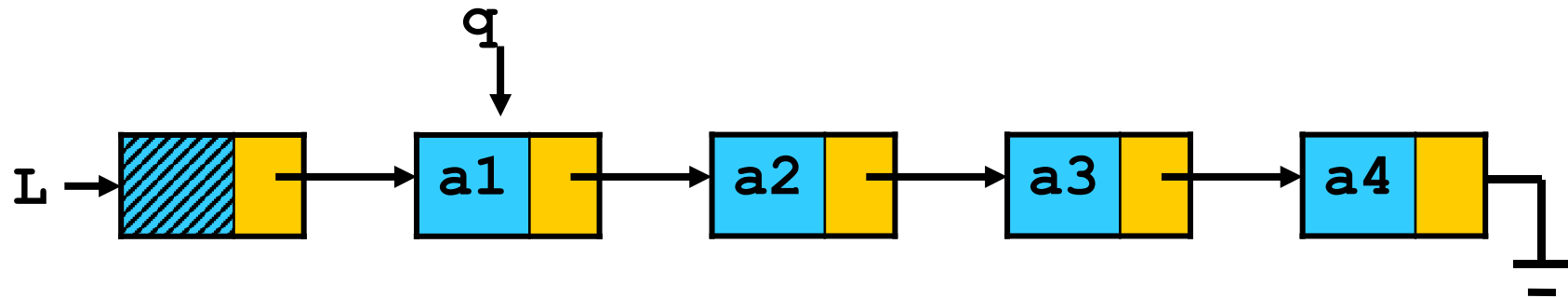
```
q =(LNode *)malloc(sizeof(LNode));  
q->data = e;  
  
q->next = L->next;  
L->next = q;
```

链表存储：头结点后插入新结点



```
q =(LNode *)malloc(sizeof(LNode));  
q->data = e;  
  
q->next = L->next;  
L->next = q;
```


链表存储：遍历链表



```
q = L->next;  
while(q){  
    Out_student(q->data);  
    q = q->next;  
}
```

```

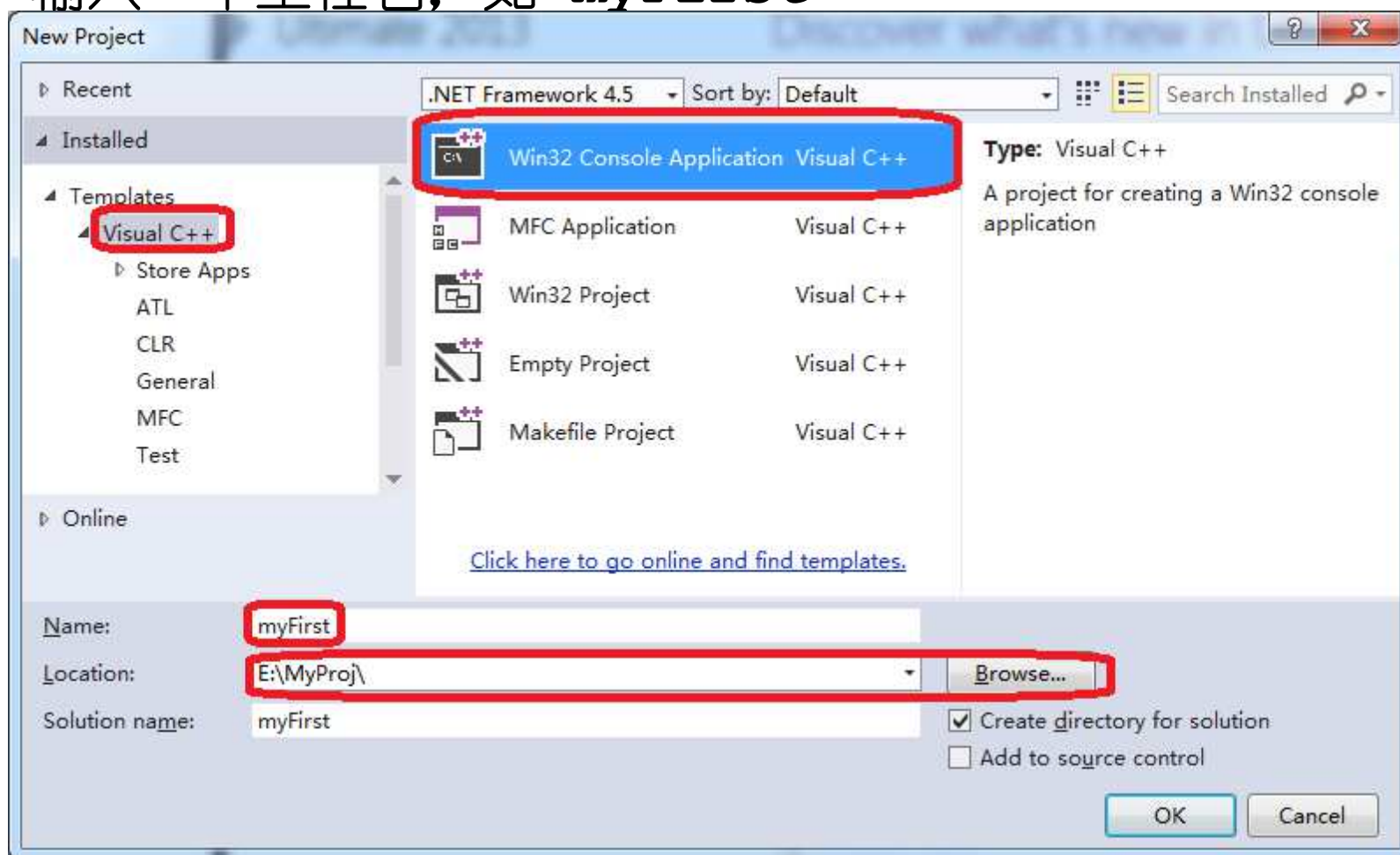
int main() {
    student s;    int i = 0, j=0;
    LNode *q = 0,
    LNode *L =(LNode *)malloc(sizeof(LNode));
    L->next = 0;
    do{In_student(s) ;
        if(s.score>=60)    j++;
        else if(s.score<0)    break;
        q =(LNode *)malloc(sizeof(LNode));
        q->data = s
        q->next = L->next;
        L->next = q;
    }while(s.score>=0) ;
    q = L->next;
    while(q){
        Out_student(q->data);        q = q->next;
    }
    printf("num of passed:%d\n",j) ;
}

```

建议用 vs201X 环境

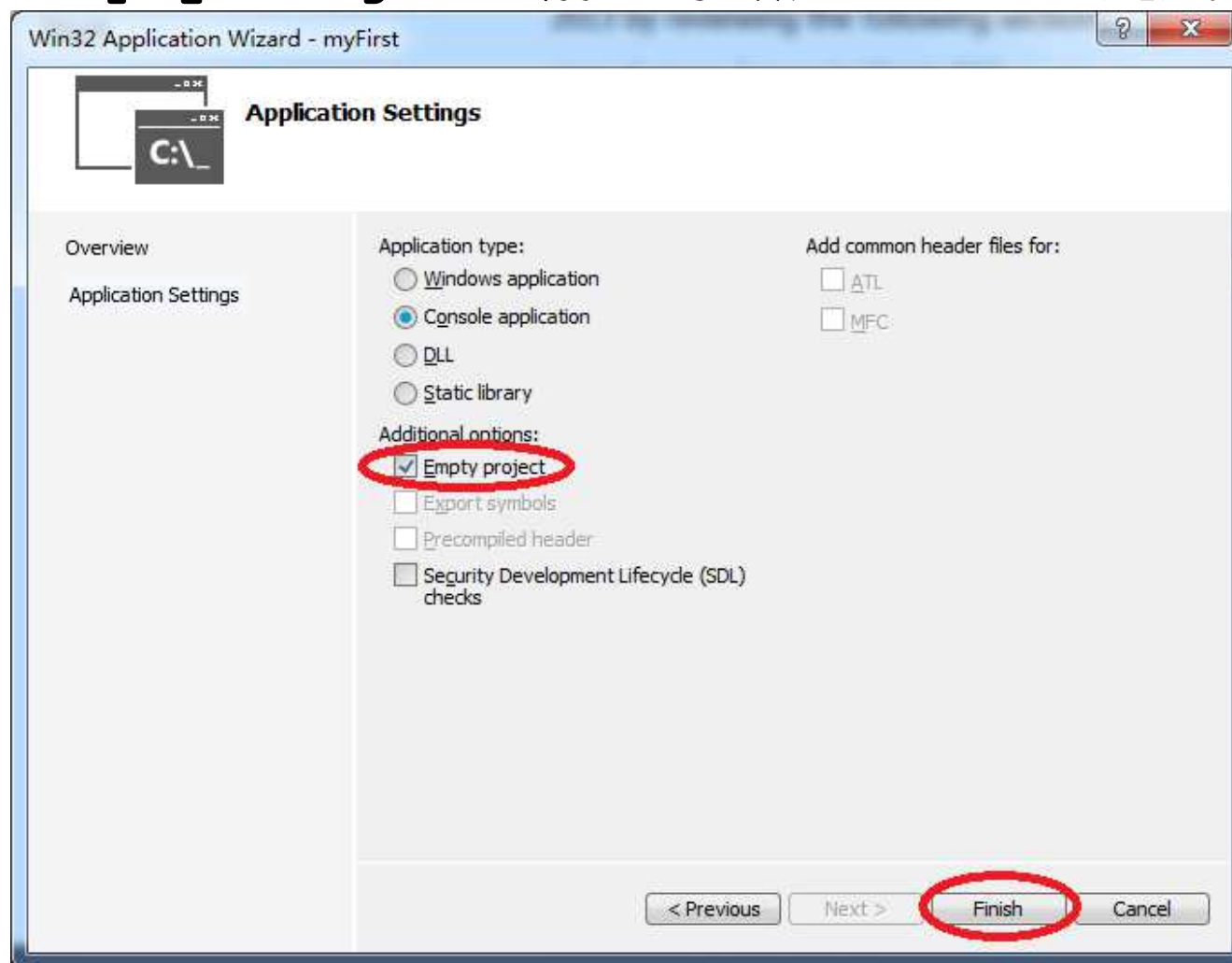
- 1) **new->Project->Visual C++ -> Win32 Console Application**

输入一个工程名，如 **myFirst**



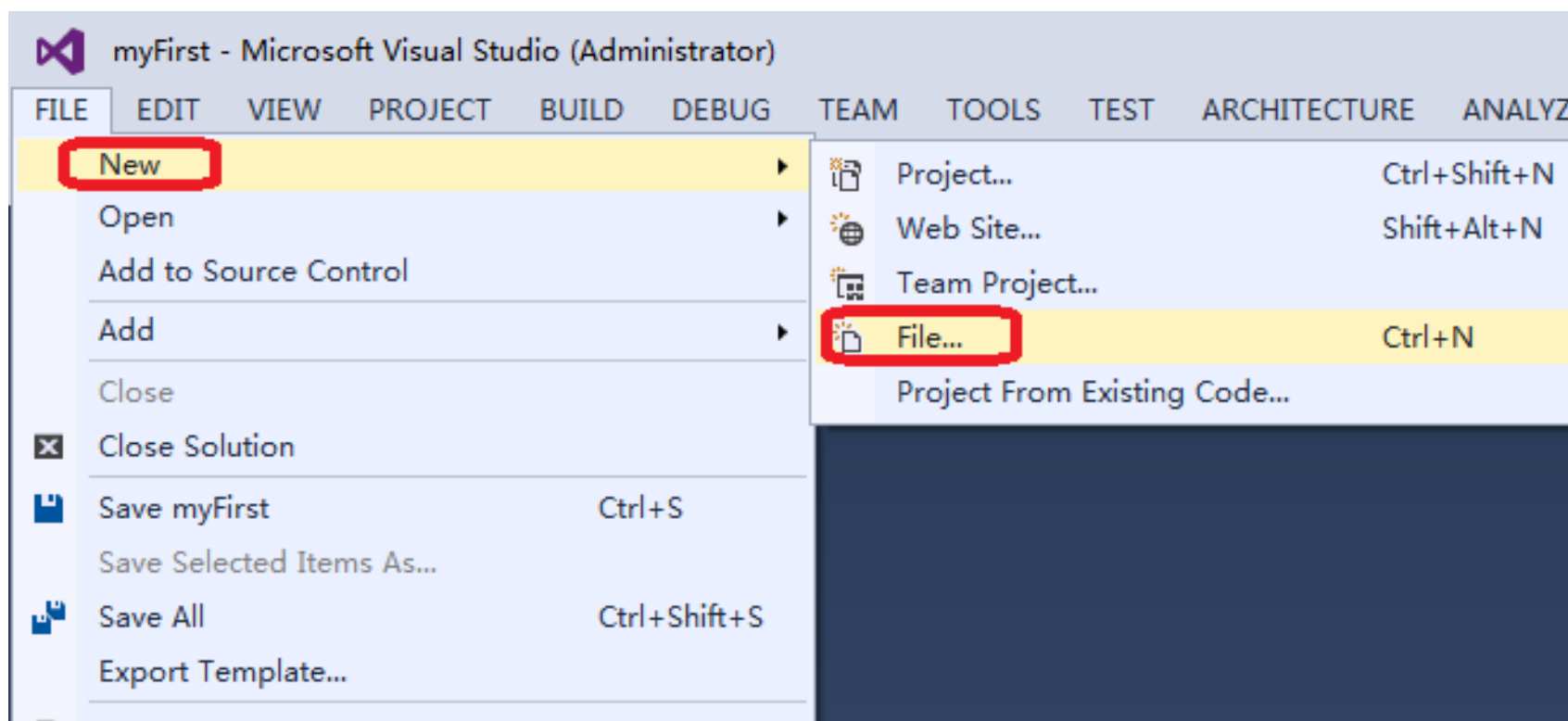
建议用 vs201x 环境

2) next, 取消” Precompiled Header”前面的勾, 勾上 “Empty Project”前的勾. 点”Finish”完成



建议用 vs201X 环境

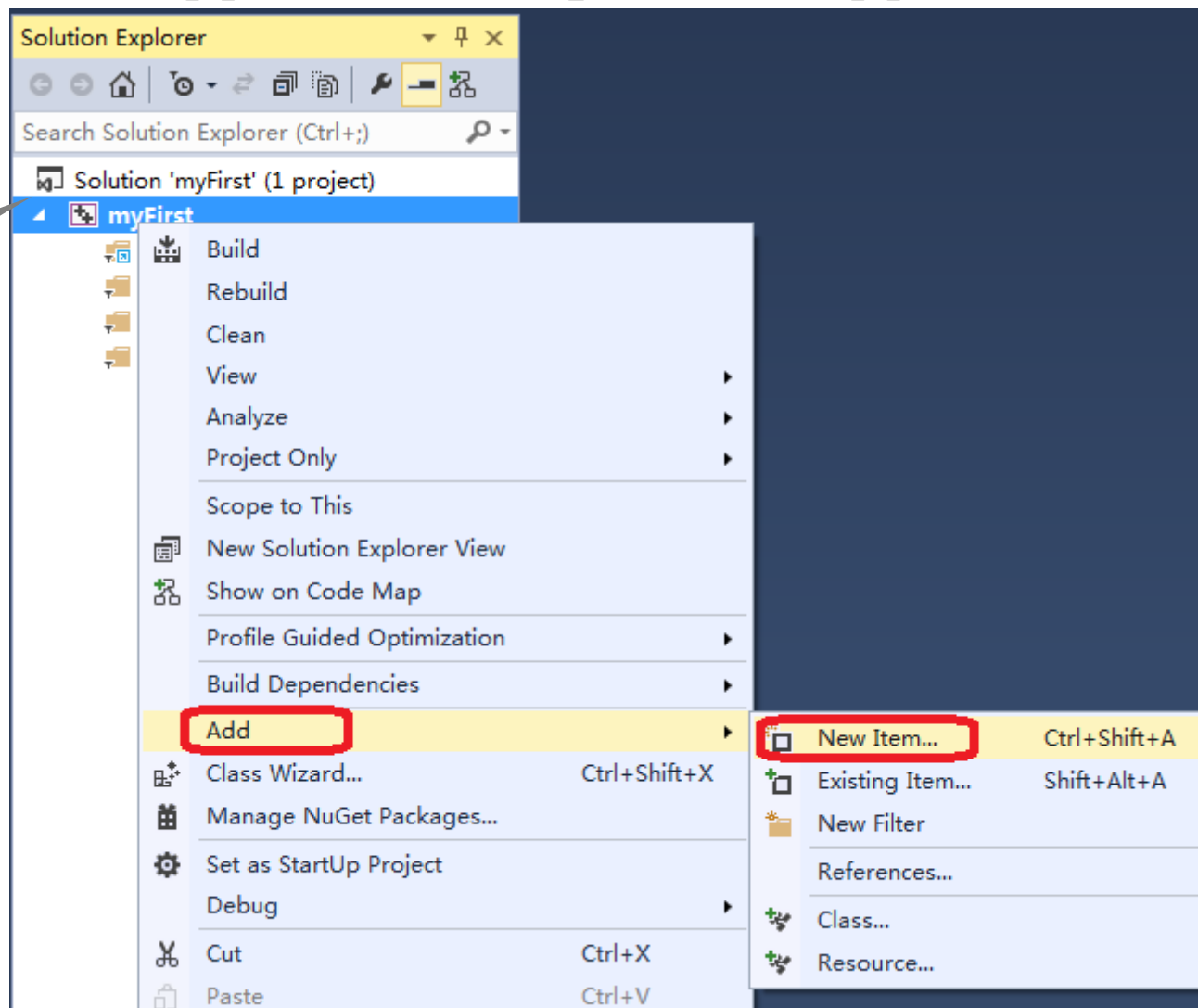
3) 同样方法生成一个 `.cpp` 文件, 如 `myFirst.cpp` 并加入到该工程中.



建议用 vs201X 环境

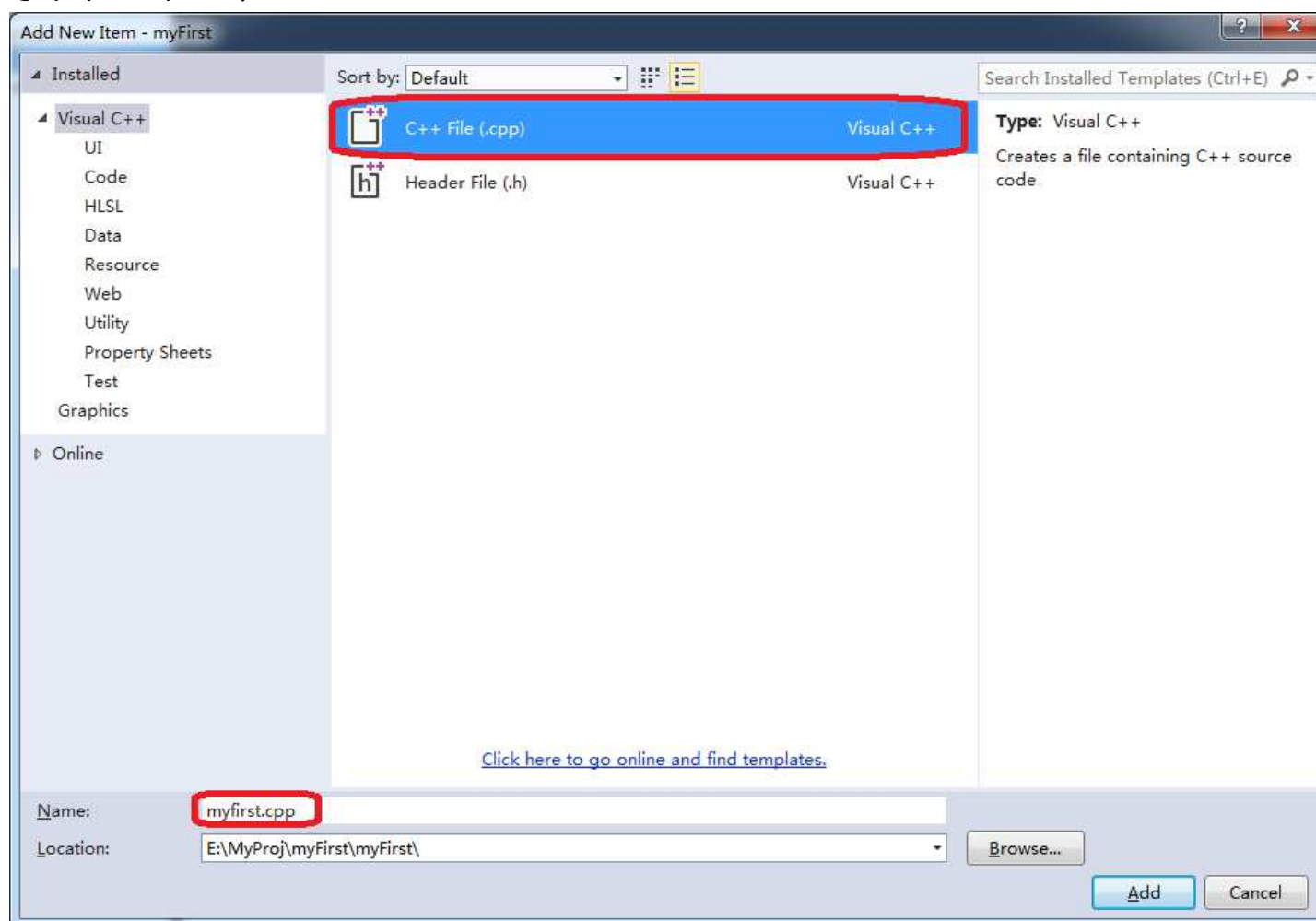
- 3) 同样方法生成一个 **.cpp** 文件, 如 **"myFirst.cpp"** 并加入到该工程中.

右键菜单



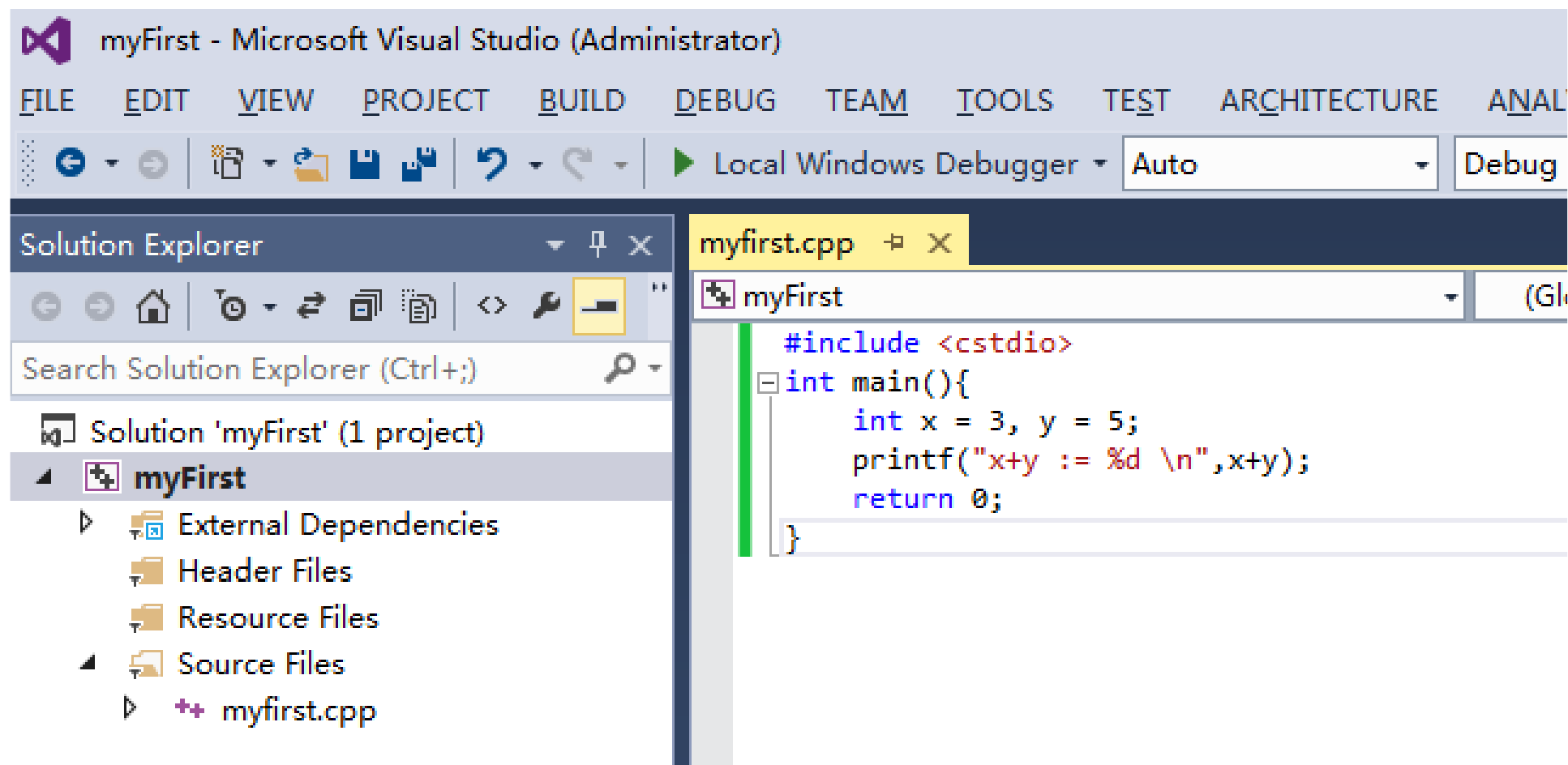
建议用 vs201X 环境

3) 同样方法生成一个 `.cpp` 文件, 如 `myFirst.cpp` 并加入到该工程中.



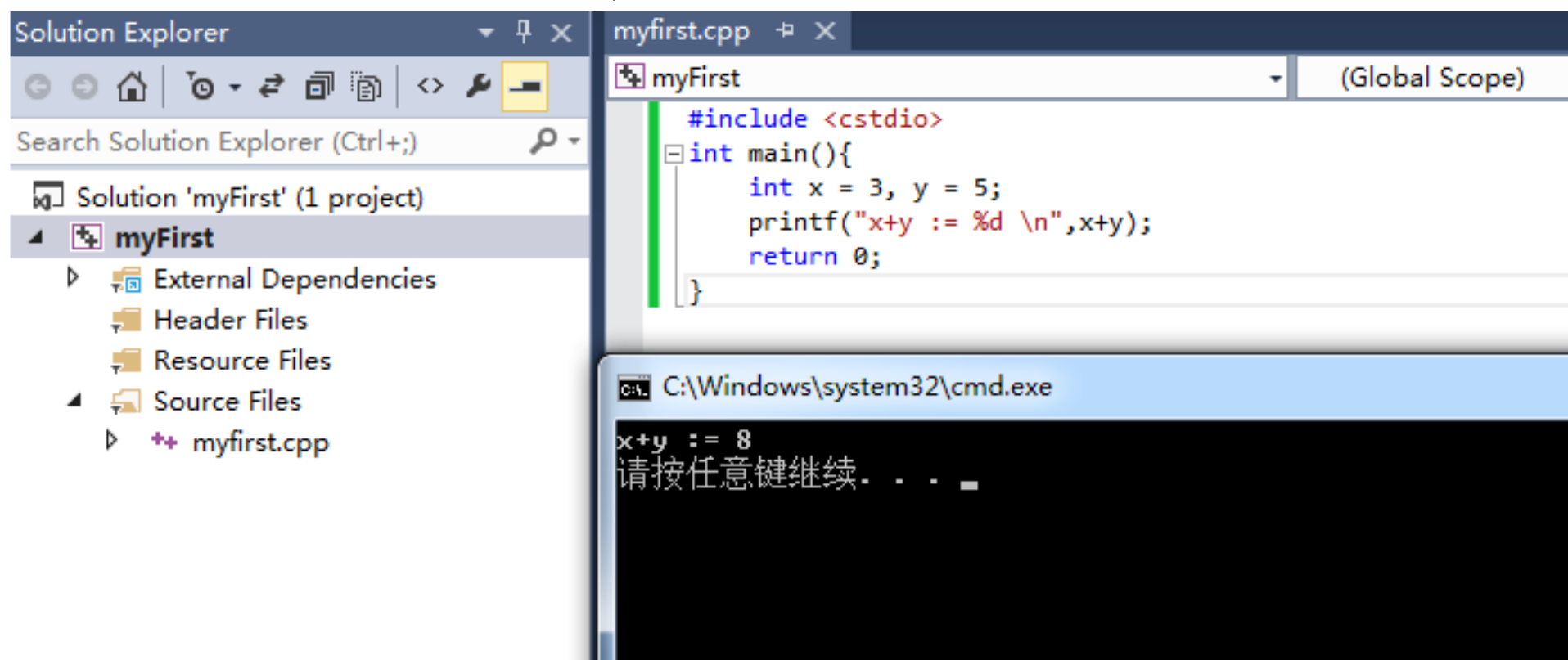
建议用 vs201x 环境

5) 输入程序代码



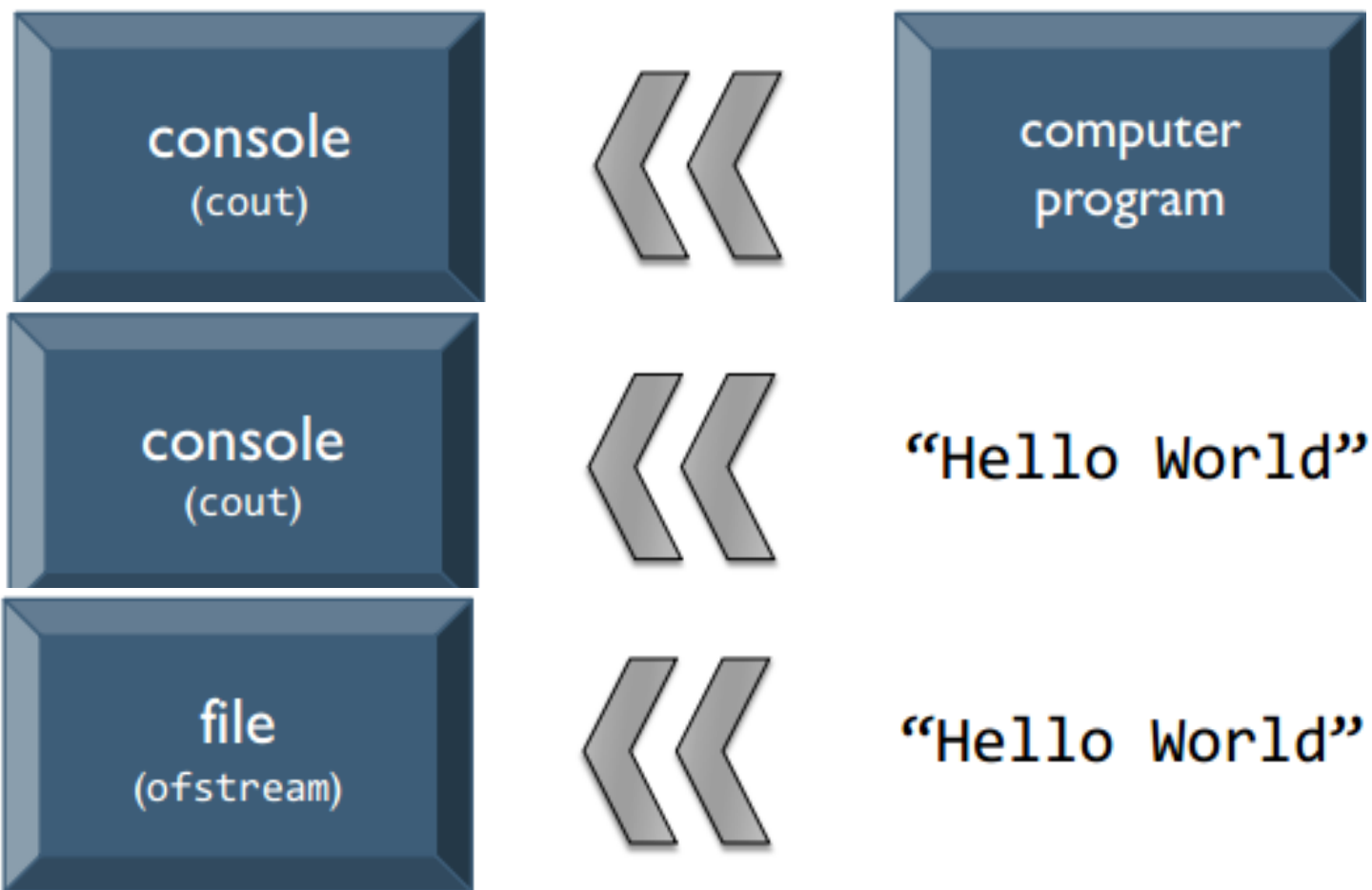
建议用 vs201X 环境

5) 然后按“CTRL+F7”编译源代码或按“F7”build该工程，再按“CTRL+F5”执行。



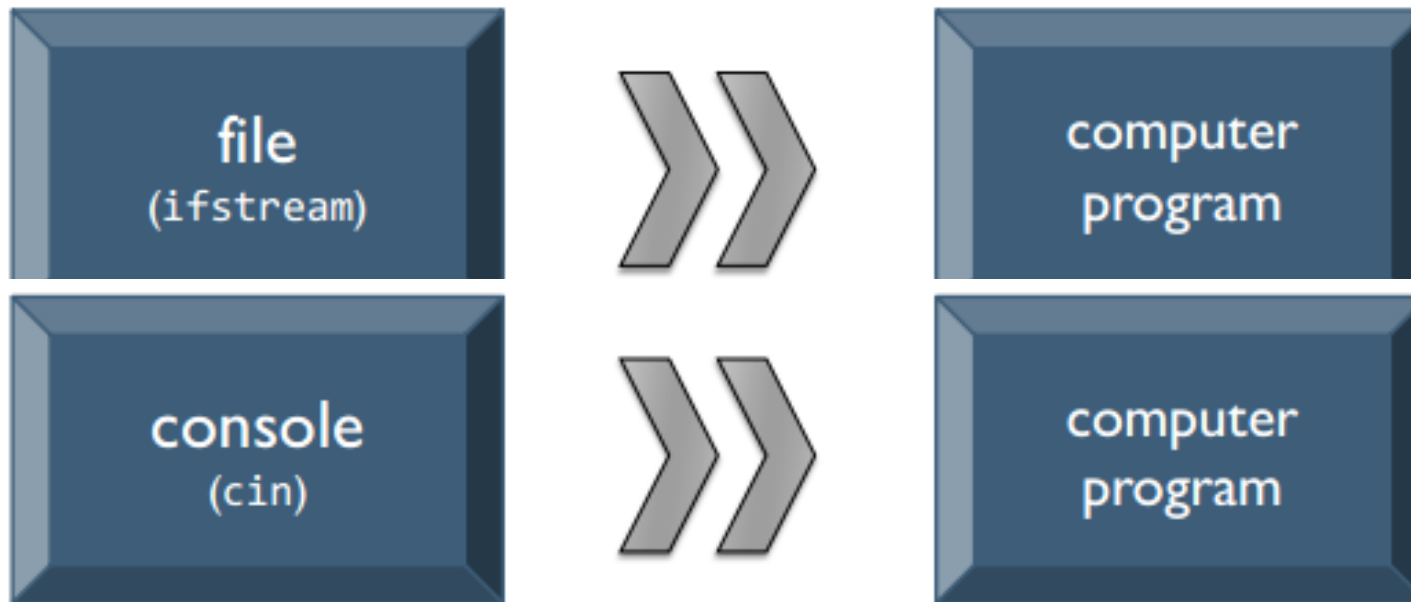
C++输入输出流

输出流运算符



C++输入输出流

输入流运算符



C++输入输出流

```
#include <iostream>
```

```
using std::cin;
```

```
using std::cout;
```

```
int main() {
```

```
    int x; double y;
```

```
    cin>>x>>y;
```

```
    cout<<x<<" "<<y<<"\n";
```

```
    return 0;
```

```
}
```

包含头文件

声明输入流对象

声明输出流对象

C++输入输出流

```
#include <fstream>
using std::ifstream;
int main(){
    ifstream iFile("a.txt");
    if(!iFile) return -1;
    double x,y,z;
    while(iFile>>x){
        iFile>>y>>z;
        std::cout<<x<<" "<<y
            <<" "<<z<<"\n";
    }
    return 0;
}
```

a.txt

20.5	31.3	99.2
10.5	21.3	39.2
30.5	11.3	9.2
	.	
	.	
	.	
60.5	1.3	3.78

C++中的string

- C++中的类是一种用户定义类型，类似于C语言中的结构类型。比如**string**就是C++的一个类。

```
#include <string>
#include <iostream>

int main(){
    string s,t("world!");
    s = "hello";
    string r = s+t;
    std::cout<<r<<"\n";

    int len = r.size();
}
```

- C++中的类是对C语言的结构**struct**的扩展，除数据成员外，还包括函数成员（也称成员函数）¹¹⁰

源文件和程序

- 大的程序经常被分成多个文件
- 编译器对每个源文件进行编译
- 连接器将编译好的目标文件和相关的库等连接成可执行文件。
- 单一定义规则：任何变量、函数等只能定义一次，但可被声明多次。
- 可能被多次引用的声明通常放在头文件中

源文件和程序

Math.h

```
#ifndef MATH_H_ $#  
#define MATH_H_ $#  
  
int add(int,int);  
extern int PI;  
  
double  
    CirArea(double);  
  
#endif
```

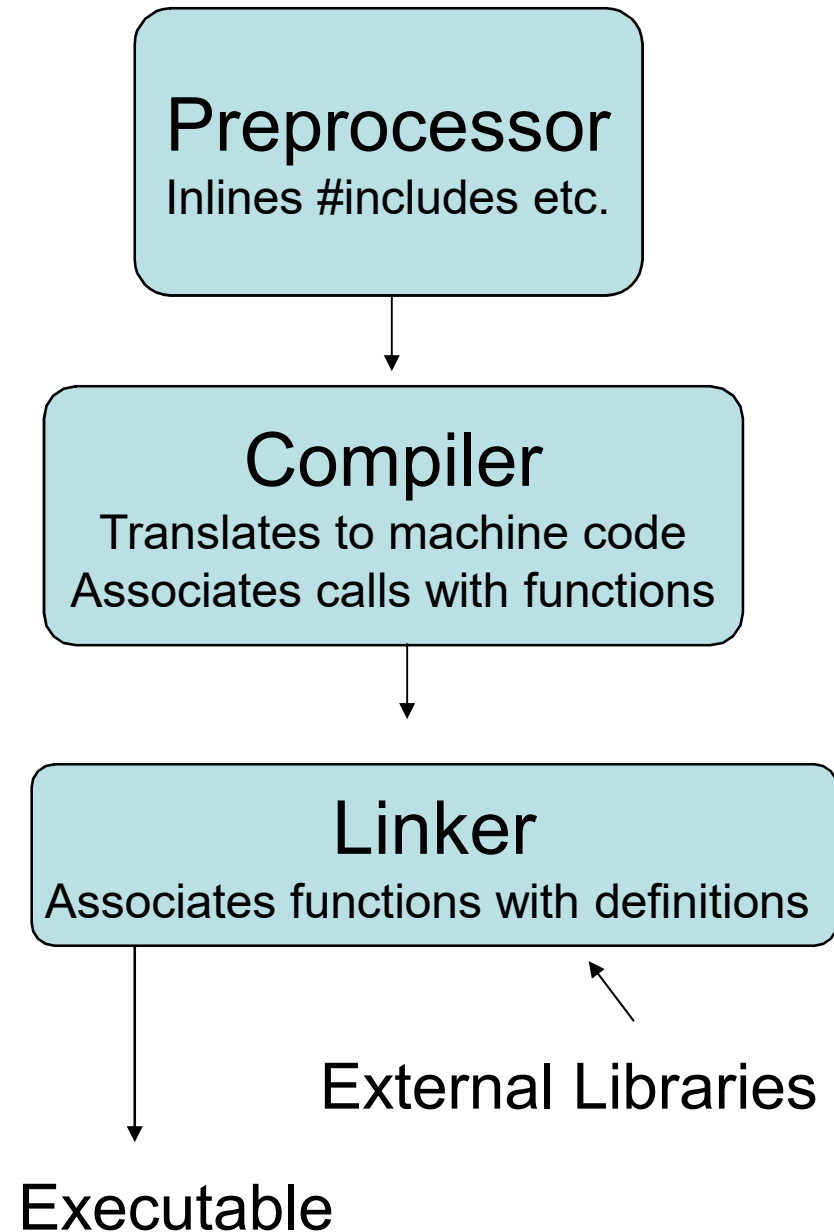
Math.cpp

```
#include "Math.h"  
int PI = 3.1415926;  
int add(int a,int b)  
{    return a+b;}  
  
double  
    CirArea(double r) {  
        return PI*r*r;  
    }
```


源文件和程序

main.cpp

```
#include "Math.h"
#include <iostream>
using namespace std;
int main() {
    double r,A;
    cin>>r;
    A = CirArea(r);
    cout<<A<<"\n";
    return 0;
}
```




作业

1. 编程实现不同版本的学生成绩表程序.
2. 搞懂幻灯片中的语法点，并编写代码比较验证这些语法点（函数参数传递、引用、类的定义和使用）. 写一个关于重要语法点的学习报告.

Download CodeBlocks

X ⓘ www.codeblocks.org/downloads/26

- FAQ
- Wiki
- Forums
- Forums (mobile)
- Nightlies
- Ticket System
- Browse SVN
- Browse SVN log



File	Date	Download from
codeblocks-16.01-setup.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01-setup-nonadmin.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01-nosetup.zip	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw-setup.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw-nosetup.zip	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw_fortran-setup.exe	28 Jan 2016	Sourceforge.net or FossHub

Download visual studio community

