

C++ programming

董洪伟

<http://hwdong.com>

C++之父

- Invented by Bjarne Stroustrup, AT&T公司贝尔实验室, now at TAMU



C++典型应用

- 浏览器



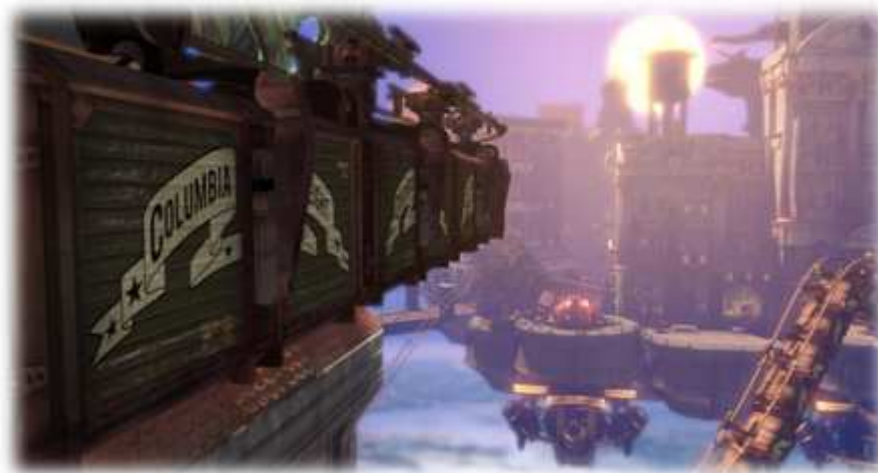
C++典型应用

- 文字处理



C++典型应用

- 娱乐



C++典型应用

- 其他



课程目标

- 学习标准C++语法
- 编写一致的、清晰的、高效的代码
- 精通C++，开发应用程序

教学信息

- 董洪伟
- 办公室： B512或B215
- 答疑： 周二或周四中午11:30-13:00
- <http://hwdong.com> 或 www.aoccm.com/bbs
- hwdong.cn@gmail.com

教学信息

- Bjarne Stroustrup. The C++ Programming Language (English or 中译本)
- Richard Johnsonbaugh等. 面向对象程序设计-c++语言描述, 机械出版社
- Stanley B.Lippman. C++ Primer. (English or 中译本)

教学信息

- 评分标准：
 - 课堂表现10%
 - 平时作业10% （抄袭者0分）
 - 实验程序40% （抄袭者0分）
 - 实验报告10% （抄袭者0分）
 - 期末考试30%

程序

- 就是对数据进行处理或加工
- 数据应有条理的存放，处理按照一定的步骤或方法进行

$$\begin{aligned}\text{程序} &= \text{数据} + \text{处理} \\ &= \text{数据结构} + \text{算法} \\ &= \text{变量} + \text{表达式}\end{aligned}$$

- 不同类型数据占据存储空间不同，是否保持不变？--数据=变量或常量
- 用运算符对数据处理或运算--表达式

程序语言-机器语言

- 电子元件'开'和'通'是两个稳定状态，用以表示1或0.
- 一组0和1二进制串可用来表示数据和运算符(操作码)。

机器指令=操作码+地址码

操作码：加、减、乘、除、移动等

地址码：第1、2操作数地址、结果地址

程序语言-机器语言

- 操作码：
0000 代表 加载 (LOAD)
0001 代表 存储 (STORE)
- 暂存器地址：
0000 代表暂存器 A
0001 代表暂存器 B
- 存储器地址：假如00000000000000 代表地址为 0 的存储器
00000000000001 代表地址为 1 的存储器
000000010000 代表地址为 16 的存储器
100000000000 代表地址为 2^{11} 的存储器
- 指令示例：
0000,0001,00000000000001 代表 LOAD B, 1
0001,0001,000000010000 代表 STORE B, 16

机器指令=操作码+地址码

汇编语言

- 汇编语言（Assembly Language）是机器语言的助记符

如已知b,c求出b+c并附给a

```
10001010 01010101 11000100 mov edx,[ebp-0x3c]
```

```
00000011 01010101 11000000 add edx,[ebp-0x40]
```

```
10001001 01010101 11001000 mov [ebp-0x38],edx
```

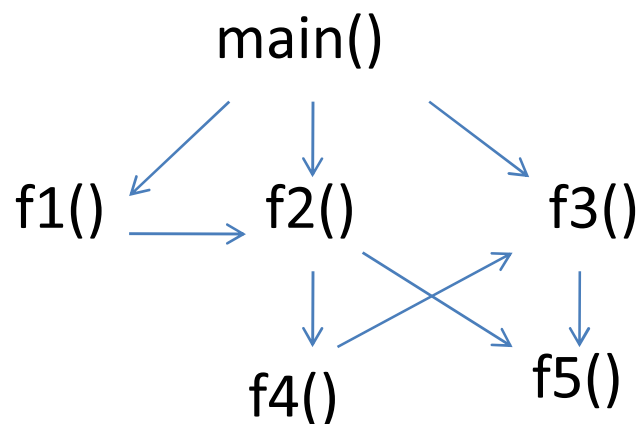
高级语言：C/C++,Java,Fortran...

- 以人们容易理解的接近自然语言的语言表示运算数和运算符。如用字母 x , y 表示变量, 用 $x*y$ 表示数学运算。
- 优点很多：易读性、易用性、易调式、开发效率高等...
- 常用高级语言：C/C++、Java、Fortran、Matlab、Basic、PHP、HTML...

C++: C的面向对象扩展

- C:过程式语言：过程-函数

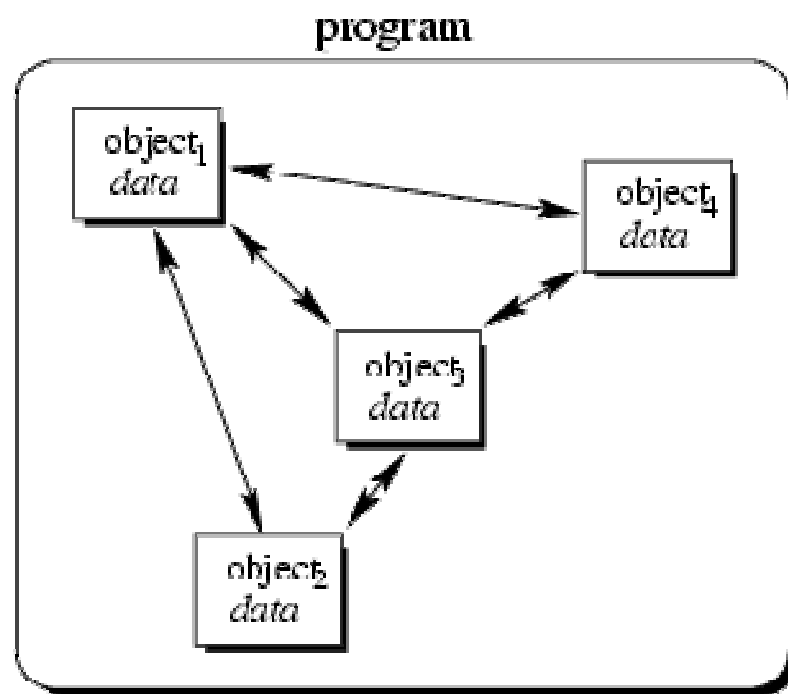
任务分解：大的任务分解为各种小任务，
一个函数完成一个特定的任务(功能)。通过
函数调用，协作完成一个复杂的功能



C++: C的面向对象扩展

- C++:过程式+面向对象

面向对象系统：一组具有独立自主功能的对象之间通过消息传递协作完成系统功能。



C++: 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);
```

```
}
```

← 注释：解释程序的功能

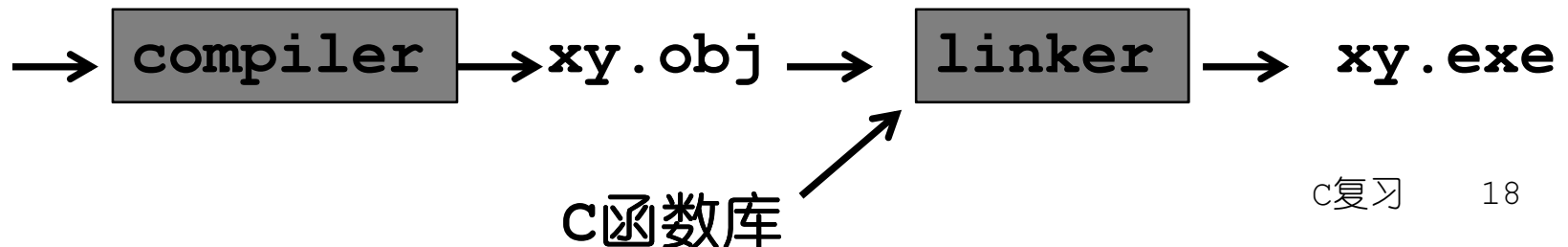
← 包含头文件：函数的定义等

← 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



C++: $z=x+y$ (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(...)`

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

?

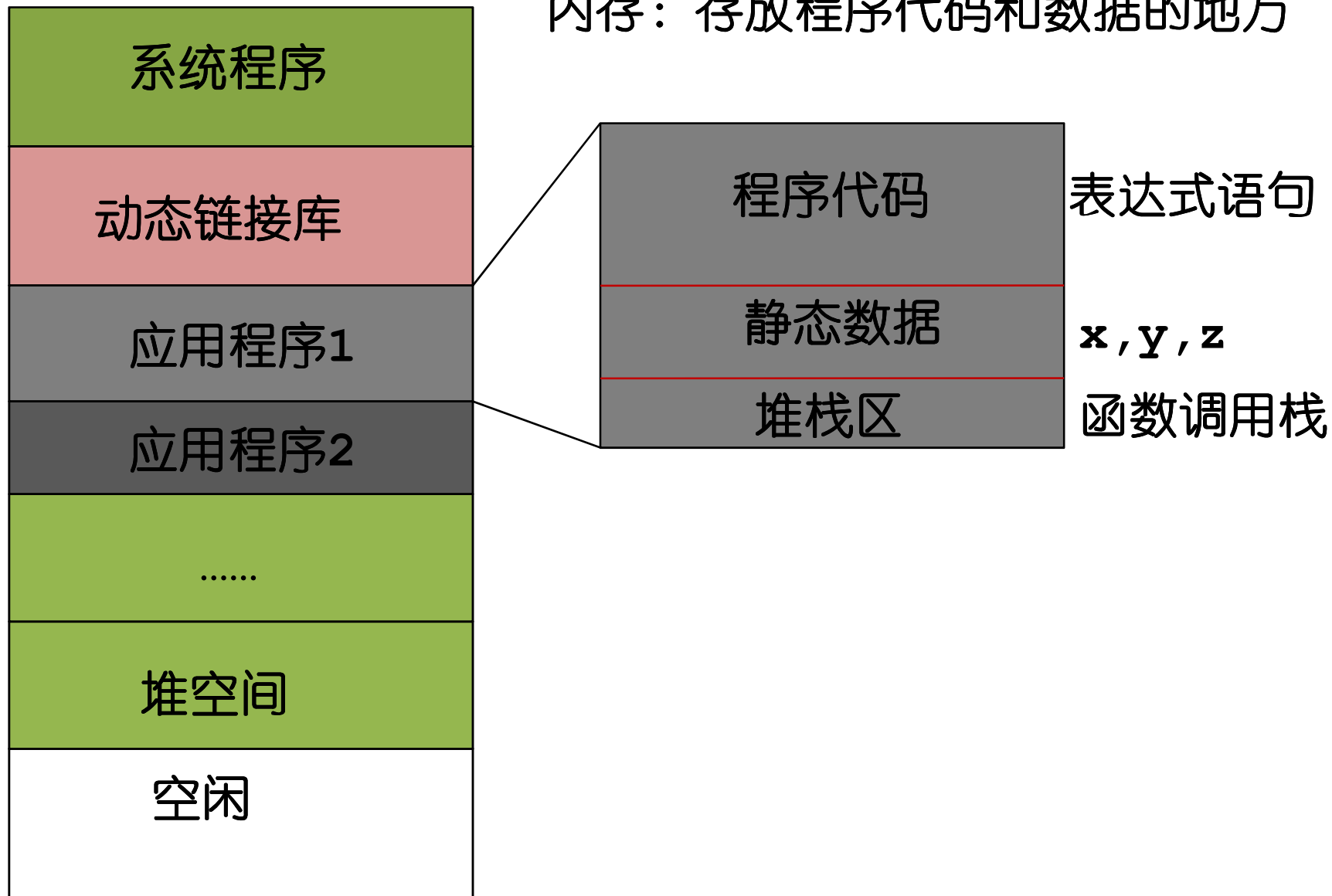
三个表达式语句

表达式: 变量、常数和运算符构成

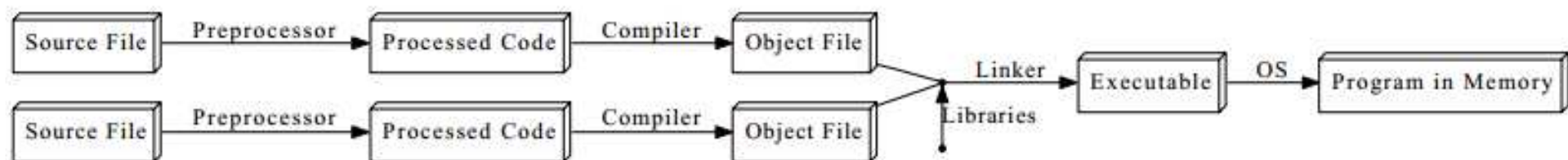
语句: 后跟 ';' 的表达式

程序内存布局

内存：存放程序代码和数据的地方



程序的编译链接



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);  
}
```

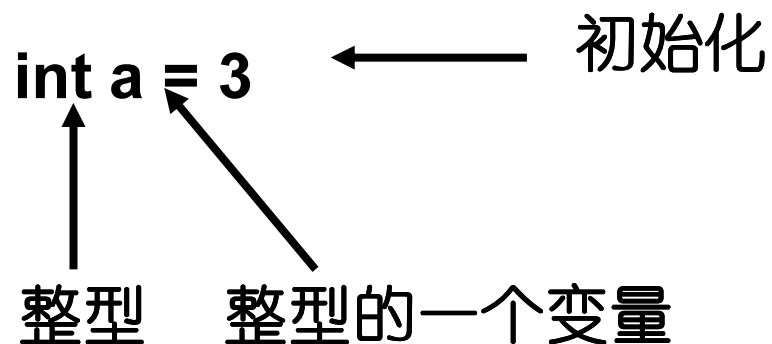
该程序编译链接没有问题，但输出结果有问题-- 逻辑错误！
因为x没有初始化！

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
- 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...

类型与变量

- 类型：规定了值集合和其上的操作
- 变量：存储一个类型值的空间



在C++中，变量也称为对象

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

```
bool f,g,h;  
int a =3;  
a = f;  
h = (f+g)/a;
```

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：enum, struct, class...

```
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
```

指针和数组

- 数组名就是数组第一个元素的指针(地址)

```
int arr[] = {10,20,30,40,50,60,70};
```

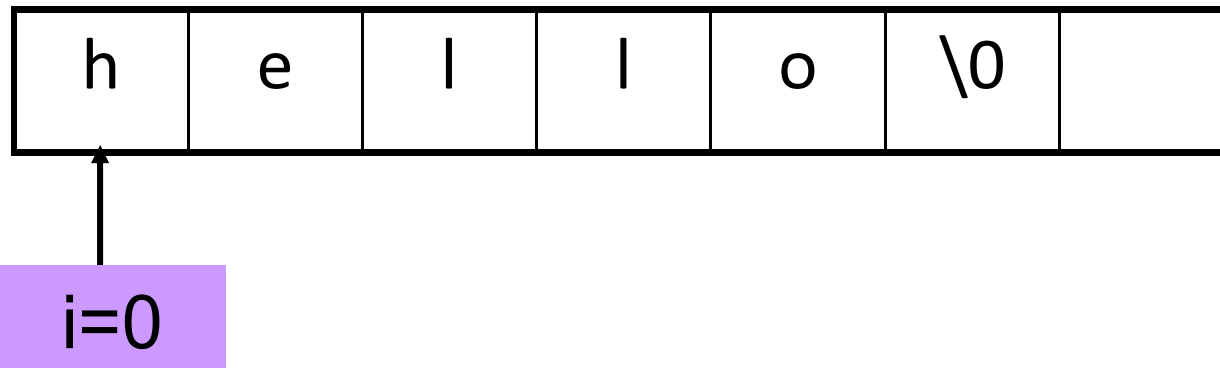
```
int *p = arr; //等价于p = &(arr[0]);
```

```
p[2] = 2; //等价于arr[2] = 2;
```

```
*(p+3) = 4; //等价于arr[3] = 4; 或 p[3] = 4;
```

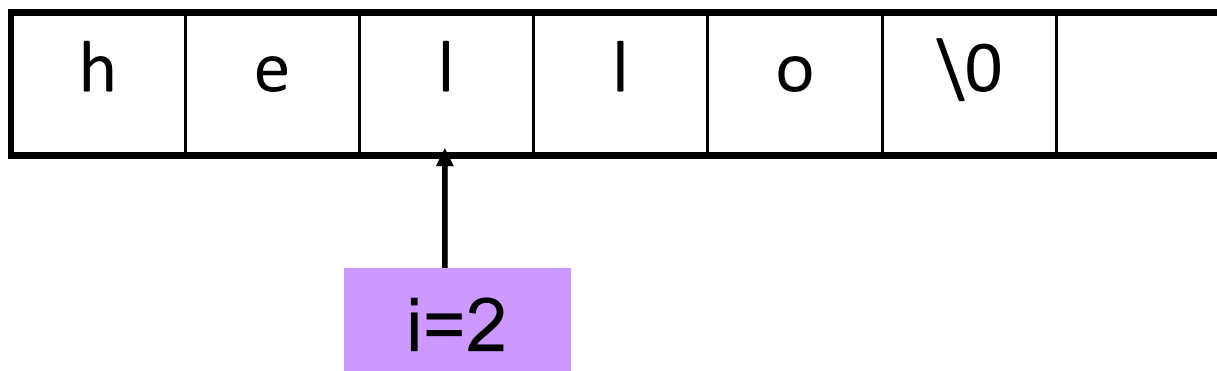
指针和数组

```
int strlen(const char *str){  
    int i = 0 ;  
    //while (str[i] != '\0') i++;  
    while (str[i++] != '\0') ;  
    return i;  
}
```



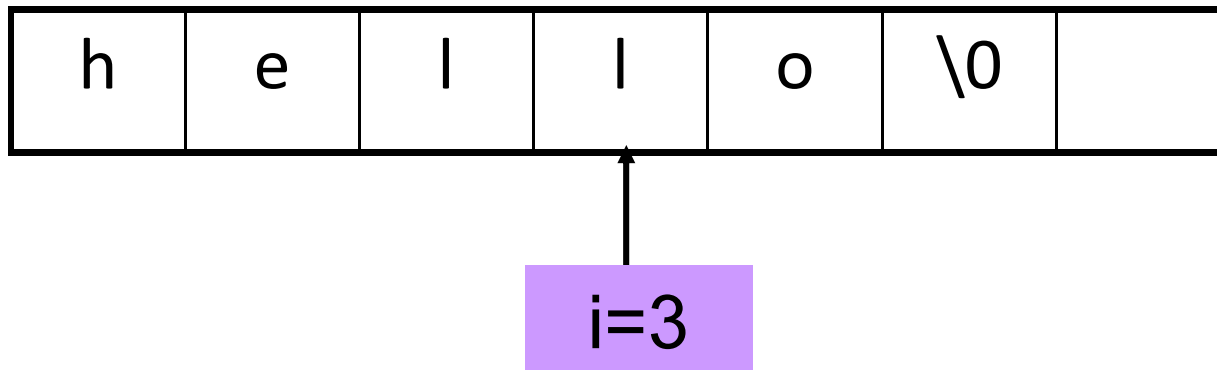
指针和数组

```
int strlen(const char *str){  
    int i = 0 ;  
    //while (str[i] != '\0') i++;  
    while (str[i++] != '\0') ;  
    return i;  
}
```



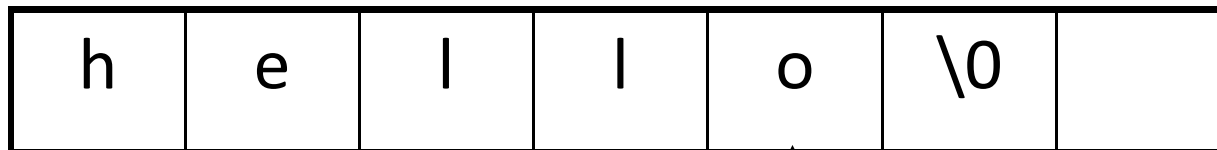
指针和数组

```
int strlen(const char *str){  
    int i = 0 ;  
    //while (str[i] != '\0') i++;  
    while (str[i++] != '\0') ;  
    return i;  
}
```



指针和数组

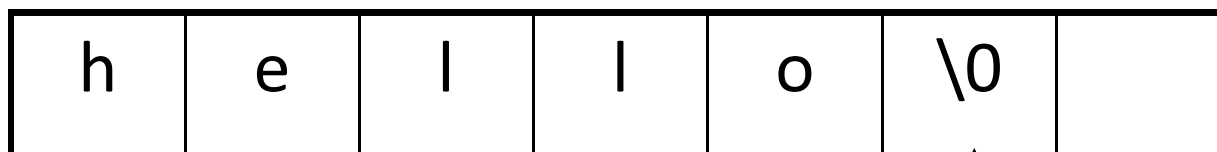
```
int strlen(const char *str){  
    int i = 0 ;  
    //while (str[i] != '\0') i++;  
    while (str[i++] != '\0') ;  
    return i;  
}
```



i=4

指针和数组

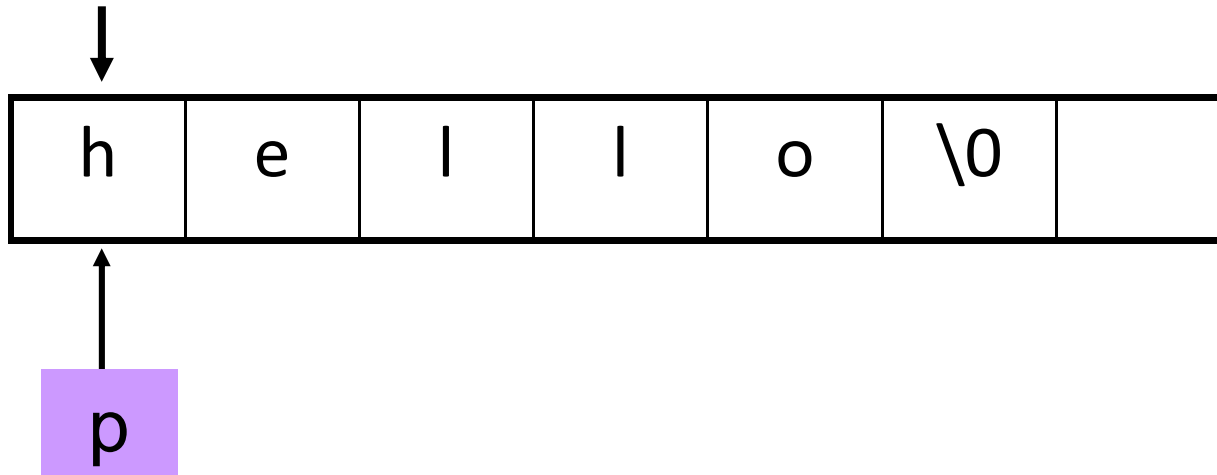
```
int strlen(const char *str){  
    int i = 0 ;  
    //while (str[i] != '\0') i++;  
    while (str[i++] != '\0') ;  
    return i;  
}
```



i=5

指针和数组

```
int strlen(const char *str){  
    int *p = str ;  
    while (*p != '\0')  p++;  
    return p-str;  
}
```



指针和数组

- 练习：实现字符串复制函数

```
void strcpy(char *dst,const char *src){  
  
}
```

指针和数组

- 下列代码哪里有问题？

```
#include <iostream>
```

```
void main( ) {
```

```
    char arr[10] = {'h','e','l','l','o'};
```

```
    std::cout<<strlen(arr);
```

```
}
```

警告： 字符数组不是字符串！ 字符串
是最后有结束字符的字符数组！

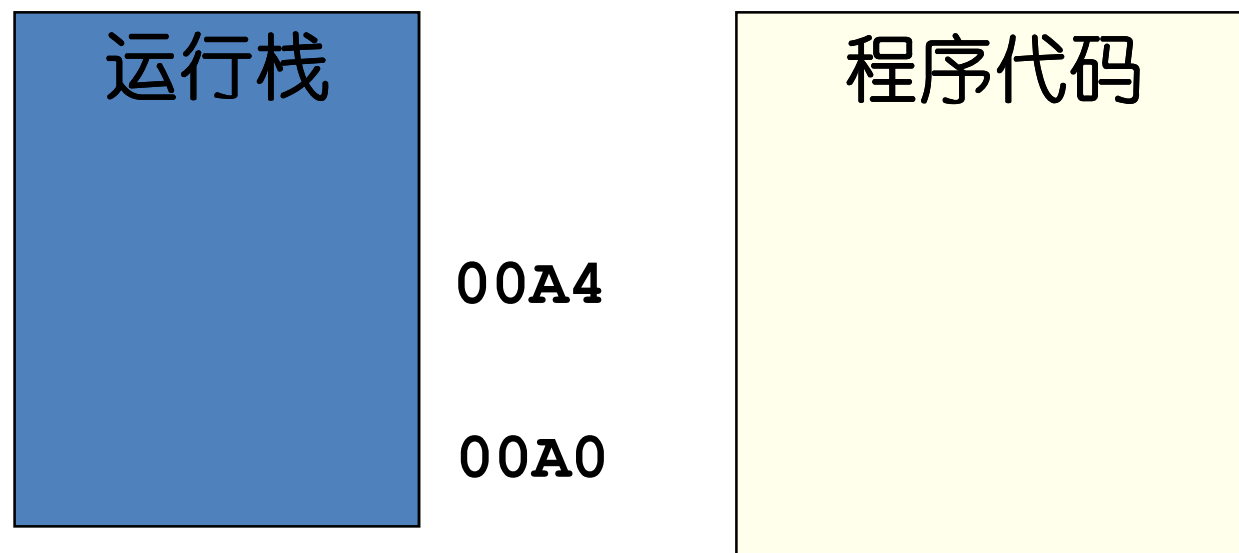
通过结构指针访问结构成员

```
struct student s;  
strcpy(s.name,"LiPin");  
s.score = 78.5;  
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

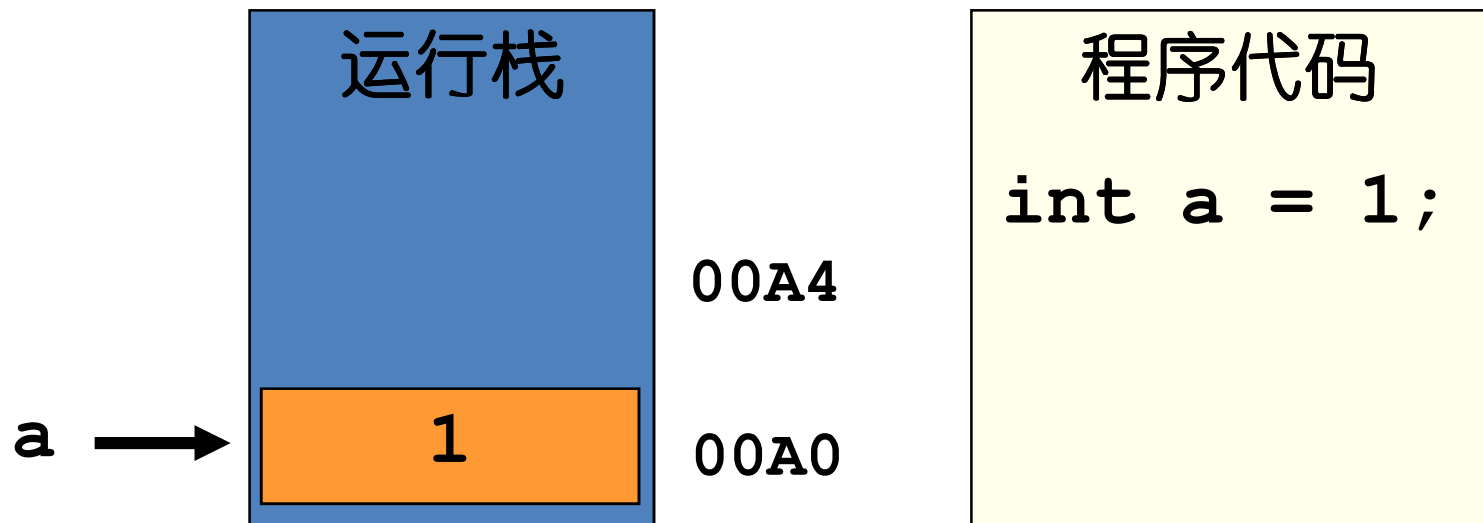
- 直接盛放自身数据
- 每个变量都有自身的值的一份拷贝
- 对一个值的修改不会影响另外一个值



值类型与引用类型

- C语言只有值类型

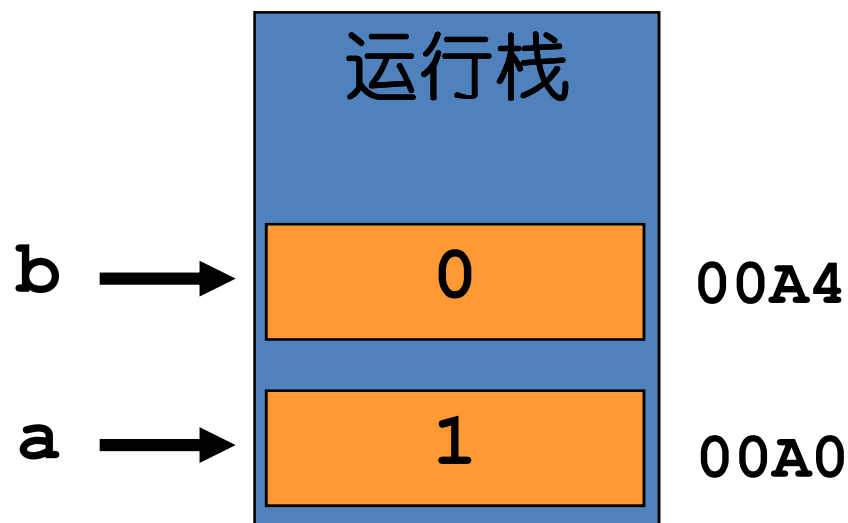
- 直接盛放自身数据
- 每个变量都有自身的值的一份拷贝
- 对一个值的修改不会影响另外一个值



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



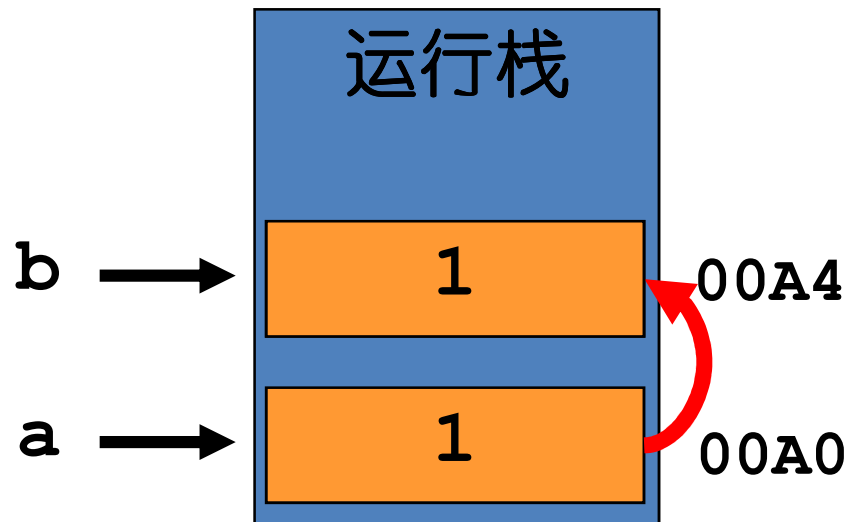
程序代码

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

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



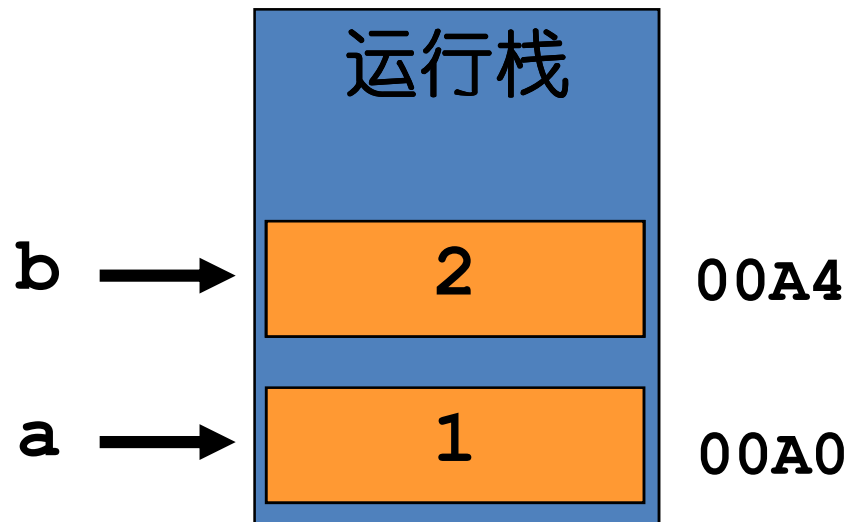
程序代码

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

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

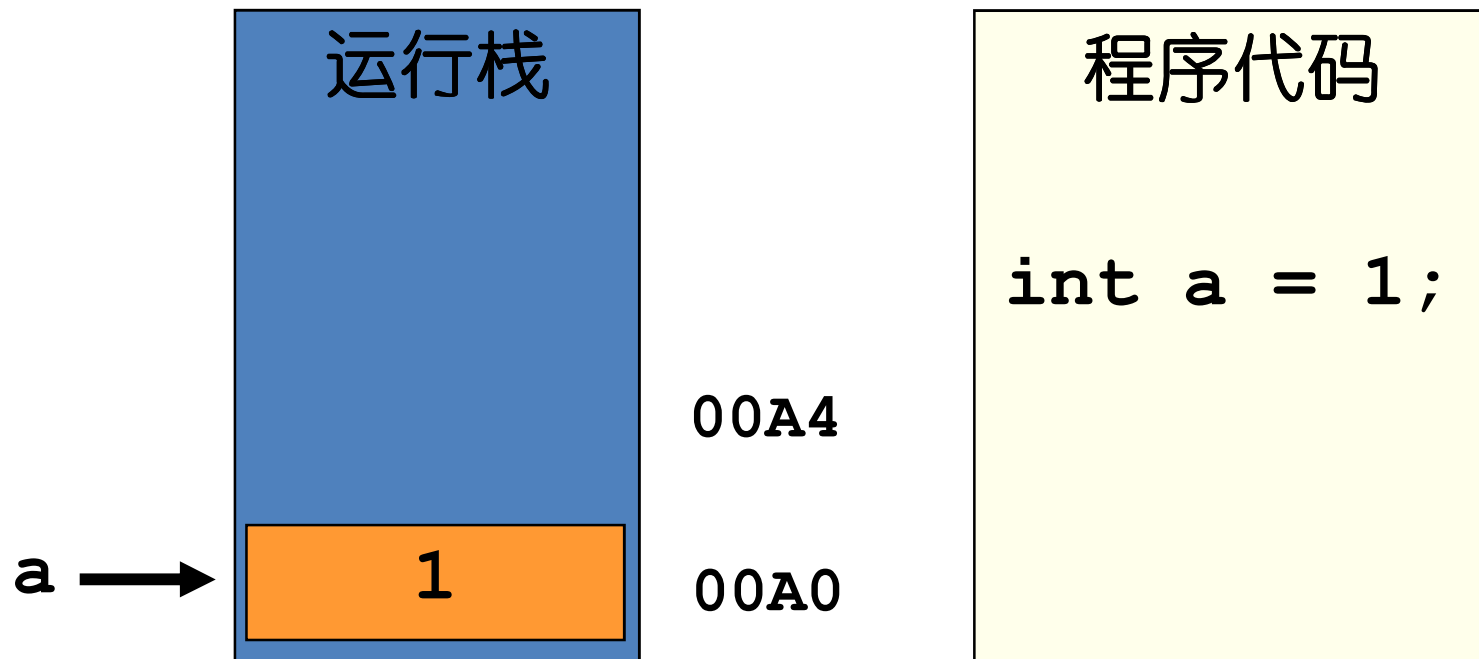


程序代码

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

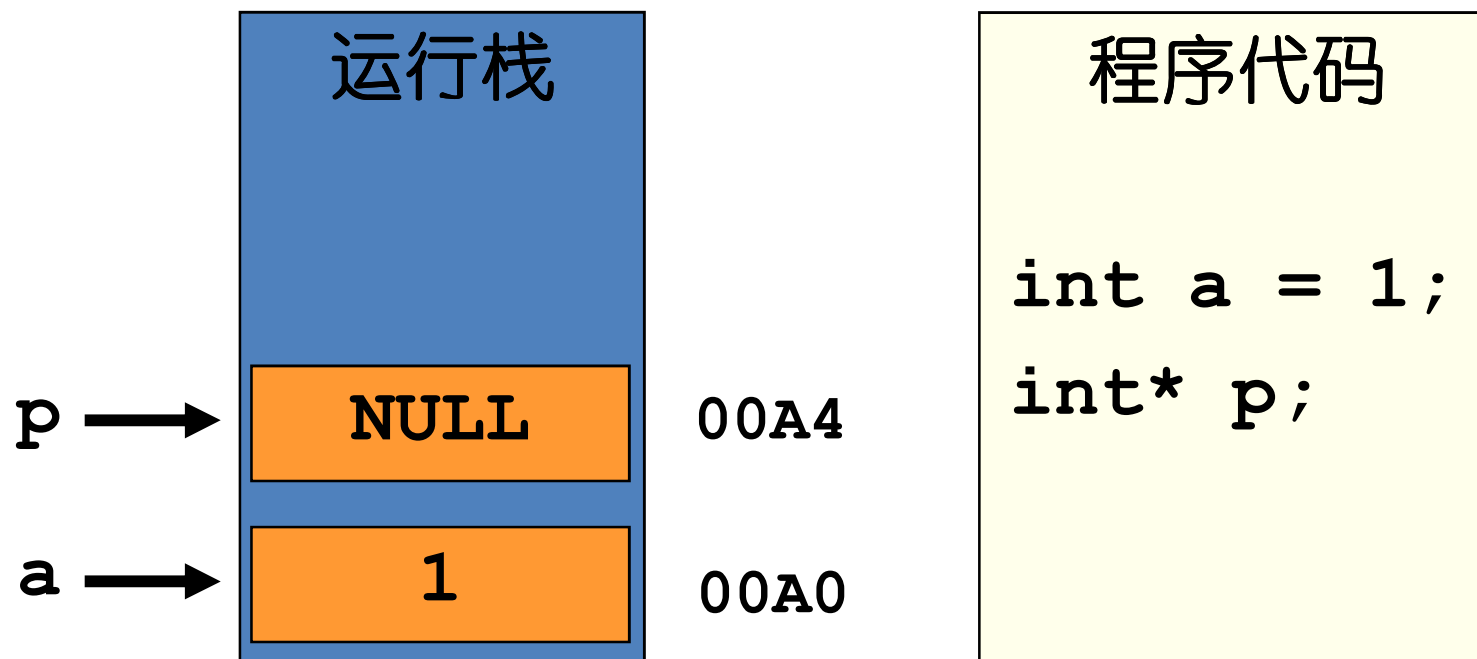
值类型与引用类型

- 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



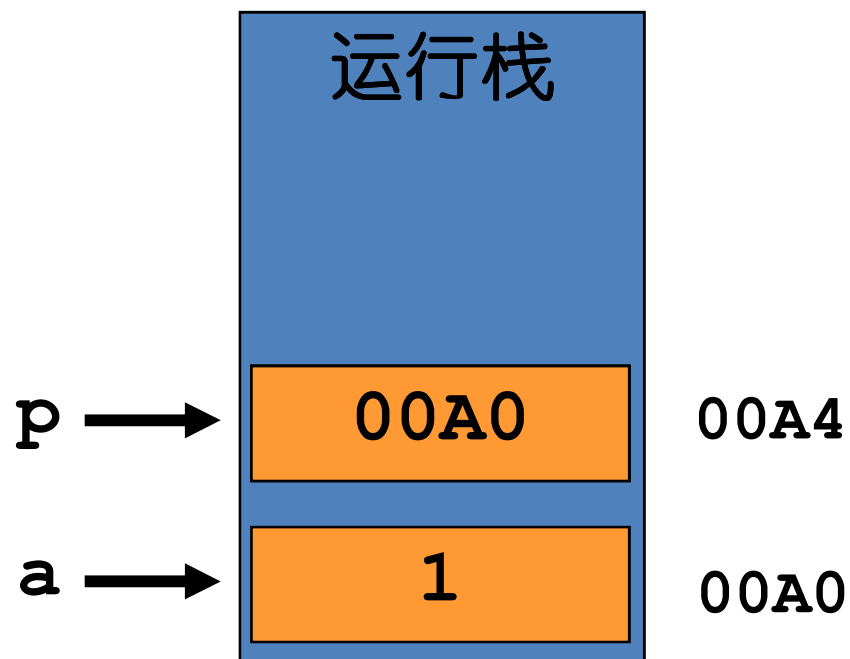
值类型与引用类型

- 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

- 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

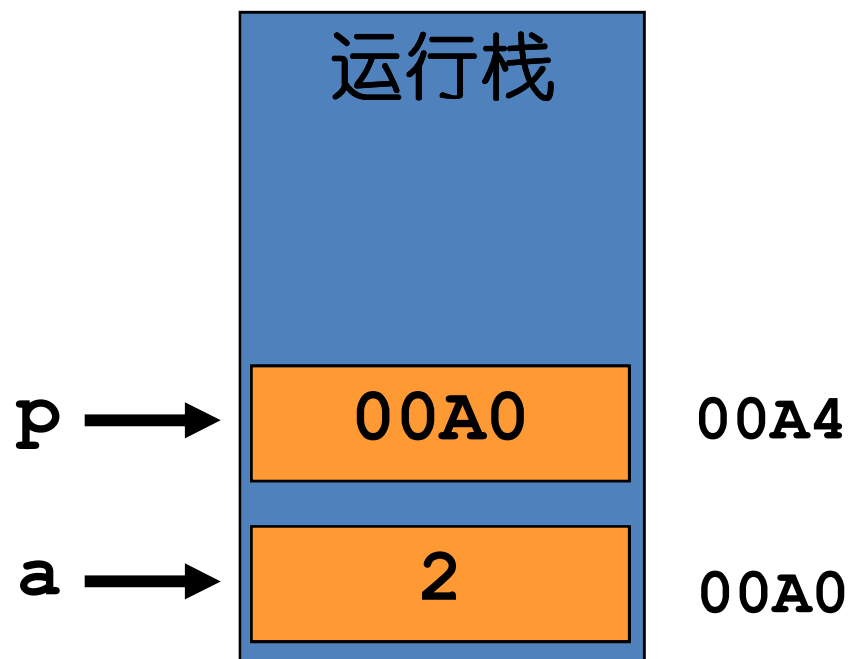


程序代码

```
int a = 1;  
int* p;  
p = &a;
```

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

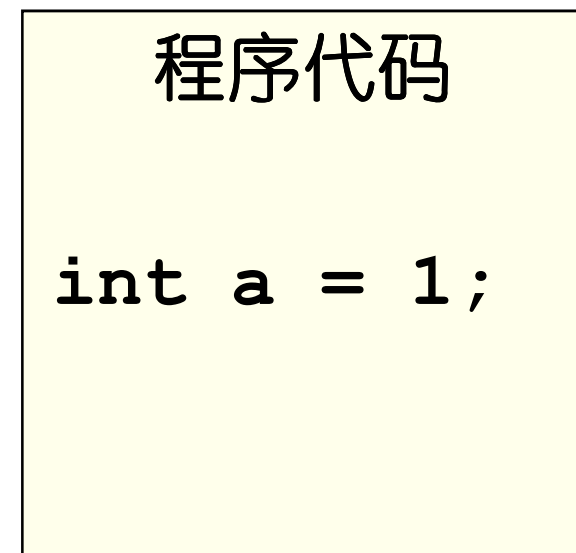
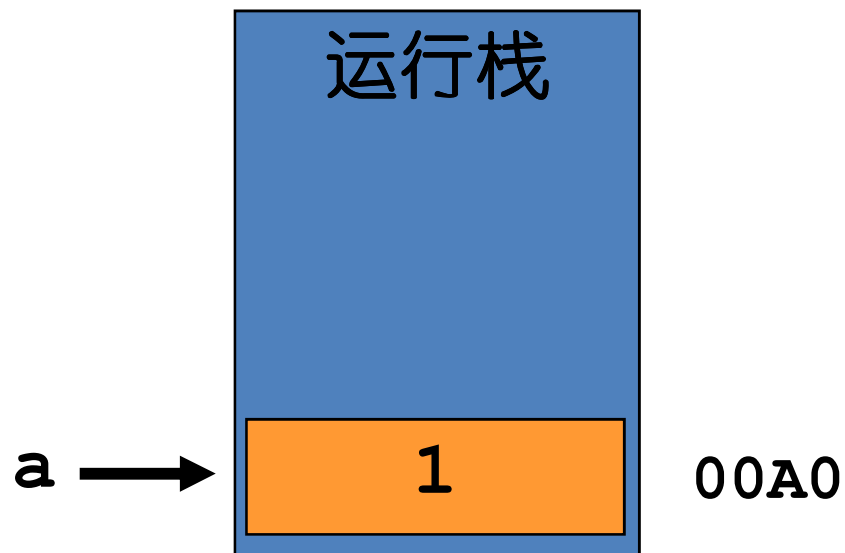


程序代码

```
int a = 1;  
int* p;  
p = &a;  
*p = 2;
```

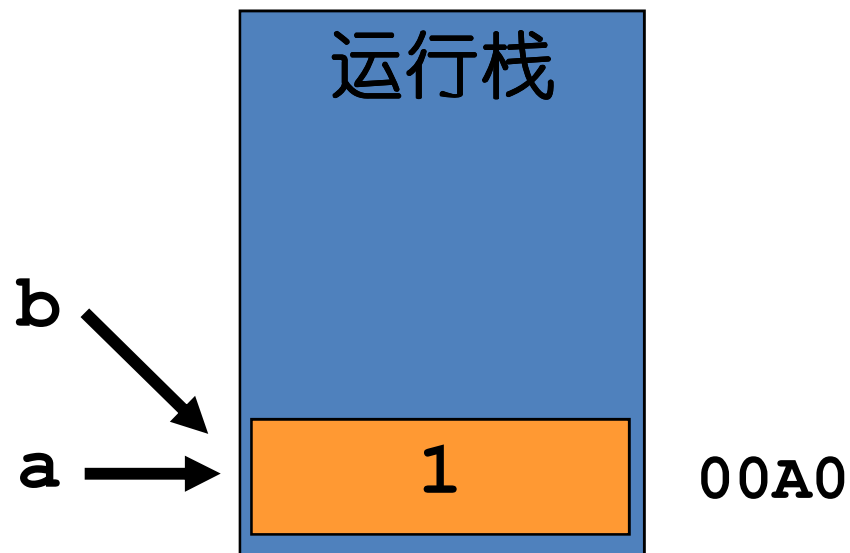
值类型与引用类型

- 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

- C++的引用类型
 - 简单理解：一个变量的别名

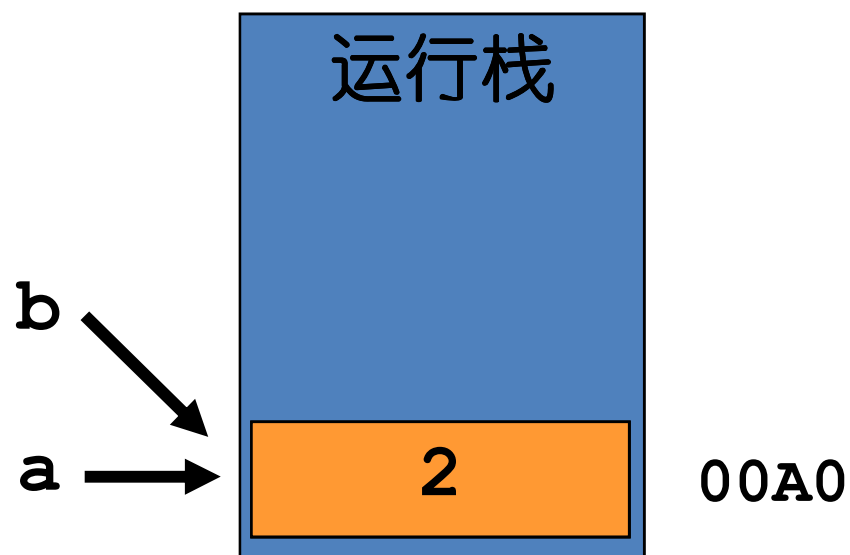


程序代码

```
int a = 1;  
int& b = a;
```

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



程序代码

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

值类型与引用类型

- 引用变量注意:
 - 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

常量const:不能改变的量

name是一个type类型的常量

Declaration	name is ...
<code>const type name = value;</code>	constant <i>type</i>
<code>type * const name = value;</code>	constant pointer to <i>type</i>
<code>const type * name = value;</code>	(variable) pointer to constant <i>type</i>
<code>const type * const name = value;</code>	constant pointer to constant <i>type</i>

name指针指向type类型的量，但name是一个不能修改的指针变量

指针变量name本身是变量，但其执行的type类型量不能改变

常量const:不能改变的量

既然不能改变，常量必须定义时就初始化！

```
int i;                // just an ordinary integer
int *ip;              // uninitialized pointer to integer
int * const cp = &i;   // constant pointer to integer
const int ci = 7;      // constant integer
const int *cip;        // pointer to constant integer
const int * const cicp = &ci; // constant pointer to constant integer
```

```
cip=&ci;
*cp = ci;
cip=&cicp;
ci = 8 ;
*cip = 8 ;
cp =&ci;
ip = cip; // 错：这回引起通过*ip改变*cip!
```

语句和表达式

- 语句，程序的基本构造块，做某些事情
- 表达式：由常量、变量和运算符构成。对数据进行加工，表达式有一个值
- 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不是main程序块内的局部变量

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(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！

- C++必须返回相应类型的返回值：

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;
}

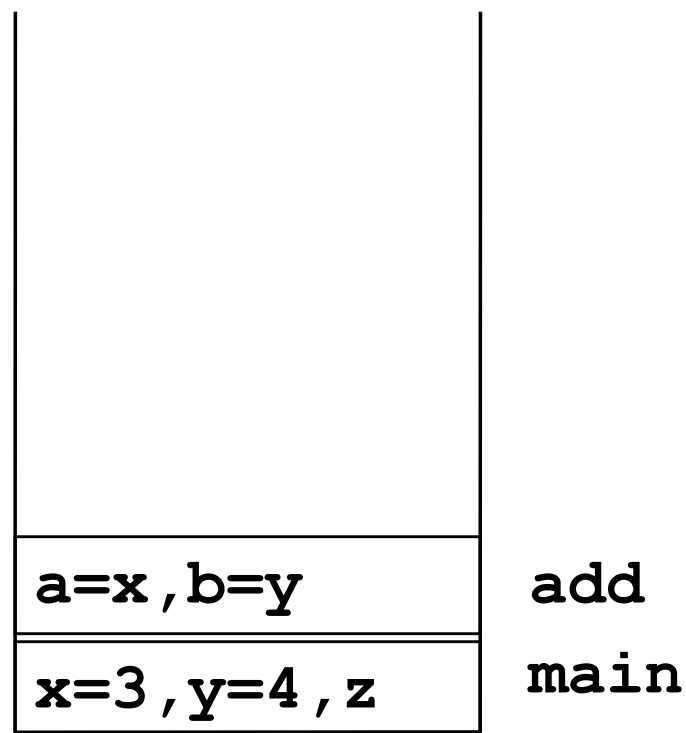
void main()
{ int x=3,y=4;
  int z = add(x,y) ;
}
```

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;
}
```

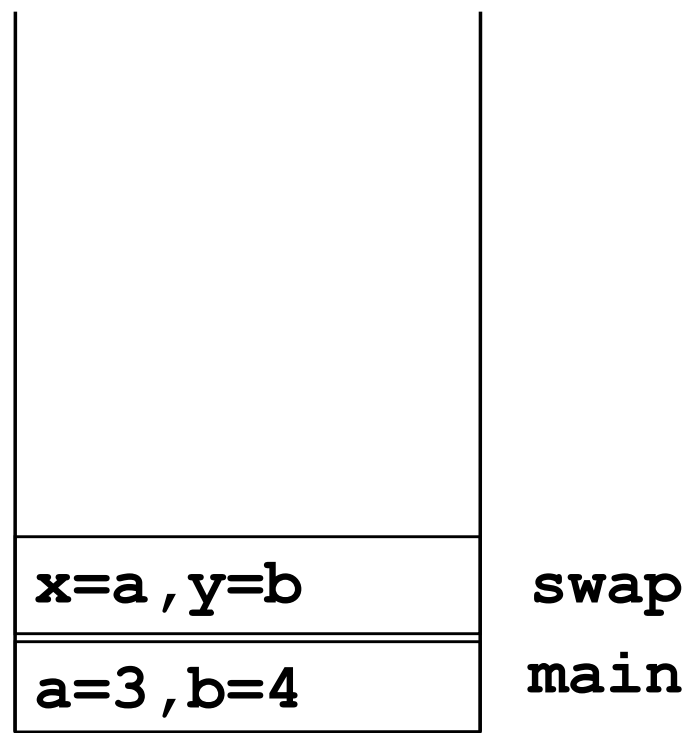
```
void main() ←
{ int x=3,y=4;
  int z = add(x,y);
}
```



函数调用：传值

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

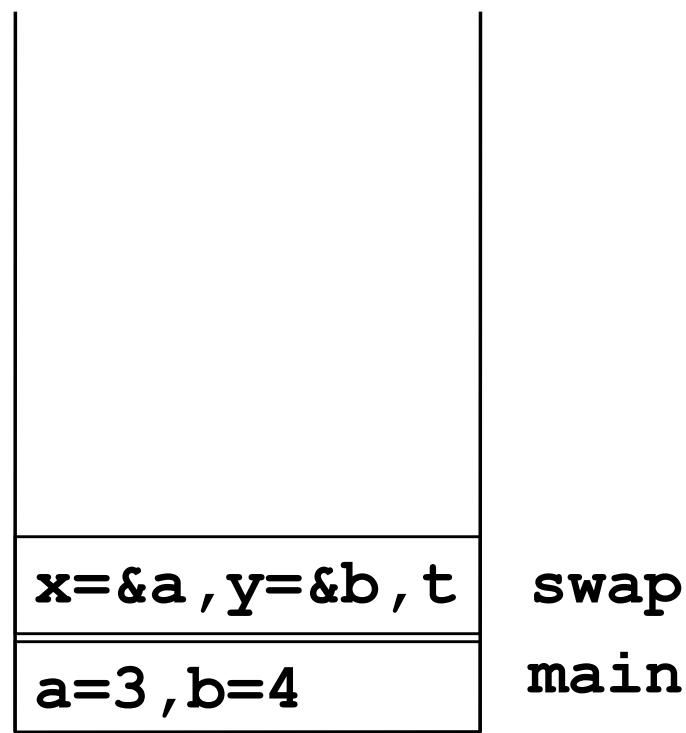
```
int main(){  
    int a = 3,b= 4;  
    swap(a,b);  
    printf("a::%d  b::%d\n",a,b);  
    return 0;  
}
```



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;  
}
```

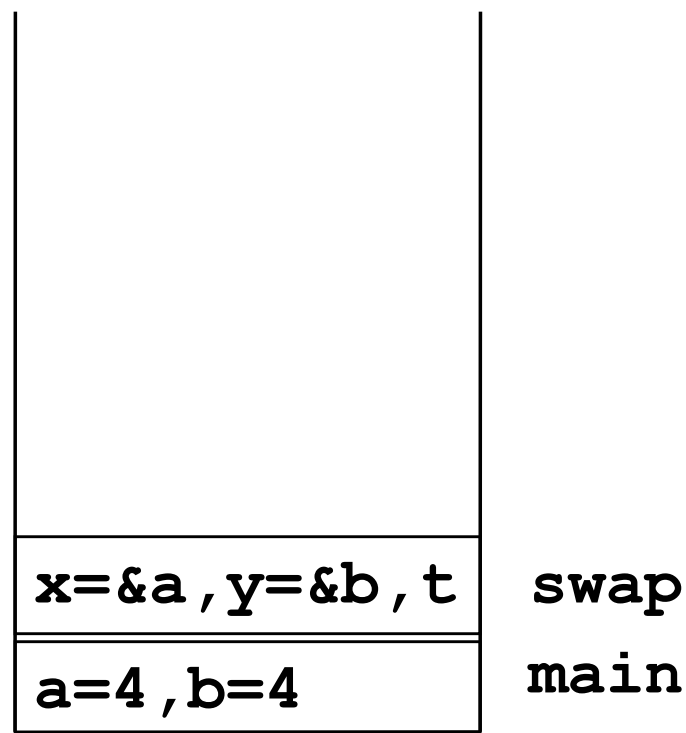


函数调用：传值

```
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;  
}
```

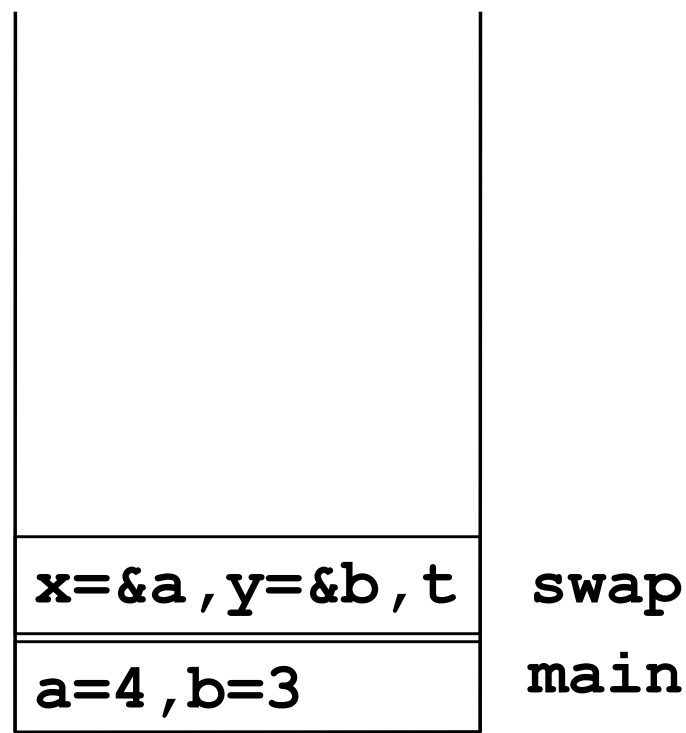


函数调用：传值

```
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;  
}
```

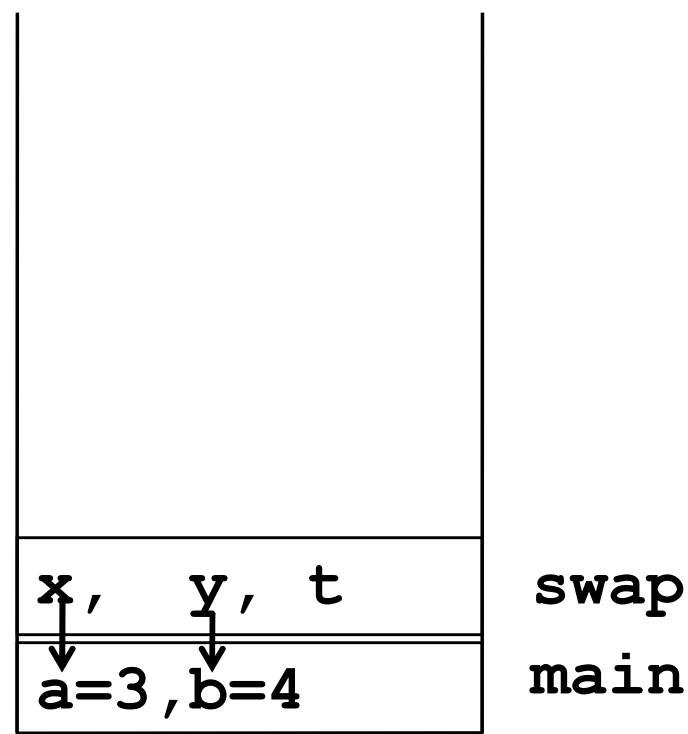


函数调用：传引用

```
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就是a， y就是b



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++;
```

```
}
```

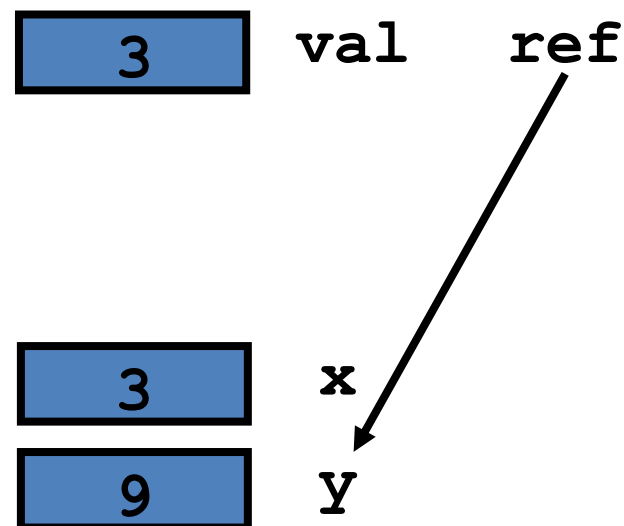
```
void main() {
```

```
    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++;
```

```
}
```

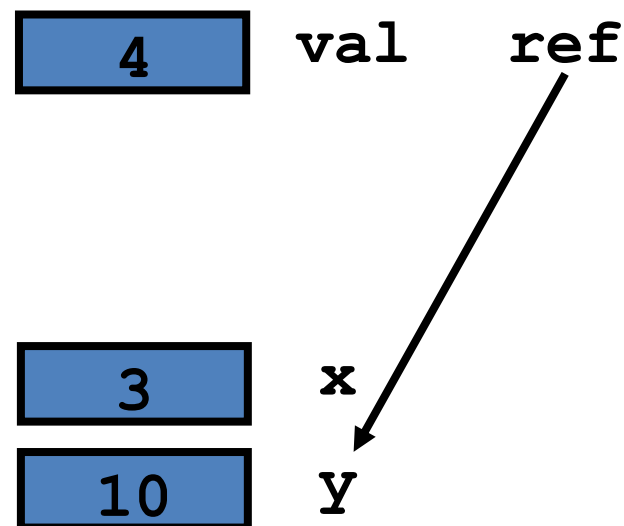
```
void main() {
```

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

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

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

```
}
```

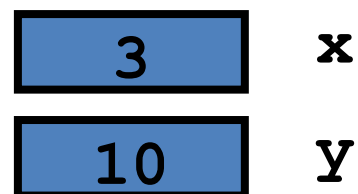


函数调用：传引用

- 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：实参是形参的别名

```
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

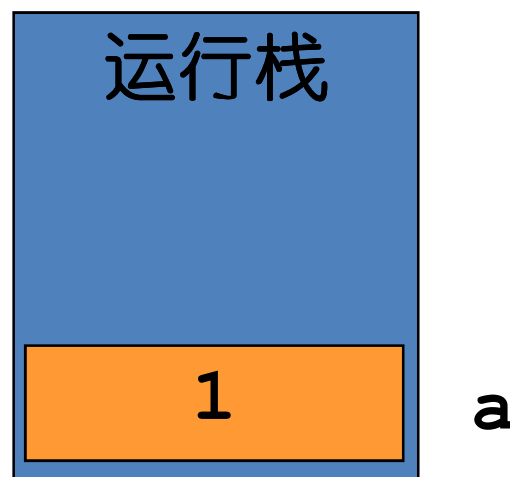
```
void foo()  
{  
    int    a = 1;  
    float f = 1.0;  
}
```

运行栈

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

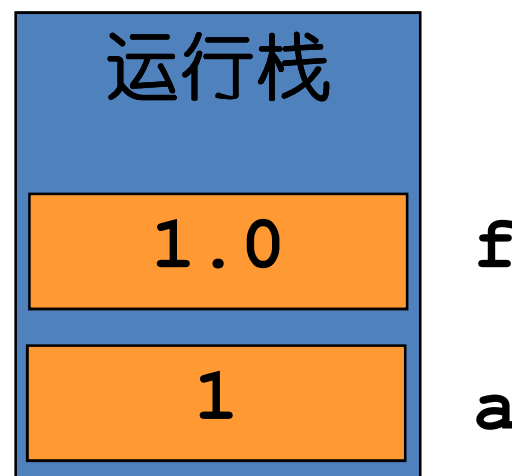
```
void foo()  
{  
    int    a = 1;  
    float f = 1.0;  
}
```



内存分配

- 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

- 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;  
}
```

运行栈

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

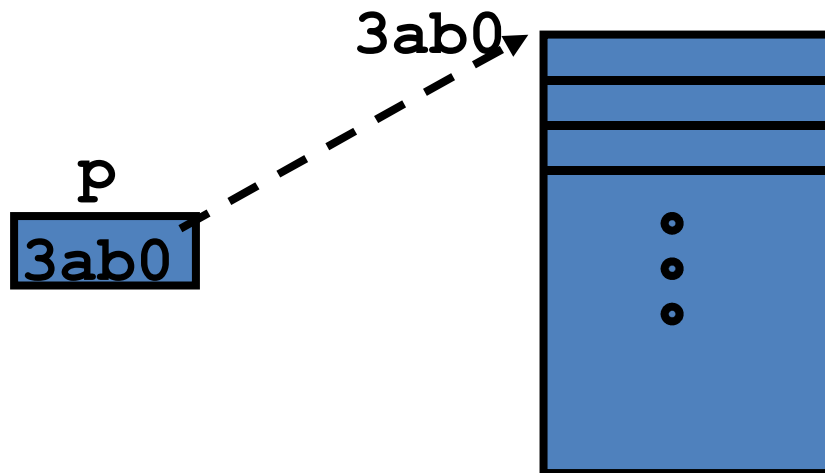
内存分配

```
int *p = malloc (30*sizeof(int)) ; //p = new int[30];
```

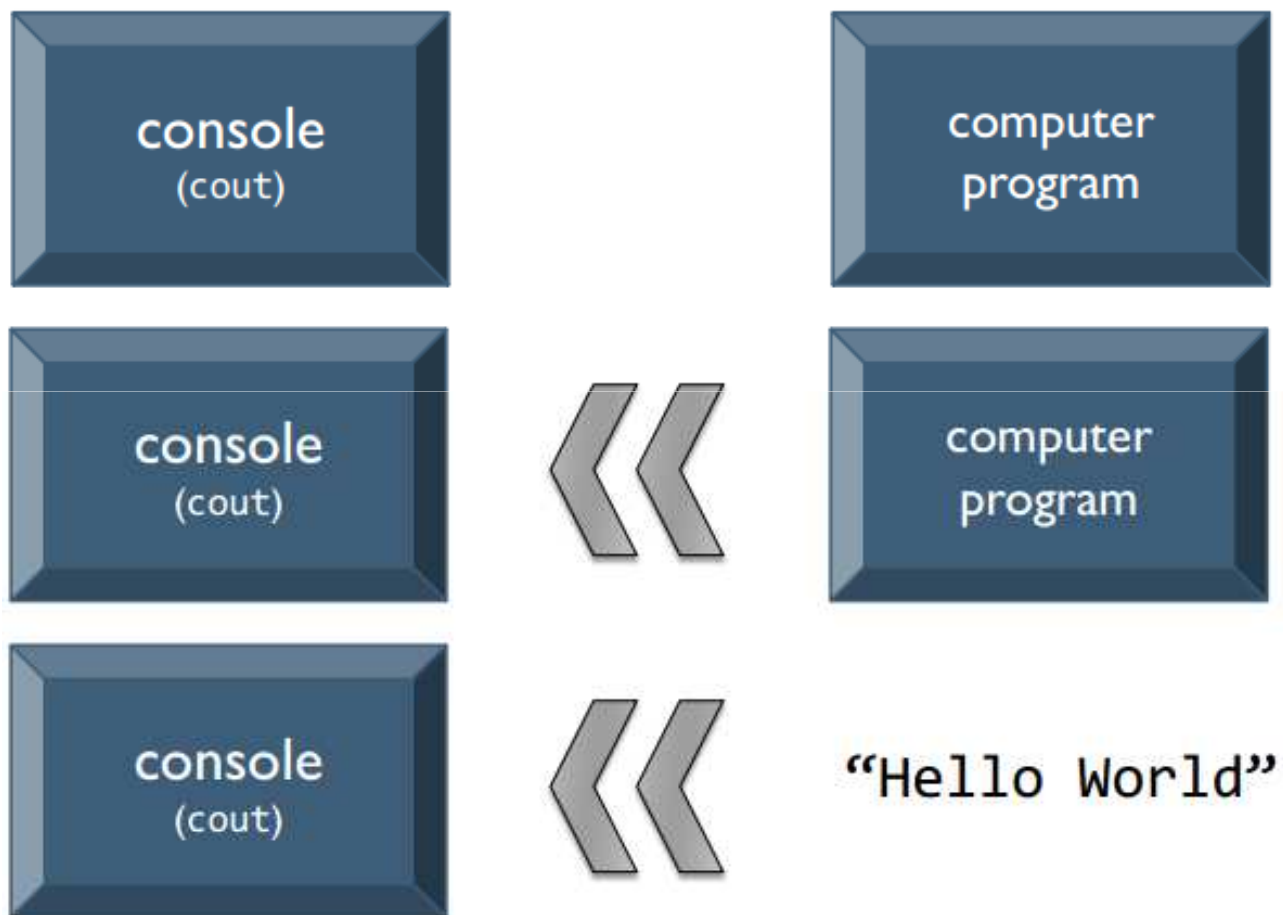
```
P[3] = 20; *(p+4) = 15;
```

...

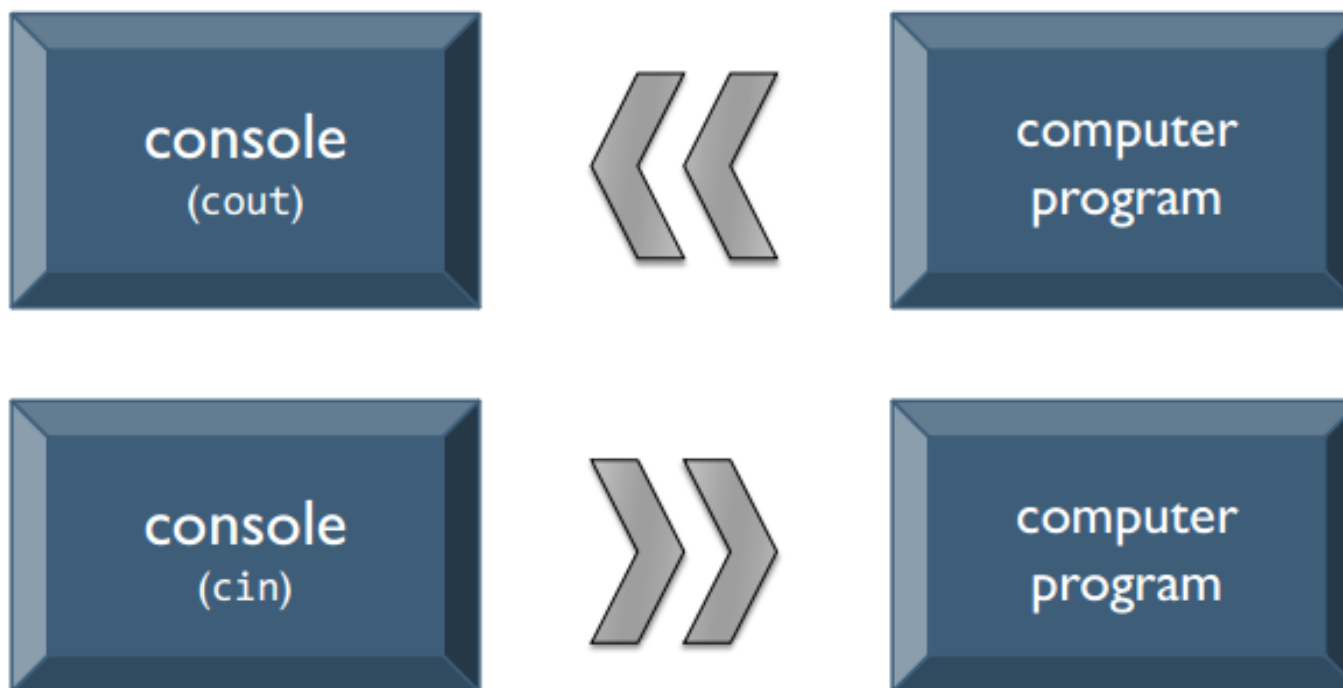
```
free(p) ;// delete[] p;
```



C++输入输出流 stream



C++输入输出流 stream



C++输入输出流 stream

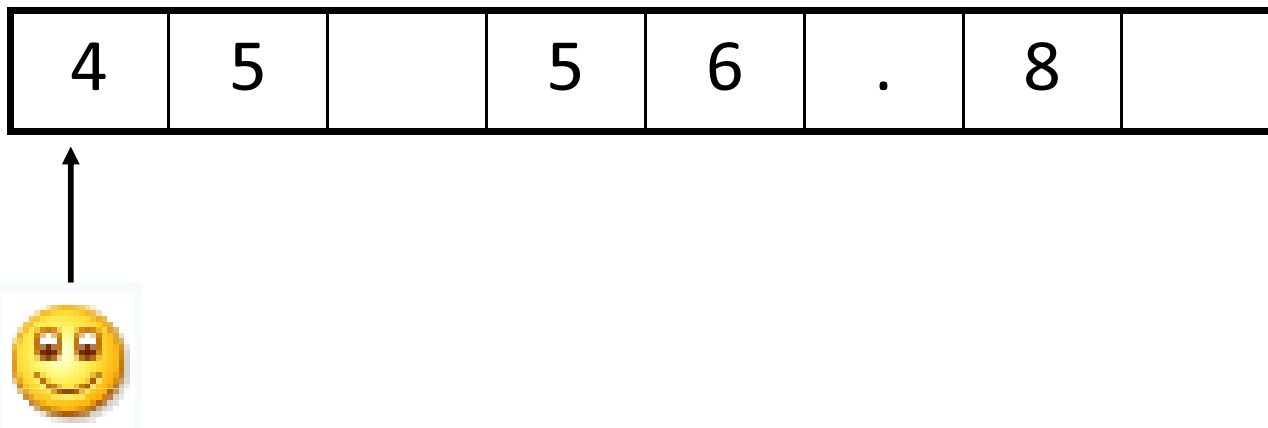


C++输入输出流 stream

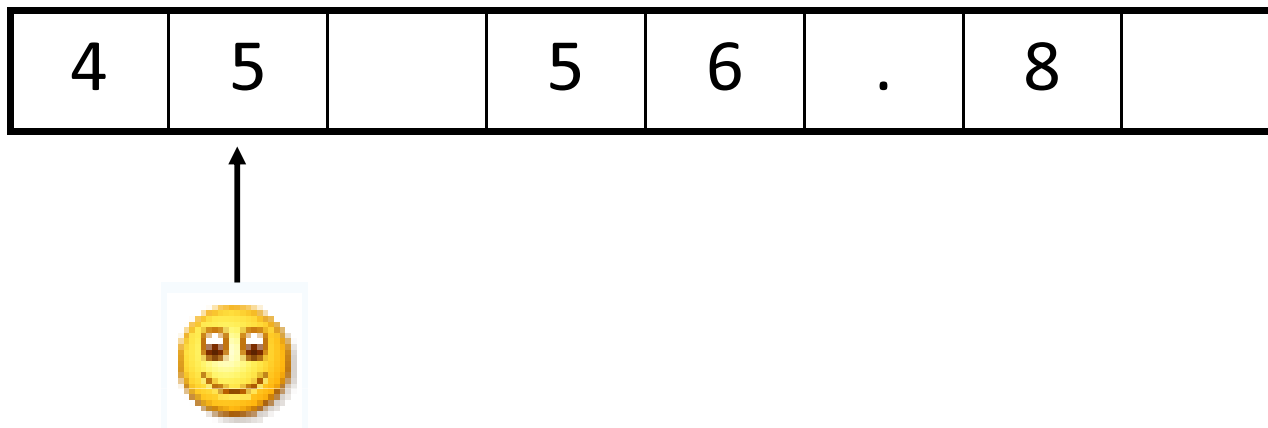
```
#include <iostream>
using std::cin;
using std::cout;
int main(){
    cout<<"hello word!";
}
```

```
#include <iostream>
using std::cin;
using std::cout;
int main(){
    int x, double y;
    cin>>x>>y;
    y = x+y;
    cout<<"y=: "<<y;
}
```

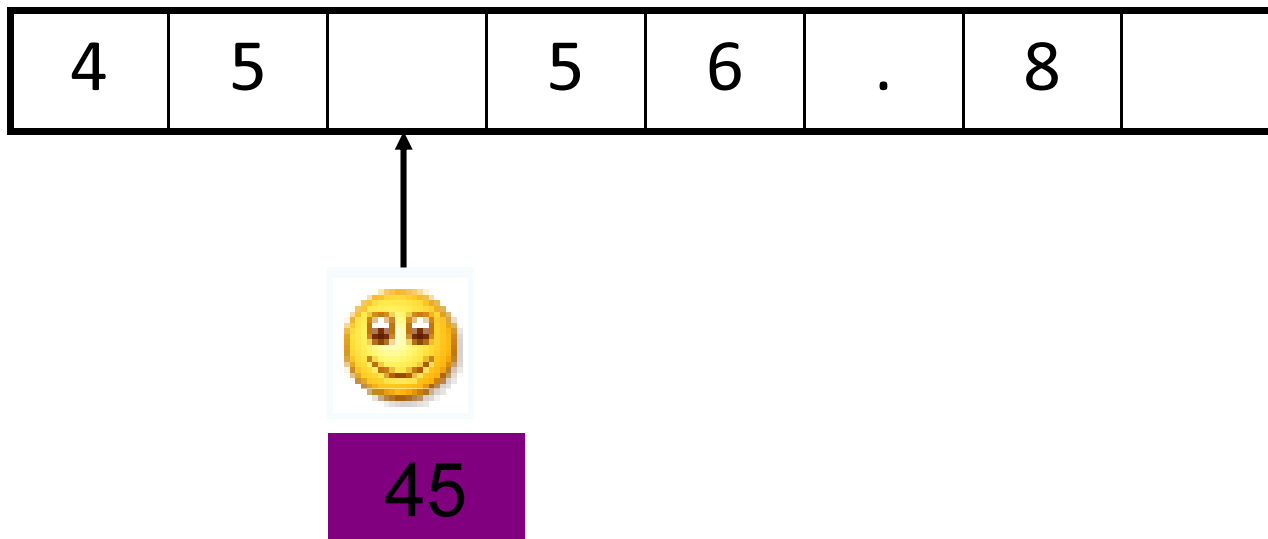
C++输入输出流 stream



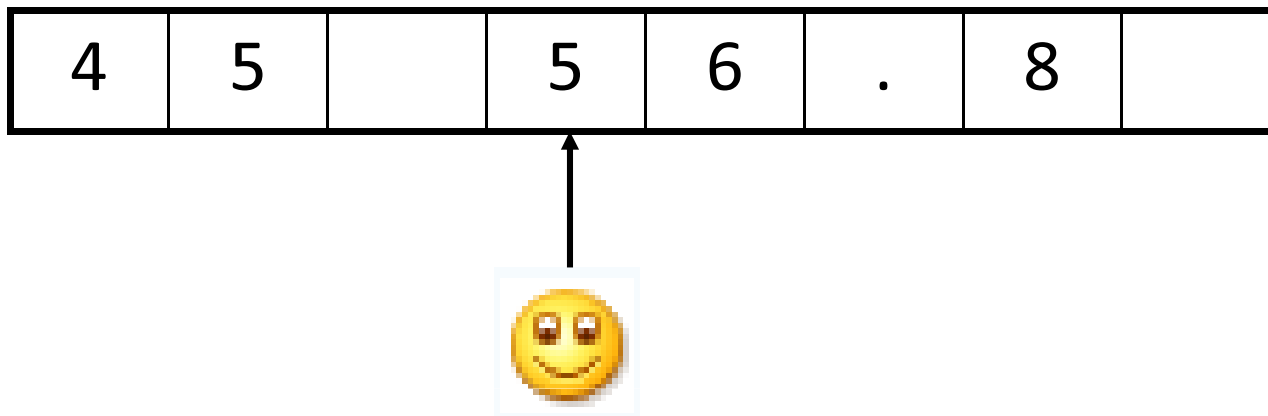
C++输入输出流 stream



C++输入输出流 stream



C++输入输出流 stream



C++输入输出流 stream

4	5		5	6	.	8	
---	---	--	---	---	---	---	--



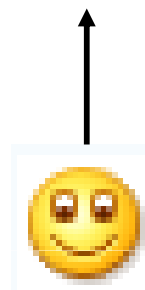
C++输入输出流 stream

4	5		5	6	.	8	
---	---	--	---	---	---	---	--



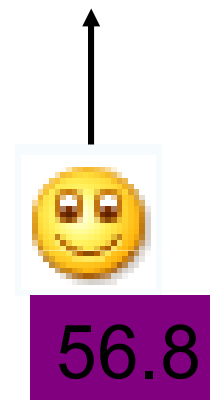
C++输入输出流 stream

4	5		5	6	.	8	
---	---	--	---	---	---	---	--



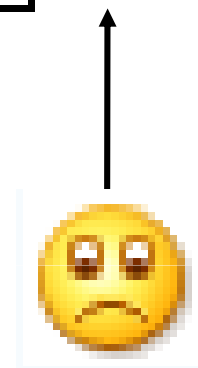
C++输入输出流 stream

4	5		5	6	.	8	
---	---	--	---	---	---	---	--



C++输入输出流 stream

4	5		5	6	.	8	
---	---	--	---	---	---	---	--



Game Over!

C++输入输出流 stream

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

```
#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;
}
```

C++: string (字符串)

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string s1,s2="world";
    cin>>s1;
    string s = s1+s2;
    s[2] = 'H';
    cout<<" length of s("<<s<<"="<<s.size();
    return 0;
}
```

C++: string (字符串)











- s1,s2,s是string类型的3个变量(对象)
- s.size()调用对象s的成员函数size(),返回字符串s的字符个数。
- google一下string类的成员(数据或函数成员)

[http://en.wikipedia.org/wiki/String_\(C%2B%2B\)](http://en.wikipedia.org/wiki/String_(C%2B%2B))

C++: string (字符串)





Overview of functions

[\[edit\]](#)







- `string::string`  (constructor) - Constructs the string from variety of sources
- `string::~~string`  (destructor) - Destructs the string
- `string::operator=`  - Replace the string
- `string::assign`  - Replace the string
- `string::get_allocator`  - Returns the allocator used to allocate memory for the bytes
- Byte access
 - `string::at`  - Accesses specified byte with bounds checking.
 - `string::operator[]`  - Accesses specified byte
 - `string::front`  - Accesses the first byte
 - `string::back`  - Accesses the last byte
 - `string::data`  - Accesses the underlying array

C++: string (字符串)










- Iterators

- `string::begin`  - Returns an iterator to the beginning of the string
- `string::end`  - Returns an iterator to the end of the string
- `string::rbegin`  - Returns a reverse iterator to the reverse beginning of the string
- `string::rend`  - Returns a reverse iterator to the reverse end of the string

- Capacity

- `string::empty`  - Checks whether the string is empty
- `string::size`  - Returns the number of bytes in the string.
- `string::max_size`  - Returns the maximum possible number of bytes in the string.
- `string::reserve`  - Reserves storage in the string
- `string::capacity`  - Returns the current reservation
- `string::shrink_to_fit`  (C++11) - Reduces memory usage by freeing unused memory

- Modifiers

- `string::clear`  - Clears the contents
- `string::insert`  - Inserts bytes or strings
- `string::erase`  - Deletes bytes
- `string::push_back`  - Appends a byte
- `string::append`  - Appends bytes or strings
- `string::operator+=`  - Appends
- `string::pop_back`  - Removes the last byte
- `string::resize`  - Changes the number of stored bytes
- `string::swap`  - Swaps the contents with another string

C++: 类和对象

//student.h

```
#include <string>
```

```
using namespace std;
```

```
class student{
```

```
    string name; double score;
```

```
public:
```

```
    student(string n="Li",double s = 60.){
```

```
        name = n; score = s;
```

```
    }
```

```
    string get_name(){return name;}
```

```
    void set_name(string n){ name = n;}
```

```
    //...
```

```
};
```


C++: 类和对象

```
//main.cpp
#include <student.h>
#include <iostream>
int main(){
    student s1,s2("Zhang",80);
    s1.set_name(s2.get_name());
    std::cout<< s1.get_name() <<"\n";
    return 0;
}
```

C++: 类和对象

- C++中的类是一种用户定义类型，而类的对象则是该类的一个变量。如

```
string s;
```

```
student stu;
```

- 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). 如

```
int n = s.size();
```

```
stu.get_name() ;
```

- 类的成员函数对类中的数据进行处理。

```
void set_name(string n){ name = n;}
```

C++: 类和对象

- 构造函数：与类名同名的函数，无返回值。用于初始化类对象的数据及分配资源。
- 析构函数：无返回值。用于销毁类对象并释放占用的资源。

```
class X{  
    //...  
    X() { ... }  
    ~X() { ... };  
};
```

```
class student{  
    string name; double score;  
public:  
    student(string n="Li",double s = 60.){  
        name = n; score = s; }  
    ~student() { }  
};
```

C++: 类和对象

- 成员函数可以在类体内或类体外定义

```
class X{  
    //...  
    X();  
    void fun( );  
    ~X();  
};
```

```
X::X(){  
    //...  
}
```

```
X::~~X(){  
    //...  
}
```

```
void X::fun( ){  
    //...  
}
```

C++: 学生成绩单

- 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

C++: 学生成绩单

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

```
typedef TypeA TypeB;
```

给类型TypeA起别名叫TypeB

```
typedef int INT;  
  
void main(){  
    int a = 3; INT b=4;  
    a = b;  
}
```

C++: 学生成绩单

```
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(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);
}
```

C++: 学生成绩单

输入输出一个学生信息的辅助函数

```
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);  
}
```


C++: 学生成绩单

```
int main(){
    student stus[100];
    int i = 0,j = 0,k=0 ;
    do{
        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);
}
```

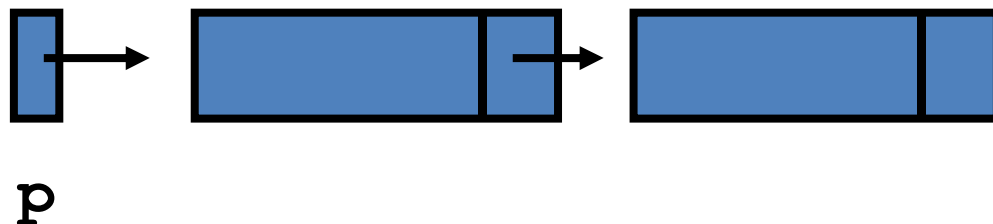
C++: 学生成绩单

- 静态数组:浪费空间和空间不够
- 解决方法:

1) 动态分配数组空间

```
student *p = malloc(N*sizeof(student));
```

2) 动态分配单个student, 并用链表串起来



C++: 学生成绩单:动态数组

```
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));
```

```
stus = stusNew;
```

用完后要释放空间: free(stus);

C++: 学生成绩单:动态数组

```
int main(){
    int size = INITSIZE; int i = 0,j = 0,k=0 ;
    student *stus = (student *) malloc(size * sizeof(student));
    do{
        if(i>=size){ size+=INC;
            student * stusNew =(student *) realloc(size*sizeof(student));
            free(stus); stus = stusNew;
        }
        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);
}
```

C++: 学生成绩单:链表

```
typedef struct{
    student data; //Type data;
    LNode *next;
} LNode;
LNode *p;

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

C++: 学生成绩单:链表

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

C++: 学生成绩单: 容器类模板

- C++提供了许多容器类: 向量vector 和链表 list

```
#include <vector> // #include<list>
```

```
//using namespace std;
```

```
void main(){
```

```
    vector<int> ints;
```

```
    vector<double> doubles;
```

```
    ints.push_back(3); ints.push_back(4);
```

```
    doubles.push_back(30); doubles.push_back(4.5);
```

```
    for(int i= 0 ; i<ints.size() ; i++)
```

```
        std::cout << ints[i] << " ";
```

```
    std::cout<<std::endl;
```

```
}
```

C++: 学生成绩单: 容器类模板

```
#include <cstdio>
int main( ){
    vector<student> students;
    student s;
    do{
        In_student(s);    if(s.score<0) break;
        students.push_back(s);
    } while(1) ;
    for(int i = 0 ; i<students.size();i++){
        Out_student(students[i]);
    }
    std::cout<<"\n";    return 0;
}
```


From C to C++

- include一个库的新方式
`#include<cmath> // #include<math.h>`
`using namespace std;` 包含名字空间的名字
- 单行注释：用// 注释单行代码
- 控制台Console输入输出流,具体的Console输入输出流对象
`std::cin` `std::cout`
- 可以在任意位置声明(定义)变量
`int x,y =4;`
`x = y+1;`
`int z = x*y;` //可以在任意位置定义变量

From C to C++

```
#include <iostream> // This is a key C++ library
#include <cmath> // The standard C library math.h
using namespace std;
int main (){
    double a;
    cin>> //从控制台读入一个实数;
    cout << a << std:: endl; //输出实数a和一个回车符
    int b= 30;    //可以在任意位置定义变量
    cout << a+b << "\n";
    return 0;
}
```

From C to C++

- 变量可以定义在一个循环语句块内
- 即使程序块内有一个同名变量，也可以访问全局变量

```
#include <iostream>
using namespace std;
int i = 100;

void main (){
    int i=487; // Simple declaration of i
    for (int i = 0; i < 4; i++) { // Local declaration of i
        cout << i <<endl; // This outputs 0, 1, 2 and 3
    }
    cout << i <<::i<< endl; // This outputs 487,100
}
```

From C to C++

- C只有值类型，而C++新增了引用类型

```
double a;    double &b = a;
```

- 名字空间用于防止名字冲突

```
namespace ns_one{
```

```
    int a, b;
```

```
}
```

```
namespace ns_two{
```

```
    double a, b;
```

```
}
```

```
void main( ){
```

```
    ns_one::a = 3;
```

```
    ns_two ::a = 37.8;
```

```
    std::cout << ns_one::a << “ “ << ns_two::a << “\n”;
```

```
}
```

From C to C++

- 函数可以有默认参数(default parameter)

```
#include <iostream>
```

```
int add(int a,int b=3) { return a+b;}
```

```
void main(){
```

```
    std::cout<<add(10)<<std::endl;
```

```
    std::cout<<add(10,7)<<std::endl;
```

```
}
```

From C to C++

- 新的动态内存分配运算符new和delete

Type *v=new Type(); // 分配单个Type类型的存储

Type *array = new Type[9]; // 分配Type类型的数组

delete v; //释放单个Type的存储

delete[] array; //释放Type的数组存储

```
void main(){
    int *arr = new int[9];
    arr[0] = 30; arr[3] = 12;
    std::cout<<arr[0]<<arr[1]<<arr[3];
    delete[] arr;
}
```

From C to C++

- 在struct和class内可以增加函数-称为成员函数

```
struct Point2D{  
    double _x,_y;  
    double getX() { return _x; }  
    double setX(int x) { return _x=x; }  
    Point2D(double x, double y=0) { _x = x; _y =y; }  
}
```

```
void main(){  
    Point2D P(3.3,5.6); P.setX(100);  
    std::cout << P.x << " " << P.y << "\n";  
}
```

From C to C++

- 模板-泛型编程

C

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

C++

普通
函数

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

C++
模板

```
template<typename T>  
T max(T x, T y) { return x>y?x:y; }
```


From C to C++

```
#include <iostream>
```

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

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

```
void main(){
```

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

```
    double a=30.5, b = 60;
```

```
    std::cout << max(x,y) << std::endl;
```

```
    std::cout << max(a,b) << std::endl;
```

```
}
```

From C to C++

```
#include <iostream>
```

```
template<typename T> T max (T x, T y) { return x>y ? x : y; }
```

```
void main(){
```

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

```
    double a=30.5, b = 60;
```

```
    std::cout << max(x,y) << std::endl;
```

```
    std::cout << max(a,b) << std::endl;
```

```
}
```

From C to C++

- more...

C++开发工具

- g++ (linux/Unix...)
- VC2010 ,VC2008, (windows) .VC6 is out
- Dev C++(free)

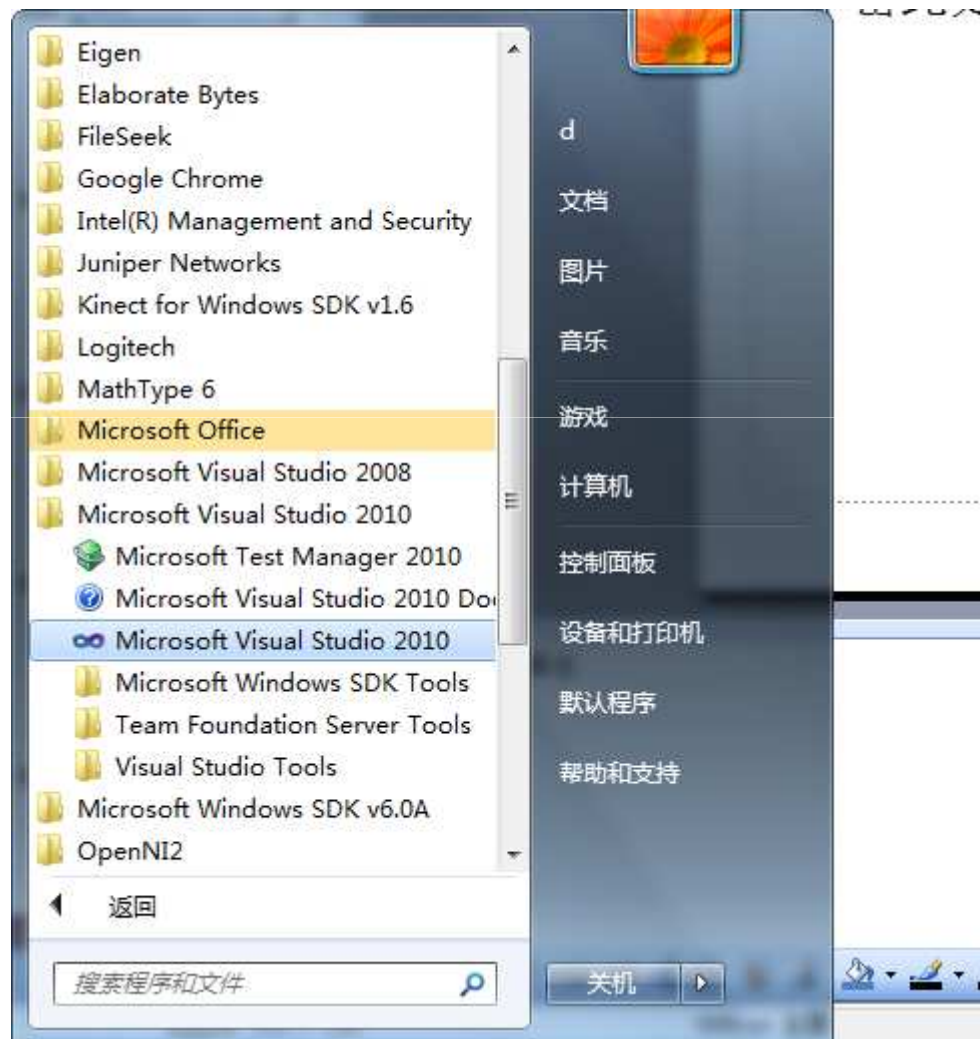
<http://www.bloodshed.net/dev/devcpp.html>

- CodeBlock

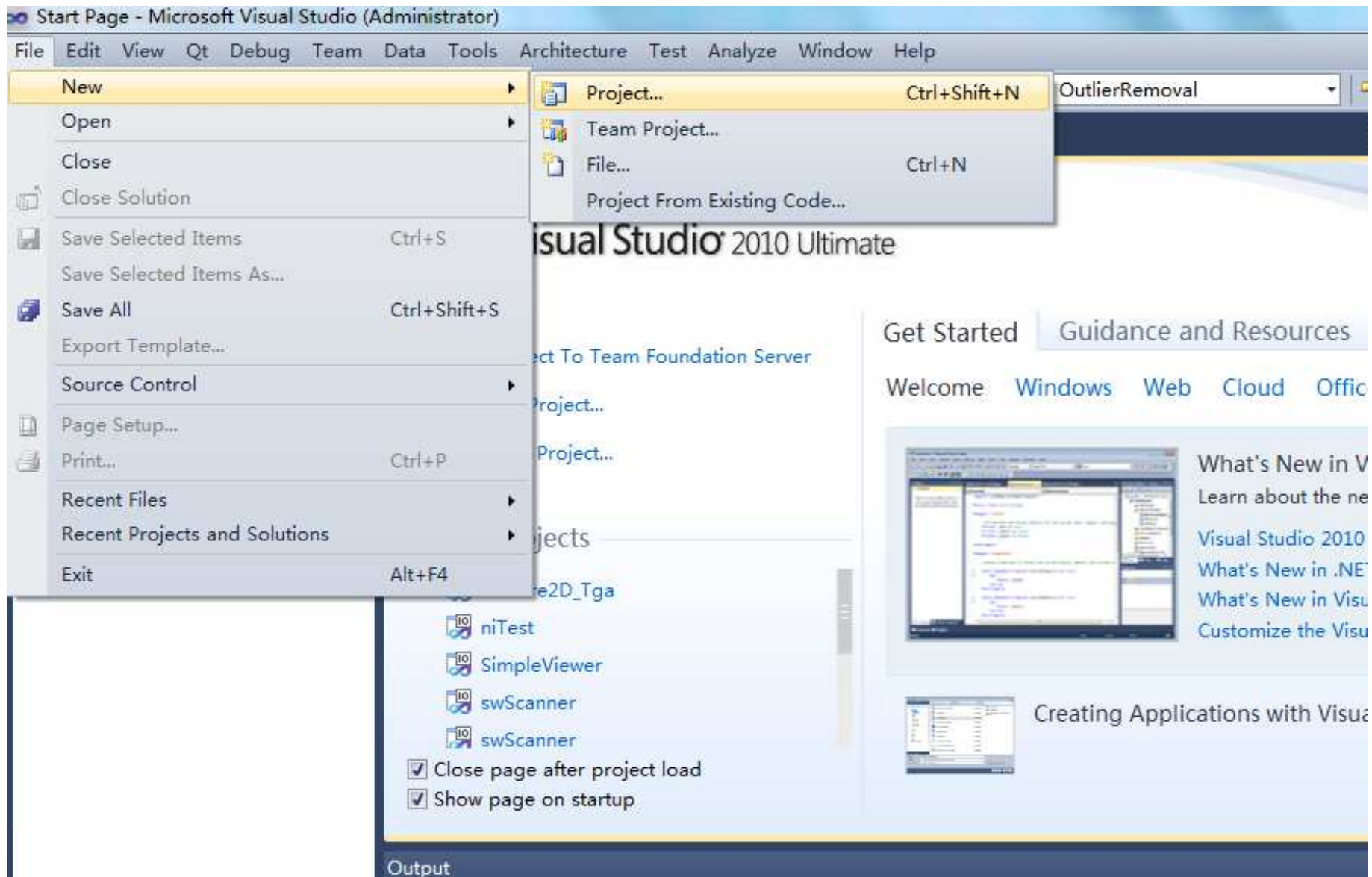
VC2010

- File->New->Project: “win32 console”,...
select “Empty project”,...
- File->New->File:
- File->Open->Project/solution: .sln

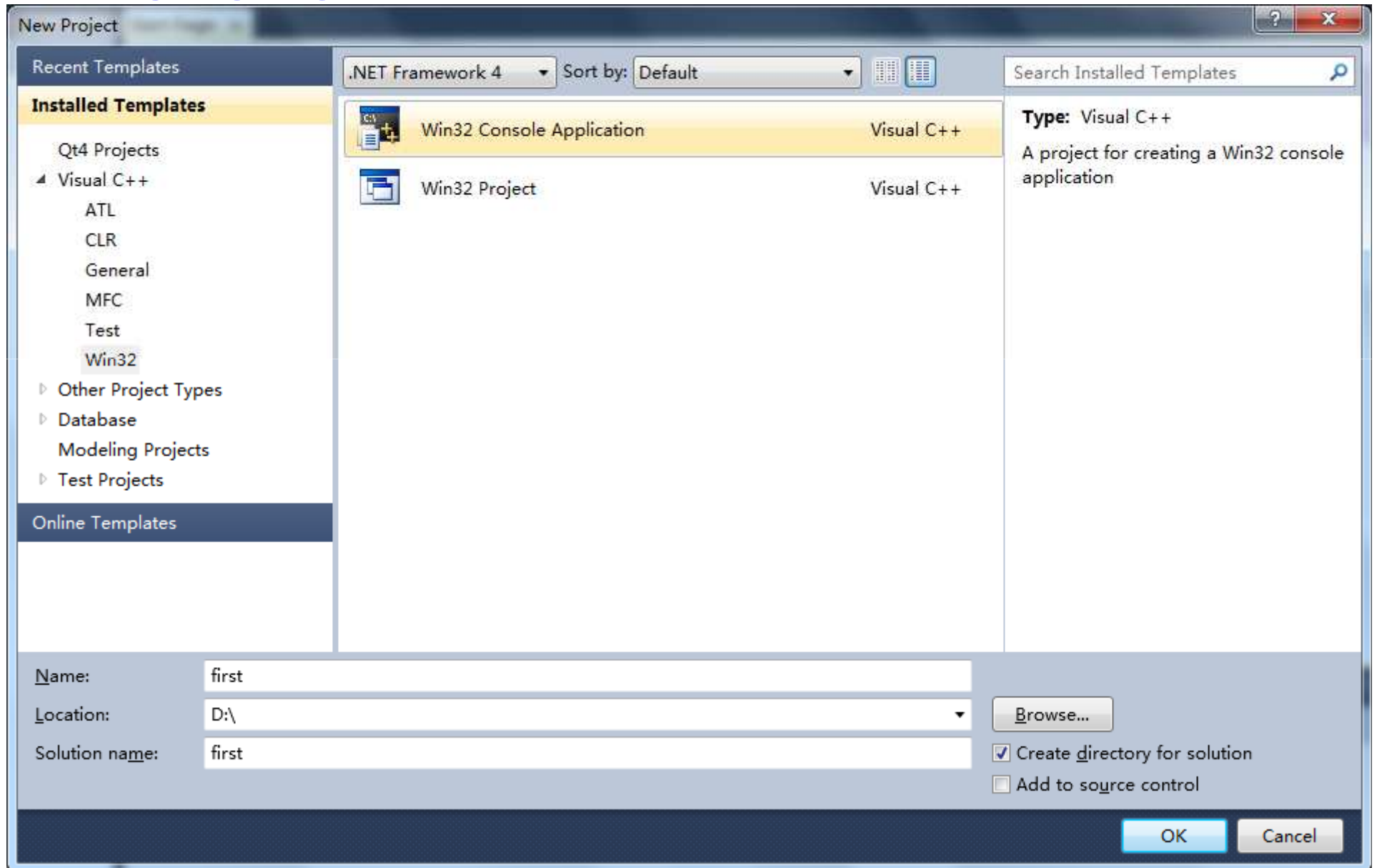
VC2010



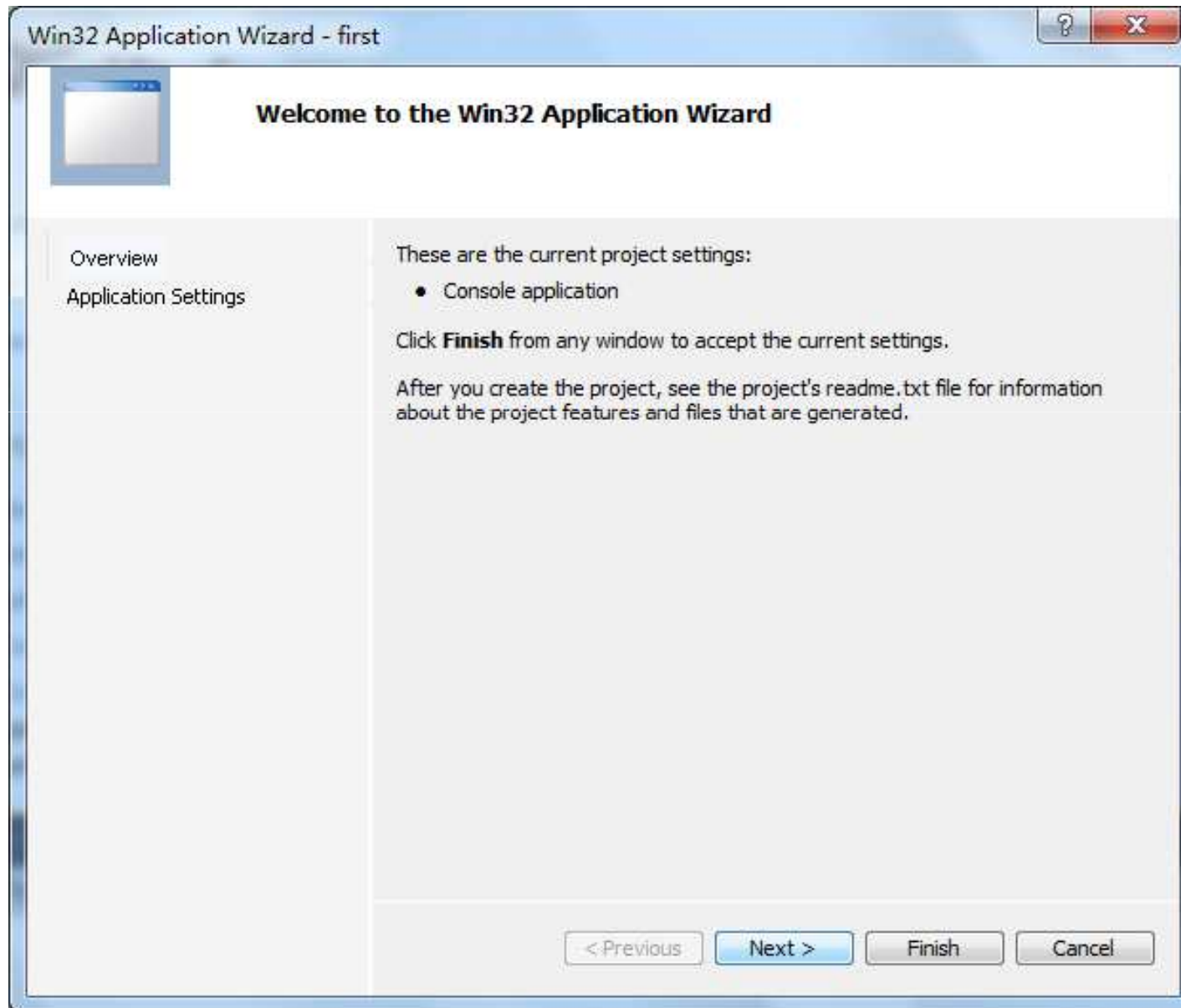
VC2010



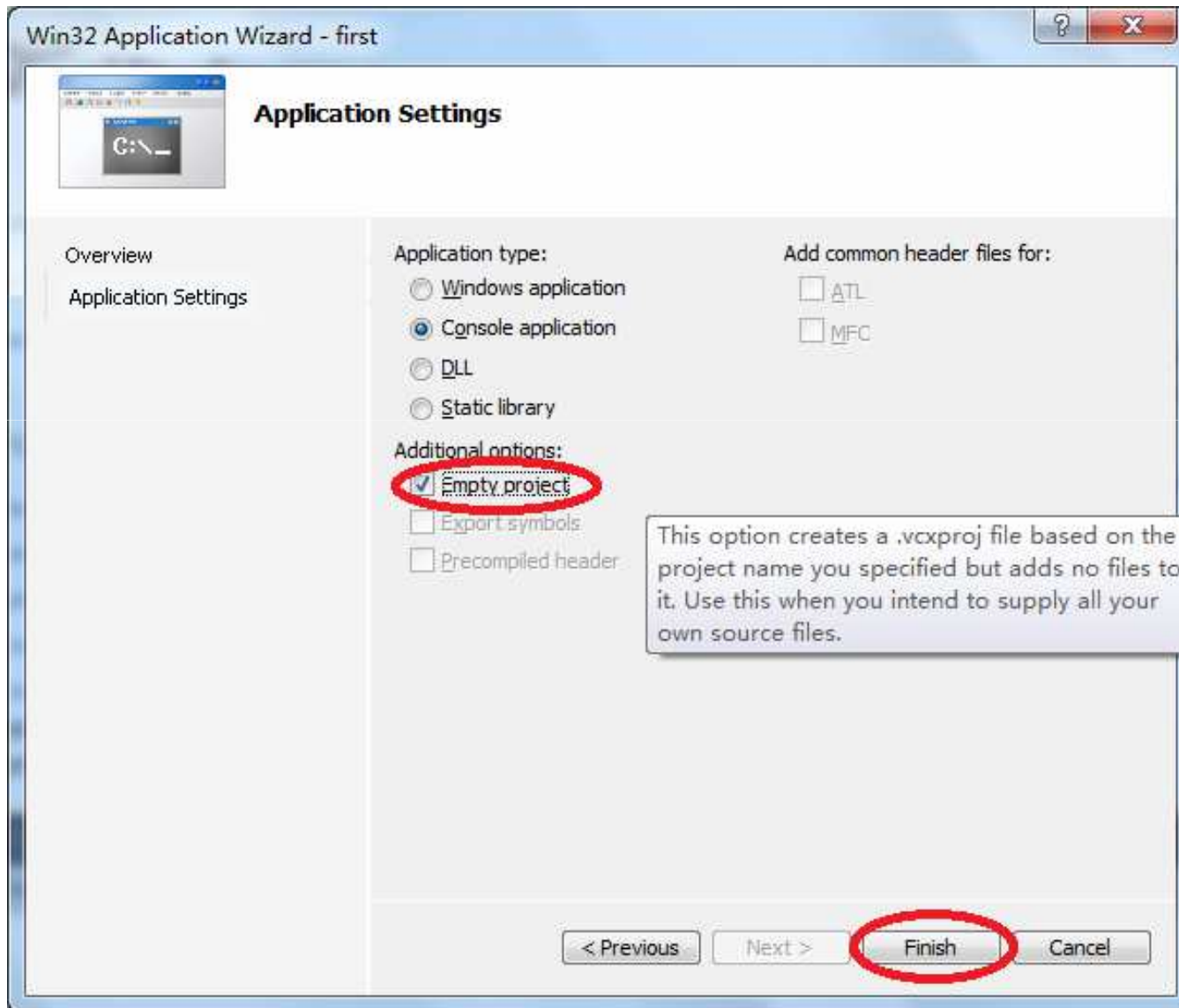
VC2010



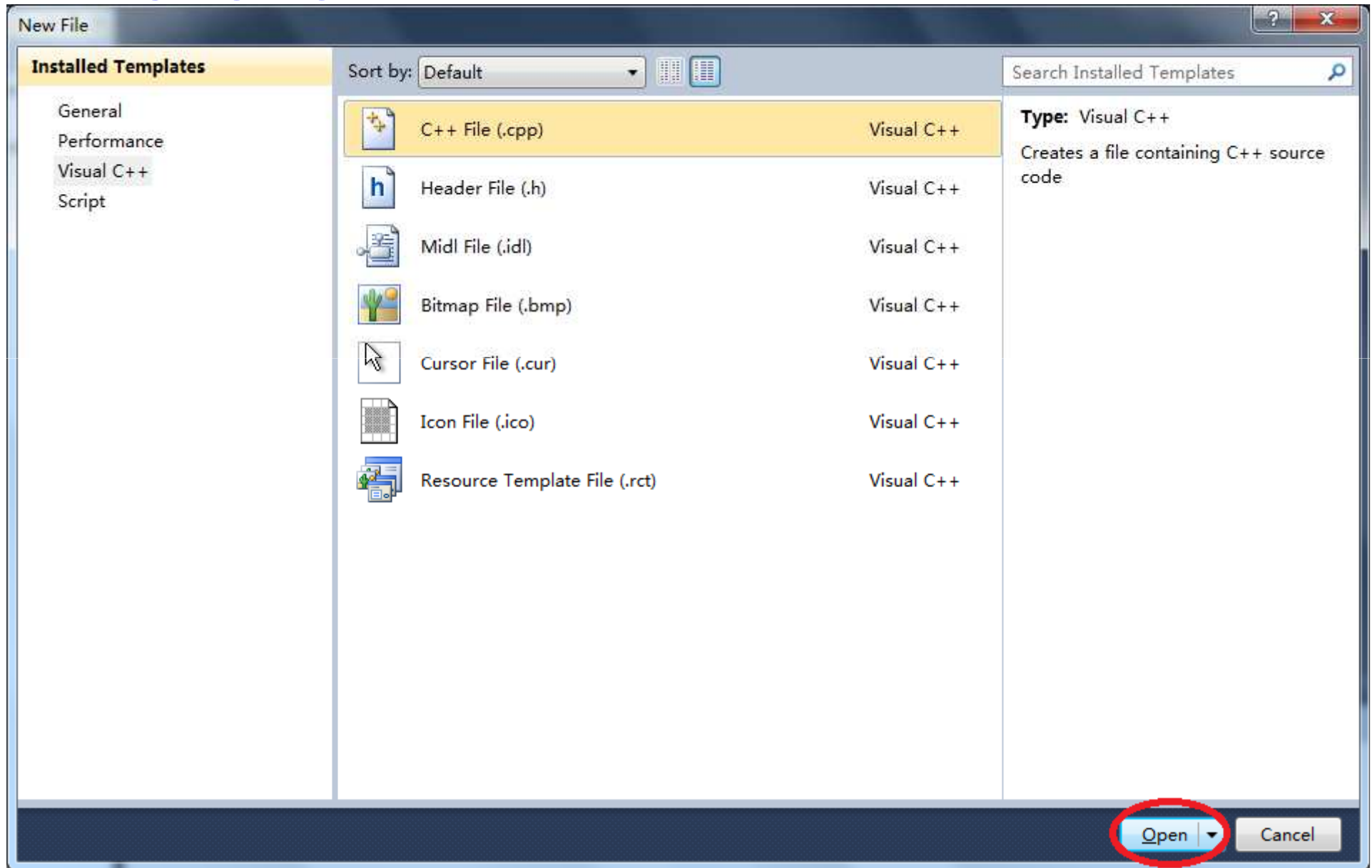
VC2010



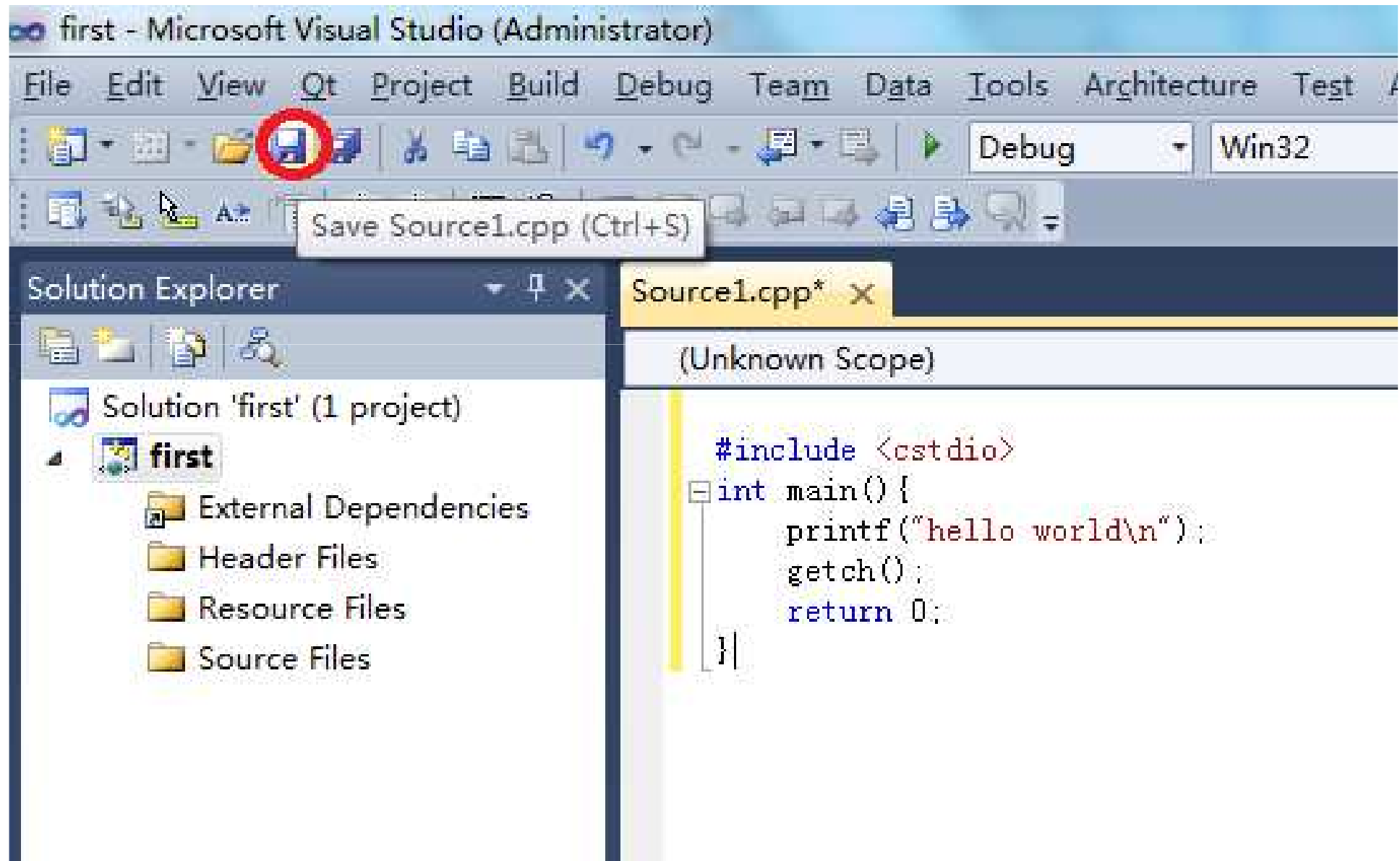
VC2010



VC2010



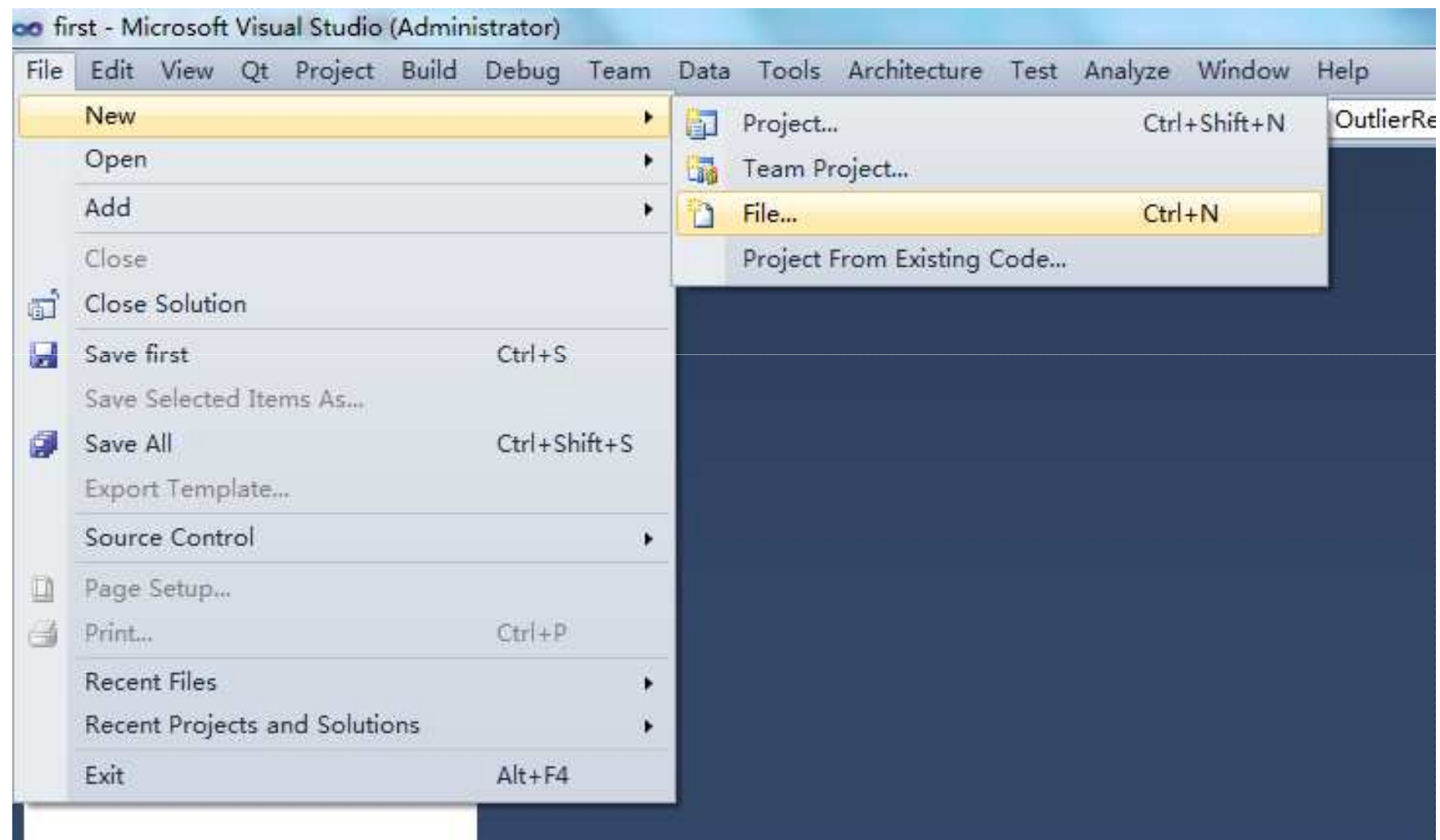
VC2010



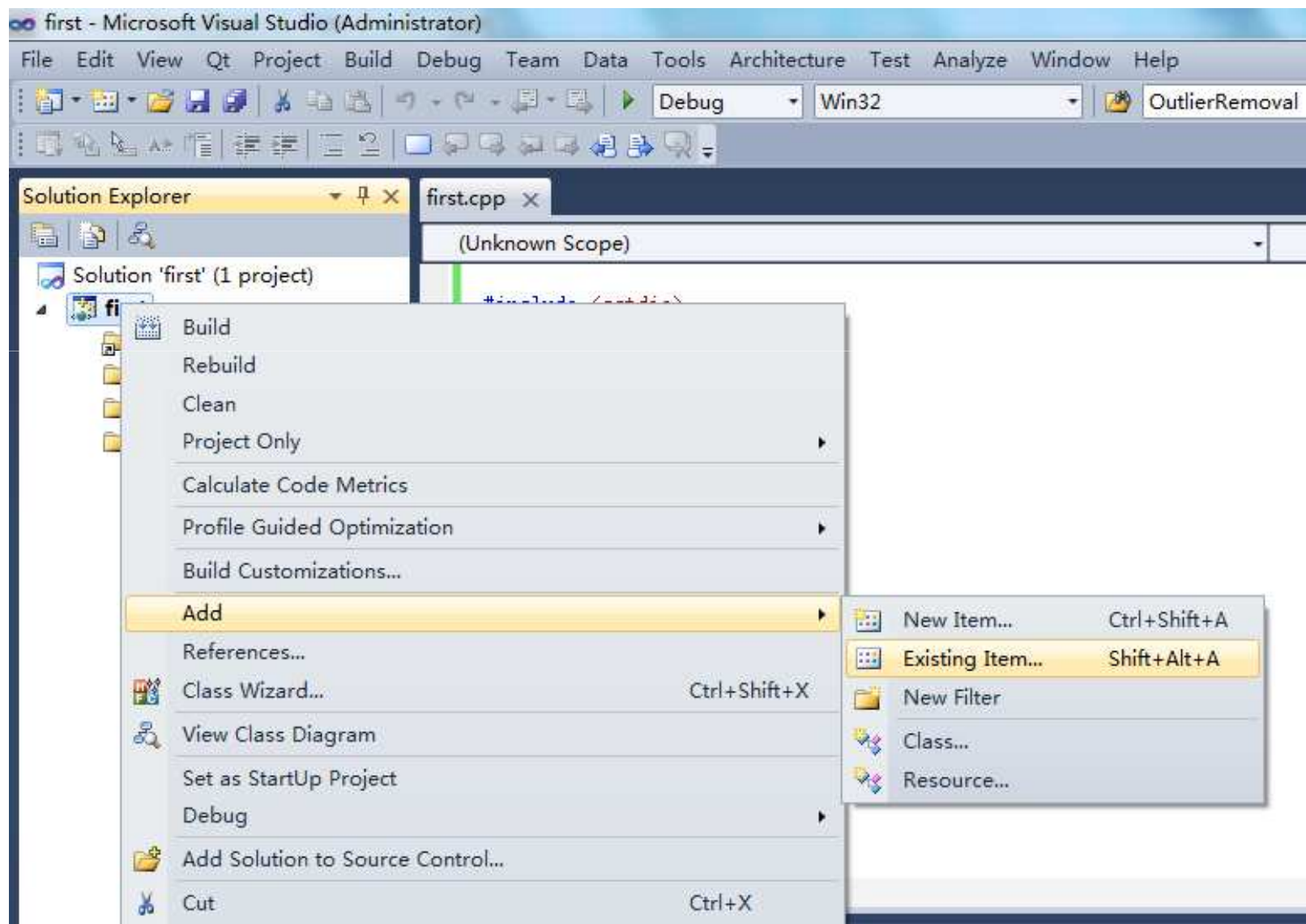
VC2010



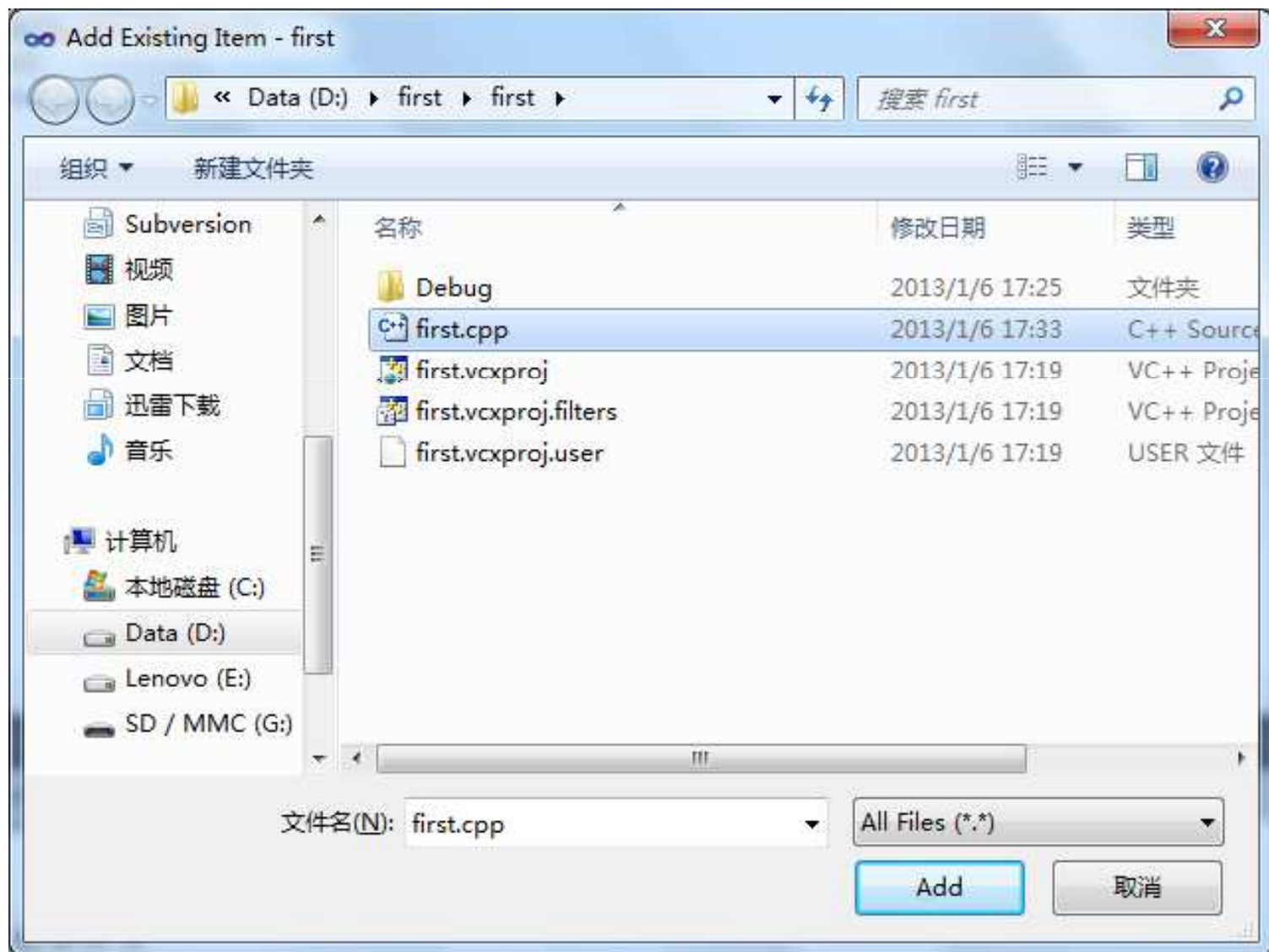
VC2010



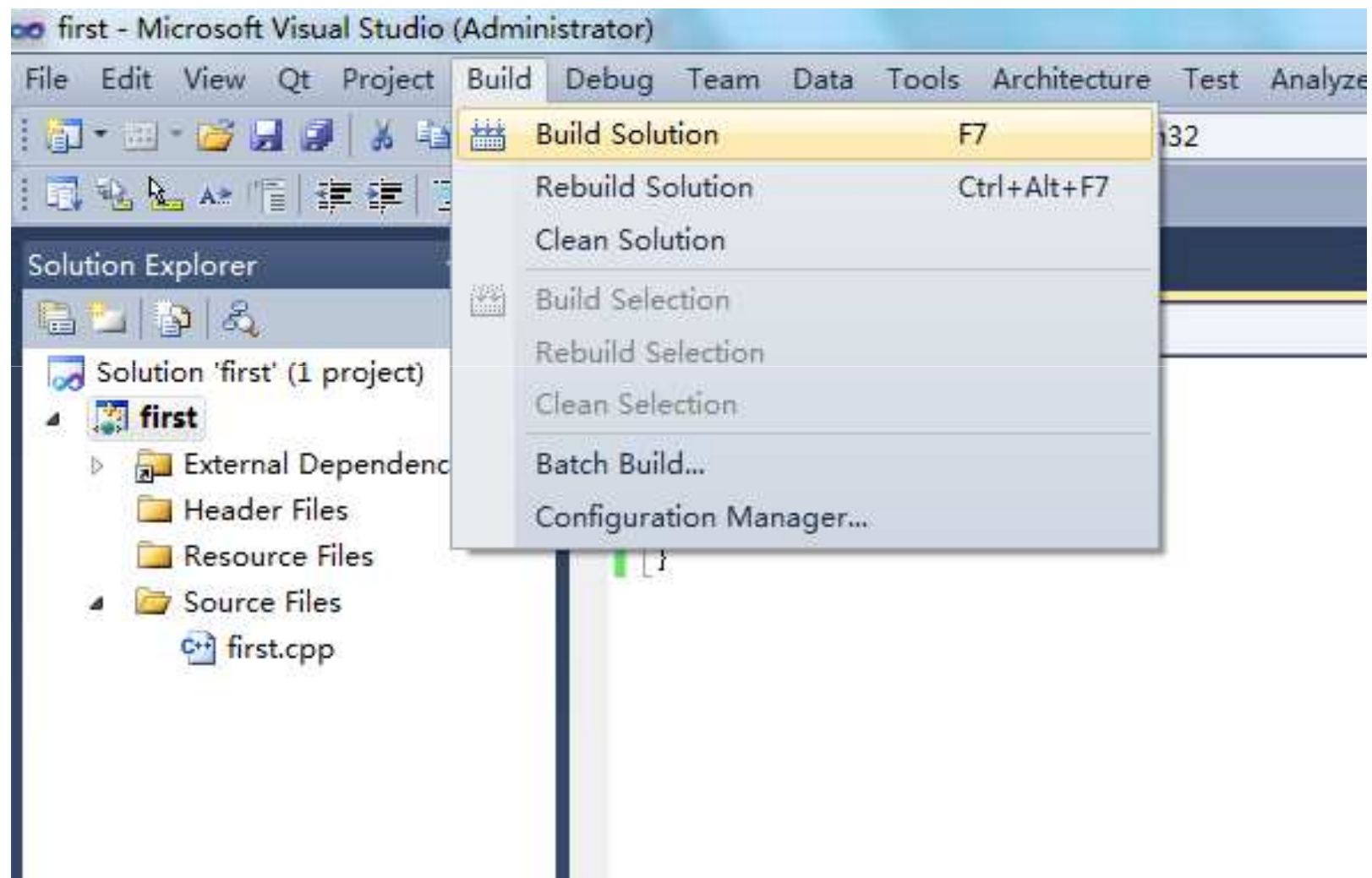
VC2010



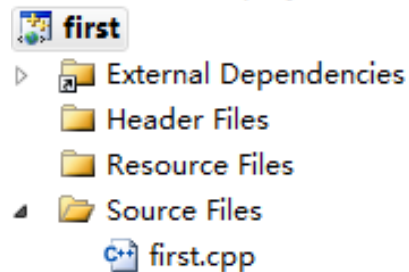
VC2010



VC2010



VC2010



```
int main() {  
    printf("hello world\n");  
    getch();  
    return 0;  
}
```

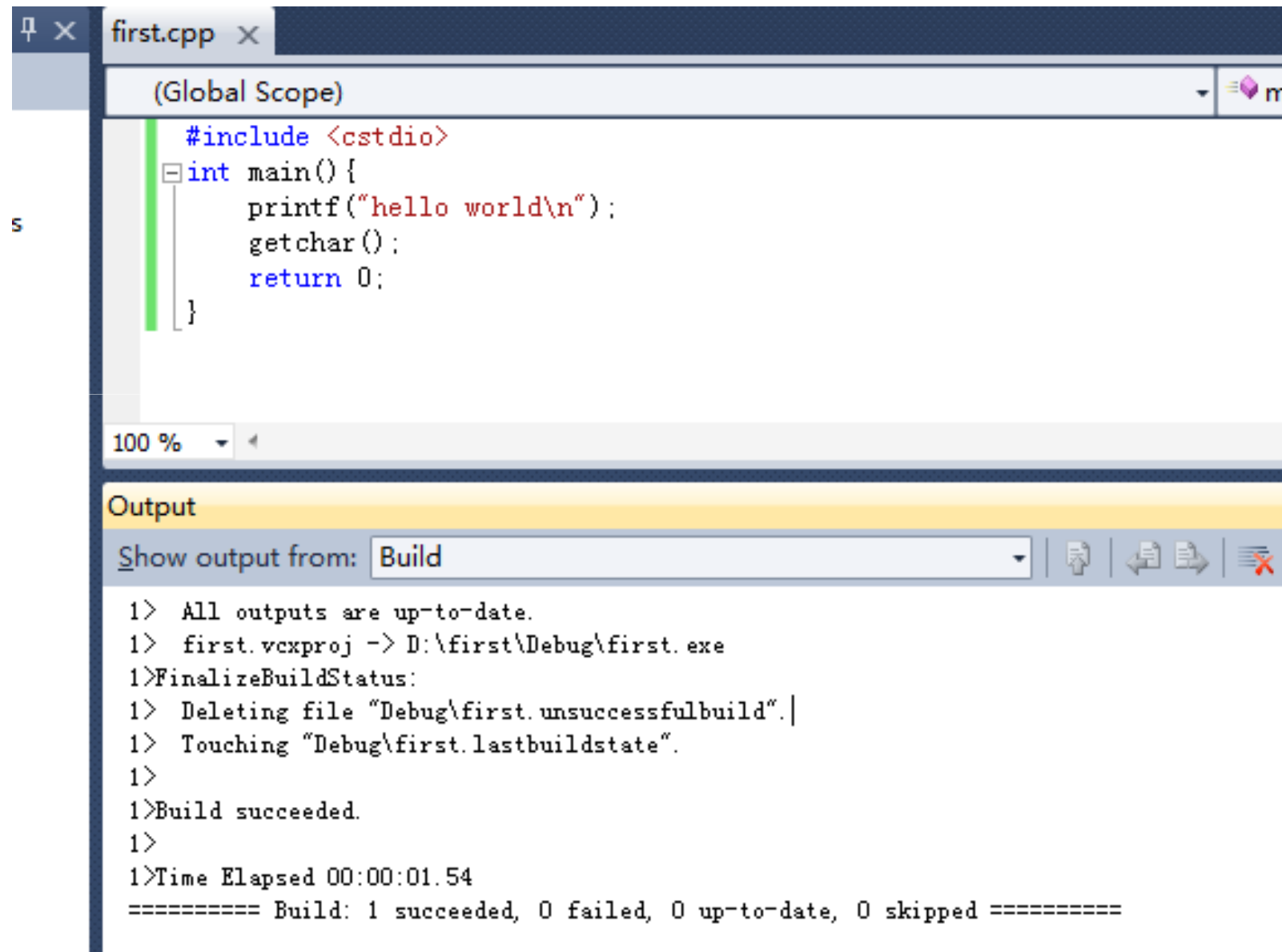
100 %

Output

Show output from: Build

```
1>InitializeBuildStatus:  
1> Creating "Debug\first.unsuccessfulbuild" because "AlwaysCreate" was specified.  
1>ClCompile:  
1> first.cpp  
1>d:\first\first\first.cpp(4): error C3861: 'getch': identifier not found  
1>  
1>Build FAILED.  
1>  
1>Time Elapsed 00:00:01.43  
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

VC2010



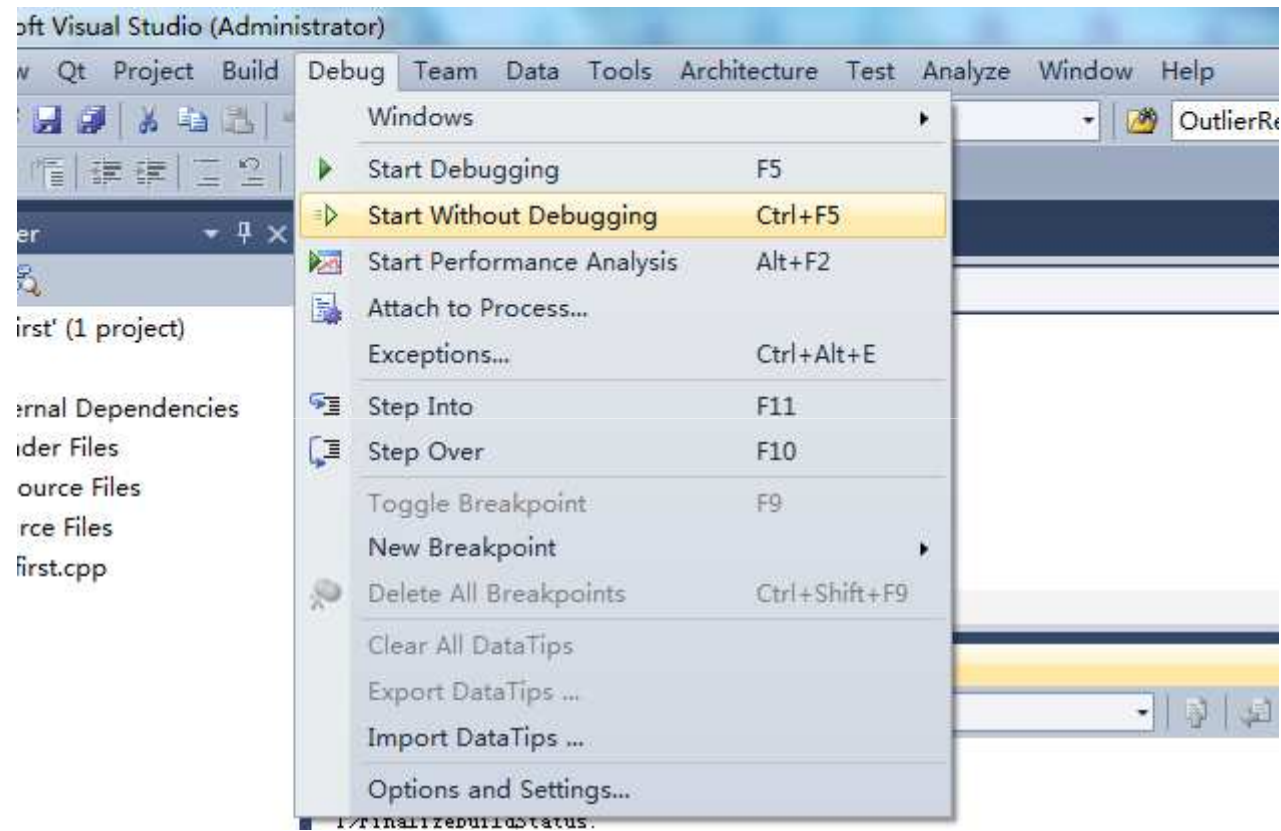
The screenshot displays the Microsoft Visual Studio 2010 interface. The top window, titled 'first.cpp', shows a C++ program with the following code:

```
#include <stdio>
int main() {
    printf("hello world\n");
    getchar();
    return 0;
}
```

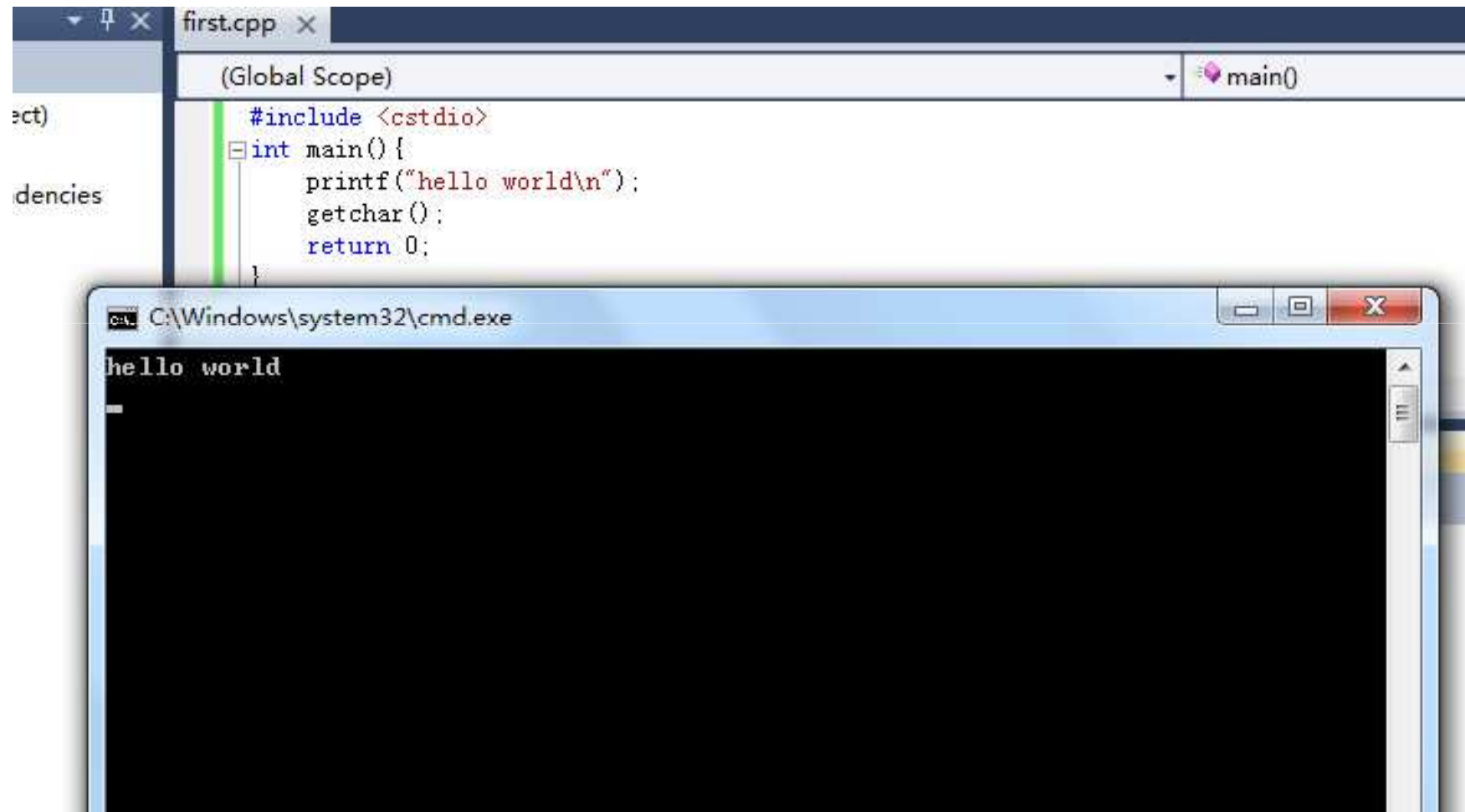
The code is displayed at 100% zoom. Below the code editor is the 'Output' window, which shows the build process. The 'Show output from:' dropdown is set to 'Build'. The output text is as follows:

```
1> All outputs are up-to-date.
1> first.vcxproj -> D:\first\Debug\first.exe
1>FinalizeBuildStatus:
1> Deleting file "Debug\first.unsuccessfulbuild".
1> Touching "Debug\first.lastbuildstate".
1>
1>Build succeeded.
1>
1>Time Elapsed 00:00:01.54
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

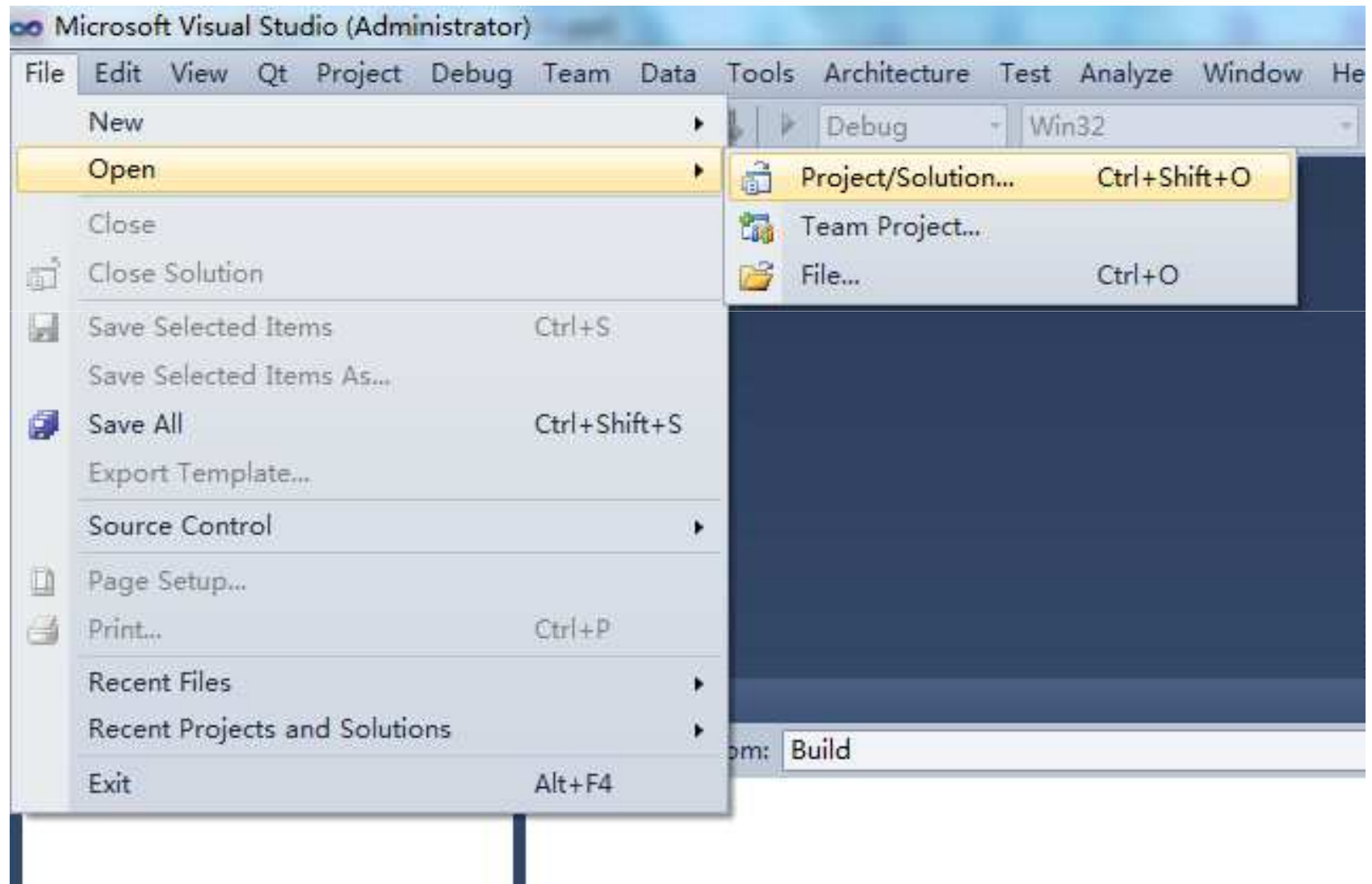
VC2010



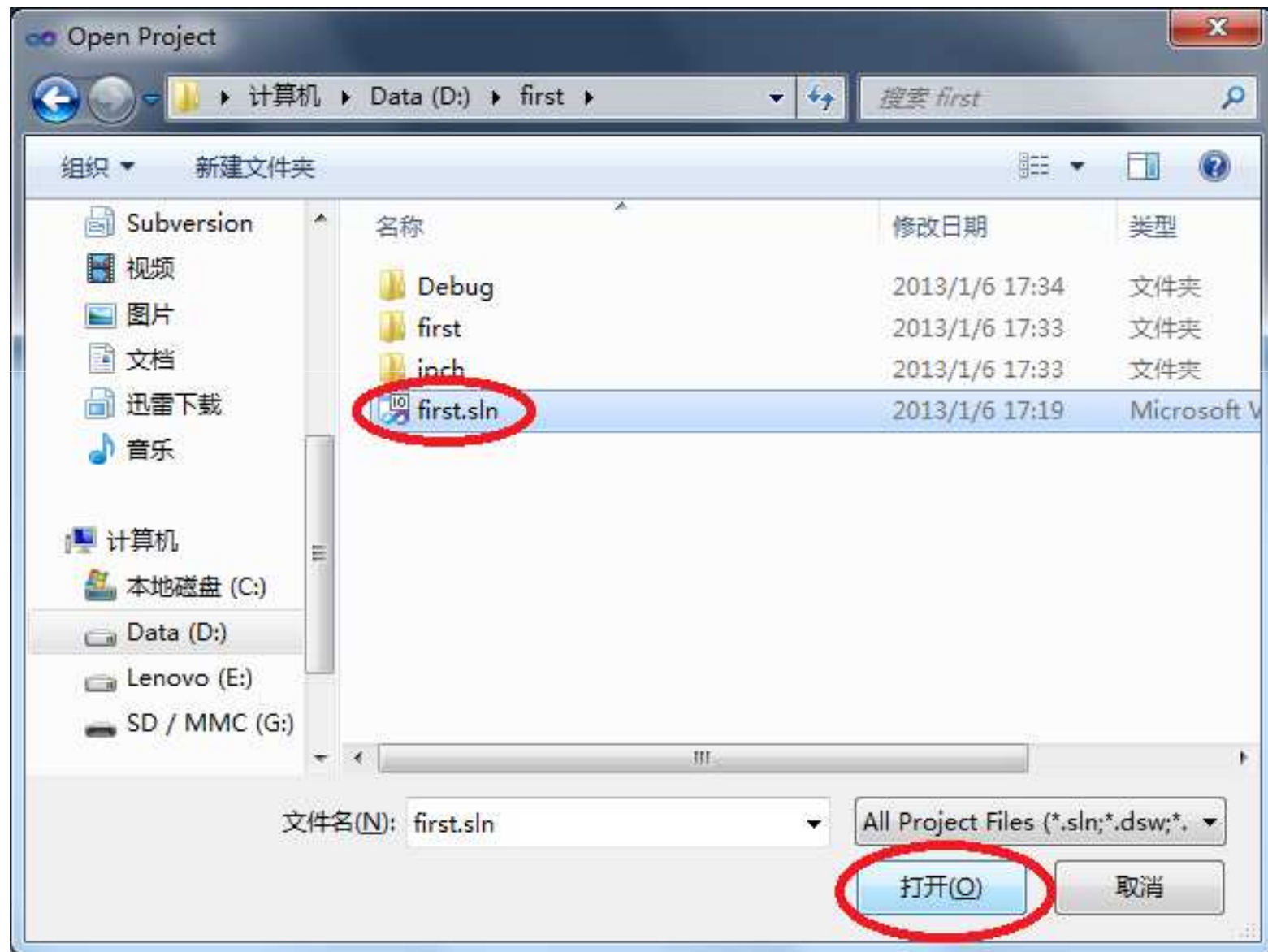
VC2010



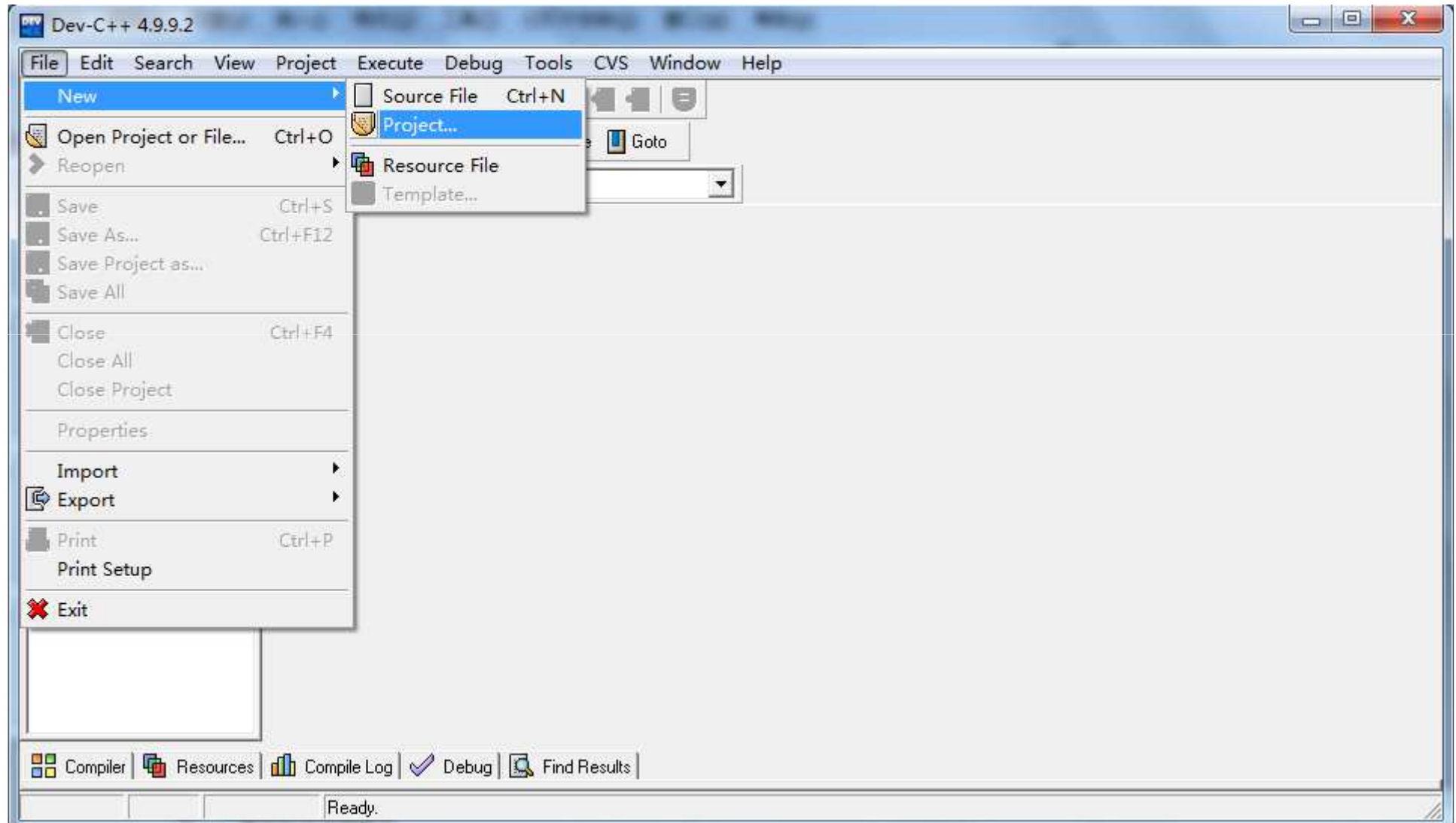
VC2010



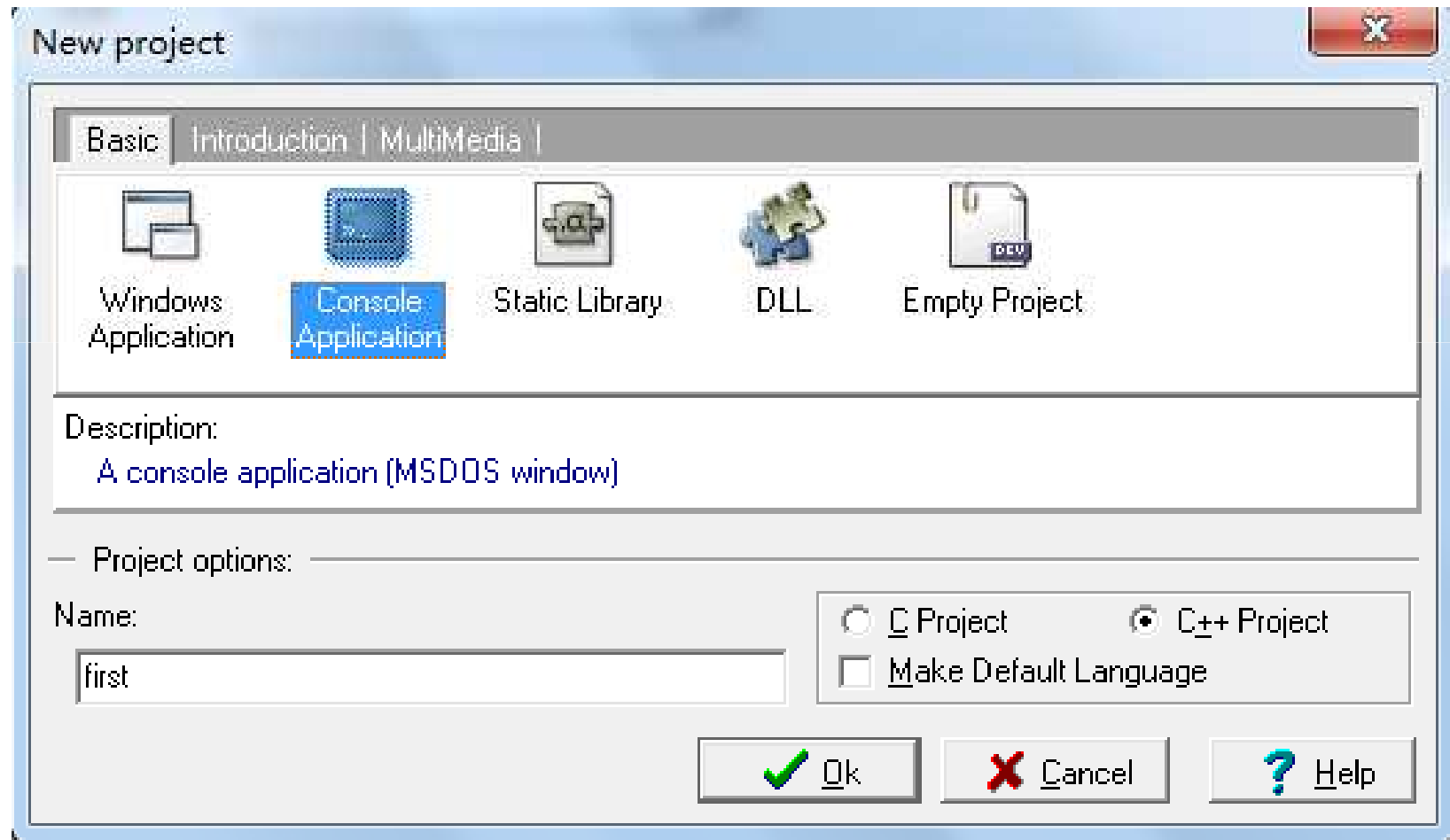
VC2010



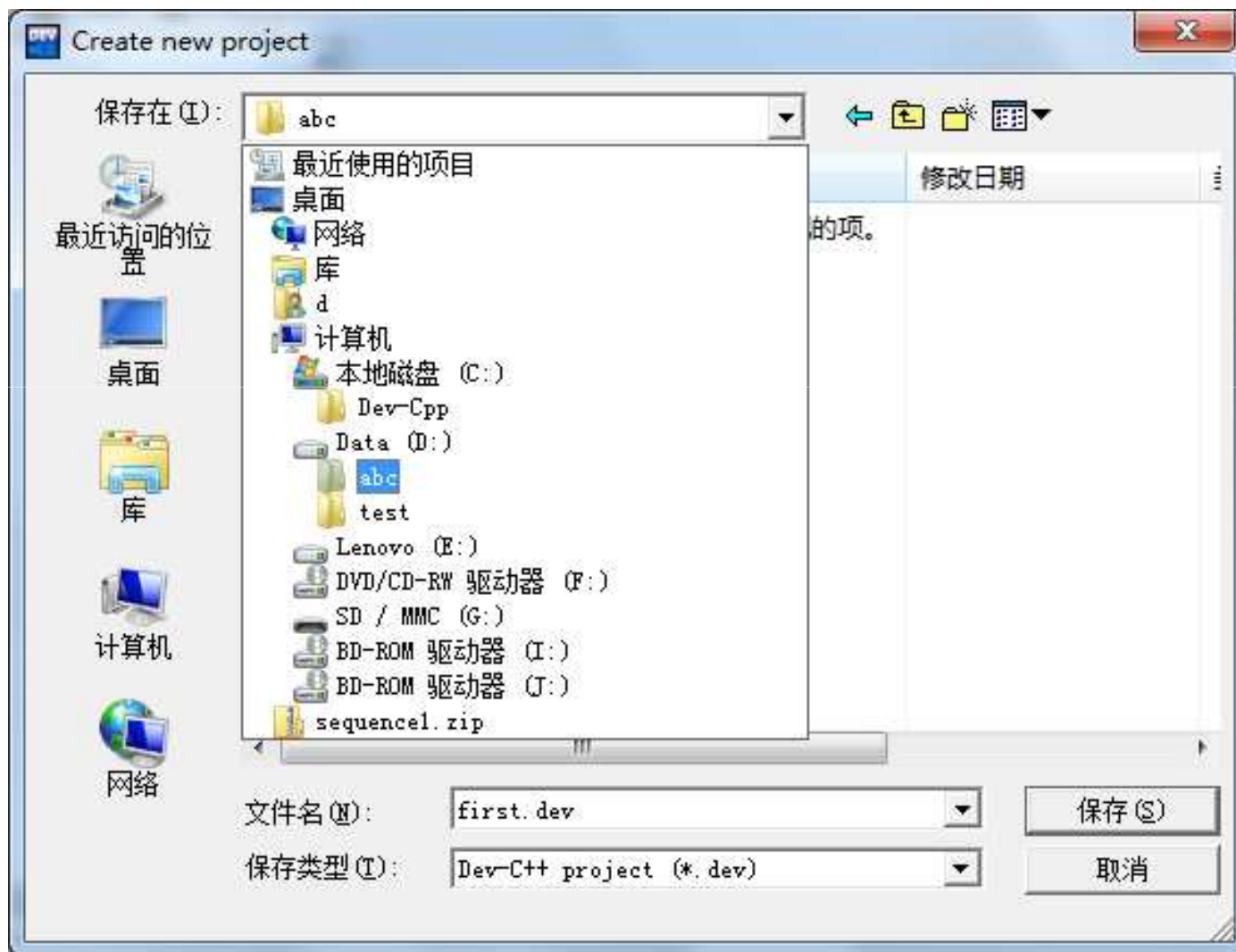
Dev C++



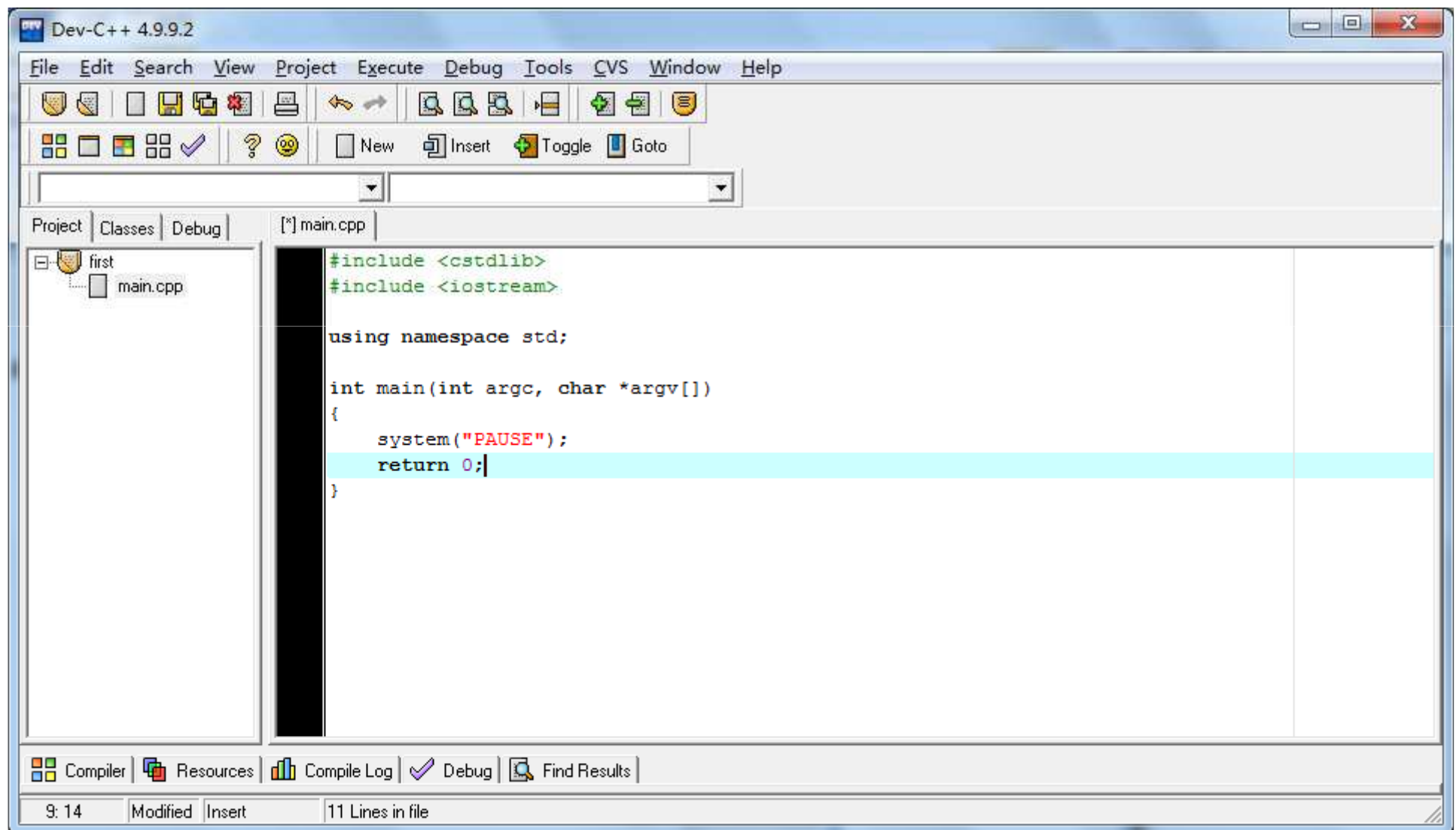
Dev C++



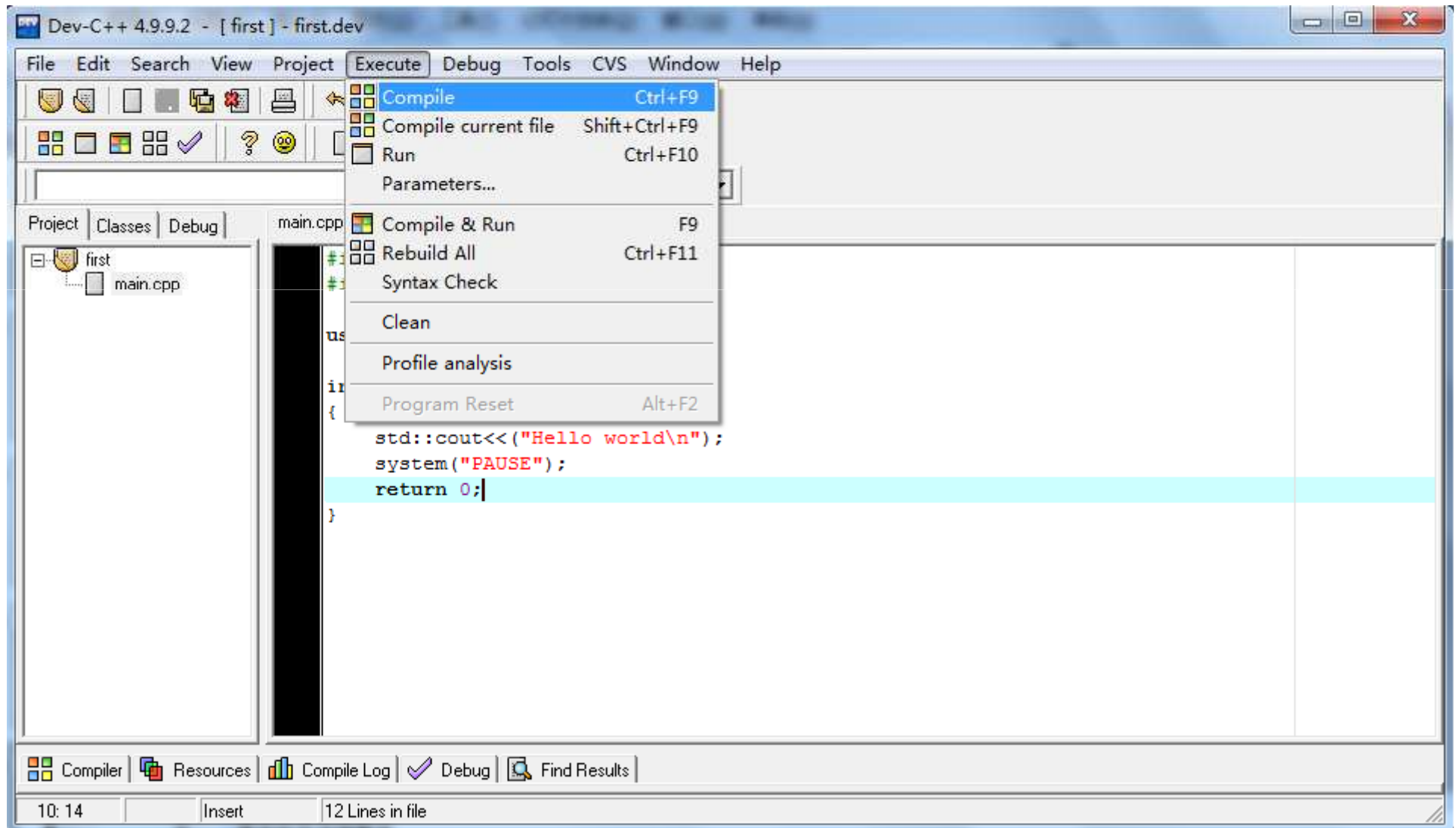
Dev C++



Dev C++



Dev C++



Dev C++

