# C/C++的秘密

**信息安全专业 江家伟 1150320117**

## C语言元素简介

### 数据类型（32位机） 存储空间（字节） 适用运算

长整形： 8 比较/算数/位/取地址

短整形 ： 2 比较/算数/位/取地址

整形 基本整形： 4 比较/算数/位/取地址

无符号： 2/4/8 比较/算数/位/取地址 有符号: 2/4/8 比较/算数/位/取地址 基本数据类型 单精度浮点: 8 比较/算数/取地址 浮点

双精度浮点: 16 比较/算数/取地址

字符类型(ASCII存储) 1 比较/算数/取地址

指针类型：存储一个内存地址 4 比较/解引用/算数/取地址

构造数据类型：结构体，数组，联合体，共用体，枚举

### 运算符

算数运算符: + - \* / ++ --

位运算符: >> << & | ^ ~

取地址运算符: &

逻辑运算符: && || ！

解引用运算符: \*

比较运算符: == != > < >= <=

### 宏

预处理：#define NAME value

宏定义方式 （简单用value替换NAME，value可以是代码，也可以是数据）

重定义类型名：typedef(不是简单替换，编译时处理)

### 函数（子程序）

定义: 返回值类型 函数名（参数1类型 参数1,……）{

函数体

……

}

调用：函数名（参数1，参数2,……）;

## 汇编语言元素简介

### 数据类型（保护模式） 定义方式

常量

整型 [{＋|－}] digits [radix]

实型 [sign]integer.[integer][exponent]

字符型 以单引号或双引号括起来的单个字符

字符串型 以单引号或双引号括起来的一串字符

占用内存（字节） 表示数据

BYTE 1 8位无符号整数

SBYTE 1 8位有符号整数

WORD 2 16位无符号整数

SWORD 2 16位有符号整数

DWORD 4 32位无符号整数

变量 SDWORD 4 32位有符号整数

FWORD 6 48位整数

QWORD 8 64位整数

TBYTE 10 80位整数

REAL4 4 32位短实数

REAL8 8 64位短实数

REAL10 10 80位扩展精度实数

### 保留字

指令助记符：如 MOV，ADD和MUL等

伪指令：用于告诉MASM如何编译程序

属性：为变量和操作数提供有关尺寸以及使用方式 的信息，例如BYTE和WORD

运算符：用在常量表达式中

预定义符号：例如@data，在编译时返回整数常量值

### 标识符 程序员选择的名字，用来标识变量、常量、过程或代码标号

### 指令

代码标号：用于程序的代码段(存放指令的地方)、 必须以冒号结尾、 可以和指令在同一行也可以独自成行

标号 数据标号：（变量名） 用于程序的数据段(定义变量 的地方)、不用冒号结尾、在同一源代码文件中必 须唯一。 标

MOV: 将一个值移动到另外一个中

ADD: 两个值相加

SUB: 从一个值中减去另一个值

指令助记符 MUL: 两个值相乘

JMP: 跳转到一个新位置

CALL: 调用一个过程

NOP: 不做任何操作,一般用于地址对齐

……

内存操作数：内存操作数由变量名或包含变量地址的 寄存器指定。 变量名表明了变量的地址

操作数 寄存器操作数：有些操作只对寄存器操作数有效

常量表达式

IO端口：用于读写IO设备

### 注释

单行注释：以分号(；)字符开始

块注释：以COMMENT伪指令以及一个用户定义的符号开始，直到一个相同的符号出现为止

注释 块注释示例：

COMMENT !

……

……!

### 过程定义

过程名PROC [USES Rigisters][param 1,param 2]

……

过程名 ENDP

### 宏

宏名MACRO[形参1,形参2,……]  
…;宏的定义体  
ENDM

## 汇编元素和C元素的比较

### 基本元素对应关系

#### 数据类型变量

C语言中的数据类型变量，在汇编的数据变量中都体现了它们各自的本质，例如：int型变量在汇编中就是DWORD类型，占4字节；char类型在汇编中就是BYTE型变量，占一字节……C语言中对变量的操作在汇编中的实现其实就是对各个内存单元的操作。

#### 宏

宏在汇编和C中的定义虽有不同，但是在用途上基本类似，都是用一个符号替换频繁使用的代码段，增加程序的抽象性和模块性。例如，汇编中的宏PROC就替代了每个过程调用中必须执行的两个语句push ebp和mov ebp ，esp；C语言中用宏常量可以使程序更具可读性，同时便于修改……

### 内存操作上的区别

C:基本变量类型大小（占用内存的空间）已经确定，在类型的转换上不方便（如想取出一个int型数据的低16位的高8位，就需要几步移位和位运算才能实现）

A：而汇编语言在数据类型上主要关心数据的大小，程序员在写汇编时能够切实感受到数据的本质—二进制。在汇编中，通过[类型+ptr]这样的强制转换，使类型的界限变得模糊，给予程序员最大的信任和操作二进制的自由。

C：C语言对内存访问限制较多，如在C语言中，除了指针能够随意读取可访问的内存之外，其余的数据类型都只能在本类型对应的存储空间内作修改。如一个int型的变量名只能访问本身所对应的4个字节空间。

A:在汇编中的DWORD类型变量中，即使后面的存储单元没有被初始化，仍然可以通过变量名+偏移直接访问。

## C++反汇编分析

### 函数定义

在VS中写了一个简单的C++函数

函数名是bingo，返回值是int型

功能简单，调用printf，打印一串字符后返回0

该代码反汇编后得到如下代码及简单分析：

int bingo() ;红色为C++源码 汇编代码分析

{

00A746C0 push ebp ……………………… 保存上一个函数的堆栈

00A746C1 mov ebp,esp ……………为本函数开辟空间

00A746C3 sub esp,0C0h

00A746C9 push ebx

00A746CA push esi ……………………寄存器压栈，保护现场

00A746CB push edi

00A746CC lea edi,[ebp-0C0h]

00A746D2 mov ecx,30h 初始化函数栈（开辟0xC0空间）

00A746D7 mov eax,0CCCCCCCCh

00A746DC rep stos dword ptr es:[edi]

printf("process!");

00A746DE mov esi,esp ………………保存栈顶指针值

00A746E0 push 0A7CC6Ch .......’process！’字串地址压栈,作为参数

00A746E5 call dword ptr ds:[0A80184h] …………调用printf

00A746EB add esp,4 ………………………回收”process!”占用的栈空间

00A746EE cmp esi,esp 检查栈平衡

00A746F0 call \_\_RTC\_CheckEsp (0A712EEh)

return 0;

00A746F5 xor eax,eax …………………….eax作为返回值，置为0

}

C++中的简单函数，在汇编语言中却涉及到了堆栈空间开辟、参数压栈、函数现场保护、堆栈平衡检测等知识。从这个简单的例子里我们可以看到C++的设计者们为我们封装了非常多的底层细节，了解这些细节，我们就能更好的理解C++以及其他高级语言的运行机理，从而更好的编程。

注意到我定义的这个bingo函数并没有局部变量，但是上面的反汇编代码中却有四行代码是用来在栈上初始化出0xC0字节大小的空间，并将这些空间填充上0xCC，这是为什么呢？

注意到上面的代码有一段是用来检测ESP平衡的。由于是dubug模式下的反汇编，可能是编译器自己加了一些调试变量在里面，导致多开了0xC0的栈空间，为了验证，切换到Release版本，在Release版本下的汇编代码中可以看到，已经没有多开0xC0的栈空间了：

; COMDAT ?bingo@@YAHHHH@Z

\_TEXT SEGMENT

\_a$ = 8 ; size = 4

\_b$ = 12 ; size = 4

\_c$ = 16 ; size = 4

?bingo@@YAHHHH@Z PROC ; bingo, COMDAT

; 8 : {

push ebp

mov ebp, esp

; 9 : b = a + c;

mov eax, DWORD PTR \_a$[ebp]

add eax, DWORD PTR \_c$[ebp]

mov DWORD PTR \_b$[ebp], eax

; 10 : return a;

mov eax, DWORD PTR \_a$[ebp]

; 11 : }

pop ebp

ret 0

?bingo@@YAHHHH@Z ENDP ; bingo

### 参数传递

我们逐渐复杂化用来分析的函数，这次我给他们加上了一些参数的传递。源码及对应反汇编分析如下：

源码 int main()

{

int va = 0,vb = 1,vc = 2;

va = bingo(partVal,vb,vc);

system("pause");

return 0;

}

int bingo(int a,int b,int c)

{

b = a + c;

return a;

}

反汇编分析

partVal = bingo(partVal,vb,vc);(参数入栈部分)

011853F7 mov eax,dword ptr [vc]

011853FA push eax

011853FB mov ecx,dword ptr [vb] 三个参数如栈,顺序为从右往左

011853FE push ecx (入栈顺序与调用约定有关)

011853FF mov edx,dword ptr [partVal]

01185402 push edx

01185403 call bingo (01181442h) ……………调用bingo函数

01185408 add esp,0Ch

0118540B mov dword ptr [partVal],eax

……….

参数传入bingo函数后,就进行一般的汇编代码,这里就不再浪费篇幅分析了,为了和下面的传引用方式进行对比,只分析bingo函数中的关键代码

b = a + c;

000F464E mov eax,dword ptr [a] 从a所在的内存单元中取出DW值放EAX

000F4651 add eax,dword ptr [c] 从c所在的内存单元中取出DW值加EAX

000F4654 mov dword ptr [b],eax 将EAX中的值放入b所在的地址中（栈地址）

return a;

000F4657 mov eax,dword ptr [a] 将a的值通过eax返回

由于是传值的方式传参,所以函数内部对a,b,c的修改不影响主函数变量va,vb,vc的值.

### 引用

接下来是改为传引用参数之后的源码及关键汇编代码

int bingo(int &a,int &b,int &c)

{

b = a + c;

return a;

}

b b = a + c; a + c;

00A8464E mov eax,dword ptr [a]

00A84651 mov ecx,dword ptr [eax]

00A84653 mov edx,dword ptr [c]

00A84656 add ecx,dword ptr [edx]

00A84658 mov eax,dword ptr [b]

00A8465B mov dword ptr [eax],ecx

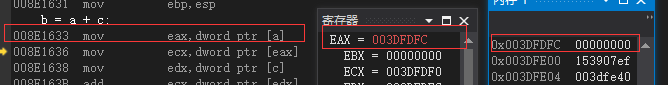
return a;

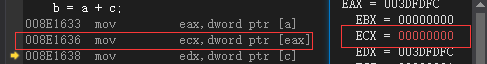
00A8465D mov eax,dword ptr [a]

00A84660 mov eax,dword ptr [eax]

汇编代码分析：

可以看到，汇编代码第一行mov eax,dword ptr [a]将a中的值（4字节）取到eax中，从内存中看，这时eax的值不再是参数va的值，而是va的地址；第二句汇编，再从eax指向的地址中取出4个字节的数，放到ecx中，这时可以观察到，ecx的值是va的值0。分析过程中用到的内存和寄存器截图如下:





从上面的分析，我们知道了，C++语言在传引用时，直接将变量的地址传入函数中，在函数中直接操作变量的地址中的值，从而直接修改了参数的值（对主函数产生影响）。到这里，不禁想看下，同样是向函数中传入参数地址的指针，在汇编中是怎么实现的，会不会和传引用一样呢？

### 指针

将程序稍作修改，向bingo函数中传入va,vb,vc的指针，部分源码及关键汇编代码如下：

源码

int bingo(int \*a,int \*b,int \*c)

{

\*b = \*a + \*c;

return \*b;

}

汇编：

\*b = \*a + \*c;

01381633 mov eax,dword ptr [a]

01381636 mov ecx,dword ptr [eax]

01381638 mov edx,dword ptr [c]

0138163B add ecx,dword ptr [edx]

0138163D mov eax,dword ptr [b]

01381640 mov dword ptr [eax],ecx

return \*b;

01381642 mov ecx,dword ptr [b]

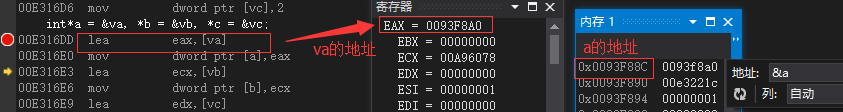
01381645 mov eax,dword ptr [ecx]

通过观察内存和寄存器的变化，可以看到指针和引用在用于传参时，并没有本质区别，它们都是直接将参数的地址传入函数中，函数中直接操作该地址中的值。

### 引用和指针的区别

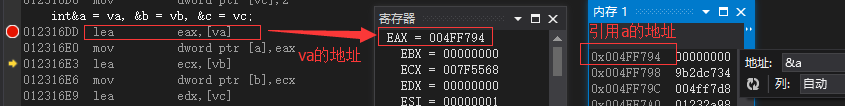
但是我们知道，指针和引用是有本质区别的，即引用只是变量的一个别名，而指针本身是一个变量，变量的值是被指向的变量的地址，它们的区别在汇编代码中也有体现：

下图是分别用va,vb,vc的地址初始化a,b,c指针时，内存和寄存器窗口观察到的值的截图：



由图可知，指针变量a的地址和被指针指向的整型变量的地址是不同的。

下图是分别用va,vb,vc的地址初始化a,b,c引用时，内存和寄存器窗口观察到的值的截图：



由图可知，指针变量a的地址和被指针指向的整型变量的地址是相同的。

### 类的创建、this指针、成员函数的调用

C++较C更为突出的特点就是对面向对象编程的支持更好，接下来就来看看C++的类在汇编语言中是怎么实现的。

我首先写了一个简单的类，为了简单起见这个类的构造函数和析构函数我都没写，这个类有一个公有方法和一个私有变量，公有方法Print（）的作用就是将私有变量打印到控制台，源代码如下：

源码（类的定义及main函数）：

class thisTest{ int main()

public: {

thisTest(){value = 100;v0 = 5;} thisTest classTest;

~thisTest(){} classTest . Print();

void Print() system("pause");

{ return 0;

printf("%d", this->value); }

}

private:

int v0;

int value = 50;

char v1 = “a”;

};

这里我们先分析一下实例化thisTest类为classTest时发生的事情：

thisTest classTest;

011816C8 lea ecx,[classTest] ……….将变量classTest的地址移到ecx中

011816CB call thisTest::thisTest (01181630h) ………调用thisTest的构造方法

011816D0 mov dword ptr [ebp-4],0………….清理栈

class thisTest{

public:

thisTest(){ value = 100; v0 = 5; }

01181630 push ebp

01181631 mov ebp,esp

01181633 push ecx ……………………保存ecx中的值（此处为thisTest变量的地址）

01181634 mov dword ptr [this],ecx ……………this是thisTest的引用（指向同一地址）

……

private:

int v0;

int value = 50;

01181637 mov eax,dword ptr [this]

0118163A mov dword ptr [eax+4],32h

char v1 = 'a'; 赋值操作，先给有初始化的成员赋初值

01181641 mov ecx,dword ptr [this]

01181644 mov byte ptr [ecx+8],61h

续上图

public:

thisTest(){ value = 100; v0 = 5; }

00083257 mov dword ptr [eax+4],64h

0008325E mov eax,dword ptr [this] 后调用构造方法，执行构造方法中的赋值操作

00083261 mov dword ptr [eax],5

00083267 mov eax,dword ptr [this]

0008326A pop edi

0008326B pop esi

0008326C pop ebx 还原调用前现场并返回

0008326D mov esp,ebp

0008326F pop ebp

00083270 ret

从上面的简单分析以及对内存的仔细观察，可以得到以下结论：

1. this的实质：当一个类被实例化时候，自动创建一个this局部变量，然后将这个变量赋值为这个实例的地址，实际上是这个实例的一个指针。
2. this指针的创建问题：既然是局部变量，那么在构造函数调用结束后，这个变量就消亡了，这样其他成员函数如何调用this指针呢？（实际应用中我们是可以在成员函数中调用this指针的）。于是跟进成员函数Print（），发现函数第一条汇编语句执行前，就重新对this指针赋值了（还是使它指向实例）：

classTest.Print();

003516D7 lea ecx,[classTest] ………….通过ECX寄存器传入classTest的值

003516DA call thisTest::Print (0351680h)

void Print()

{

00351680 push ebp

00351681 mov ebp,esp 重新对this指针赋值

00351683 push ecx

00351684 mov dword ptr [this],ecx

1. 类成员变量的初始化顺序：变量的初始化工作在构造函数中调用后完成，调用构造函数后，首先被初始化的是类中已经指明初始值的变量，然后调用构造方法，进行相应赋值。在调用了构造函数之后，实例的大小就确定了，其大小为类中包含的所有变量所占内存空间的总和（有的编译器为了寻址方便，每个成员变量的大小都是4的整数倍，VS2013的编译器就是这样），这个大小不会改变，但是每个成员变量的值可以被成员函数改变（如thisTest中的value既作为成员变量被赋值，又在构造方法中被赋值，后者覆盖了前者）。
2. 成员函数的执行：当成员函数被调用，且this指针被重新赋值之后，之后的函数执行就和一般函数没有什么区别了，只不过成员函数根据自己的权限不同能够访问的成员变量不同而已。
3. 成员变量的访问：在成员函数中，成员变量的访问方式可以是点访问，也可以用this指针访问，在成员函数的参数和类的成员变量同名时，为防止混淆，必须用this指针来访问成员变量。

### 重载

C++的函数之所以能够重载，是因为它细致的命名方式，也就是说，C++会根据成员函数的不同元素，如返回值、参数类型、参数个数等的不同，来区分各个成员函数，也就是说，程序员写的函数名不是区分两个函数的主要依据，而是以编译器综合各函数的元素考虑，得出的名字来区分的，下面用两个重载函数来更清楚地解释重载在底层的实现保障：

class thisTest{

public:

thisTest(){ value = 100; }

~thisTest(){};

void Print() ……………… ?Print@thisTest@@QAEXXZ

{

printf("%d", this->value);

}

void Print(int a) ………………. ?Print@thisTest@@QAEXH@Z

{

value = a;

printf("%d", this->value);

}

private:

int value = 50;

};

上面的代码中有两个重载的函数void Print()和void Print(int a)。这两个函数在编译器输出的.asm文件中的标识符是上图中各自虚线后的字串。可以看到，这两个函数虽然函数名相同，但参数个数不同，通过编译器编译后，产生了不同的标识符，从而达到重载的目的。

### 全局变量、局部变量

全局变量和局部变量的区别就在于，一旦程序被装载入内存，全局变量的地址就被确定，而局部变量的地址则是在函数栈中动态产生；他们的寿命也不同，局部变量随函数的消亡而消亡，全局变量则与程序共存亡，下面来看一个简单例子：

源代码：

int globalVal = 0;

int main()

{

int a = 10;

printf("%d %d", globalVal, a);

system("pause");

return 0;

}

关键汇编代码:

int a = 10;

00A31634 mov dword ptr [a],0Ah …...局部变量赋值，注意到这里a是个标号

printf("%d %d", globalVal, a);

00A3163B mov eax,dword ptr [a]

00A3163E push eax

00A3163F mov ecx,dword ptr ds:[0A34430h] ……全局变量赋值，这里则是个段:偏移地址

00A31645 push ecx

00A31646 push 0A3319Ch

00A3164B call dword ptr ds:[0A330B4h]

00A31651 add esp,0Ch

这里局部变量a的值之所以是个标号,是因为程序载入内存尚未运行时,还不能确定a的地址(栈底指针还没确定);而全局变量globalVal则是一个确切的数值ds:[0xA34430],程序载入内存后，ds寄存器中的值被确定，则globalVal在内存中的位置就确定了。

### Switch语句与if……else if的区别

当一个数的取值有多种情况，而每种情况将导致执行不同的代码块时，我们将会写switch语句或者if….else if语句。那么，这两种写法产生的汇编代码有什么不同呢？

首先来看看swtich语句的源码和汇编代码：

源码：

int main(){

int a = 10;

switch (a){

case 5:a = 100;break;

case 10:a = 1;break;

default:a = 0;

}

printf("%d",a);

system("pause");

return 0;

}

部分汇编代码

int main()

{

012B1630 push ebp

012B1631 mov ebp,esp

012B1633 sub esp,8

int a = 10;

012B1636 mov dword ptr [a],0Ah

switch (a)

012B163D mov eax,dword ptr [a]

012B1640 mov dword ptr [ebp-8],eax switch语句的汇编代码，实际上是

012B1643 cmp dword ptr [ebp-8],5 多个cmp、jmp、je指令的组合使用

012B1647 je main+21h (012B1651h) 当条件成立（相等）时，跳到相应的

012B1649 cmp dword ptr [ebp-8],0Ah 代码块执行

012B164D je main+2Ah (012B165Ah)

012B164F jmp main+33h (012B1663h)

{

case 5:

a = 100;

012B1651 mov dword ptr [a],64h ………..与某个switch值相应的指令

break;

012B1658 jmp main+3Ah (012B166Ah) …….break语句的效果，跳出switch

case 10:

a = 1;

012B165A mov dword ptr [a],1

break;

012B1661 jmp main+3Ah (012B166Ah)

default:

a = 0;

012B1663 mov dword ptr [a],0 …………….每个switch条件都不成立的情况

}

可以看到，switch语句的判断语句集中在一起，这么做的主要目的是为了实现C语言的语法需求，即在被执行的case语句执行后，若没有break语句，将顺序执行之后的代码。

再来看看if….else if 的汇编实现有什么区别**：**

源码：

int main()

{

int a = 10;

if (a == 5)a = 100;

else if (a == 10) a = 100;

else a = 0;

printf("%d",a);

system("pause");

return 0;

}

部分反汇编

int a = 10;

008A1634 mov dword ptr [a],0Ah

if (a == 5)a = 100;

008A163B cmp dword ptr [a],5 ……………判断a是否为5

008A163F jne main+1Ah (08A164Ah) ……不等则跳到下一个比较

008A1641 mov dword ptr [a],64h ………..相等则执行相应语句块

008A1648 jmp main+30h (08A1660h) ……语句块执行后跳转到主函数中继续执行

else if (a == 10) a = 100;

008A164A cmp dword ptr [a],0Ah

008A164E jne main+29h (08A1659h) 分析同上

008A1650 mov dword ptr [a],64h

008A1657 jmp main+30h (08A1660h)

else a = 0;

008A1659 mov dword ptr [a],0

可以看到，if….else if……else语句并没有像switch语句一样，将所有的判断放在一起，而是判断和相应语句交错，这样的结构与它的功能也是相匹配的。

从上面的比较来看，switch语句出了在功能上和if…else if结构有一些区别外，在效率上似乎没什么区别，但实际上，switch语句在case的判定值有明显的线性关系时，switch的优化特性便凸显出来。

### switch语句的优越性

来看下面这个例子：

源码：

int main()

{

int a = 10;

switch (a){

case 1:a = 1; break;

case 2:a = 2; break;

case 3:a = 3; break;

case 4:a = 4; break;

case 5:a = 5; break;

default:a = 100;

}

return 0;

}

部分反汇编及关键语句简要分析：

int a = 10;

011B1636 mov dword ptr [a],0Ah

switch (a){

011B163D mov eax,dword ptr [a]

011B1640 mov dword ptr [ebp-8],eax ……自动创建一个新的局部变量存储a的值

011B1643 mov ecx,dword ptr [ebp-8]

011B1646 sub ecx,1 将a的值减1，平衡下标

011B1649 mov dword ptr [ebp-8],ecx

011B164C cmp dword ptr [ebp-8],4 ….总共有5种情况，故下标最大为4

011B1650 ja $LN6+2Dh (011B1689h) …若平衡后的下标大于4，则跳到default

011B1652 mov edx,dword ptr [ebp-8] ….否则，查switch表

011B1655 jmp dword ptr [edx\*4+11B16B8h]….直接跳转到switch表相应下标存放的地址

当case的值具有强线性关系时，switch语句的形式有了很大的改变，我们不再能看到许多cmp、jmp指组合的身影，取而代之的是一句简单的判断和一句跳转，从比较次数的期望看，这样的方法，显然是快于if…else语句的。

那么，为什么可以用所谓的‘下标’来实现switch跳转呢？这是因为，switch语句的case值有强线性关系时，编译器会在汇编代码中构造一个‘跳转表’,本质上是一个函数指针数组。表的首地址中存放的是第一个case对应的语句块的首地址，之后的每一个下标都对应相应的case语句块的首地址。这样就可以实现上述跳转了。

在上面的例子程序中，首先将a的值减一，这么做是为了‘平衡下标’，因为第一个case值是1，减去第一个case的值，才是跳转表的下标索引，而这个索引，由于case的最大值是5，所以不会大于4，若a的值大于4，则可以直接判断出a的值不在case里，直接跳到default语句块；若不大于4，则根据得到的下标值，直接取出跳转表的地址进行跳转即可，效率着实提升了不少。

对case值没有强线性关系或不是顺序的switch语句，编译器都有办法构造相应的跳转表来进行查询和跳转。

### for循环

对于常用的for循环语句，观察其反汇编代码的实现：

源码：

int a = 1;

for (int i = 0; i < 10;i ++){

a += i;

}

关键代码反汇编及简要分析：

int a = 1;

01371636 mov dword ptr [a],1

for (int i = 0; i < 10;i ++){

0137163D mov dword ptr [ebp-4],0 ……i初始化为0

01371644 jmp main+1Fh (0137164Fh) ….跳转到比较，比较i和10的大小

01371646 mov eax,dword ptr [ebp-4]

01371649 add eax,1 i++的代码

0137164C mov dword ptr [ebp-4],eax

0137164F cmp dword ptr [ebp-4],0Ah ….比较语句

01371653 jge main+30h (01371660h) ….若i不小于10，则跳出循环

a += i;

01371655 mov ecx,dword ptr [a]

01371658 add ecx,dword ptr [ebp-4] for循环执行的语句块

0137165B mov dword ptr [a],ecx

}

0137165E jmp main+16h (01371646h)……跳转到i++的代码

上面的分析已经完整分析出for循环的汇编实现。

## 为什么要学习汇编

学完汇编语言这门课，回想汇编能做的事情，似乎C和C++大部分都能够实现，那么为什么我们要花大量时间来学一个功能拓展不多，但是学起来令人晕头转向的语言呢？我认为，这是因为作为一个立志成为高素质程序员的计算机系的学生，我们每一个人都应该知道，我们每天写的那些英文字符串：那些定义、声明、赋值、判断、调用、算法……究竟是怎么实现的，为啥我写了c = a+b，计算机就能把a和b的和赋值给c？ 了解这些不是没有意义的，正相反，如果你不了解这些系统底层实现方法，你基本上不可能写出快速、稳定的高性能程序，而在任何一个大工程里，对程序性能的高效性和稳定性的要求都是十分高的。

汇编语言就是我们了解系统底层的比较便捷的途径，通过阅读汇编代码，我们可以知道我们写的程序运行速度的瓶颈，从而修改代码，让程序跑得更快，更稳。

《深入理解计算机系统》（csapp）这本书的第五章中有一个例子，有力的证明了通过汇编语言理解计算机底层实现对写出更好程序的重要性：

int i ,j = 0;

char a[] = “hello word!”

for(i = 0;i < strlen(a);i++){

j += i;

}

上面的程序是一个但凡学过C语言都能写出来的程序，程序功能我就不多说了。乍看过去，这个程序似乎没有什么需要优化的地方，但是，学过会汇编，用汇编看过for循环的汇编代码实现的我们能够知道，在字符数组a的长度十分大，比如几千万字节时，这个程序的效率是非常低的。原因在于：在高级语言的机器实现中，每次for循环结束后，程序都要重新计算边界条件，在这里就是调用strlen这个函数计算字串a的长度。明白了这一点，上述函数的时间复杂度就不再是我们学汇编以前认为的O（n）的，而是O(n^2)的。这样一个程序在处理短长度的字符串的情况下固然可行，但是一旦放到其他地方使用，而且很不幸地用于处理超长的字串，那么情况就不乐观了。谁也不希望自己辛苦写的大项目不能很好运行的原因竟然是这样一个简单的循环，对吧？

上述代码可以优化成：

int i ,j = 0;

char a[] = “hello word!”

int len = strlen(a);

for(i = 0;i <len;i++){

j += i;

}

我们在进入循环前，将strlen(a)的值赋给一个变量，这样就可以避免重复计算的问题，这样的循环，时间复杂度就是O(n)的。

汇编语言除了帮助我们优化程序代码以外，还是信息安全专业学生的一门必修课。现在的网络世界中病毒横行，当你截获一个病毒样本时，你所能获取的对这个病毒的最真实的资料，除了观察病毒行为外，就是这个病毒程序的反汇编代码了。读懂了病毒的汇编代码，就能够很好的根除、防范这类病毒。