**深入理解C语言**

**马玉坤**

院 （系）：计算机科学与技术 专 业：计算机科学与技术

学　　号：1150310618 指导教师：史先俊

**2016年11月**



**课程作业（论文）**

题 目 深入理解C语言

专 业 计算机科学与技术

学　　 号 1150310618

学 生 马玉坤

指 导 教 师 史先俊

# 摘 要

汇编语言（英语：assembly language）是一种用于电子计算机、微处理器、微控制器，或其他可编程器件的低级语言。 在不同的设备中，汇编语言对应着不同的机器语言指令集。本文详细论述了C与汇编的不同方面的区别，并且描述了C的底层实现。包括以下方面：C与汇编关于数据类型、宏、函数、过程的区别，C的数据、结构、函数、参数、类的底层实现。

关键词：汇编语言, C++, C, 底层实现, 操作系统

**目 录**

[摘 要 - 0 -](#_Toc467620350)

[第1章 C与汇编的语言要素 - 3 -](#_Toc467620351)

[1.1 数据类型比较 - 3 -](#_Toc467620352)

[1.1.1 基本数据类型比较 - 3 -](#_Toc467620353)

[1.1.2 其他数据类型比较 - 3 -](#_Toc467620354)

[1.2 常量、变量与表达式 - 5 -](#_Toc467620355)

[1.2.1常量与常量表达式 - 5 -](#_Toc467620356)

[1.2.2 变量与变量表达式 - 6 -](#_Toc467620357)

[1.3 赋值、运算与流程控制 - 6 -](#_Toc467620358)

[1.3.1 赋值 - 6 -](#_Toc467620359)

[1.3.2 运算 - 7 -](#_Toc467620360)

[1.3.3 流程控制语句 - 8 -](#_Toc467620361)

[1.4 子程序 - 9 -](#_Toc467620362)

[1.5 宏 - 9 -](#_Toc467620363)

[1.6类 - 10 -](#_Toc467620364)

[1.7 与外部共享变量、子程序 - 10 -](#_Toc467620365)

[1.8头文件 - 10 -](#_Toc467620366)

[1.9 源程序模板 - 11 -](#_Toc467620367)

[1.10第一条语句标识 - 14 -](#_Toc467620368)

[第2章 C语言各要素的底层实现 - 15 -](#_Toc467620369)

[2.1 C语言的编译运行过程 - 15 -](#_Toc467620370)

[2.1.1 C语言的编译链接 - 15 -](#_Toc467620371)

[2.1.2 C语言程序的运行过程 - 15 -](#_Toc467620372)

[2.2 C语言类型的存储 - 15 -](#_Toc467620373)

[2.2.1 基本类型 - 15 -](#_Toc467620374)

[2.2.2 复杂类型 - 16 -](#_Toc467620375)

[2.3 C语言中的变量与指针 - 17 -](#_Toc467620376)

[2.4 C语言的运算与表达式 - 17 -](#_Toc467620377)

[2.4.1 C语言中的算术运算与算术表达式 - 17 -](#_Toc467620378)

[2.4.2 C语言中的条件表达式 - 19 -](#_Toc467620379)

[2.4.3 C语言中的逻辑运算与关系表达式 - 19 -](#_Toc467620380)

[2.5 C语言的程序结构 - 20 -](#_Toc467620381)

[2.5.1 顺序结构 - 20 -](#_Toc467620382)

[2.5.2 分支结构 - 20 -](#_Toc467620383)

[2.5.3 循环结构 - 22 -](#_Toc467620384)

[2.6 C语言中的函数 - 23 -](#_Toc467620385)

[2.7 C语言中的函数参数 - 25 -](#_Toc467620386)

[2.7.1 不可变参数 - 25 -](#_Toc467620387)

[2.7.2 可变参数函数原理 - 26 -](#_Toc467620388)

[2.8 C语言中的宏 - 28 -](#_Toc467620389)

[2.9 外部功能的调用 - 29 -](#_Toc467620390)

[2.9.1 C源程序函数的调用 - 29 -](#_Toc467620391)

[2.9.2 C库函数的调用 - 29 -](#_Toc467620392)

[2.9.3 ASM函数的调用 - 29 -](#_Toc467620393)

[2.9.4 OS函数的调用 - 29 -](#_Toc467620394)

[2.10 C++中类的本质 - 29 -](#_Toc467620395)

[2.11 new/delete与动态数组 - 33 -](#_Toc467620396)

[2.12 main函数的参数与返回值 - 34 -](#_Toc467620397)

[第3章 C语言与汇编语言的比较 - 35 -](#_Toc467620398)

[第4章 总结与分析，顿悟与畅想 - 36 -](#_Toc467620399)

[第5章 其他收获与希望 - 37 -](#_Toc467620400)

[参考文献 - 38 -](#_Toc467620401)

[致 谢 - 39 -](#_Toc467620402)

# 第1章 C与汇编的语言要素

## 1.1 数据类型比较

### 1.1.1 基本数据类型比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C语言 | 汇编语言 | 表示范围 | 字节 | 有无符号 |
| char | BYTE | -128~127 | 1 | 有 |
| unsigned char | 0~255 | 1 | 无 |
| short | WORD | -32768~32767 | 2 | 有 |
| unsigned short | 0~65535 | 2 | 无 |
| int | DWORD | -2147483648~2147483647 | 4 | 有 |
| unsigned int | 0~4294967295 | 4 | 无 |
| long long | QWORD | -9223372036854775808~9223372036854775807 | 8 | 有 |
| unsigned long long | 0~ 18446744073709551615 | 8 | 无 |
| float | REAL4 | -3.40E-38~3.40E38 | 4 | 有 |
| double | REAL8 | -1.7E-308~1.7E308 | 8 | 有 |
| 无 | REAL10 | 3.37E-4932~1.18E4932 | 10 | 有 |
| long double | 未指定 | 未指定 | 未指定 | 有 |
| bool | BYTE | C:{True,False},  ASM:0~255 | 1 | 无 |
| [type]\*(指针) | PTR [type] | 0~4294967295(x86)  0~18446744073709551615 | 4(x86),8(x64) | 无 |

表 1 数据类型比较表

### 1.1.2 其他数据类型比较

**(1) 多维数组：**

在汇编语言中，多维数组的表示实际上是将其展开成一维数组来实现的。

例如以下代码

#include<stdio.h>

int arr[10][10];

int main() {

int i = 3, j = 5;

arr[i][j] = 1;

return 0;

}

反汇编后可得到：

arr[i][j] = 1;

imul eax,dword ptr [i],28h

mov ecx,dword ptr [j]

mov dword ptr arr (0358138h)[eax+ecx\*4],1

在对a[i][j]元素取值时，编译器先将i与数组第二维的大小(0x28)相乘，得到a数组第i行的首地址，然后a[i]+j，得到a[i][j]的地址。

**(2)结构**

将以下代码编译运行

#include<stdio.h>

struct S {

int a, b;

long long c;

}s1, s2;

int main() {

s1.a = 1;

s1.b = 2;

s1.c = 3;

s2.a = 4;

s2.b = 5;

s2.c = 6;

return 0;

}

在return 0;语句前设断点。这时访问内存中s1的地址，可以得到：

s1.a = 1;

mov dword ptr [ebp-14h],1

s1.b = 2;

mov dword ptr [ebp-10h],2

s1.c = 3;

mov dword ptr [ebp-0Ch],3

可以看出， s1内a,b,c三个元素的地址连续。在汇编语言中，一个结构体实际上可以理解为一个内存地址连续且内部元素类型不一定相同的数组。

**(3)联合**

将以下C语言代码编译运行

#include<stdio.h>

union S {

char s[9];

int x;

long long y;

};

int main() {

S s1;

s1.x = 10;

s1.y = 11;

return 0;

}

可以得到以下汇编语句：

s1.x = 10;

mov dword ptr [ebp-14h],0Ah

s1.y = 11;

mov dword ptr [ebp-14h],0Bh

mov dword ptr [ebp-10h],0

可以看出，在联合中，所有成员的地址相同，都是该联合体的地址，且联合体的长度即能容纳最大的成员，也满足内存对齐。

**(4)枚举**

将以下C语言代码编译运行

#include<stdio.h>

enum S {

S1, S2, S3

};

int main() {

S s1=S1;

s1 = S2;

return 0;

}

获得以下反汇编代码：

S s1=S1;

mov dword ptr [ebp-8],0

s1 = S2;

mov dword ptr [ebp-8],1

可以看出，所谓的联合，本质上只是一个int变量，当这个int变量取不同值时，该联合体取对应不同的值。

**(5)类**

详见1.6

## 1.2 常量、变量与表达式

### 1.2.1常量与常量表达式

汇编语言中，主要有三种常量：立即数、符号常量与数据返回算符。

立即数储存在代码段，可以直接在代码段被访问。

符号常量为伪指令（如=）定义过后的字符。

数值返回算符：例如OFFSET产生的数字，会在编译过程中被替换成立即数。

C语言中，常量的类型与汇编相似。两者的常量可以互相转化。

无论是C语言还是汇编语言，如果一个表达式中，所有参与运算的元素均为常量，那么我们称这个表达式为常量表达式。常量表达式会由编译器在编译阶段提前算好，以立即数的形式储存在代码段中，供程序使用。为了提高程序运行的效率，这些表达式都不需要在运行时重新计算。

### 1.2.2 变量与变量表达式

|  |  |  |  |
| --- | --- | --- | --- |
|  | | C语言 | 汇编语言 |
| 全局变量 | 定义 | 在函数外 | 数据段(.data)或寄存器 |
| 使用 | 变量名 | 变量名或寄存器名 |
| 局部变量 | 定义 | 代码块中 | 将esp减去变量大小或LOCAL语句 |
| 使用 | 变量名 | 使用ebp加减指针或变量名 |

表 2 C语言与汇编语言对于变量的比较

由上表，C语言与汇编关于变量的部分大同小异，但汇编语言使用变量尤其是局部变量时，比起C语言来说更为繁琐，需要指针加减，移到寄存器等一系列操作。

在使用变量表达式时，汇编语言极为繁琐。对于普通的二元或一元运算（不涉及逻辑运算），对于每个运算符，我们至少需要写一条语句来执行。例如(x+y)\*z语句。如果使用C语言，那么我们只需要按照数学的方法表示即可，然而如果使用汇编语言，我们需要取得x变量地址，然后放入寄存器，再使用add指令，将寄存器与内存中的数相加，结果储存到寄存器中。然后取得z的地址，用mul指令，将寄存器与内存中的数相乘，结果储存到寄存器中。

对于条件表达式与关系表达式来说，汇编语言就显得更为繁琐。一个简单的用C语言表示的条件表达式，例如z?x>y。如果使用汇编语言写，就要使用条件分支。先要test z，然后用je或jle语句跳转到不同的分支来实现条件表达式。对于关系表达式尤其是复杂的关系表达式，如果使用汇编来写，将十分冗长，会有许多判断与跳转语句，非常容易出现编写错误的情况。

## 1.3 赋值、运算与流程控制

### 1.3.1 赋值

汇编语言定义全局变量时的初始化语句为如下格式：

变量名     类型   初始值1, 初始值2, …

变量名     类型   重复数量  dup  (初始值1, 初始值2, …)

而C语言定义全局变量时的初始化语句如下：

类型 变量名 = 初始值;

类型 变量名[数组大小] = {初始值1，初始值2}; //未涉及化到的元素填充为0

汇编语言无法在定义局部变量时同时初始化这些局部变量，只能用其他方式赋初值。而使用C语言可以在定义局部变量时同时初始化局部变量，例如:int x = 1;

C语言对全局变量以及局部变量赋值时，只需要使用“变量名=值”这样的语句即可。而汇编语言对变量赋值时，需要使用“mov 变量名, 寄存器或立即数”来进行。

### 1.3.2 运算

**(1) 整数与浮点数**

对于加减乘除等算术运算，使用C语言时，整数与浮点数不会在写法上有太大的区别。而使用汇编语言时，由于CPU结构的原因，使用整数进行算术运算与浮点数算术运算有一些差异。

|  |  |  |  |
| --- | --- | --- | --- |
|  | | C语言 | 汇编语言 |
| 加法 | 整数 | + | add |
| 浮点数 | addss |
| 减法 | 整数 | - | sub |
| 浮点数 | subs |
| 乘法 | 整数 | \* | mul |
| 浮点数 | mulss |
| 除法 | 整数 | / | div |
| 浮点数 | divss |
| 逻辑运算 | 整数 | &、|、^、~ | and,or,xor,neg |
| 浮点数 | 无 | 无 |

表 3 C语言与汇编关于运算的差异

**(2) 有符号与无符号**

与整数与浮点数的差异类似，C语言对于有符号与无符号的运算符相同，但程序运行时，会根据变量的类型，使用不同的语句。而汇编语言中，数据没有明确的有无符号之分，只能靠程序员明晰每个变量的类型，从而使用恰当的运算指令。

例如以下C语言程序：

#include<stdio.h>

int main() {

int x=-10;

unsigned int y = -10;

x = x >> 1;

y = y >> 1;

return 0;

}

编译运行后反汇编，可以得到如下语句：

x = x >> 1;

mov eax,dword ptr [x]

sar eax,1

mov dword ptr [x],eax

y = y >> 1;

mov eax,dword ptr [y]

shr eax,1

mov dword ptr [y],eax

显然，虽然x与y在内存中无法区分，但是由于两者在C语言中定义时的类型不一样，所以进行右移操作时，会调用不同的指令，一个是sar，一个是shr。而汇编语言没有有无符号的明确区分。

### 1.3.3 流程控制语句

**(1) 分支转移**

对于分支转移，C语言中可以方便地使用if、else、switch等语句，如果使用汇编语言，则需要用如下指令进行条件判断，然后跳往不同的地址，继续执行代码。

JE ;等于则跳转

JNE ;不等于则跳转

JZ ;为 0 则跳转

JNZ ;不为 0 则跳转

JS ;为负则跳转

JNS ;不为负则跳转

JC ;进位则跳转

JNC ;不进位则跳转

JO ;溢出则跳转

JNO ;不溢出则跳转

JA ;无符号大于则跳转

JNA ;无符号不大于则跳转

JAE ;无符号大于等于则跳转

JNAE ;无符号不大于等于则跳转

JG ;有符号大于则跳转

JNG ;有符号不大于则跳转

JGE ;有符号大于等于则跳转

JNGE ;有符号不大于等于则跳转

JB ;无符号小于则跳转

JNB ;无符号不小于则跳转

JBE ;无符号小于等于则跳转

JNBE ;无符号不小于等于则跳转

JL ;有符号小于则跳转

JNL ;有符号不小于则跳转

JLE ;有符号小于等于则跳转

JNLE ;有符号不小于等于则跳转

JP ;奇偶位置位则跳转

JNP ;奇偶位清除则跳转

JPE ;奇偶位相等则跳转

JPO ;奇偶位不等则跳转

**(2) 循环语句**

在C语言中，如果想要实现循环结构，那么就有三种不同的循环可以使用，它们分别为:for、while、do-while。而在汇编语言中，循环结构的实现则选择十分有限，只能使用rep+标号或者使用jump类语句+标号的方式来进行循环。但实际上，C语言中的三种循环语句都十分容易转化为jump类语句+标号。

例如一个简单的for循环:

for (i = 1; i <= 5; i++);

使用汇编语言，可以写作：

011616AE mov dword ptr [ebp-8],1

011616B5 jmp main+30h (011616C0h)

011616B7 mov eax,dword ptr [ebp-8]

011616BA add eax,1

011616BD mov dword ptr [ebp-8],eax

011616C0 cmp dword ptr [ebp-8],5

011616C4 jg main+38h (011616C8h)

可以看出，该for循环“翻译”成的汇编语句仍然有一定的可读性，也比较容易实现。

## 1.4 子程序

|  |  |  |
| --- | --- | --- |
|  | C语言 | 汇编语言 |
| 函数声明 | 类型 函数名() {} | proc 函数名  endp 函数名 |
| 函数调用 | 函数名(参数列表); | call 函数名/invoke 函数名，参数列表 |
| 函数返回 | return ; | ret |
| 参数传递 | 函数名(参数列表); | push 参数1  push 参数2  …  call 函数名 |
| 返回值 | 函数类型 returnVal = 函数名(参数); | 一般储存在eax中，如果位数不足高位在ecx中。 |

表 4 C语言与汇编语言关于函数的差异

值得注意的是，无论C语言还是汇编语言，对于传值与传地址，两者都十分类似。而且传地址本质上也是传值，不过传的值是一个地址而已。

## 1.5 宏

|  |  |  |
| --- | --- | --- |
|  | C语言 | 汇编语言 |
| 声明 | #define语句，例如  #define max(x, y) \  (x>y?x:y) | macro，例如:  max macro x, y  push eax  push ebx  mov eax, x  mov ebx, y  cmp eax, ebx  jg RXL  exitm y  RXL:  exitm x  end macro |
| 调用及参数传递 | max(x, y) | max(x, y) |

## 1.6类

类是C语言中特有的数据类型，其结构十分复杂，类的底层实现详见2.10。

## 1.7 与外部共享变量、子程序

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 汇编语言 | C语言 |
| 变量 | 对外提供 | public C变量名 | extern “C” {  变量类型 变量1；  变量类型 变量2;  …  } |
| 使用外部变量 | extern C 变量名:类型 | extern “C” 变量类型 变量名； |
| 外部子程序 | 对外提供子程序 | public 子程序名 | extern “C” {  函数类型 函数名(参数列表) ; |
| 使用外部子程序 | 只需用 PROTO C语句声明即可 | extern “C” 函数类型 函数名(参数); |

表 5 C语言汇编语言关于共享变量、子程序的区别

值得注意的是，在C语言中，使用外部变量与使用外部子程序写法类似，对外提供变量与对外提供子程序也十分类似。而汇编语言中，对外提供变量与对外提供子程序类似，但使用外部变量必须extern声明，而使用外部子程序则不需要。

## 1.8头文件

|  |  |  |
| --- | --- | --- |
|  | C语言 | 汇编语言 |
| 拓展名 | .h | .inc |
| 内容 | 对过程（子程序等）、变量、宏等的声明，以及其他头文件的引用 | 对过程（子程序等）、变量、宏等的声明，以及其他头文件的引用 |
| 引用格式 | #include “头文件.h” | INCLUDE 头文件.inc |

表 6 C语言与汇编语言关于头文件的区别

## 1.9 源程序模板

以下为一份比较典型的C语言程序模板：

#include <stdio.h>

#include <conio.h>

#include <string.h>

// 外部变量与过程声明

extern "C" int n;

extern "C" int qi(char \*buf, int \*pi, char chr);

extern "C" int qo(char \*buf, int \*po, char \*chr);

extern "C" int qp(char \*buf, int pi, int po);

extern "C" {

/\*

声明为外部提供的变量与过程

\*/

void incp(int \*p);

int QUEUE\_SIZE = 16;

}

void incp(int \*p) {

(\*p)++;

if (\*p == QUEUE\_SIZE)

\*p = 0;

}

char buf[16];

int po, pi;

int main() {

printf("请选择：ESC 退出；\n");

printf(" - 从队列提取元素显示\n");

printf(" + 打印当前队列\n");

printf(" [0-9A-Z] 打印当前队列\n");

printf("其他抛弃\n");

while (true) {

char chr = \_getche();

if (chr == '-') {

char chr\_out;

int status = qo(buf, &po, &chr\_out);

if (status == 0) {

printf("\n队列已空\n");

} else {

printf("提取的元素为:%c\n", chr\_out);

}

} else if (chr == '+') {

int status = qp(buf, pi, po);

} else if (('0' <= chr && chr <= '9') || ('A' <= chr && chr <= 'Z')) {

int status = qi(buf, &pi, chr);

if (status == 0) {

printf("\n队列已满\n");

}

} else if (chr == 27) {

break;

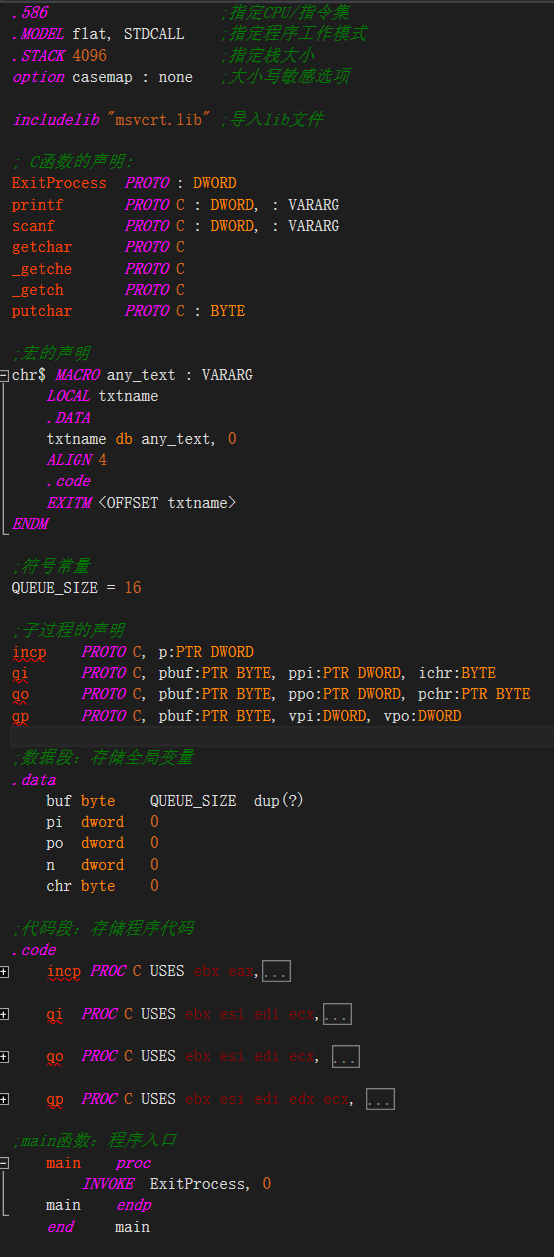
}

}

return 0;

}

以下为一份典型的汇编语言程序代码：



可以看到，C程序与汇编程序的代码风格相差甚大，而且C语言比汇编语言更有层次性和结构性。汇编语言的全局变量与函数声明的位置与C语言比起来，更为严格。而且汇编语言只有行内注释，而C语言除行内注释外还有段注释。

## 1.10第一条语句标识

C语言中，主函数默认为名为“main”的函数，所以如果没有main函数，程序将无法通过编译。而在汇编语言中，程序的入口可以任意指定，只需在程序末尾添加一行

*end* 入口函数名

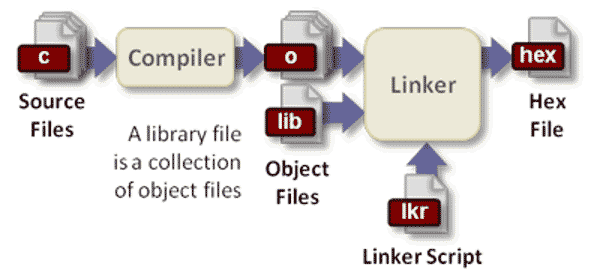
即可。

# 第2章 C语言各要素的底层实现

## 2.1 C语言的编译运行过程

### 2.1.1 C语言的编译链接

C原因的编译链接过程如下：



在通常的编译过程中，C语言源代码首先会被编译器(Compiler)编译成.o与.lib文件，编译成.o与.lib文件需要经历预处理#define, #include等预处理语句、编译成汇编语言文件两个过程。之后经过连接器(Linker)把外部函数的代码（通常是后缀名为.lib和.a的文件），添加到可执行文件中。

### 2.1.2 C语言程序的运行过程

以静态链接为例，操作系统会最先读取可执行的文件头，因为里面有运行程序的信息，如段表位置，程序入口，程序类型等。对于操作系统最重要的是段表与程序入口。其中段表就是elf中有多少段，每个段在文件中的偏移，入口则是常说得main函数的虚拟地址。

由于操作系统使用的是虚拟地址，因此对于程序来说，其每条语句的段地址始终不变，而虚拟地址到内存真实地址的映射则是由操作系统来完成。

另外，在进入main函数之前，操作系统还会同时初始化程序的栈空间与堆空间，载入main函数的参数，即char argc与char\* argv。

## 2.2 C语言类型的存储

### 2.2.1 基本类型

将以下代码编译运行

int main(int argc, char const \*argv[]) {

int intVar = 012;

unsigned int uintVar = 0x12;

long long llVar = 12;

unsigned long long ullVar = 12;

char chrVar = 12;

unsigned char uchrVar = 12;

int \*ip = &intVar;

float floatVar = 1.2;

double doubleVar = 1.2;

return 0;

}

反汇编后可以得到如下汇编代码：

int intVar = 012;

mov dword ptr [ebp-0Ch],0Ah

unsigned int uintVar = 0x12;

mov dword ptr [ebp-18h],12h

long long llVar = 12;

mov dword ptr [ebp-28h],0Ch

mov dword ptr [ebp-24h],0

unsigned long long ullVar = 12;

mov dword ptr [ebp-38h],0Ch

mov dword ptr [ebp-34h],0

char chrVar = 12;

mov byte ptr [ebp-41h],0Ch

unsigned char uchrVar = 12;

mov byte ptr [ebp-4Dh],0Ch

int \*ip = &intVar;

lea eax,[ebp-0Ch]

mov dword ptr [ebp-5Ch],eax

float floatVar = 1.2;

movss xmm0,dword ptr ds:[00F36B44h]

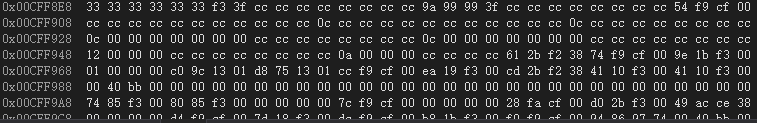
movss dword ptr [ebp-68h],xmm0

double doubleVar = 1.2;

movsd xmm0,mmword ptr ds:[00F36CB0h]

movsd mmword ptr [ebp-78h],xmm0

这时，堆栈中这些临时变量在内存中的表示如下：



可以看出，实际上，C语言中的各个基本类型都采用8-64位二进制整数存储，而且有符号整数与无符号整数使用补码存储，在内存中并无差别，只是在进行具体操作时，会使用不同的CPU指令（例如sar与shr，参见1.1.1）。

### 2.2.2 复杂类型

对于比较复杂的类型，例如结构体、枚举与联合，1.1.2做了详细的说明。由1.1.2的测试结果，可以得知：所谓的结构体在内存中只是一段连续的数据，当访问结构体中的某一元素时，只需要使用该结构体首地址以及该元素的偏移量即可算出该元素的地址，从而对该元素进行一系列操作；而枚举本质上是一个int变量，其能根据不同的取值来对应不同的用户自定的值，这个映射是由程序来完成的；联合中的每个成员的首地址与该联合体的首地址相同，共用一段内存。

## 2.3 C语言中的变量与指针

将以下C语言代码编译运行：

int globalVar = 10;

int main(int argc, char const \*argv[]) {

int tempVar = 20;

static int staticVar = 30;

register int regVar = 40;

int \*pg = &globalVar;

pg = &staticVar;

\*pg = 50;

return 0;

}

编译运行得到如下汇编代码：

int tempVar = 20;

009242FE mov dword ptr [ebp-8],14h

static int staticVar = 30;

register int regVar = 40;

00924305 mov dword ptr [ebp-14h],28h

int \*pg = &globalVar;

0092430C mov dword ptr [ebp-20h],928034h

pg = &staticVar;

00924313 mov dword ptr [ebp-20h],928038h

\*pg = 50;

0092431A mov eax,dword ptr [ebp-20h]

0092431D mov dword ptr [eax],32h

全局变量globalVar的地址为0x00E78034，该地址所指向的内存在数据段内。局部变量tempVar的地址为0x00CFF868，在堆栈内。因此，全局变量在数据段中，局部变量在堆栈中。

静态变量staticVar的地址为0x00E78038，与globalVar地址相邻，都在数据段中。从而可以得出，静态变量本质与全局变量没有区别，都储存在数据段中，只是在编写C程序时，编译器会对全局变量与静态变量的作用域做检查。两者的作用域不同是C语言强加的一种语言特性，与二者的本质无关。

另外，指针变量本质是一个32位整数，其值为这个指针指向元素的地址，例如当把&globalVar赋值给\*pg时，其值变为0x0E78034，指向了globalVar这个元素，当使用”\*”操作符时，程序首先将pg的值取出到寄存器中，然后用寄存器间接取址得到globalVar这个元素进行操作。

## 2.4 C语言的运算与表达式

### 2.4.1 C语言中的算术运算与算术表达式

将以下C语言程序编译运行：

#include<stdio.h>

int main() {

int x = 0;

int y = 1, z = 2;

x = y + z;

x = y - z;

x = y \* z;

x = y / z;

x = y % z;

x = (y + z) \* y + y \* z;

return 0;

}

可以得到如下汇编代码：

int x = 0;

0035168E mov dword ptr [ebp-8],0

int y = 1, z = 2;

00351695 mov dword ptr [ebp-14h],1

0035169C mov dword ptr [ebp-20h],2

x = y + z;

003516A3 mov eax,dword ptr [ebp-14h]

003516A6 add eax,dword ptr [ebp-20h]

003516A9 mov dword ptr [ebp-8],eax

x = y - z;

003516AC mov eax,dword ptr [ebp-14h]

003516AF sub eax,dword ptr [ebp-20h]

003516B2 mov dword ptr [ebp-8],eax

x = y \* z;

003516B5 mov eax,dword ptr [ebp-14h]

003516B8 imul eax,dword ptr [ebp-20h]

003516BC mov dword ptr [ebp-8],eax

x = y / z;

003516BF mov eax,dword ptr [ebp-14h]

003516C2 cdq

003516C3 idiv eax,dword ptr [ebp-20h]

003516C6 mov dword ptr [ebp-8],eax

x = y % z;

003516C9 mov eax,dword ptr [ebp-14h]

003516CC cdq

003516CD idiv eax,dword ptr [ebp-20h]

003516D0 mov dword ptr [ebp-8],edx

x = (y + z) \* y + y \* z;

003516D3 mov eax,dword ptr [ebp-14h]

003516D6 add eax,dword ptr [ebp-20h]

003516D9 imul eax,dword ptr [ebp-14h]

003516DD mov ecx,dword ptr [ebp-14h]

003516E0 imul ecx,dword ptr [ebp-20h]

003516E4 add eax,ecx

003516E6 mov dword ptr [ebp-8],eax

参考上述代码反汇编的结果，尤其是x=(y+z)\*y+y\*z语句。可见，编译器在处理算术表达式时，会按照算符优先级，使用寄存器变量记录中间结果，使用add、sub、mul、idiv等语句进行加减乘除取模运算，来求得表达式的值。

### 2.4.2 C语言中的条件表达式

将以下C语言程序编译运行

#include<stdio.h>

int main() {

int x=0, y=1;

int z = x > y ? x : y;

return 0;

}

得到以下汇编语句：

int z = x > y ? x : y;

00C816BC mov eax,dword ptr [x]

00C816BF cmp eax,dword ptr [y]

00C816C2 jle main+3Fh (0C816CFh)

00C816C4 mov ecx,dword ptr [x]

00C816C7 mov dword ptr [ebp-0E8h],ecx

00C816CD jmp main+48h (0C816D8h)

00C816CF mov edx,dword ptr [y]

00C816D2 mov dword ptr [ebp-0E8h],edx

00C816D8 mov eax,dword ptr [ebp-0E8h]

00C816DE mov dword ptr [z],eax

可以看出，所谓的条件表达式实际上只是一个条件分支。当x,y满足不同关系时，程序会跳到不同的地址。

### 2.4.3 C语言中的逻辑运算与关系表达式

C语言中的逻辑运算符有两套，一套支持短路运算，例如&&、||等，一套不支持短路运算，例如&、|等。如果使用短路运算符&&，在计算a&&b时，如果a的值为false，那么b的值将不会被计算，从而提高程序运行的效率。而非短路运算符则相反。

1. **非短路运算**

将以下C语言程序进行编译运行：

z = (x > y&&x < y) || (x == y&&x != y);

011E169C mov eax,dword ptr [ebp-8]

011E169F cmp eax,dword ptr [ebp-14h]

011E16A2 jle 011E16AC

011E16A4 mov ecx,dword ptr [ebp-8]

011E16A7 cmp ecx,dword ptr [ebp-14h]

011E16AA jl 011E16C8

011E16AC mov edx,dword ptr [ebp-8]

011E16AF cmp edx,dword ptr [ebp-14h]

011E16B2 jne 011E16BC

011E16B4 mov eax,dword ptr [ebp-8]

011E16B7 cmp eax,dword ptr [ebp-14h]

011E16BA jne 011E16C8

011E16BC mov dword ptr [ebp+FFFFFF18h],0

011E16C6 jmp 011E16D2

011E16C8 mov dword ptr [ebp+FFFFFF18h],1

011E16D2 mov ecx,dword ptr [ebp+FFFFFF18h]

011E16D8 mov dword ptr [ebp-20h],ecx

经过分析，可以得出：条件表达式是采用cmp之后jmp的方式。非短路逻辑运算使用了若干次cmp，且将运算的结果储存到临时变量中，之后按顺序依次计算逻辑表达式的值，然后使用逻辑运算进行综合。

1. **短路运算**

将以下C语言代码编译运行

#include<stdio.h>

int main() {

int x=0, y=1;

int z;

z = (x > y&&x < y) || (x == y&&x != y);

return 0;

}

可以得到如下反汇编后的代码：

z = (x > y&&x < y) || (x == y&&x != y);

00E916BC mov eax,dword ptr [x]

00E916BF cmp eax,dword ptr [y]

00E916C2 jle main+3Ch (0E916CCh)

00E916C4 mov ecx,dword ptr [x]

00E916C7 cmp ecx,dword ptr [y]

00E916CA jl main+58h (0E916E8h)

00E916CC mov edx,dword ptr [x]

00E916CF cmp edx,dword ptr [y]

00E916D2 jne main+4Ch (0E916DCh)

00E916D4 mov eax,dword ptr [x]

00E916D7 cmp eax,dword ptr [y]

00E916DA jne main+58h (0E916E8h)

00E916DC mov dword ptr [ebp-0E8h],0

00E916E6 jmp main+62h (0E916F2h)

00E916E8 mov dword ptr [ebp-0E8h],1

00E916F2 mov ecx,dword ptr [ebp-0E8h]

00E916F8 mov dword ptr [z],ecx

我们可以得到一个大致的脉络：首先可以将一个复杂的关系表达式表示成若干层关系表达式，每一层之间由多个关系表达式用一种运算（“与”、“或”或“异或”）连接。多个表达式相“与”时，程序将会按顺序判断各个表达式，一旦遇到值为假的表达式，便会直接跳到上一层对应的值为“假”的位置。多个表达式相“或”时，一旦遇到一个表达式值为真，便跳到上一层对应的值为“真”的位置。

## 2.5 C语言的程序结构

### 2.5.1 顺序结构

C语言的顺序结构程序在编译成汇编语言时，会按执行顺序依次转换为机器码，在程序执行时按顺序执行语句。

### 2.5.2 分支结构

将以下C语言程序编译运行：

#include<stdio.h>

int main() {

int x = 1, y = 2;

if (x < y) {

x++;

} else if(x > y) {

y++;

} else {

x++;

}

switch (x) {

case 1:

case 3:

x++;

case 2:

y++;

default:

y++;

}

return 0;

}

反汇编后可以得到以下代码：

if (x < y) {

011D169C mov eax,dword ptr [ebp-8]

011D169F cmp eax,dword ptr [ebp-14h]

011D16A2 jge 011D16AF

x++;

011D16A4 mov eax,dword ptr [ebp-8]

011D16A7 add eax,1

011D16AA mov dword ptr [ebp-8],eax

011D16AD jmp 011D16CB

} else if(x > y) {

011D16AF mov eax,dword ptr [ebp-8]

011D16B2 cmp eax,dword ptr [ebp-14h]

011D16B5 jle 011D16C2

y++;

011D16B7 mov eax,dword ptr [ebp-14h]

011D16BA add eax,1

011D16BD mov dword ptr [ebp-14h],eax

} else {

011D16C0 jmp 011D16CB

x++;

011D16C2 mov eax,dword ptr [ebp-8]

011D16C5 add eax,1

011D16C8 mov dword ptr [ebp-8],eax

}

switch (x) {

011D16CB mov eax,dword ptr [ebp-8]

011D16CE mov dword ptr [ebp+FFFFFF24h],eax

011D16D4 cmp dword ptr [ebp+FFFFFF24h],1

011D16DB je 011D16F1

011D16DD cmp dword ptr [ebp+FFFFFF24h],2

011D16E4 je 011D16FA

011D16E6 cmp dword ptr [ebp+FFFFFF24h],3

011D16ED je 011D16F1

011D16EF jmp 011D1703

case 1:

case 3:

x++;

011D16F1 mov eax,dword ptr [ebp-8]

011D16F4 add eax,1

011D16F7 mov dword ptr [ebp-8],eax

case 2:

y++;

011D16FA mov eax,dword ptr [ebp-14h]

011D16FD add eax,1

011D1700 mov dword ptr [ebp-14h],eax

default:

y++;

011D1703 mov eax,dword ptr [ebp-14h]

011D1706 add eax,1

分析其中的if-else分支，我们可以看出，C语言的if-else分支采用的是cmp后使用适当的jump语句来跳转的。而switch-case分支也同样。

### 2.5.3 循环结构

编译运行下列C语言代码：

#include<stdio.h>

int main() {

int i;

for (i = 1; i <= 10; i++);

i = 1;

while (i <= 10)i++;

i = 1;

do {

i++;

}while(i <= 10);

return 0;

}

可以得到如下汇编程序代码：

for (i = 1; i <= 10; i++);

0013168E mov dword ptr [ebp-8],1

00131695 jmp 001316A0

00131697 mov eax,dword ptr [ebp-8]

0013169A add eax,1

0013169D mov dword ptr [ebp-8],eax

001316A0 cmp dword ptr [ebp-8],0Ah

001316A4 jg 001316A8

001316A6 jmp 00131697

i = 1;

001316A8 mov dword ptr [ebp-8],1

while (i <= 10)i++;

001316AF cmp dword ptr [ebp-8],0Ah

001316B3 jg 001316C0

001316B5 mov eax,dword ptr [ebp-8]

001316B8 add eax,1

001316BB mov dword ptr [ebp-8],eax

001316BE jmp 001316AF

i = 1;

001316C0 mov dword ptr [ebp-8],1

do {

i++;

001316C7 mov eax,dword ptr [ebp-8]

001316CA add eax,1

001316CD mov dword ptr [ebp-8],eax

}while(i <= 10);

001316D0 cmp dword ptr [ebp-8],0Ah

001316D4 jle 001316C7

可以发现，for循环分支使用的是jump类语句跳转的方式进行的。类似地，while循环也是为该结构，与for循环逻辑几乎一致。而do-while结果最为简单，因为其不需要进行第一次判断，只需执行循环体，然后做判断，之后决定跳转。

## 2.6 C语言中的函数

函数f用于计算1到10的和，C语言代码如下：

int f() {

int res = 0, i;

for (i = 1; i <= 10; i++)

res += i;

return res;

}

反汇编后得到的代码则是：

int f() {

000C1670 push ebp

000C1671 mov ebp,esp

000C1673 sub esp,0D8h

000C1679 push ebx

000C167A push esi

000C167B push edi

000C167C lea edi,[ebp+FFFFFF28h]

000C1682 mov ecx,36h

000C1687 mov eax,0CCCCCCCCh

000C168C rep stos dword ptr es:[edi]

int res = 0, i;

000C168E mov dword ptr [ebp-8],0

for (i = 1; i <= 10; i++)

000C1695 mov dword ptr [ebp-14h],1

000C169C jmp 000C16A7

000C169E mov eax,dword ptr [ebp-14h]

000C16A1 add eax,1

000C16A4 mov dword ptr [ebp-14h],eax

000C16A7 cmp dword ptr [ebp-14h],0Ah

000C16AB jg 000C16B8

res += i;

000C16AD mov eax,dword ptr [ebp-8]

000C16B0 add eax,dword ptr [ebp-14h]

000C16B3 mov dword ptr [ebp-8],eax

000C16B6 jmp 000C169E

return res;

000C16B8 mov eax,dword ptr [ebp-8]

}

主函数中的掉用语句如下：

f();

000C16EE call 000C133E

分析这份汇编代码的逻辑，我们可以得出：C语言中，函数的临时变量储存在栈中，并且在函数具体语句执行前，函数内会有一系列push语句用来保存现场。并在ret语句执行前将push到栈中的“现场”恢复。

但是eax和ecx的值并不会被保护，这两个寄存器需要主函数自行保护，而且eax是会作为函数返回值来使用。

在调用函数时，会使用call语句。函数返回时使用ret语句。

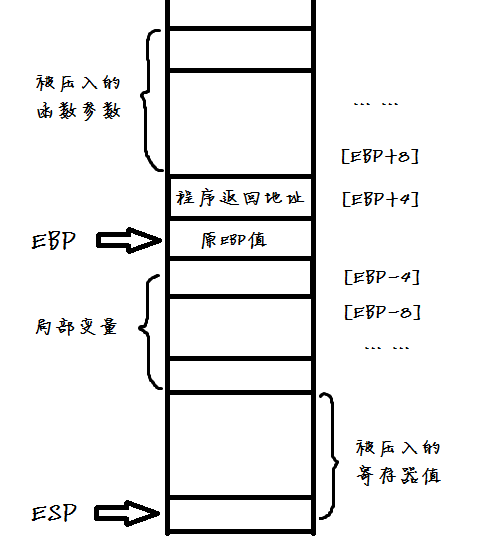


图 1 堆栈框架

如上图所示，堆栈框架可以用来存储函数参数、程序返回地址、局部变量以及现场。

## 2.7 C语言中的函数参数

### 2.7.1 不可变参数

定义结构体S:

struct S {

int mS=10, mT=2;

char mC = 1;

};

然后在主函数中定义如下变量，并调用f函数。

int x=0;

char y=1;

S s;

int \*p=&x;

int v[10];

f(x, y, s, p, v);

观察其反汇编代码，如下：

int x=0;

01324B58 mov dword ptr [ebp-0Ch],0

char y=1;

01324B5F mov byte ptr [ebp-15h],1

S s;

01324B63 lea ecx,[ebp-2Ch]

S s;

01324B66 call 01321352

int \*p=&x;

01324B6B lea eax,[ebp-0Ch]

01324B6E mov dword ptr [ebp-38h],eax

int v[10];

f(x, y, s, p, v);

01324B71 lea eax,[ebp-68h]

01324B74 push eax

01324B75 mov ecx,dword ptr [ebp-38h]

01324B78 push ecx

01324B79 sub esp,0Ch

01324B7C mov edx,esp

01324B7E mov eax,dword ptr [ebp-2Ch]

01324B81 mov dword ptr [edx],eax

01324B83 mov ecx,dword ptr [ebp-28h]

01324B86 mov dword ptr [edx+4],ecx

01324B89 mov eax,dword ptr [ebp-24h]

01324B8C mov dword ptr [edx+8],eax

01324B8F movzx ecx,byte ptr [ebp-15h]

01324B93 push ecx

01324B94 mov edx,dword ptr [ebp-0Ch]

01324B97 push edx

01324B98 call 0132134D

01324B9D add esp,1Ch

可以发现，主函数在调用f函数时，会使用内存对齐，将char填充到4byte，高位填0。而且在传递数组时，传递了数组的首地址。在传递结构体时，C语言将会把成员变量一一放入栈中，因此C语言在传递结构体时使用了传值的方法。

观察f函数对x与y进行操作的代码如下：

x = 10;

013216EE mov dword ptr [ebp+8],0Ah

\*p = 10;

013216F5 mov eax,dword ptr [ebp+1Ch]

013216F8 mov dword ptr [eax],0Ah

可知：函数体在使用传入的值时，会直接访问对应的参数。在使用传入的地址时，会先将参数放入寄存器，然后用寄存器相对寻址，得到指针指向的内容，并加以使用。

### 2.7.2 可变参数函数原理

将以下C代码编译运行：

#include<stdio.h>

#include<stdarg.h>

void f(int n, ...) {

va\_list pArgs = NULL;

va\_start(pArgs, n);

while (n--) {

va\_arg(pArgs, int);

}

va\_end(pArgs);

}

int main() {

int n = 3, x = 0, y = 1, z = 3;

f(n, x, y, z);

return 0;

}

可以得到调用函数f时的汇编代码：

f(n, x, y, z);

mov eax,dword ptr [ebp-2Ch]

push eax

mov ecx,dword ptr [ebp-20h]

push ecx

mov edx,dword ptr [ebp-14h]

push edx

mov eax,dword ptr [ebp-8]

push eax

call 0109135C

似乎跟不可变参数的参数传递方式一模一样？不要急。接下来是f函数的反汇编代码：

va\_list pArgs = NULL;

mov dword ptr [ebp-8],0

va\_start(pArgs, n);

call 01091357

lea eax,[ebp+0Ch]

mov dword ptr [ebp-8],eax

while (n--) {

mov eax,dword ptr [ebp+8]

mov dword ptr [ebp+FFFFFF30h],eax

mov ecx,dword ptr [ebp+8]

sub ecx,1

mov dword ptr [ebp+8],ecx

cmp dword ptr [ebp+FFFFFF30h],0

je 01093976

va\_arg(pArgs, int);

mov eax,dword ptr [ebp-8]

add eax,4

mov dword ptr [ebp-8],eax

}

jmp 01093950

va\_end(pArgs);

mov dword ptr [ebp-8],0

}

实际上，可变参数传递与不可变参数传递确实没有太大不同。在可变参数的函数中，我们需要维护一个指针指向传递的参数，并且需要通过某种约定好的方式来获得每个可变参数的类型（例如scanf的%d可以提示下一个参数的类型为int），从而可以判断出指针需要加的值，以此来指向下一个参数。

如果查阅资料，便可以知道va\_list、va\_start、va\_arg以及va\_end宏定义如下：

typedef char \* va\_list;

#define \_INTSIZEOF(n) ( (sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1) )

#define va\_start(ap,v) ( ap = (va\_list)&v + \_INTSIZEOF(v) )

#define va\_arg(ap, type) ( \*(type \*)((ap += \_INTSIZEOF(type)) - \_INTSIZEOF(type)) )

#define va\_end(ap) ( ap = (va\_list)0 )

这也与我们刚才的结论不矛盾。

## 2.8 C语言中的宏

将如下C语言代码反汇编：

#include<stdio.h>

#include<stdarg.h>

int max(int x, int y) {

return x > y ? x : y;

}

#define MAX(x, y) (x>y?x:y)

int main() {

int x = 1, y = 2;

max(x, y);

MAX(x, y);

return 0;

}

可以得到：

int x = 1, y = 2;

mov dword ptr [ebp-8],1

mov dword ptr [ebp-14h],2

max(x, y);

mov eax,dword ptr [ebp-14h]

max(x, y);

push eax

mov ecx,dword ptr [ebp-8]

push ecx

call 00AF1361

add esp,8

MAX(x, y);

mov eax,dword ptr [ebp-8]

cmp eax,dword ptr [ebp-14h]

jle 00AF396F

mov ecx,dword ptr [ebp-8]

mov dword ptr [ebp+FFFFFF24h],ecx

jmp 00AF3978

mov edx,dword ptr [ebp-14h]

mov dword ptr [ebp+FFFFFF24h],edx

x > y ? x : y;

mov eax,dword ptr [ebp-8]

cmp eax,dword ptr [ebp-14h]

jle 00AF398B

mov ecx,dword ptr [ebp-8]

mov dword ptr [ebp+FFFFFF24h],ecx

jmp 00AF3994

mov edx,dword ptr [ebp-14h]

mov dword ptr [ebp+FFFFFF24h],edx

可以看出，MAX(x,y)与x>y?x:y几乎完全等价，而与同样作用的过程max(x,y)却大不一样。使用宏后，编译器在编译时会自动将其替换成对应的代码，这样虽然会增大程序大小，但是可以提高效率，毕竟减少了参数传递等一系列耗时的操作。

## 2.9 外部功能的调用

### 2.9.1 C源程序函数的调用

C语言的程序之间互相调用函数。编译器在编译时首先将每个函数的短地址确定，然后根据每个程序所需要用到的函数去找对应的函数段地址，再将对应的call语句填充完整。因此C源程序在调用外部函数时，与调用内部函数并不太大区别。

### 2.9.2 C库函数的调用

库函数调用分为静态调用与动态调用，两者的区别为：静态调用是在编译时调用，编译器将调用的代码写入程序；而动态调用是在运行时调用，程序调用内存中的库函数，可以减少内存占用以及程序大小。

由于调用C库函数时，使用的是静态调用，因此被调用的C库函数就在代码段中。与程序内部定义的函数使用类似。

### 2.9.3 ASM函数的调用

ASM函数调用尽管在写法上与C源程序之间的调用有些许不同，但原理都近似。

### 2.9.4 OS函数的调用

在普通的应用程序中，如果想要调用系统函数（即系统调用）需要从用户态转为内核态，因为只有在内核空间的进程才能调用系统函数。应用程序调用系统调用的过程是：

1. 把系统调用的编号存入 EAX；
2. 把函数参数存入其它通用寄存器；
3. 触发 0x80 号中断（int 0x80）。

## 2.10 C++中类的本质

编译如下C++源代码：

// stack.cpp : Defines the entry point for the console application.

//

#include <stdio.h>

#include <conio.h>

class Stack

{

int \*a; int len, top;

public: Stack(int n); //初始化堆栈

~Stack(); //析构堆栈空间

Stack(const Stack& t); //拷贝构造函数

void push(int x); //将x压入堆栈

int pop(); //将堆栈元素弹出返回

};

Stack::Stack(int n)

{

a = new int[n];

len = n; top = 0;

}

Stack::~Stack()

{

delete[]a;

}

Stack::Stack(const Stack& t)

{

a = new int[t.len];

len = t.len;

top = t.top;

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

a[i] = t.a[i];

}

void Stack::push(int x)

{

if (top == len)

{

printf("Stack is Full !\n");

return;

}

a[top++] = x;

}

int Stack::pop()

{

return a[--top];

}

int main(int argc, char \* argv[])

{

Stack s(100);

int i;

for (i = 1; i <= 100; i++)

s.push(i);

for (i = 1; i <= 100; i++)

printf(" %d ", s.pop());

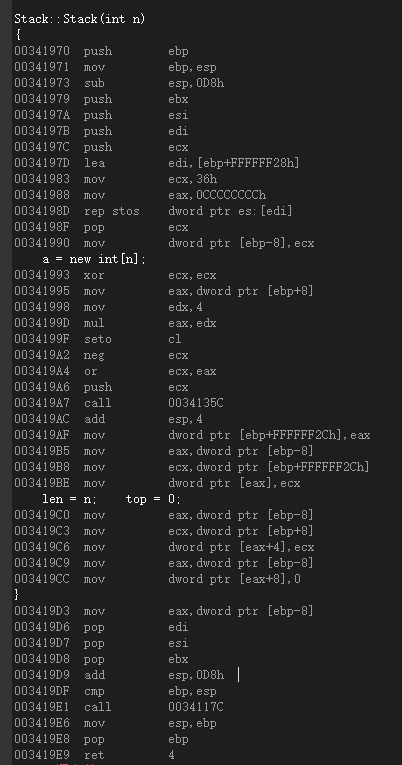
getchar();

return 0;

}

经过测试，可得出以下结论：

1. 构造函数储存在代码段中，在程序开始运行时便已存在。
2. 类的各个非静态成员方法必须在构造函数过程中分配到地址，在构造函数钱选择“转到反汇编”会报错。其构造函数的汇编代码如下：

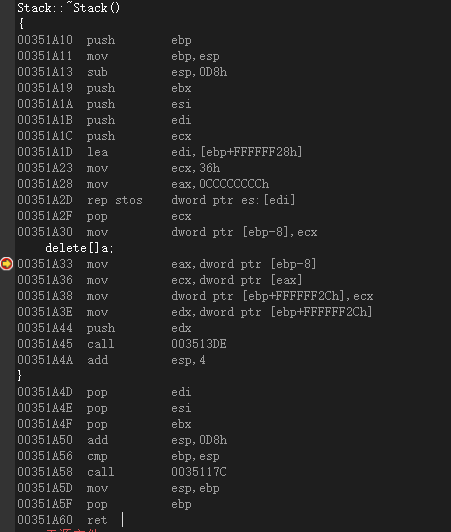


1. Stack类的实例s的各个属性在构造函数前已经存在，并且都在堆栈中。s在构造函数前在内存中的表示如下图：



三个成员，一共12byte。

1. 由上述两点，可得到C++中类的属性和方法的存储位置是分开的，并不相邻。
2. 在析构过程中，函数所占用的空间会被回收，其代码如下：



1. 类中的私有和共有成员在内存中并无差别，只是作为面向对象的特性，只在编译时进行使用的检查。

## 2.11 new/delete与动态数组

在使用new语句时，程序会向堆申请内存空间。堆是库函数使用一系列算法维护的，能够使用\_nh\_malloc\_dbg来申请内存空间。——《C++反汇编与逆向分析技术揭秘》

动态数组则是先申请一定大小的容纳若干元素的内存空间，设当前最大容纳的元素个数为n，当元素个数增大到n时，动态数组会重新申请能够容纳2n各元素的内存空间，然后将之前的n个元素复制到新的内存空间中，并将原来的内存空间销毁。

## 2.12 main函数的参数与返回值

将下列程序编译运行：

#include<stdio.h>

int main(int argc, char \*argv[])

{

for (int i = 0; i < argc; i++)

{

printf("第%0d个参数为：%s\n", i, argv[i]);

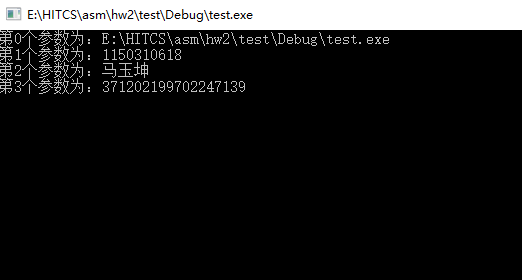
}

getchar();

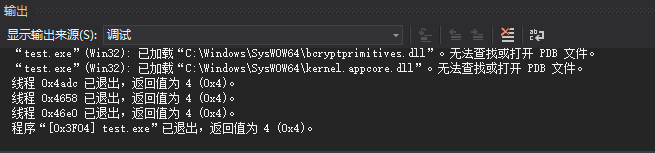
return argc++;

}

运行结果为：



程序正常终止后，VS的输出为：



由此可知：该程序是Visual Studio调用的，返回值返回到了调用者——即Visual Studio。

# 第3章 C语言与汇编语言的比较

|  |  |  |
| --- | --- | --- |
|  | C语言 | 汇编语言 |
| 优点 | 1. 结构清晰，逻辑简单，符合人类思维 2. 语句简单，开发效率高 3. 使用广泛，有众多库 4. 跨平台，可移植性强 | 1. 运行效率高，能够以极高的效率使用硬件 2. 贴近硬件，易于直接操作硬件 3. 可以由机器语言直接翻译而来，所以可用于可执行文件的逆向分析 4. 运行所需要的硬件要求较低（内存等） |
| 缺点 | 1. 运行效率较差 2. 远离硬件，如果要对硬件进行操作，则比较困难 | 1. 逻辑混乱 2. 语句繁琐，开发效率差 3. 极大依赖硬件，可移植性差 |
| 适应场合 | 1. 软件开发 2. 科学计算 | 1. 单片机编程 2. 嵌入式软件系统开发 3. 逆向工程 |

表 7 C语言与汇编语言比较

# 第4章 总结与分析，顿悟与畅想

通过这门课，我学习了汇编语言，重新认识了计算机的硬件及（底层）软件知识。也让我能够从汇编语言与C语言的比较中，意识到C语言的不足与优势。具体地讲，我收获了以下知识：

1. 汇编语言程序的编写
2. 操作系统管理用户程序的原理
3. 中断、异常与系统调用机制
4. C与汇编的联合编程
5. 计算机安全初识
6. 对C语言底层的重新认识与理解

出于兴趣，接下来我将继续学习有关计算机硬件的知识，尤其是并行计算方面。

# 第5章 其他收获与希望

通过这门课，我真的是收获了许多。在过去，由于高中参加信息学竞赛的基础，我自认为对C++乃至计算机的理解已经十分透彻，而这门课向我展示了计算机以及程序真正的一面，让我意识到计算机的魅力无穷无尽，仍然需要我去探索。

我认为最伟大的事莫过于把人们习以为常的事物解释清楚。计算机的操作系统、用户程序我们每天都在用，向多用户多任务这样的系统我们已经习以为常，然而却一直没有意识到这背后的高深奥妙的知识。史老师可以说为我讲明了这些我早已习以为常，已经忘了去问“为什么”的问题，让我能够进一步认识计算机，也认识到汇编语言的魅力，也能让我重新回到过去如饥似渴学习计算机知识的日子。

令我印象最深的莫过于实验3，破解csec.exe程序。我花了两个小时才通过搜索内存找到了名字对应的密码，之后又经过4个小时，第一次将一个可执行文件翻译成C++代码，十分有成就感。

我希望接下来学习计算机的日子能够像汇编课一样有趣又能学到许多知识。

最后，忠心感谢无所不知、无问不答的史先俊老师，以及各位负责的助教！

# 参考文献

[1] 欧文. Intel汇编语言程序设计[M]. 中国电力出版社, 2007.

[2] 钱林松. C++反汇编与逆向分析技术揭秘[M]. 机械工业出版社, 2011.

[3] 迈耶斯. Effective C++中文版[M]. 华中科技大学出版社, 2001.

[4] 王霄. 逆向工程技术及其应用[M]. 化学工业出版社, 2004.

# 致 谢

衷心感谢史先俊老师对本人的精心指导。

感谢汇编语言助教的帮助。