

# 了解Visual Studio 编译器调研报告

李科

学号：1711344

## Abstract

本文主要从Visual Studio 2019 Preview（以下简称VS）编译器出发，简要描述和探讨了编译器如何把C++代码最后翻译成目标机器语言的四个过程——预处理、编译、汇编、链接/加载，并且观察分析了各个步骤处理的结果，以及对优化选项的进行了了解。

**Keywords:** Visual Studio; 汇编; C++; 优化

## 1 引言

一个使用高级语言（便于人编写、阅读与维护，如：C++、JAVA等）编写的程序只有经过编译器的处理才能成为能被计算机识别的可以执行的机器语言程序被用户调用，处理输入并输出。由此可见：编译过程在计算机的世界里扮演了尤为重要的角色。了解透了编译过程，对我们编写代码、制作编译器有很好的启发作用。VS是一个具有强大功能的现代语言处理器，源程序在经过该语言处理器时会经过一个系统：预处理器->编译器->汇编器->链接器/加载器。VS可以对源程序的运行进行人为的调控，从而实现我们想实现的结果。文章以C++语言为例，介绍VS编译器的编译过程。一个C++程序，编写好后保存为.cpp文件，首先会经过预处理程序，预处理程序会把.cpp文件加工为.i文件，接下来就是核心步骤，也就是计算机会把.i文件送给编译器加工，编译器在加工时会分为6个阶段，然后生成汇编代码.asm文件，汇编代码再经过汇编器生成可重定位的机器代码.obj文件，最后通过链接器和加载器，调用库文件的内容和其他可重定位的文件形成目标机器代码（程序）.exe文件，我们就可以对程序进行输入使用了。

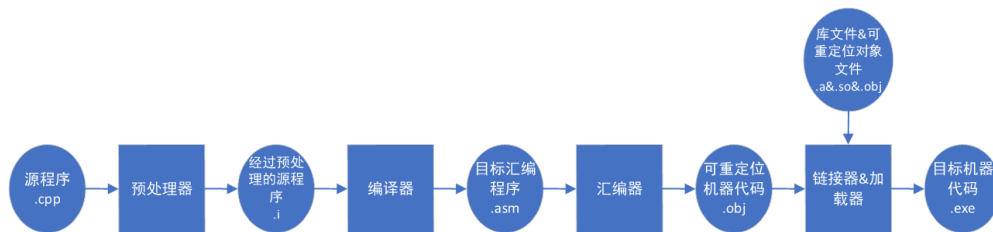


Figure 1: VS处理源代码过程

## 2 预处理器

预编译顾名思义就是对源程序进行预先处理，在VS中，如果直接点击“本地Windows调试器”进行调试，是不会生成预处理后的文件的，需要在如图2的界面进行设置，方可在项目目录（图3）中找到预编译后的.i文件。

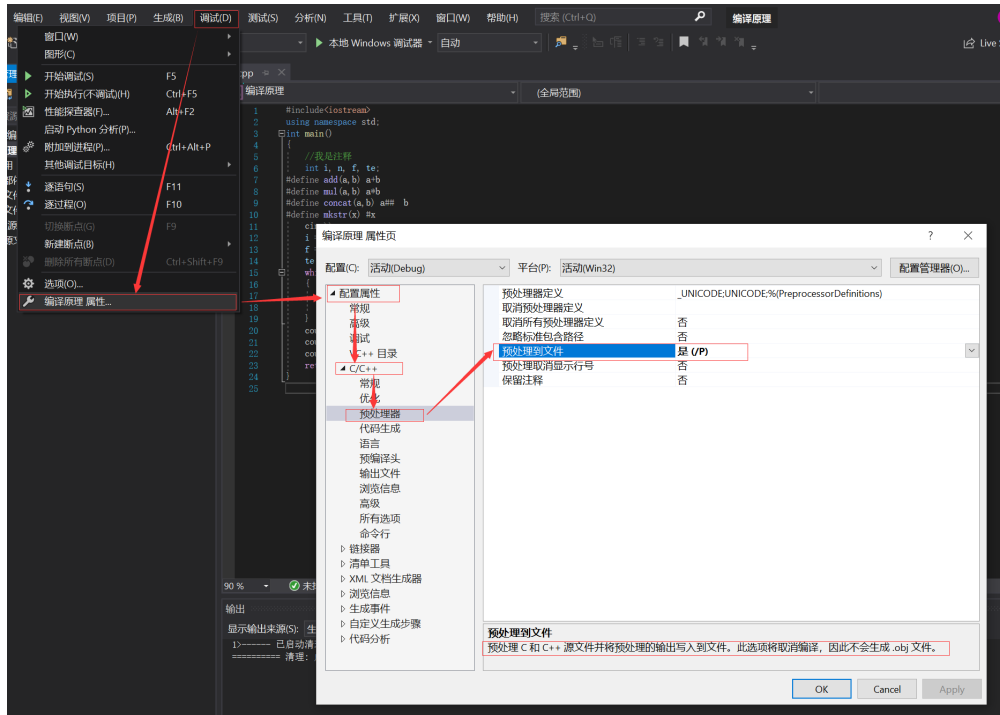


Figure 2: 预处理到文件

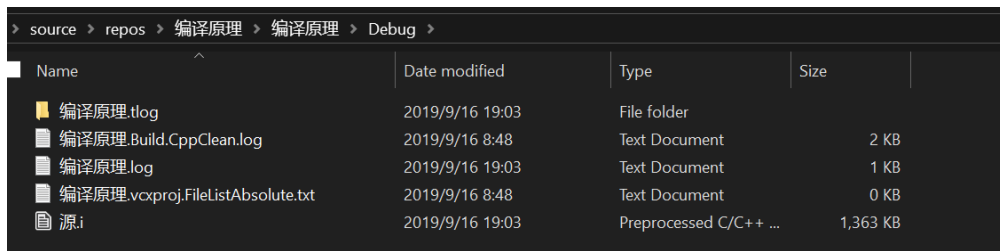
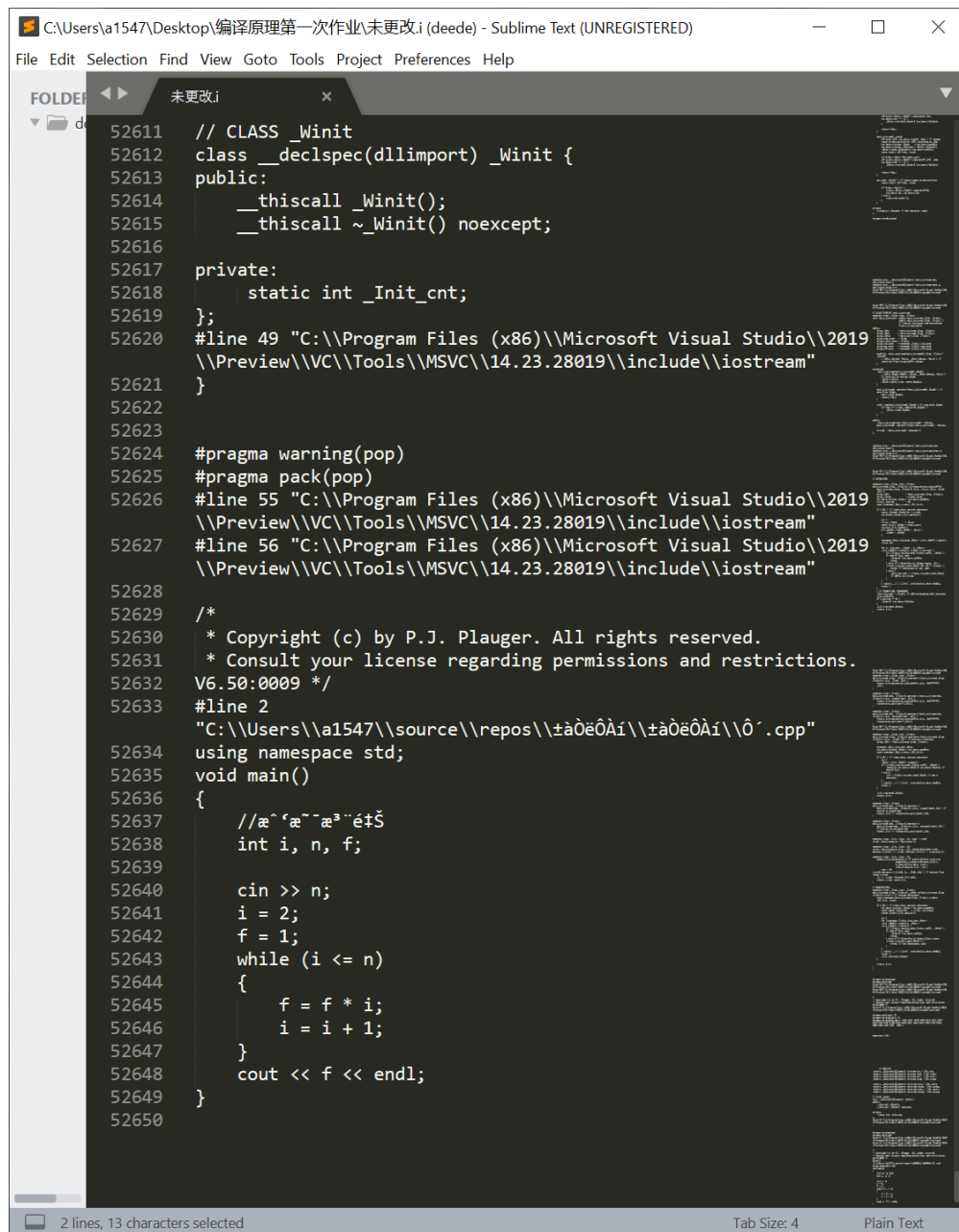


Figure 3: 项目目录

## 2.1 #include预处理

找到文件后就要分析预编译到底做了什么，打开该文件，发现文件有52649行，第一反应当然是惊讶，然后终于找到我能看懂的部分，也就是最后的代码。查阅了资料，知道了前面一大堆看不懂的就是预处理的库文件，也就是代码里的*iostream*。实际上我们输入的“*#include*”是一个指令，告诉预编译器这一块是需要他进行处理的。也就是会把头文件里的内容包含在*.i*文件里。



```
C:\Users\1547\Desktop\编译原理第一次作业\未更改.i (deede) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

FOLDER: d
未更改.i x

52611 // CLASS _Winit
52612 class __declspec(dllimport) _Winit {
52613 public:
52614     __thiscall _Winit();
52615     __thiscall ~_Winit() noexcept;
52616
52617 private:
52618     static int _Init_cnt;
52619 };
52620 #line 49 "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019
\\Preview\\VC\\Tools\\MSVC\\14.23.28019\\include\\iostream"
52621 }
52622
52623 #pragma warning(pop)
52624 #pragma pack(pop)
52625 #line 55 "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019
\\Preview\\VC\\Tools\\MSVC\\14.23.28019\\include\\iostream"
52626 #line 56 "C:\\Program Files (x86)\\Microsoft Visual Studio\\2019
\\Preview\\VC\\Tools\\MSVC\\14.23.28019\\include\\iostream"
52627
52628 /*
52629  * Copyright (c) by P.J. Plauger. All rights reserved.
52630  * Consult your license regarding permissions and restrictions.
52631  V6.50:0009 */
52632 #line 2
52633 "C:\\Users\\1547\\source\\repos\\1547\\1547\\0'.cpp"
52634 using namespace std;
52635 void main()
52636 {
52637     //æ`æ`æ`é$
52638     int i, n, f;
52639
52640     cin >> n;
52641     i = 2;
52642     f = 1;
52643     while (i <= n)
52644     {
52645         f = f * i;
52646         i = i + 1;
52647     }
52648     cout << f << endl;
52649 }
52650
```

Figure 4: 预处理代码

## 2.2 #define预处理

#define 命令可以定义一个标识符的含义，如：一个数字，一个函数。首先是一个数字，我定义一个num为1

```
#include<iostream>
using namespace std;
#define num 1
int main()
{
    //这是注释
    int i, n, f;
```

```

cin >> n;
i = 2;
f = 1;
while (i <= n)
{
    f = f * i;
    i = i + num;
}
cout << f << endl;
}

```

在经过预处理后，num直接被替换为了1

```

using namespace std;
void main()
{
    //æ~ 'æ~-æ³'é†§
    int i, n, f;

    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
}

```

然后对程序里的乘法与加法进行定义

```

#include<iostream>
using namespace std;
#define add(a,b) a+b
#define mul(a,b) a*b

void main()
{
    //我是注释
    int i, n, f;

    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = mul(f,i);
        i = add(i,1);
    }
    cout << f << endl;
}

```

```
}
```

生成的.i文件也和上面一样。因此知道了预处理其实就是一个替换的过程，把define的标识符进行一一替换为相应的内容，这样使得后面的编译器处理更加统一。

## 2.3 #和##运算

根据查阅资料，还发现了一个很好玩的运算，就是#和##运算，#运算是单目运算就是把后面的内容变为字符或者字符串，##运算是双目运算，作用是链接两个运算单元，使之变为一个新的标识符，应该是把这两个运算单元符号化进行链接，然后再让之变为标识符。

```
#include<iostream>
using namespace std;
void main()
{
    //我是注释
    int i, n, f,te;
    #define add(a,b) a+b
    #define mul(a,b) a*b
    #define concat(a,b) a## b
    #define MKSTR(x) #x
    cin >> n;
    i = 2;
    f = 1;
    te = 88;
    while (i <= n)
    {
        f = mul(f,i);
        i = add(i,1);
    }
    cout << MKSTR(te)<<endl;
    cout << concat(t, e) <<concat(f,i)<< endl;
    cout << f << endl;
}
```

生成的.i文件如下

```
using namespace std;
void main()
{
    //æ~ 'æ~æ³·é†Š
    int i, n, f,te;

    cin >> n;
    i = 2;
    f = 1;
    te = 88;
```

```

while (i <= n)
{
    f = f*i;
    i = i+1;
}
cout << "te"<<endl;
cout << te <<fi<< endl;
cout << f << endl;
}

```

可以看到#让te变成了字符串“te”，而##让单独的f与i，t与e链接而成了新的标识符 fi，te。再这个例子中  
可以发现，在预处理阶段，不管代码是不是正确的都可以生成相应的预处理文件，因此预编译器的作用不包含  
代码分析，而这一部分是由下面即将介绍的编译器实现。据资料显示，C++中预定义的宏还有：

**Table 1:** 其他的宏

宏	描述
<code>__LINE__</code>	这会在程序编译时包含当前行号。
<code>__FILE__</code>	这会在程序编译时包含当前文件名。
<code>__DATE__</code>	这会包含一个形式为 month/day/year 的字符串，它表示把源文件转换为目标代码的日期。
<code>__TIME__</code>	这会包含一个形式为 hour:minute:second 的字符串，它表示程序被编译的时间。

### 3 编译器

在经过预处理器处理后的代码，下一步就是进入编译器了，编译器分为两个基本的组成部分：分析、综合，或者叫前端与后端。我们课程主要研究的就是分析部分，分析是将源程序分解为基本组成部分，并且在  
这些部分上加上语法结构，或者语义上不一致就提供可供修正的信息，最后生成中间表示形式，主要包括：  
词法分析、语法分析、符号表创建、语义分析、中间代码生成、部分代码优化、错误处理。综合则是从中间  
表示形式构建目标程序，包括代码优化与代码生成。



**Figure 5:** 编译器的处理过程

#### 3.1 词法分析

编译器在对预处理文件进行第一步处理就是词法分析，也叫扫描。词法分析器读入预处理后的字符流，  
把之处理为词素，对于词素，词法分析器产生词法单元作为输出，从字符流到单词流。根据我的理解，词法  
分析器就是把代码里的出现的各种符号进行归类形成集合，如标识符集合、运算符集合、数字集合、届集合  
等。词法单元的形式如下：<抽象符号,属性值>，标识符既有抽象符号id，又有属性值，也就是在符号表里对  
应的条目，而有的词素就没有属性值，比如运算符。其实运算符本应该也拥有自己的抽象符号，但为了方便  
就用本身作为抽象符号的名字了。

## 3.2 语法分析

第二步处理叫语法分析，也叫解析，输入是词法分析器各个词法单元的第一个分量，输出是可以表示词法单元流的语法结构的语法树，树中的每个内部节点表示一个运算，子节点就表示运算的分量。我们将使用上下文无关文法来描述语法结构。在进行语法分析时也是使用递归分析，最后运算的，最先处理也就是在树的顶层。

## 3.3 语义分析

第三步处理叫语义分析，主要功能是查错，用上一步生成的语法树和符号表中的信息来检查源程序是否和语言定义的一致，当然它也要收集类型信息，并把信息放在语法树或者符号表中。语义分析器提供类型检查、自动类型转换的功能。类型检查就是检查每个运算符是否具有匹配的运算分量，该是什么类型就应该是什麼类型，比如数组的下标必须是整型，如果是其他类型就会报错。自动类型转换，可能允许某些类型的转换，比如浮点与整型的相互转换。

## 3.4 中间代码生成

第四步处理叫中间代码生成，在经过中间代码生成器处理后，会生成一个明确的低级的或类机器语言的中间表示，也就是某个抽象机器的程序。该程序容易生成，并且能够被轻松的翻译为目标机器上的语言。常见的形式为三地址代码，每个三地址赋值指令的右部最多只有一个运算符。编译器会生成一个临时名字来存放计算得到的值。当然有时指令运算分量少于三个。

## 3.5 代码优化

第五步处理叫代码优化，也是机器无关的代码优化，优化的目的是让代码运行更快，或者更短或能耗更低的目标代码。处理的是经过中间代码生成器生成的代码。我的理解就是减少只有两个运算分量的三地址指令，这些临时名字可以直接替换掉。有的简单的代码优化方法，可以很大程度地提高目标程序的运行效率而不会花费过多的编译时间。

## 3.6 代码生成

第六步处理，也是最后一步叫代码生成，输入时经过优化后的中间表示形式，输出则是目标语言，在C++的处理中，目标语言就是机器代码，所以就必须为程序使用的每个变量选择寄存器或内存位置，然后中间表示形式就可以被翻译成可以完成相同任务的机器指令序列。代码生成器最重要的功能就是合理分配寄存器供程序使用。

## 3.7 具体实现

### 3.7.1 汇编代码生成

因为使用的是VS，所以想要生成汇编代码就需要对VS进行配置。首先是关闭“预处理到文件”，只有这样程序才能正常编译

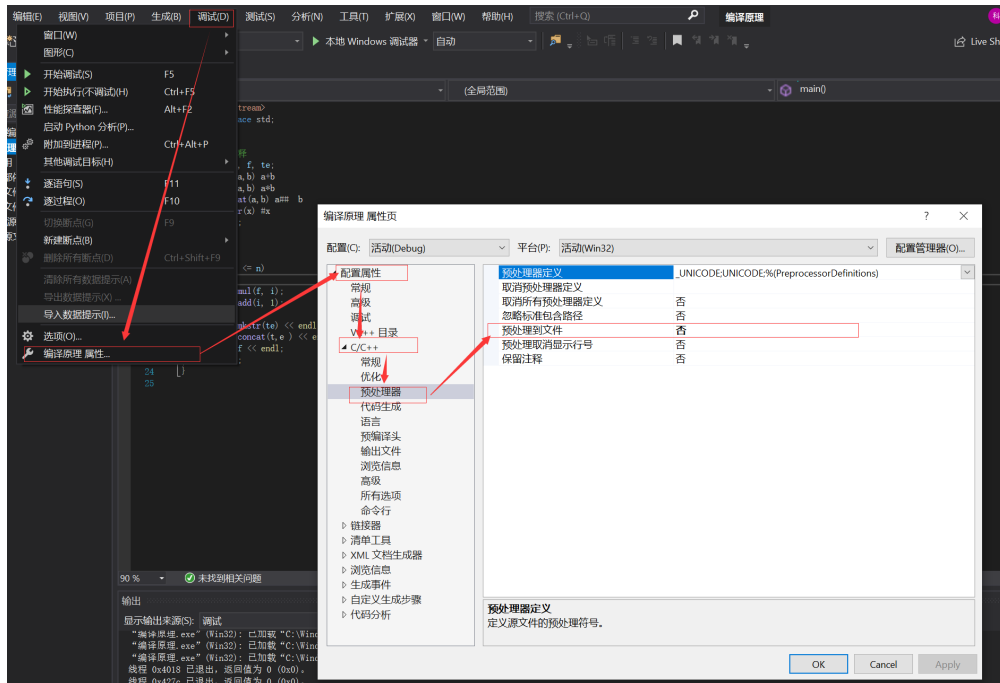


Figure 6: 关闭预处理到文件

然后设置输出文件，也就是输出汇编语言。

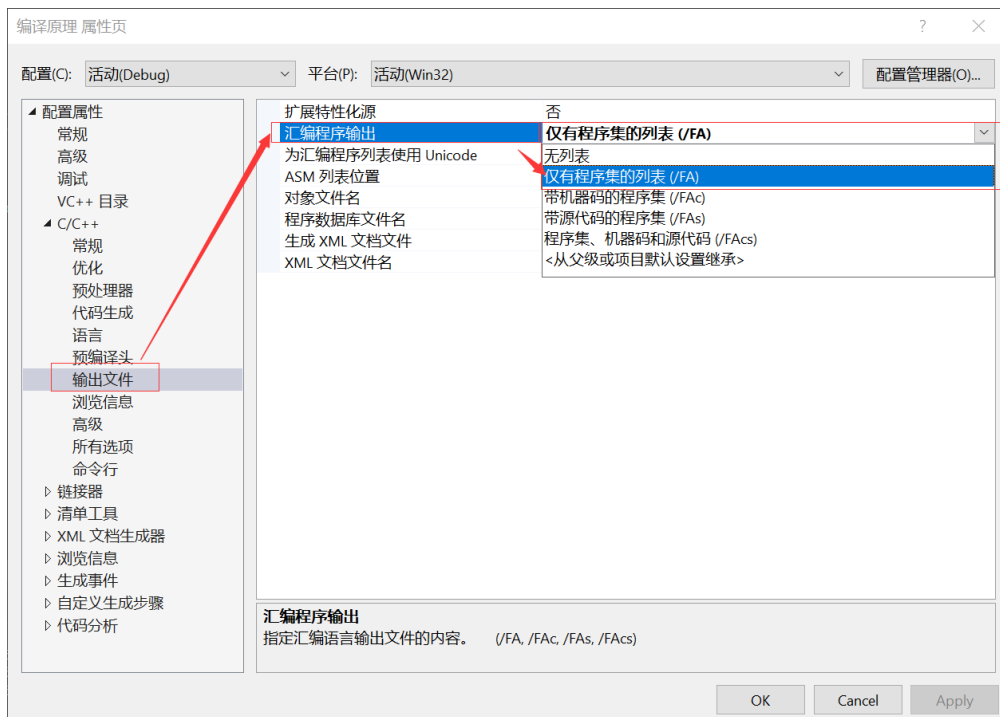


Figure 7: 汇编语言输出设置

点击调试后，进入项目debug目录就发现有了.asm文件



source > repos > 编译原理 > 编译原理 > Debug			
Name	Date modified	Type	Size
编译原理.tlog	2019/9/17 10:18	File folder	
vc142.idb	2019/9/17 10:18	VC++ Minimum Reb...	139 KB
vc142.pdb	2019/9/17 10:18	PDB File	396 KB
编译原理.Build.CppClean.log	2019/9/16 8:48	Text Document	2 KB
编译原理.log	2019/9/17 10:18	Text Document	1 KB
编译原理.vcxproj.FileListAbsolute.txt	2019/9/16 8:48	Text Document	0 KB
源.asm	2019/9/17 10:18	Assembler Source	45 KB
源.i	2019/9/17 10:17	Preprocessed C/C++ ...	1,363 KB
源.obj	2019/9/17 10:18	3D Object	54 KB

Figure 8: 汇编语言输出目录

打开该文件

```

1239 DD -24 ; ffffffff8H
1240 DD 4
1241 DD $LN5@main
1242 $LN5@main:
1243 DB 110 ; 0000006eH
1244 DB 0
1245 _main ENDP
1246 _TEXT ENDS
1247 ; Function compile flags: /Odtp /RTCsu /ZI
1248 ; COMDAT ?eof@?$_Narrow_char_traits@DH@std@@@SAHXZ
1249 _TEXT SEGMENT
1250 __$EHRec$ = -12 ; size = 12
1251 ?eof@?$_Narrow_char_traits@DH@std@@@SAHXZ PROC ;
std::_Narrow_char_traits<char,int>::eof, COMDAT
1252 ; File C:\Program Files (x86)\Microsoft Visual
Studio\2019\Preview\VC\Tools\MSVC\14.23.28019\include\xstring
; Line 410
1253 push ebp
1254 mov ebp, esp
1255 push -1
1256 push __ehandler$?eof@?$_Narrow_char_traits@DH@std@@@SAHXZ
1257 mov eax, DWORD PTR fs:0
1258 push eax
1259 sub esp, 192 ; 000000c0H
1260 push ebx
1261 push esi
1262 push edi
1263 lea edi, DWORD PTR [ebp-204]
1264 mov ecx, 48 ; 00000030H
1265 mov eax, -858993460 ; ccccccccH
1266 rep stosd
1267 mov eax, DWORD PTR __security_cookie
1268 xor eax, ebp
1269 push eax
1270 lea eax, DWORD PTR __$EHRec$[ebp]
1271 mov DWORD PTR fs:0, eax
1272 mov ecx, OFFSET __81E947AA_xstring
1273 call @__CheckForDebuggerJustMyCode@4
1274 ; Line 411
1275 or eax, -1
1276 ; Line 412
1277 mov ecx, DWORD PTR __$EHRec$[ebp]
1278 mov DWORD PTR fs:0, ecx
1279 pop ecx
1280 pop edi
1281 pop esi
1282 pop ebx
1283

```

Figure 9: 汇编代码

发现是一堆汇编代码，那么编译器的工作就完成了。因为在刚才的设置中看到有多种选择方式，这次选

择了第四个，也就是带有程序集、机器码和源代码

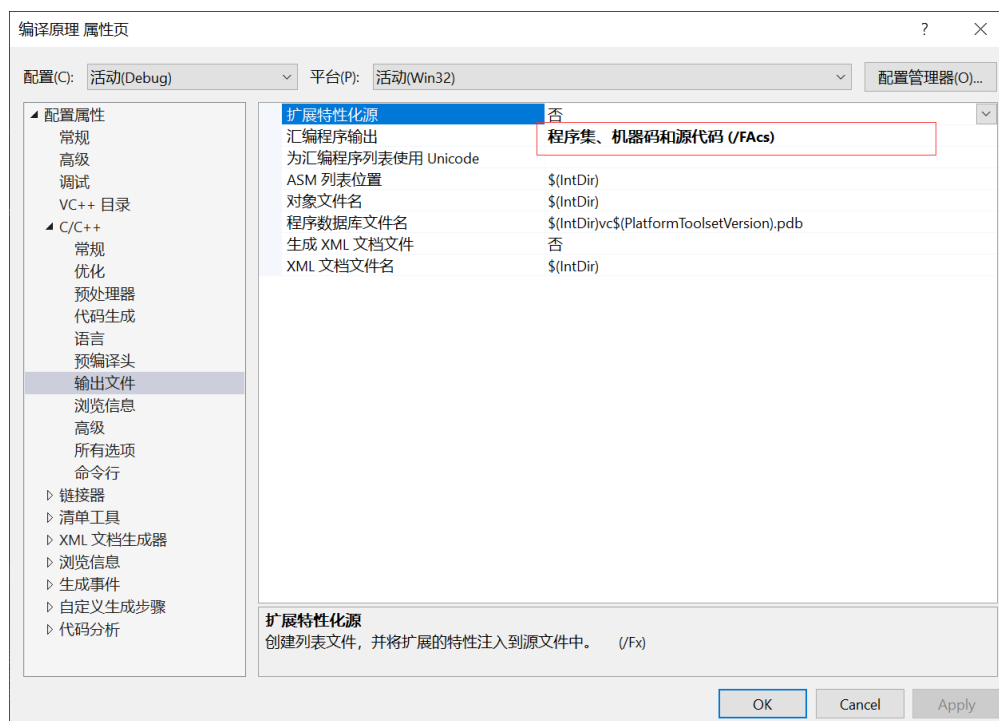


Figure 10: 程序集、机器码和源代码

调试后就发现在debug目录下的.asm文件变成了.cod文件

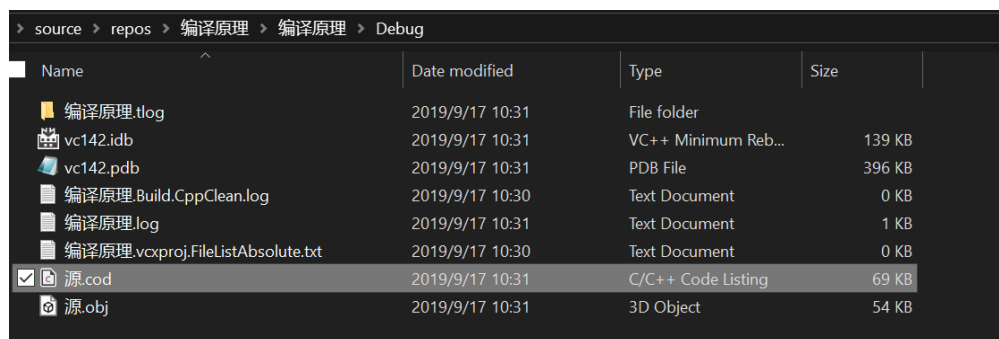


Figure 11: cod目录

打开文件翻到最后面就发现和刚刚相比多了源代码和相应的机器码

```
@D@std@@@std@@@QAEAAV01@AAH@Z
1437 00044 3b f4    cmp     esi, esp
1438 00046 e8 00 00 00 00    call  __RTC_CheckEsp
1439
1440 ; 12  :  i = 2;
1441
1442 0004b c7 45 f4 02 00
1443 00 00    mov     DWORD PTR _i$[ebp], 2
1444
1445 ; 13  :  f = 1;
1446
1447 00052 c7 45 dc 01 00
1448 00 00    mov     DWORD PTR _f$[ebp], 1
1449
1450 ; 14  :  te = 88;
1451
1452 00059 c7 45 d0 58 00
1453 00 00    mov     DWORD PTR _te$[ebp], 88 ; 00000058H
1454 $LN2@main:
1455
1456 ; 15  :  while (i <= n)
1457
1458 00060 8b 45 f4    mov     eax, DWORD PTR _i$[ebp]
1459 00063 3b 45 e8    cmp     eax, DWORD PTR _n$[ebp]
1460 00066 7f 15      jg      SHORT $LN3@main
1461
1462 ; 16  :  {
1463 ; 17  :      f = mul(f, i);
1464
1465 00068 8b 45 dc    mov     eax, DWORD PTR _f$[ebp]
1466 0006b 0f af 45 f4 imul    eax, DWORD PTR _i$[ebp]
1467 0006f 89 45 dc    mov     DWORD PTR _f$[ebp], eax
1468
1469 ; 18  :      i = add(i, 1);
1470
1471 00072 8b 45 f4    mov     eax, DWORD PTR _i$[ebp]
1472 00075 83 c0 01    add     eax, 1
1473 00078 89 45 f4    mov     DWORD PTR _i$[ebp], eax
1474
1475 ; 19  :  }
1476
1477 0007b eb e3      jmp     SHORT $LN2@main
1478 $LN3@main:
1479
1480 ; 20  :  cout << mkstr(te) << endl;
1481
1482 0007d 8b f4      mov     esi, esp
```

Figure 12: 含有机码和源代码的汇编代码

这样对于我们理解代码和编译原理有了很大的帮助。其他选项类似。

### 3.7.2 代码优化实现

现在对代码进行优化，同样在VS的编译属性里进行设置

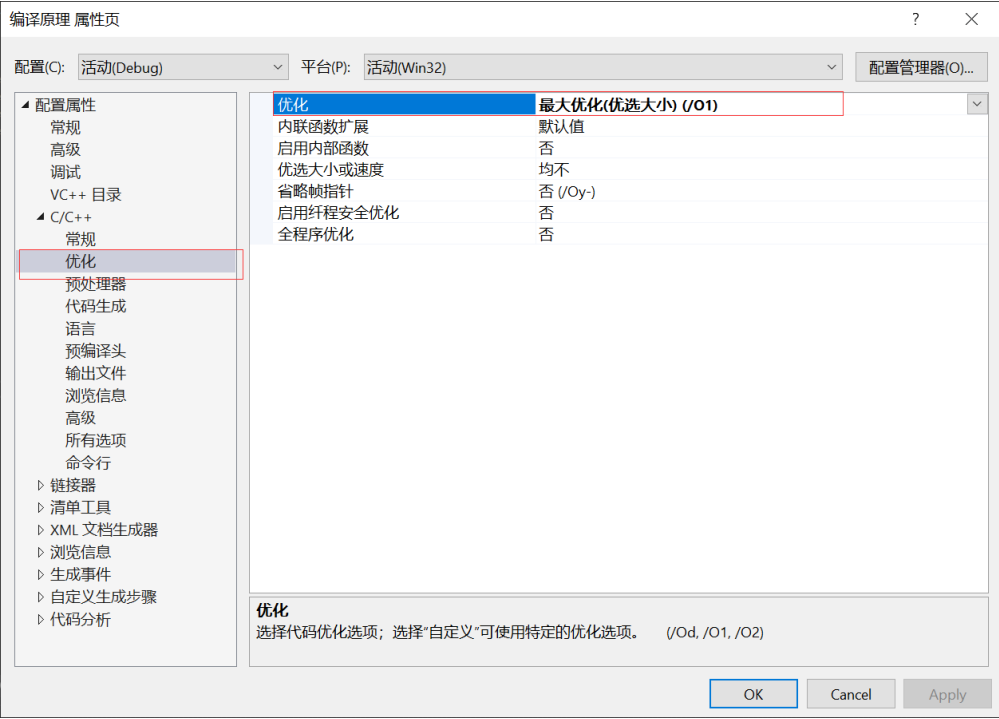


Figure 13: 优化设置

先选择最大优化在执行之前需要更改基本运行时检查为默认值，否则会报错

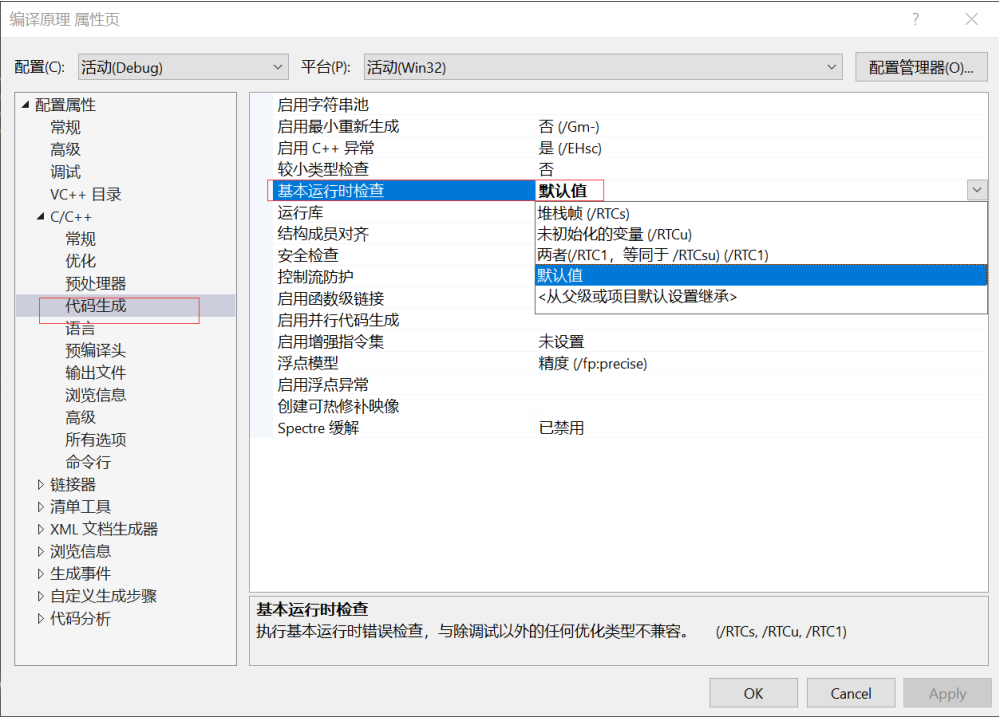


Figure 14: 基本运行时检查设置

在最大优化（优化大小）后的Debug文件夹

Name	Date modified	Type	Size
编译原理.tlog	2019/9/17 11:58	File folder	
vc142.idb	2019/9/17 11:57	VC++ Minimum Reb...	147 KB
vc142.pdb	2019/9/17 11:57	PDB File	396 KB
编译原理.Build.CppClean.log	2019/9/17 10:34	Text Document	0 KB
编译原理.log	2019/9/17 11:58	Text Document	1 KB
编译原理.vcxproj.FileListAbsolute.txt	2019/9/17 10:34	Text Document	0 KB
源.asm	2019/9/17 11:57	Assembler Source	29 KB
源.obj	2019/9/17 11:57	3D Object	49 KB

Figure 15: 优化大小后的Debug文件夹

对比优化前的文件夹

Name	Date modified	Type	Size
编译原理.tlog	2019/9/17 12:01	File folder	
vc142.idb	2019/9/17 12:01	VC++ Minimum Reb...	139 KB
vc142.pdb	2019/9/17 12:01	PDB File	396 KB
编译原理.Build.CppClean.log	2019/9/17 12:01	Text Document	0 KB
编译原理.log	2019/9/17 12:01	Text Document	1 KB
编译原理.vcxproj.FileListAbsolute.txt	2019/9/17 12:01	Text Document	0 KB
源.asm	2019/9/17 12:01	Assembler Source	40 KB
源.obj	2019/9/17 12:01	3D Object	51 KB

Figure 16: 优化前的Debug文件夹

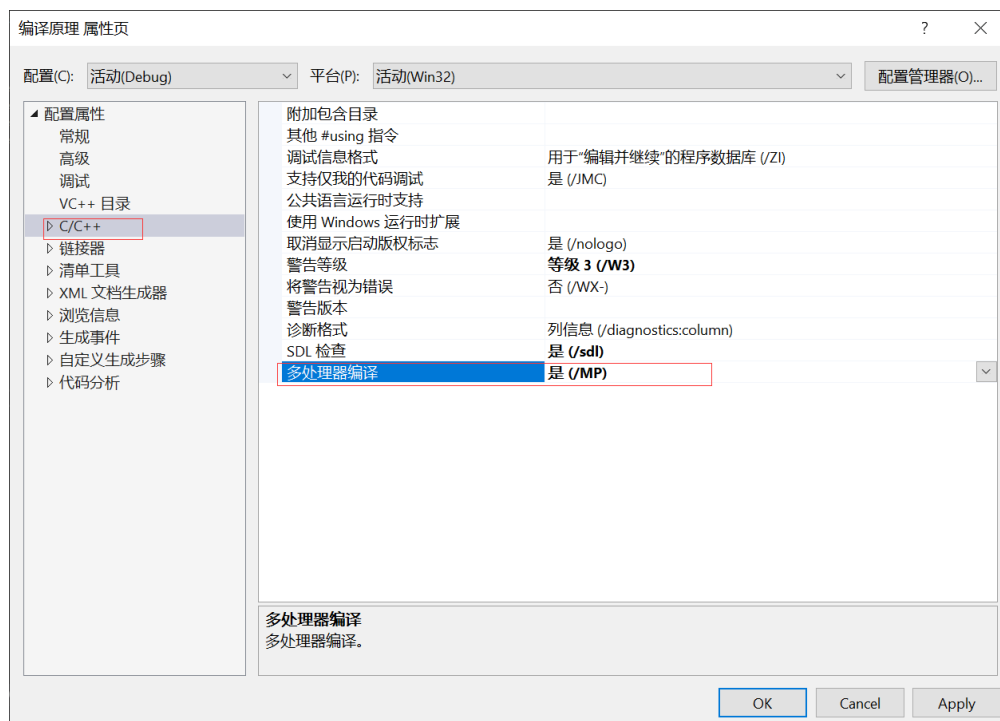
可见.asm文件明显减小这一次我们选择最大优化（优化速度）发现优化后的文件变得更小了，而程序执行的速度肉眼可见也变快了。

Name	Date modified	Type	Size
编译原理.tlog	2019/9/17 12:04	File folder	
vc142.idb	2019/9/17 12:04	VC++ Minimum Reb...	139 KB
vc142.pdb	2019/9/17 12:04	PDB File	396 KB
编译原理.Build.CppClean.log	2019/9/17 12:03	Text Document	0 KB
编译原理.log	2019/9/17 12:04	Text Document	1 KB
编译原理.vcxproj.FileListAbsolute.txt	2019/9/17 12:03	Text Document	0 KB
源.asm	2019/9/17 12:04	Assembler Source	28 KB
源.obj	2019/9/17 12:04	3D Object	48 KB

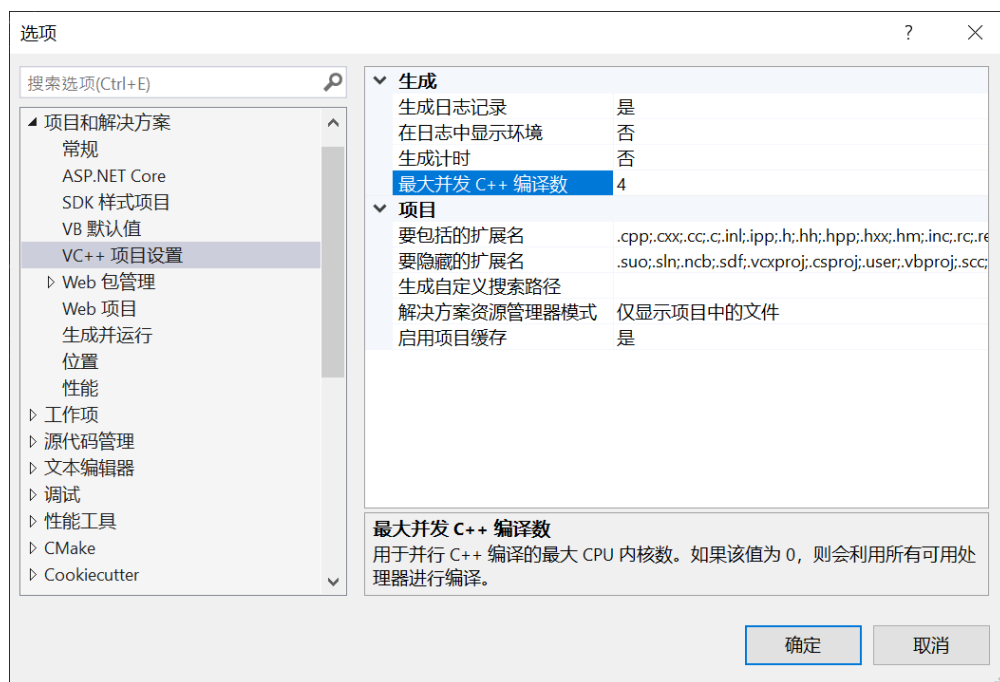
Figure 17: 优化速度后的Debug文件夹

### 3.7.3 并行优化

要进行并行优化，现在属性页进行设置



**Figure 18: 多处理器编译设置**



**Figure 19:** 最大并发编译数设置

但学生看开启后的线程数和开启前的线程数一致，都是三。

```
"编译原理.exe" (Win32): 已加载 "C:\Windows\SysWOW64\sechost.dll"。  
线程 0x10c0 已退出, 返回值为 0 (0x0)。  
线程 0x2580 已退出, 返回值为 0 (0x0)。  
线程 0x2564 已退出, 返回值为 0 (0x0)。  
程序 "[18196] 编译原理.exe" 已退出, 返回值为 0 (0x0)。
```

Figure 20: 程序线程

## 4 汇编器

### 4.1 主要过程

汇编器从本质上讲也是编译器，汇编器也会进行词法分析、语法分析、语义分析、符号表管理和代码生成等阶段。汇编器把解析出的指令简介地映射到正确的机器代码相对比较复杂，但是汇编器输入的代码时经过编译器处理后的代码，所以是正确的汇编文件，因此不需要考虑源文件出错，所以它语法分析的目的是识别出汇编文件里的语法结构并进行解析引导机器代码生成。汇编器允许符号后置定义，所以必须采用两边扫描的方式进行设计。汇编器的结构如下：

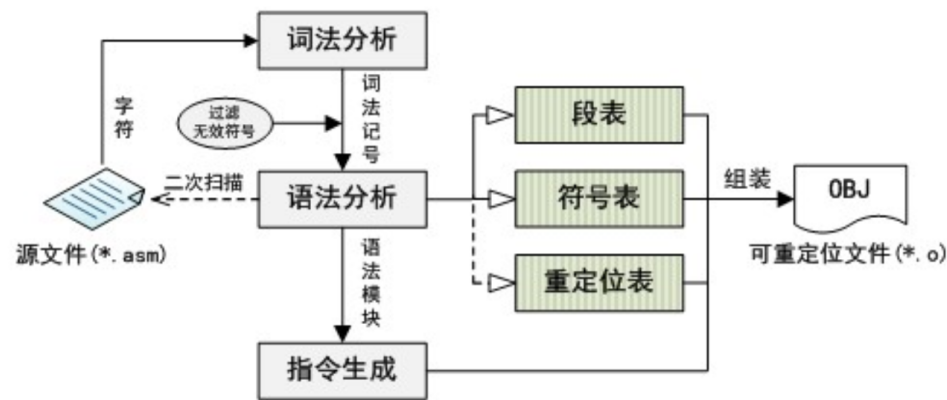


Figure 21: 汇编器处理过程

在语法分析前的结构与编译器完全相同，就只是在语法分析时需要进行二次扫描。第一遍扫描获取文件定义的所有段的信息和符号信息，第二遍扫描则根据第一次扫描后的结果把所有的重定位信息收集到重定位表中，通过指令生成了代码段数据，最后从符号表中抽取有效数据定义形成数据段，符号导出到文件符号表段，再把所有段按照elf文件的格式进行组装，最终形成了可重定位的目标文件.obj。

### 4.2 实现

直接点击调试就可以在目录里找到.obj文件

source > repos > 编译原理 > 编译原理 > Debug					Search Debug
Name	Date modified	Type	Size		
编译原理.tlog	2019/9/17 16:37	File folder			
vc142.idb	2019/9/17 16:37	VC++ Minimum Reb...	147 KB		
vc142.pdb	2019/9/17 16:37	PDB File	396 KB		
编译原理.Build.CppClean.log	2019/9/17 16:37	Text Document	0 KB		
编译原理.log	2019/9/17 16:37	Text Document	1 KB		
编译原理.vcxproj.FileListAbsolute.txt	2019/9/17 16:37	Text Document	0 KB		
源.cod	2019/9/17 16:37	C/C++ Code Listing	54 KB		
源.obj	2019/9/17 16:37	3D Object	51 KB		

Figure 22: 机器码目录

打开这个文件就看到时一堆二进制代码。

```

3216 6368 6172 5+74 7261 6974 7340 4440 7374
3217 6440 4040 3040 4141 5631 3040 5042 4440
3218 5a00 5f5f 7472 7962 6c6f 636b 7461 626c
3219 6524 3f3f 243f 3655 3f24 6368 6172 5f74
3220 7261 6974 7340 4440 7374 6440 4040 7374
3221 6440 4059 4141 4156 3f24 6261 7369 635f
3222 6f73 7472 6561 6d40 4455 3f24 6368 6172
3223 5f74 7261 6974 7340 4440 7374 6440 4040
3224 3040 4141 5631 3040 5042 4440 5a00 5f5f
3225 6361 7463 6873 796d 243f 3f24 3f36 553f
3226 2463 6861 725f 7472 6169 7473 4044 4073
3227 7464 4040 4073 7464 4040 5941 4141 563f
3228 2462 6173 6963 5f6f 7374 7265 616d 4044
3229 553f 2463 6861 725f 7472 6169 7473 4044
3230 4073 7464 4040 4030 4041 4156 3130 4050
3231 4244 405a 2433 005f 5f65 6866 756e 6369
3232 6e66 6f24 3f3f 315f 5365 6e74 7279 5f62
3233 6173 6540 3f24 6261 7369 635f 6f73 7472
3234 6561 6d40 4455 3f24 6368 6172 5f74 7261
3235 6974 7340 4440 7374 6440 4040 7374 6440
3236 4051 4145 4058 5a00 5f5f 6568 6675 6e63
3237 696e 666f 243f 3f30 7365 6e74 7279 403f
3238 2462 6173 6963 5f6f 7374 7265 616d 4044
3239 553f 2463 6861 725f 7472 6169 7473 4044
3240 4073 7464 4040 4073 7464 4040 5141 4540
3241 4141 5631 3240 405a 005f 5f75 6e77 696e
3242 6474 6162 6c65 243f 3f30 7365 6e74 7279
3243 403f 2462 6173 6963 5f6f 7374 7265 616d
3244 4044 553f 2463 6861 725f 7472 6169 7473
3245 4044 4073 7464 4040 4073 7464 4040 5141
3246 4540 4141 5631 3240 405a 005f 5f65 6866
3247 756e 6369 6e66 6f24 3f3f 3173 656e 7472
3248 7940 3f24 6261 7369 635f 6f73 7472 6561
3249 6d40 4455 3f24 6368 6172 5f74 7261 6974
3250 7340 4440 7374 6440 4040 7374 6440 4051
3251 4145 4058 5a00 5f5f 696d 705f 3f63 696e
3252 4073 7464 4040 3356 3f24 6261 7369 635f
3253 6973 7472 6561 6d40 4455 3f24 6368 6172
3254 5f74 7261 6974 7340 4440 7374 6440 4040
3255 3140 4100 5f5f 696d 705f 3f63 6f75 7440
3256 7374 6440 4033 563f 2462 6173 6963 5f6f
3257 7374 7265 616d 4044 553f 2463 6861 725f
3258 7472 6169 7473 4044 4073 7464 4040 4031
3259 4041 003f 3f5f 4340 5f30 3245 4c4d 4d48
3260 4a49 4440 7465 4000 5f5f 5f73 6563 7572
3261 6974 795f 636f 6f6b 6965 00

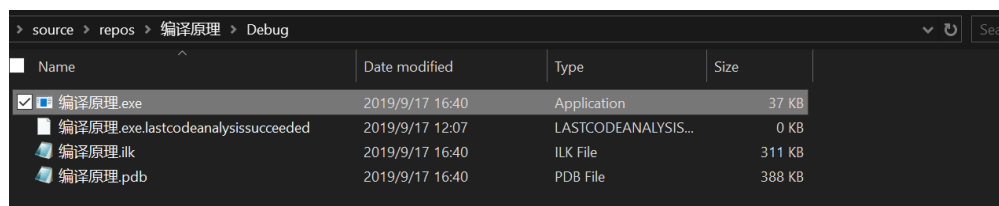
```

Figure 23: 机器码



## 5 链接器/加载器

链接器和加载器主要完成重定位和符号解析，对于大型的项目使得项目可以分割到多个文件，多线程编译，然后通过链接器连接起来，一个编译器产生的目标文件中包括**bss**段、**data**段、**text**段、重定位表、符号表、还可能有调试信息。**bss**段保存了未初始化的数据，分配空间时初始为0。重定位表里面包含了重定位需要的信息，包括位置，变量地址，转移地址等。假设所有的位置的基地址都为0，链接器把段合并的时候，某些目标文件相应的及地址一定会发生改变，那么链接器就要根据表找到这些位置对它们按照新的基地址进行更新。重定位表和符号表是最重要的两个表，主要完成重定位和符号解析。对于一个目标文件来说，也需要链接系统库，则链接器就需要去库里找到需要的模块链接到目标文件里。这里的符号表主要用来让链接器和加载器用在合并文件、查找外部引用、重定位的时候使用的，与编译器生成的符号表有类似的作用，但不是同一个。有的目标文件，在链接时就确定了地址，加载器在加载的时候就可以直接加载到那个地址，这叫静态链接。但有时不能确定地址，就需要动态链接，需要保存重定位信息和符号信息，在运行时重新定位，由加载器根据加载时确定的基地址对程序做最后一次处理。在进行调试后就可以在项目目录里找到.exe文件了



Name	Date modified	Type	Size
编译原理.exe	2019/9/17 16:40	Application	37 KB
编译原理.exe.lastcodeanalysis...	2019/9/17 12:07	LASTCODEANALYSIS...	0 KB
编译原理.ilc	2019/9/17 16:40	ILK File	311 KB
编译原理.pdb	2019/9/17 16:40	PDB File	388 KB

Figure 24: exe文件目录

双击文件运行输入6，就可以输出相应的结果了，分别是字符串“te”、te的值、以及6的阶乘。源代码如下：

```
#include<iostream>
#include"omp.h"
using namespace std;
int main()
{
    //我是注释
    int i, n, f, te;
    #define add(a,b) a+b
    #define mul(a,b) a*b
    #define concat(a,b) a## b
    #define mkstr(x) #x
    cin >> n;
    i = 2;
    f = 1;
    te = 88;

    #pragma omp parallel
    while (i <= n)
    {
        f = mul(f, i);
        i = add(i, 1);
    }
```

```

}
cout << mkstr(te) << endl;
cout << concat(t,e ) << endl;
cout << f << endl;
system("pause");
return 0;
}

```

结果如下：

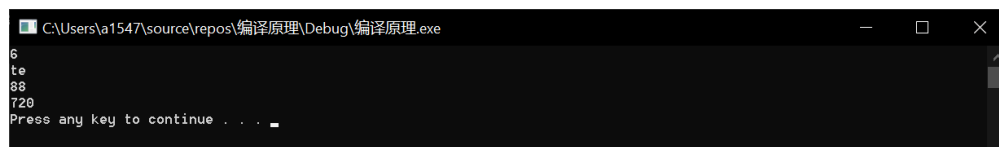


Figure 25: 运行结果

## 6 总结

一个C++程序在最后到我们能够运行之间主要经过了预编译器、汇编器、链接器/加载器的处理，在处理程序里又有更加详细的步骤，如汇编器的六个步骤等，并且在处理源程序时有着不同的优化，让我们的程序能够以更快的速度，更稳定的运行，给开发者以最好的编程体验，给用户以最好的感受。

## References

AHO A V, S.LAM M, SETHI R, 等. 编译原理（第二版）[J].

FLORIAN. 汇编器构造[J/OL]. <https://www.cnblogs.com/fanzhidongyzby/p/5812140.html>.

PHYLIPS@BMV. 链接器和加载器原理[J/OL]. <http://duanple.blog.163.com/blog/static/7097176720094874045208/>.

菜鸟教程. C++ 预处理器[J/OL]. <https://www.runoob.com/cplusplus/cpp-preprocessor.html>.