

“初识 VC 编译器” 调研报告

姓名：皮春莹
学号：1711436
专业：计算机科学与技术专业

摘要

本文以 VC 编译器作为研究对象，探究语言处理系统的完整工作过程。以一个简单的 C++ 阶乘源程序为例，利用编译器的程序选项获得各阶段的输出，研究它们与源程序的关系。

在语言处理的过程中，预处理器（`cpp`）根据以字符`#`开头的命令，修改原始的 C++ 程序，将文本文件`.cpp` 翻译成文本文件`.i`；编译器（`cc1`）将文本文件`.i` 翻译成文本文件汇编语言程序`.asm`；汇编器（`as`）将`.asm` 文件翻译成机器语言指令，把这些指令打包成一种可重定位目标程序的格式，并将结果保存在二进制目标文件`.obj` 中；链接器（`ld`）负责处理合并目标代码，生成一个可执行目标文件（`.exe`），可以被加载到内存中，由系统执行。

另外，本文通过细微修改程序，以及调整编译器的程序选项，观察各阶段输出的变化，加深对编译器的理解。

关键字：VC 编译器 语言处理系统 预处理器 汇编器 链接器

目录

引言	3
编译器的发展简要	3
语言处理系统的过程简要	3
阶乘程序编译实例	4
源程序	4
预处理器	4
编译器	7
汇编器	9
链接器	10
拓展部分	11
程序微调	11
修改编译参数	12
结论	14
参考文献	14

引言

最初,计算机通过编写机器代码控制执行运算。这些代码在普通人看来晦涩难懂。由于缺乏理论指导,把可理解的符号语言翻译为机器语言一直难以实现。随着乔姆斯基文法定义的产生以及 John Backus 在语法描述语言 BNF 中所做的工作和公理化语义学的发展,编译技术在不断进步。

编译器是一种计算机程序,它可以将便于人编写、阅读、维护的高级计算机语言所写作的源代码程序,翻译为计算机能解读、运行的机器语言程序。现代编译器主要工作流程是:源代码→预处理器→编译器→汇编程序→目标代码→链接器→可执行文件。本文以 VC 编译器作为研究对象,探究语言处理的整个过程。

下面各章中,首先简要介绍了编译器的发展历史以及语言处理系统的工作过程。接着以一个简单的 C++阶乘源程序为例,利用编译器的程序选项,获得各阶段的输出,研究它们与源程序的关系。

编译器的发展简要^[1]

在 20 世纪 40 年代,即早期冯诺依曼计算机时期,程序都是以机器语言编写,编写程序十分枯燥。后来汇编语言代替机器语言,但汇编语言仍有许多缺点,阅读理解起来很难,而且必须依赖于特定的机器。

在 20 世纪 50 年代,IBM 的 John Backus 带领一个研究小组对 FORTRAN 高级语言及其编译器进行开发;与此同时, Noam Chomsky 开始了对自然语言结构的研究;对有限状态自动机和正则表达式的研究,与 Chomsky 的研究几乎同时开始,并且引出了表示程序设计语言的单词的符号方式;接着,人们又深化了生成有效目标代码的方法,得到了最初的编译器^[6]。

在 20 世纪 70 年代,编译程序的自动生成工具初现端倪, Steve Johnson 为 Unix 系统编写了 Yacc, Mike Lesk 为 Unix 系统开发了 Lex。

在 20 世纪 90 年代,, 作为 GNU 项目或其它开放源代码项目的一部分,许多免费编译器和编译器开发工具被开发出来。

经过不断发展现代编译技术已经较为成熟,多种高级语言发展迅速都离不开编译技术的进步。

语言处理系统的过程简要

C++语言编译过程主要有四个阶段^[2]:

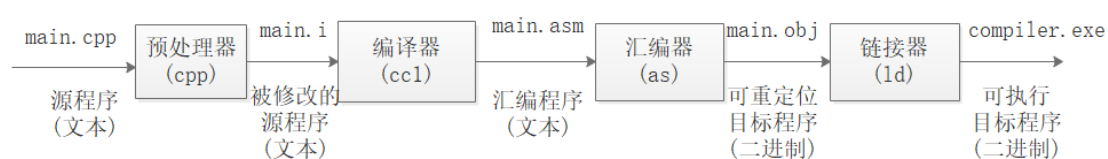


图 1 编译器的工作阶段

注:在本文的阶乘程序的例子中,源代码的文件名为 main.cpp,项目名称为 compiler。

-
- 1) 预处理阶段。预处理器（`cpp`）根据以字符`#`开头的命令，修改原始的 C++ 程序，将文本文件 `main.cpp` 翻译成文本文件 `main.i`。
 - 2) 编译阶段。编译器（`cc1`）将文本文件 `.i` 翻译成文本文件 `main.asm`，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切的描述了一条低级机器语言指令。
 - 3) 汇编阶段。汇编器（`as`）将 `main.asm` 翻译成机器语言指令，把这些指令打包成一种可重定位目标程序的格式，并将结果保存在目标文件 `main.obj` 中。`main.obj` 文件是一个二进制文件，它的字节编码是机器语言指令。
 - 4) 链接阶段。链接器（`ld`）负责处理合并目标代码，生成一个可执行目标文件（`compiler.exe`），可以被加载到内存中，由系统执行^[4]。

阶乘程序编译实例

源程序

```
#include<iostream>
using namespace std;
int main()
{
    int i, n, f;

    cin >> n;
    i = 2;
    f = 1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    cout << f << endl;
    return 0;
}
```

预处理器

预处理器在编译之前执行，预处理器拥有源文件翻译能力，预处理的结果是将被传递给实际编译器的单个文件。预处理器主要进行以下过程^[3]：

1. 以实现定义方式，将源文件的各个单独字节，映射为基本源字符集的字符。任何无法被映射到基本源字符集中的字符的源文件字符，均被替换为其通用字符名，（用 `\u` 或 `\U` 转义），或某种被等价处理的由实现定义的形式。将各个三字符序列替换为其对应的单字符表示。将宏定义展开。

2. 凡在反斜杠出现于行尾（其后紧跟换行符）时，删除反斜杠和换行符，把两个物理源码行组合成一个逻辑源码行。若有非空源文件不以换行符结束（无论是原本就无换行，还是以反斜杠结束），则其行为未定义（C++11 前）添加一个终止的换行符（C++11 起）。
3. 将源文件分解为注释，空白字符（空格、水平制表、换行、垂直制表和换页）的序列，和下列各种预处理记号：
 - a) 头文件名，如 `<iostream>` 或 `"myfile.h"`
 - b) 标识符
 - c) 预处理数字
 - d) 字符与字符串字面量，包含用户定义的（C++11 起）
 - e) 运算符与标点（包括代用记号），如 `+`、`<=<`、`<%`、`##` 或 `and`
 - f) 不属于任何其他类别的单独非空白字符以一个空格字符替换每段注释。
4. `#include` 指令所引入的每个文件都经历阶段 1 到 3 的处理，递归执行。此阶段结束时，从源码移除所有预处理器指令。

在 vs2017 中查看 C/C++ 预处理后的文件，步骤如下：

1. 右键工程(我的工程名为 `compiler`)，选择【属性】，在弹出的对话框中，选择【配置属性】-->【C/C++】-->【预处理器】，将【预处理到文件】该为【是】，应用，确认。
2. 在 VS 的菜单栏中点击【生成】-->【生成 `compiler`】。
3. 右键【`compiler`】--->【在文件资源管理器中打开文件夹】---> 在 `Debug` 目录下有个 `main.i` 文件，就是预处理后的文件。
4. 如果最后要调试运行，需要把上述的改动再改回去，不然生成不了 `obj` 文件。

文件开头：

```
1 #line 1 "e:\\vs_code\\compiler\\compiler\\main.cpp"
2 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\iostream"
3
4 #pragma once
5
6
7
8 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\istream"
9
10 #pragma once
11
12
13
14 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\ostream"
15
16 #pragma once
17
18
19
20 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\ios"
21
22 #pragma once
23
24
25
26 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\xlocnum"
27
28 #pragma once
```

图 2 main.i 文件开头

文件结尾：

```

51723 #pragma warning(pop)
51724 #pragma pack(pop)
51725 #line 51 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\iostream"
51726 #line 52 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\iostream"
51727
51728
51729
51730
51731
51732 #line 2 "e:\\vs_code\\compiler\\compiler\\main.cpp"
51733 using namespace std;
51734 int main()
51735 {
51736     int i, n, f;
51737
51738     cin >> n;
51739     i = 2;
51740     f = 1;
51741     while (i <= n)
51742     {
51743         f = f * i;
51744         i = i + 1;
51745     }
51746     cout << f << endl;
51747     return 0;
51748 }
51749

```

图 3 main.i 文件结尾

main.cpp 中第一行的#include<iostream>命令告诉预处理器读取系统头文件iostream 的内容，并把它插入程序文本中，而 iostream 实际上又引用了 istream，istream 引用了 ostream，ostream 引用了 ios.....因此这是一个递归的过程。为了证明预处理器插入的结果，我们可以在 main.i 中找到与头文件相同的代码段。以被引用的文件 utility 为例，图 5 左侧为 utility 部分代码，右侧为 main.i 的部分代码，我们可以看到 utility 的代码被插入到了程序文本中。

```

28259
28260 #line 1 "c:\\program files (x86)\\microsoft visual studio\\2017\\community\\vc\\tools\\msvc\\14.12.25827\\include\\utility"
28261
28262 #pragma once

```

图 4 main.i 中引用 utility 的语句

<pre> 106 107 108 template<class _Uty1 = _Tyl, 109 class _Uty2 = _Ty2, 110 enable_if_t<conjunction_v< 111 is_copy_constructible<_Uty1>, 112 is_copy_constructible<_Uty2>, 113 is_convertible<const _Uty1&, _Uty1>, 114 is_convertible<const _Uty2&, _Uty2> 115 >, int> = 0> 116 constexpr pair(const _Uty1& _Val1, const _Uty2& 117 : first(_Val1), second(_Val2) 118 { // construct from specified values 119 } 120 121 template<class _Uty1 = _Tyl, 122 class _Uty2 = _Ty2, 123 enable_if_t<conjunction_v< 124 is_copy_constructible<_Uty1>, 125 is_copy_constructible<_Uty2>, 126 negation<conjunction< 127 is_convertible<const _Uty1&, _Uty1>, 128 is_convertible<const _Uty2&, _Uty2>>> 129 >, int> = 0> 130 constexpr explicit pair(const _Uty1& _Val1, co 131 : first(_Val1), second(_Val2) 132 { // construct from specified values 133 } </pre>	<pre> 29809 29810 template<class...> 29811 class tuple; 29812 29813 template<class _Tyl, 29814 class _Ty2> 29815 struct pair 29816 { 29817 using first_type = _Tyl; 29818 using second_type = _Ty2; 29819 29820 template<class _Uty1 = _Tyl, 29821 class _Uty2 = _Ty2, 29822 enable_if_t<conjunction_v< 29823 is_default_constructible<_Uty1>, 29824 is_default_constructible<_Uty2> 29825 >, int> = 0> 29826 constexpr pair() 29827 : first(), second() 29828 { 29829 } 29830 29831 template<class _Uty1 = _Tyl, 29832 class _Uty2 = _Ty2, 29833 enable_if_t<conjunction_v< 29834 is_copy_constructible<_Uty1>, 29835 is_copy_constructible<_Uty2>, 29836 is_convertible<const _Uty1&, _Uty1>, 29837 is_convertible<const _Uty2&, _Uty2> </pre>
---	---

图 5 utility 与 main.i 的对比

编译器

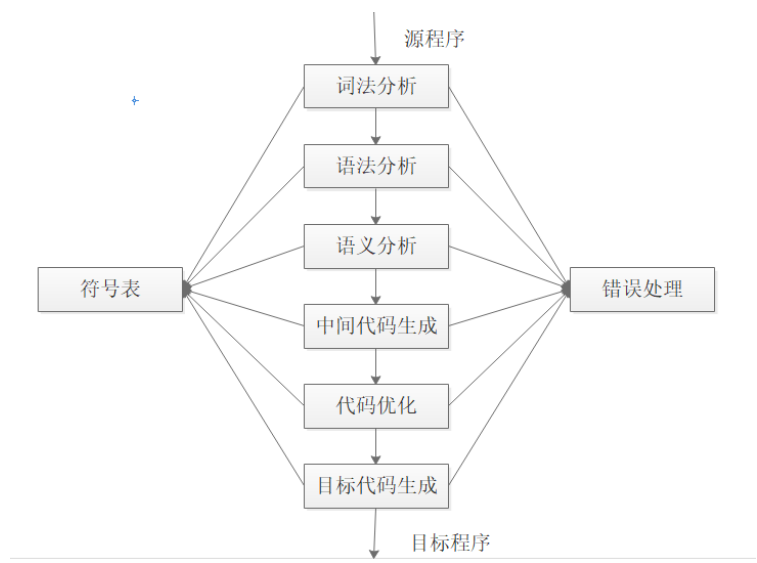


图 6 编译器的多个阶段

编译器会将上文中提到的 main.i 进一步翻译成保存汇编语言的文本，后缀为.asm。

查看 C/C++程序编译后的汇编代码，步骤如下：打开项目的属性页，选择“配置属性” → “C/C++” → “命令行”，在右下方“其它选项”中填写“/FAs”，然后重新生成工程，在工程目录中就会有对应的汇编代码文件。

用 Understand 软件查看汇编源文件，接近三百行，部分截图如下：
文件开头：

```
1 ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.12.25835.0
2
3
4     TITLE      E:\vs_code\compiler\compiler\main.cpp
5     .686P
6     .XMM
7     include listing.inc
8     .model flat
9
10    INCLUDELIB MSVCRTD
11    INCLUDELIB OLDNAMES
12
13    PUBLIC  ?_empty_global_delete@@YAXPAX@Z      ; __empty_global_delete
14    PUBLIC  ?_empty_global_delete@@YAXPAXI@Z      ; __empty_global_delete
15    PUBLIC  _main
16    PUBLIC  ??sendl@DU?$char_traits@D@std@@@std@@YAAAV?$basic_ostream@DU?$char_traits@D@std@@@0@AAV10@@@Z ; sta
17    EXTRN  __imp__widen@@?$basic_ios@DU?$char_traits@D@std@@@std@@@QBEDD@Z:PROC
18    EXTRN  __imp__?6?$basic_ostream@DU?$char_traits@D@std@@@std@@@QAEAAV01@P6AAAV01@AAV01@@@Z@Z:PROC
19    EXTRN  __imp__?6?$basic_ostream@DU?$char_traits@D@std@@@std@@@QAEAAV01@H@Z:PROC
20    EXTRN  __imp__put@@?$basic_ostream@DU?$char_traits@D@std@@@std@@@QAEAAV12@D@Z:PROC
21    EXTRN  __imp__flush@@?$basic_ostream@DU?$char_traits@D@std@@@std@@@QAEAAV12@XZ:PROC
22    EXTRN  __imp__?5?$basic_istream@DU?$char_traits@D@std@@@std@@@QAEAAV01@AAH@Z:PROC
23    EXTRN  @ _RTC_CheckStackVars@8:PROC
24    EXTRN  @_security_check_cookie@4:PROC
25    EXTRN  __RTC_InitBase:PROC
26    EXTRN  __RTC_Shutdown:PROC
27    EXTRN  __imp__?cin@std@@@3V?$basic_istream@DU?$char_traits@D@std@@@1@A:BYTE
28    EXTRN  __imp__?cout@std@@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A:BYTE
29    EXTRN  __security_cookie:DWORD
30    ; COMDAT rtc$TMZ
31    rtc$TMZ SEGMENT
```

图 7 utility 与 main

图 8 main.asm 开头部分

主要部分：


```

111  _main  PROC                                     ; COMDAT
112
113  ; 4      : {
114
115      push    ebp
116      mov     ebp, esp
117      sub     esp, 232                          ; 000000e8H
118      push    ebx
119      push    esi
120      push    edi
121      lea     edi, DWORD PTR [ebp-232]
122      mov     ecx, 58                          ; 0000003aH
123      mov     eax, -858993460                  ; ccccccccH
124      rep     stosd
125      mov     eax, DWORD PTR ___security_cookie
126      xor     eax, ebp
127      mov     DWORD PTR __$ArrayPad$[ebp], eax
128
129  ; 5      :     int i, n, f;
130  ; 6      :
131  ; 7      :     cin >> n;
132
133      mov     esi, esp
134      lea     eax, DWORD PTR _n$[ebp]
135      push    eax
136      mov     ecx, DWORD PTR __imp_?cin@std@@3V?$basic_istream@DU?$char_traits@@@
137      call    DWORD PTR __imp_??5?$basic_istream@DU?$char_traits@@@
138      cmp     esi, esp
139      call    __RTC_CheckEsp

```

图 9 main.asm 主要部分(1)

```

141  ; 8      :     i = 2;
142
143      mov     DWORD PTR _i$[ebp], 2
144
145  ; 9      :     f = 1;
146
147      mov     DWORD PTR _f$[ebp], 1
148  $LN2@main:
149
150  ; 10     :     while (i <= n)
151
152      mov     eax, DWORD PTR _i$[ebp]
153      cmp     eax, DWORD PTR _n$[ebp]
154      jg      SHORT $LN3@main
155
156  ; 11     :     {
157  ; 12     :         f = f * i;
158
159      mov     eax, DWORD PTR _f$[ebp]
160      imul    eax, DWORD PTR _i$[ebp]
161      mov     DWORD PTR _f$[ebp], eax
162
163  ; 13     :         i = i + 1;
164
165      mov     eax, DWORD PTR _i$[ebp]
166      add     eax, 1
167      mov     DWORD PTR _i$[ebp], eax
168
169  ; 14     :     }
170
171      jmp     SHORT $LN2@main
172  $LN3@main:
173
174  ; 15     :     cout << f << endl;

```

图 10 main.asm 主要部分(2)

文件结尾:

```

249     pop ebp
250     ret 0
251 ?__empty_global_delete@@YAXPAXI@Z ENDP          ; __empty_global_delete
252 _TEXT ENDS
253 ; Function compile flags: /Odtp /RTCsu /ZI
254 ; File e:\vs_code\compiler\compiler\main.cpp
255 ; COMDAT ?__empty_global_delete@@YAXPAX@Z
256 _TEXT SEGMENT
257     __formal$ = 8 ; size = 4
258 ?__empty_global_delete@@YAXPAX@Z PROC          ; __empty_global_delete, COMDAT
259
260     push ebp
261     mov ebp, esp
262     sub esp, 192 ; 000000c0H
263     push ebx
264     push esi
265     push edi
266     lea edi, DWORD PTR [ebp-192]
267     mov ecx, 48 ; 00000030H
268     mov eax, -858993460 ; ccccccccH
269     rep stosd
270     pop edi
271     pop esi
272     pop ebx
273     mov esp, ebp
274     pop ebp
275     ret 0
276 ?__empty_global_delete@@YAXPAX@Z ENDP          ; __empty_global_delete
277 _TEXT ENDS
278 END
279

```

图 11 main.asm 结尾部分

汇编器

汇编器会将上述的 main.asm 文件翻译成机器语言指令，并把这些指令打包可重定位目标程序的格式，将结果保存在新生成的二进制文件 main.obj 中。

main.obj 的部分截图如下：

```

00000040h: 75 67 24 53 00 00 00 00 00 00 00 00 64 63 00 00 ;
00000050h: 4B 03 00 00 AF 66 00 00 00 00 00 00 04 00 00 00 ;
00000060h: 40 00 10 42 2E 64 65 62 75 67 24 54 00 00 00 00 ;
00000070h: 00 00 00 00 4C 00 00 00 D7 66 00 00 00 00 00 00 ;
00000080h: 00 00 00 00 00 00 00 00 40 00 10 42 2E 74 65 78 ;
00000090h: 74 24 6D 6E 00 00 00 00 00 00 00 00 7B 00 00 00 ;
000000a0h: 23 67 00 00 9E 67 00 00 00 00 00 00 07 00 00 00 ;
000000b0h: 20 10 50 60 2E 64 65 62 75 67 24 53 00 00 00 00 ;
000000c0h: 00 00 00 00 34 01 00 00 E4 67 00 00 18 69 00 00 ;
000000d0h: 00 00 00 00 0B 00 00 00 40 10 10 42 2E 74 65 78 ;
000000e0h: 74 24 6D 6E 00 00 00 00 00 00 00 00 25 00 00 00 ;
000000f0h: 86 69 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000100h: 20 10 50 60 2E 64 65 62 75 67 24 53 00 00 00 00 ;
00000110h: 00 00 00 00 D4 00 00 00 AB 69 00 00 7F 6A 00 00 ;
00000120h: 00 00 00 00 05 00 00 00 40 10 10 42 2E 74 65 78 ;
00000130h: 74 24 6D 6E 00 00 00 00 00 00 00 00 25 00 00 00 ;
00000140h: B1 6A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000150h: 20 10 50 60 2E 64 65 62 75 67 24 53 00 00 00 00 ;
00000160h: 00 00 00 00 EC 00 00 00 D6 6A 00 00 C2 6B 00 00 ;
00000170h: 00 00 00 00 05 00 00 00 40 10 10 42 2E 74 65 78 ;
00000180h: 74 24 6D 6E 00 00 00 00 00 00 00 00 E2 00 00 00 ;
00000190h: F4 6B 00 00 D6 6C 00 00 00 00 00 00 10 00 00 00 ;
000001a0h: 20 10 50 60 2E 64 65 62 75 67 24 53 00 00 00 00 ;
000001b0h: 00 00 00 00 94 01 00 00 76 6D 00 00 0A 6F 00 00 ;
000001c0h: 00 00 00 00 11 00 00 00 40 10 10 42 2E 72 74 63 ;
000001d0h: 24 49 4D 5A 00 00 00 00 00 00 00 00 04 00 00 00 ;

```

图 12 main.obj 的部分截图(1)

```

000011f0h: 6D 65 72 69 63 5F 6C 69 6D 69 74 73 3C 73 69 67 ; meric_limits<sig
00001200h: 6E 65 64 20 63 68 61 72 3E 3A 3A 69 73 5F 73 69 ; ned char>::is_si
00001210h: 67 6E 65 64 00 31 00 07 11 5E 10 00 00 07 00 73 ; gned.1...^.....s
00001220h: 74 64 3A 3A 6E 75 6D 65 72 69 63 5F 6C 69 6D 69 ; td::numeric limi
00001230h: 74 73 3C 73 69 67 6E 65 64 20 63 68 61 72 3E 3A ; ts<signed char>:
00001240h: 3A 64 69 67 69 74 73 00 33 00 07 11 5E 10 00 00 ; :digits.3...^...
00001250h: 02 00 73 74 64 3A 3A 6E 75 6D 65 72 69 63 5F 6C ; ..std::numeric_l
00001260h: 69 6D 69 74 73 3C 73 69 67 6E 65 64 20 63 68 61 ; imits<signed cha
00001270h: 72 3E 3A 3A 64 69 67 69 74 73 31 30 00 36 00 07 ; r>::digits10.6..
00001280h: 11 AD 11 00 00 00 00 73 74 64 3A 3A 6E 75 6D 65 ; .?....std::nume
00001290h: 72 69 63 5F 6C 69 6D 69 74 73 3C 75 6E 73 69 67 ; ric_limits<unsig
000012a0h: 6E 65 64 20 63 68 61 72 3E 3A 3A 69 73 5F 73 69 ; ned char>::is_si
000012b0h: 67 6E 65 64 00 33 00 07 11 5E 10 00 00 08 00 73 ; gned.3...^.....s
000012c0h: 74 64 3A 3A 6E 75 6D 65 72 69 63 5F 6C 69 6D 69 ; td::numeric limi
000012d0h: 74 73 3C 75 6E 73 69 67 6E 65 64 20 63 68 61 72 ; ts<unsigned char
000012e0h: 3E 3A 3A 64 69 67 69 74 73 00 35 00 07 11 5E 10 ; >::digits.5...^
000012f0h: 00 00 02 00 73 74 64 3A 3A 6E 75 6D 65 72 69 63 ; ....std::numeric
00001300h: 5F 6C 69 6D 69 74 73 3C 75 6E 73 69 67 6E 65 64 ; _limits<unsigned
00001310h: 20 63 68 61 72 3E 3A 3A 64 69 67 69 74 73 31 30 ; char>::digits10
00001320h: 00 2E 00 07 11 AD 11 00 00 01 00 73 74 64 3A 3A ; .....?....std::
00001330h: 6E 75 6D 65 72 69 63 5F 6C 69 6D 69 74 73 3C 73 ; numeric_limits<s
00001340h: 68 6F 72 74 3E 3A 3A 69 73 5F 73 69 67 6E 65 64 ; hort>::is_signed
00001350h: 00 2B 00 07 11 5E 10 00 00 0F 00 73 74 64 3A 3A ; .+...^.....std::
00001360h: 6E 75 6D 65 72 69 63 5F 6C 69 6D 69 74 73 3C 73 ; numeric_limits<s
00001370h: 68 6F 72 74 3E 3A 3A 64 69 67 69 74 73 00 2D 00 ; hort>::digits.-.

```

图 13 main.obj 的部分截图(2)

链接器

链接器是一个独立程序，将一个或多个库或目标文件（先前由编译器或汇编器生成）链接到一块生成可执行程序。链接过程的主要内容是把各个模块之间相互引用的部分都处理好，使得各个模块之间可以正常的拼接，包括以下三个部分：

1. 地址和空间的分配;
2. 符号决议;
3. 重定位。

链接分为静态链接和动态链接, 根据百度百科的介绍^[5], 静态链接是由链接器在链接时将库的内容加入到可执行程序中的做法。动态链接所调用的函数代码并没有被拷贝到应用程序的可执行文件中去, 而是仅仅在其中加入了所调用函数的描述信息 (往往是一些重定位信息)。仅当应用程序被装入内存开始运行时, 在 Windows 的管理下, 才在应用程序与相应的 DLL 之间建立链接关系。当要执行所调用 DLL 中的函数时, 根据链接产生的重定位信息, Windows 才转去执行 DLL 中相应的函数代码。

对于本文中具体的例子, 通过链接器, 生成 `compiler.exe` 可执行文件。双击该文件, 会出现计算阶乘的界面:

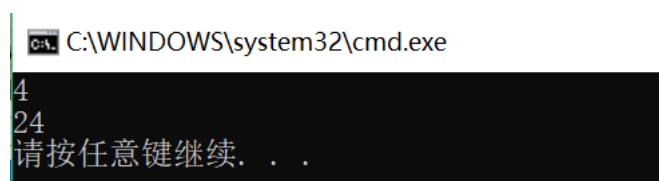


图 14 compiler.exe

拓展部分

程序微调

1. 将整数 `i` 和 `f` 的定义和初始化写在头文件 `fac.h` 中, 在 `main.cpp` 中引用 `fac.h`;
 2. 增加注释 “//输入一个整数 `n`, 输出 `n` 的阶乘”;
 3. 在 `main.cpp` 中进行宏定义, 令 `x=3`, 并在 `main()` 函数中输出 `x` 的值。
- 修改后的程序如图 11, 图 12 所示:

```
1 #pragma once
2 #include<iostream>
3 int i=2, f=1;
```

图 15 fac.h

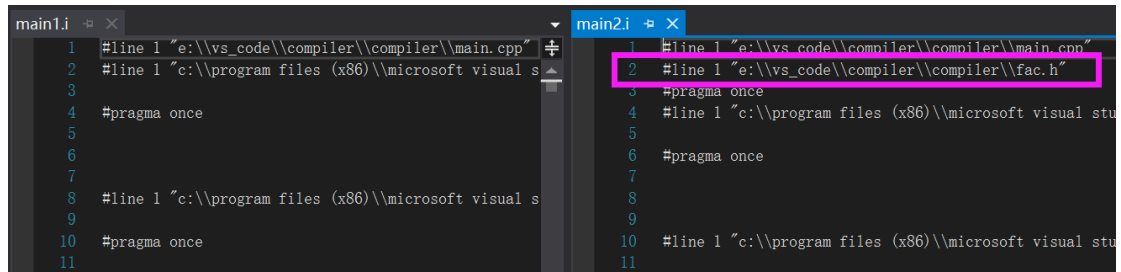
```
1 #include"fac.h"
2 using namespace std;
3 #define x 3
4 //输入一个整数n, 输出n的阶乘
5 int main() {
6     cout << "The value of x is" << x << endl;
7     int n;
8     cin >> n;
9     while (i <= n) {
10         f = f * i;
11         i = i + 1;
12     }
13     cout << f << endl;
14     return 0;
15 }
```

图 16 mian.cpp

另外, 在上完第二次编译原理的课程之后, 我学习到在头文件中进行变量声

明是不合适的，因为可能在文本替换中发生重复声明的错误。但是在进行这一步小测试时，我为了直观地体现文本替换的效果，暂时忽略了这一点。

重复上面写的预编译过程，得到 main.i。为了方便前后对比，将未改动代码之前的预编译文件命名为 main1.i，经过代码改动后得到的预编译文件命名为 main2.i，具体的区别如图 13 和图 14 所示：（左侧为 main1.i，右侧为 main2.i）文件开头：

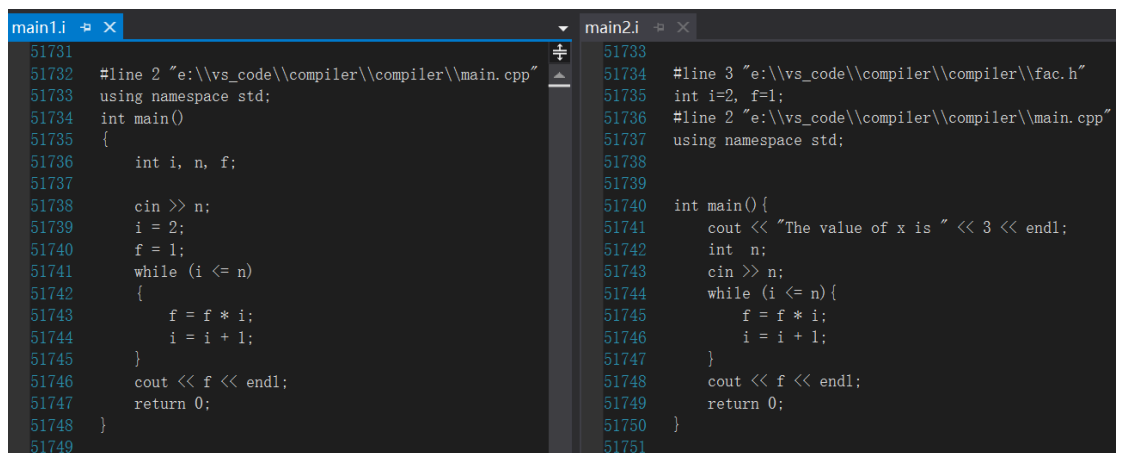


```
main1.i  main2.i
1 #line 1 "e:\\vs_code\\compiler\\compiler\\main.cpp"
2 #line 1 "c:\\program files (x86)\\microsoft visual s
3
4 #pragma once
5
6
7
8 #line 1 "c:\\program files (x86)\\microsoft visual s
9
10 #pragma once
11

1 #line 1 "e:\\vs_code\\compiler\\compiler\\main.cpp"
2 #line 1 "e:\\vs_code\\compiler\\compiler\\fac.h"
3 #pragma once
4 #line 1 "c:\\program files (x86)\\microsoft visual stu
5
6 #pragma once
7
8
9
10 #line 1 "c:\\program files (x86)\\microsoft visual stu
11
```

图 17 main.i 开头

文件结尾：



```
main1.i  main2.i
51731
51732 #line 2 "e:\\vs_code\\compiler\\compiler\\main.cpp"
51733 using namespace std;
51734 int main()
51735 {
51736     int i, n, f;
51737
51738     cin >> n;
51739     i = 2;
51740     f = 1;
51741     while (i <= n)
51742     {
51743         f = f * i;
51744         i = i + 1;
51745     }
51746     cout << f << endl;
51747     return 0;
51748 }
51749

51733 #line 3 "e:\\vs_code\\compiler\\compiler\\fac.h"
51734 int i=2, f=1;
51735 #line 2 "e:\\vs_code\\compiler\\compiler\\main.cpp"
51736 using namespace std;
51737
51738 int main(){
51739     cout << "The value of x is " << 3 << endl;
51740     int n;
51741     cin >> n;
51742     while (i <= n){
51743         f = f * i;
51744         i = i + 1;
51745     }
51746     cout << f << endl;
51747     return 0;
51748 }
51751
```

图 18 main.i 结尾

对比代码改动前后生成的.i 文件，我们可以更具体地看出编译器做的工作。在预处理的过程中，fac.h 的内容被替换写入.i 文件中；注释也被删除；另外，可以看到预处理对 x 进行了宏替换。

修改编译参数

这一部分我们在修改后的阶乘代码的基础上，对编译器的程序选项进行一些修改，观察输出的变化。编译参数的设置主要是在【属性】→【配置属性】→【C/C++】里进行。

对程序进行调试，通过编译器生成 main2.obj。

选中“优化”选项，在右侧对优化选项进行修改。默认的优化选项是“/Od”，这是为了简化调试的选项。

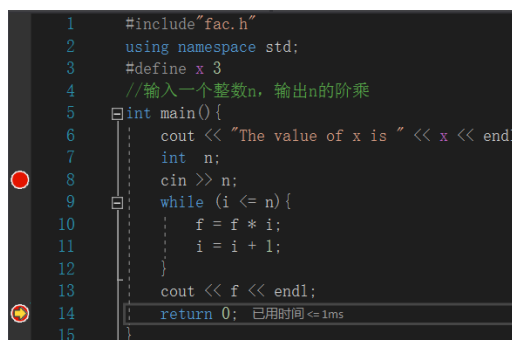


图 19 程序调试

优化	已禁用 (/Od)
内联函数扩展	默认值
启用内部函数	否
优选大小或速度	均不
省略帧指针	否 (/Oy-)
启用线程安全优化	否
全程序优化	否

图 20 优化选项

根据微软的文档解释^[7]，/O 选项控制有助于创建具有最高速度或最小大小的代码的各种优化，具体包括以下参数选项：

- ♦ /O1、/O2（最小化大小、最大化速度） 为获得最小大小而优化代码。
- ♦ /O1、/O2（最小化大小、最大化速度） 为获得最高速度而优化代码。
- ♦ /Ob（内联函数展开） 控制内联函数扩展。
- ♦ /Od（禁用（调试）） 禁用优化，从而加快编译并简化调试。
- ♦ /Og（全局优化） 启用全局优化。
- ♦ /Oi（生成内部函数） 为适当的函数调用生成内部函数。
- ♦ /Os、/Ot（代码大小优先、代码速度优先） 通知编译器优选大小优化而非速度优化。
- ♦ /Os、/Ot（代码大小优先、代码速度优先）（默认设置）通知编译器优选速度优化而非大小优化。
- ♦ /Ox（完全优化） 选择完全优化。
- ♦ /Oy（框架指针省略） 取消在调用堆栈上创建框架指针，以更快地进行函数调用。

修改参数时要注意命令行选项是否兼容的问题，我最初将优化选项改为/O1时，编译报错：“/O1”和“/RTC1”命令行选项不兼容。通过在 Stack Overflow 上查找^[8]，发现原因是/RTC 不能和编译器优化选项同时使用，因此在“代码生成”的目录下，将“基本运行时检查”设置为默认，即可成功编译。

启用字符串池	
启用最小重新生成	是 (/Gm)
启用 C++ 异常	是 (/EHsc)
较小类型检查	否
基本运行时检查	两者 (/RTC1, 等同于 /RTCsu) (/RTC1)
运行库	多线程调试 DLL (/MDd)
结构成员对齐	默认设置
安全检查	启用安全检查 (/GS)
控制流防护	
启用函数级链接	
启用并行代码生成	
启用增强指令集	未设置
浮点模型	精度 (/fp:precise)
启用浮点异常	
创建可热修补映像	

图 21 代码生成的设置

为了对比修改优化选项前后的差别，将“/Od”对应生成的 obj 文件命名为

main2.obj, 将“/O1”对应生成的 obj 文件命名为 main3.obj, 我们可以直观的从文件大小上看出优化前后的差别: main2.obj 大小为 46.2kb, main3.obj 大小为 42.6kb, 符合我们为了获得最小大小而做出的修改。

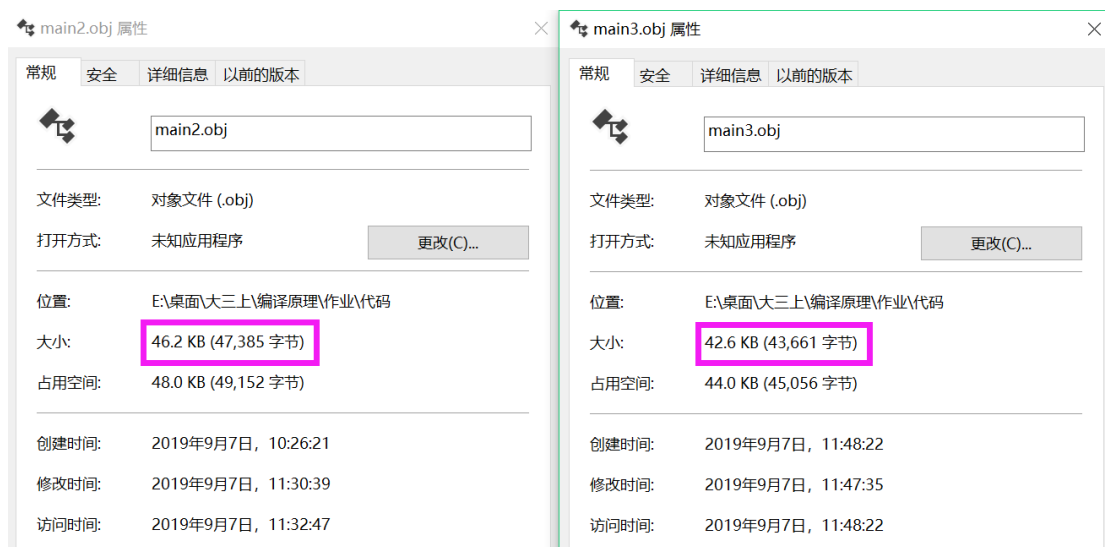


图 22 main.obj 大小对比

结论

通过对阶乘源程序的编译过程的探究, 我初步认识了预处理器、编译器、汇编器和链接器所做的工作, 在语言处理的过程中, 预处理器 (cpp) 根据以字符#开头的命令, 修改原始的 C++ 程序, 将文本文件 main.cpp 翻译成文本文件 main.i; 编译器 (cc1) 将文本文件.i 翻译成文本文件汇编语言程序 main.asm; 汇编器 (as) 将 main.asm 翻译成机器语言指令, 把这些指令打包成一种可重定位目标程序的格式, 并将结果保存在二进制目标文件 main.obj 中; 链接器 (ld) 负责处理合并目标代码, 生成一个可执行目标文件(compiler.exe), 可以被加载到内存中, 由系统执行。

通过细微修改程序, 以及调整编译器的程序选项, 观察各阶段输出的变化, 加深了对编译器的理解。

参考文献

- [1] 刘雪飞.浅谈编译原理[J].中外企业家,2018(03):35.
- [2] https://blog.csdn.net/baidu_33604078/article/details/79091049
- [3] <https://zh.cppreference.com/w/c/preprocessor>
- [4] <https://blog.csdn.net/liquanhai/article/details/51569692>
- [5] <https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E9%93%BE%E6%8E%A5/4142691?fr=aladdin>
- [6] <https://blog.csdn.net/mmpire/article/details/620714>

-
- [7] [https://docs.microsoft.com/zh-cn/previous-versions/visualstudio/visual-studio-2008/k1ack8f1\(v=vs.90\)](https://docs.microsoft.com/zh-cn/previous-versions/visualstudio/visual-studio-2008/k1ack8f1(v=vs.90))
 - [8] <https://stackoverflow.com/questions/37007939/command-line-error-d8016-o2-and-rtc1-command-line-options-are-incompatibl>