C++编译链接过程

```
#include <Stdio.h>
int main(){
    printf("hello world!\n");
}
```

```
gcc hello.c #编译
./a.out #执行
```

上述gcc命令实际执行了四步操作：

1. 预处理（Preprocessing）
2. 编译（Compilation）
3. 汇编（Assemble）
4. 链接（Linking）

示例

```
├── test.c
└── inc
    ├── mymath.h
    └── mymath.c
```

假设文件结构如上，test中使用mymath实现的函数

```c
//test.c
#include <stdio.h>
#include "mymath.h"// 自定义头文件
int main(){
    int a = 2;
    int b = 3;
    int sum = add(a, b);
    printf("a=%d, b=%d, a+b=%d\n", a, b, sum);
}
```

```c
// mymath.h
#ifndef MYMATH_H
#define MYMATH_H
int add(int a, int b);
int sum(int a, int b);
#endif
```

```c
// mymath.c
int add(int a, int b){
    return a+b;
}
int sub(int a, int b){
    return a-b;
}
```

1. 预处理（Preprocessing）

预处理用于将所有的**#include头文件以及宏定义替换成真正的内容**，预处理之后得到的仍然是文本文件，但文件体积会变大很多。gcc的预处理是**预处理器cpp**完成的，可通过如下命令对test.c进行预处理

```
$ gcc -E -I./inc test.c -o test.i
```

或者直接调用cpp命令

```
$ cpp test.c -I./inc -o test.i
```

上述命令中-E是让编译器在预处理之后就退出，不进行后续编译过程；-I指定头文件目录，这里指定的是我们自定义的头文件目录；-o指定输出文件名

2. 编译（Compilation）

这里的编译不是指程序从源文件到二进制程序的全部过程，而是指将经过预处理之后的程序转换成特定汇编代码的过程

```
$ gcc -S -I./inc test.c -o test.s
```

上述命令中-S让编译器在编译之后停止，不进行后续过程。编译过程完成后，**将生成程序的汇编代码test.s，这也是文本文件**，生成的指令如下

```
    .file   "main.cpp"
    .text
    .section    .text$_Z6printfPKcz,"x"
```

```
    .linkonce discard
    .globl  _Z6printfPKcz
    .def    _Z6printfPKcz;  .scl    2;  .type 32; .endef
    .seh_proc   _Z6printfPKcz
_Z6printfPKcz:
.LFB9:
    pushq   %rbp
    .seh_pushreg    %rbp
    pushq   %rbx
    .seh_pushreg    %rbx
    subq    $56, %rsp
    .seh_stackalloc 56
    leaq    48(%rsp), %rbp
    .seh_setframe   %rbp, 48
    .seh_endprologue
    movq    %rcx, 32(%rbp)
    movq    %rdx, 40(%rbp)
    movq    %r8, 48(%rbp)
    movq    %r9, 56(%rbp)
    leaq    40(%rbp), %rax
    movq    %rax, -16(%rbp)
    movq    -16(%rbp), %rbx
    movl    $1, %ecx
    movq    __imp___acrt_iob_func(%rip), %rax
    call    *%rax
    movq    %rax, %rcx
    movq    32(%rbp), %rax
    movq    %rbx, %r8
    movq    %rax, %rdx
```

```
    call    __mingw_vfprintf
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    addq    $56, %rsp
    popq    %rbx
    popq    %rbp
    ret
    .seh_endproc
    .def    __main; .scl    2;  .type   32;
.endef
    .section .rdata,"dr"
.LC0:
    .ascii "a=%d, b=%d, a+b=%d\12\0"
    .text
    .globl  main
    .def    main;   .scl    2;  .type   32;
.endef
    .seh_proc   main
main:
.LFB45:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $48, %rsp
    .seh_stackalloc 48
    .seh_endprologue
    call    __main
    movl    $2, -4(%rbp)
    movl    $3, -8(%rbp)
```

```
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    movl    %eax, %ecx
    call    _Z3addii
    movl    %eax, -12(%rbp)
    movl    -12(%rbp), %ecx
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    movl    %ecx, %r9d
    movl    %edx, %r8d
    movl    %eax, %edx
    leaq    .LC0(%rip), %rax
    movq    %rax, %rcx
    call    _Z6printfPKcz
    movl    $0, %eax
    addq    $48, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .ident  "GCC: (x86_64-win32-seh-rev1, Built by MinGW-W64 project) 12.2.0"
    .def    __mingw_vfprintf;   .scl    2;  .type   32; .endef
    .def    _Z3addii;   .scl    2;  .type   32; .endef
```

3. 汇编（Assemble)

汇编过程将上一步的汇编代码转换成机器码，产生的文件叫做目标文件，是二进制格式

```
as test.s -o test.o
```

等价于

```
gcc -c test.s -o test.o
```

这一步会为每一个源文件产生一个目标文件，因此mymath.c也需要产生一个mymath.o文件

4. 链接（Linking)

```
$ ld -o test.out test.o inc/mymath.o
...libraries...
```

链接的详细过程如下

合并段

在elf文件中字节对齐是以4字节对齐的，在可执行程序中是以页的方式对齐的（一个页的大小为4k），因此如果我们在链接时将各个.o文件各个段单独的加载到可执行文件中，将会非常浪费空间：

因此我们需要合并段，调整段偏移，把每个.o文件的.text段合并在一起.data段合并在一起，这样在生成的可执行文件中，各个段都只有一个，如下图，由于在链接时只需要加载代码段(.text段)和数据段（.data段和.bss段）。因此合并段之后，在系统给我们分配内存时，只需要分配两个页面大小就可以，分别存放代码和数据

调整段偏移

汇总所有符号

每个obj文件在编译时都会生成自己的符号表，我们要把这些符号都合并起来进行符号解析

完成符号的重定位

在进行合并段，调整段偏移时，输入文件的各个段在连接后的虚拟地址就已经确定了，这一步完成后，连接器开始计算各个符号的虚拟地址，因为各个符号在段内的相对位置是固定的，所以段内各个符号的地址也已经是确定的了，只不过连接器需要给每个符号加上一个偏移量，使他们能够调整到正确的虚拟地址，这就是符号的重定位过程

在 elf 文件中，有一个叫重定位表的结构专门用来保存这些与从定位有关的信息，重定位表在elf文件中往往是一个或多个段

5. 数据和指令

上面是代码编译链接的过程，得到了可执行的文件后，程序在内存中是如何运行的，数据和指令又分别是什么

所有的全局变量和静态变量都是数据，除此之外都是指令（包括**局部变量**）

虚拟地址空间

在每个程序运行的时候，我们的操作系统都会给他分配一个固定大小的虚拟地址空间（x86，32bit，Linux内核下默认大小为4G），那这段内存分配结构如下：

整个4G的空间有1G是供操作系统使用的内核空间，用户无法访问，还有3G是我们的用户空间，以供该虚拟地址空间上进程的运行，在这3G的用户空间中又被分成了很多段，从0地址开始的128M大小是系统的预留空间，用户也是无法访问的。

接下来是.text段，该段空间中存放的是代码，然后是.data段和.bss段，这两段里面存放的都是数据，但又有不同：**data段中存放的数据是已经初始化并且初始化值不为0的数据**，**而.bss段中存放的是未经初始化或者初始化为0的数据**(注：bssbetter save space(更好的节省空间))

结语

经过以上分析，我们发现编译过程并不像想象的那么简单，而是要经过预处理、编译、汇编、链接。尽管我们平时使用gcc命令的时候没有关心中间结果，但每次程序的编译都少不了这几个步骤。也不用为上述繁琐过程而烦恼，因为你仍然可以：

```
$ gcc hello.c # 编译
$ ./a.out # 执行
```