C++编译过程

编译与链接的过程可以分解为 4 个步骤，分别是预处理（Prepressing）、编译（Compilation）、汇编（Assembly）和链接（Linking），具体如图 4-1 所示。

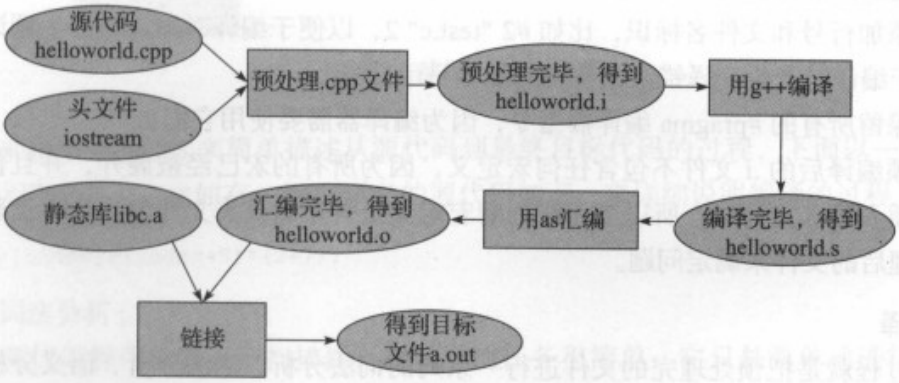


图 4-1　编译与链接的过程图

预处理

首先是源代码文件 helloworld.cpp 和相关的头文件，如 iostream 等被预处理器处理成一个 .i 文件。 第一步预处理的过程相当于如下命令（ - E 表示只进行预处理）：

```
g++ -E helloworld.cpp -o helloworld.i
```

其中：`- E` 的编译选项，意味着只执行到预编译，直接输出预编译结果 。
预处理过程主要处理那些源代码文件只能够的以“#”开始的预编译指令 。 比如#include 、#define 等，主要处理规则如下所述 。

- 将所有的**#define**删除，并且展开所有的**宏定义**，如：

  ```
  \#define a b
  ```

  对于这种伪指令，预编译所要做的是将程序中的所有 a 用 b 替换，但作为字符串常量的a 则不被替换 。 还有 #undef，则将取消对某个宏的定义，使以后该串的出现不再被替换。

- 处理所有**条件预编译指令**，比如**#if、#ifdef、#elif、#else、#endif。**
  这些伪指令的引人使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。 预编译程序将根据有关的文件，将那些不必要的代码过滤掉 。

- 处理**#include 预编译指令**，将被包含的文件插入到该预编译指令的位置。 注意： 这个过程是**递归进行**的，也就是说**被包含的文件可能还包含其他文件** 。
  过滤所有的**注释**“ // ”和“/*/ ”中的内容 。

- 添加**行号和文件名标识**，比如#2”test.c" 2 ，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号 。

- 保留所有的 **#pragma 编译器指令**，因为编译器需要使用它们 。
  经过预编译后的 .i 文件不包含任何宏定义，因为所有的宏已经被展开，并且包含的文件也已经被插入到 .i 件中 。

编译&汇编

译过程就是把预处理完的文件进行一系列的词法分析、语法分析 、 语义分析以及优化后产生相应的汇编代码文件。上面的编译过程相当于如下命令：

```
g++ -S helloworld .i -o hellowor ld.s
```

其中，-S 的编译选项，表示只执行到源代码到汇编代码的转换，输出汇编代码 。 注意，这里是用大写 S ，而不是小写 s 。 - s 表示直接生成与运用 strip 同样效果的可执行文件（删除了所有符号信息）。

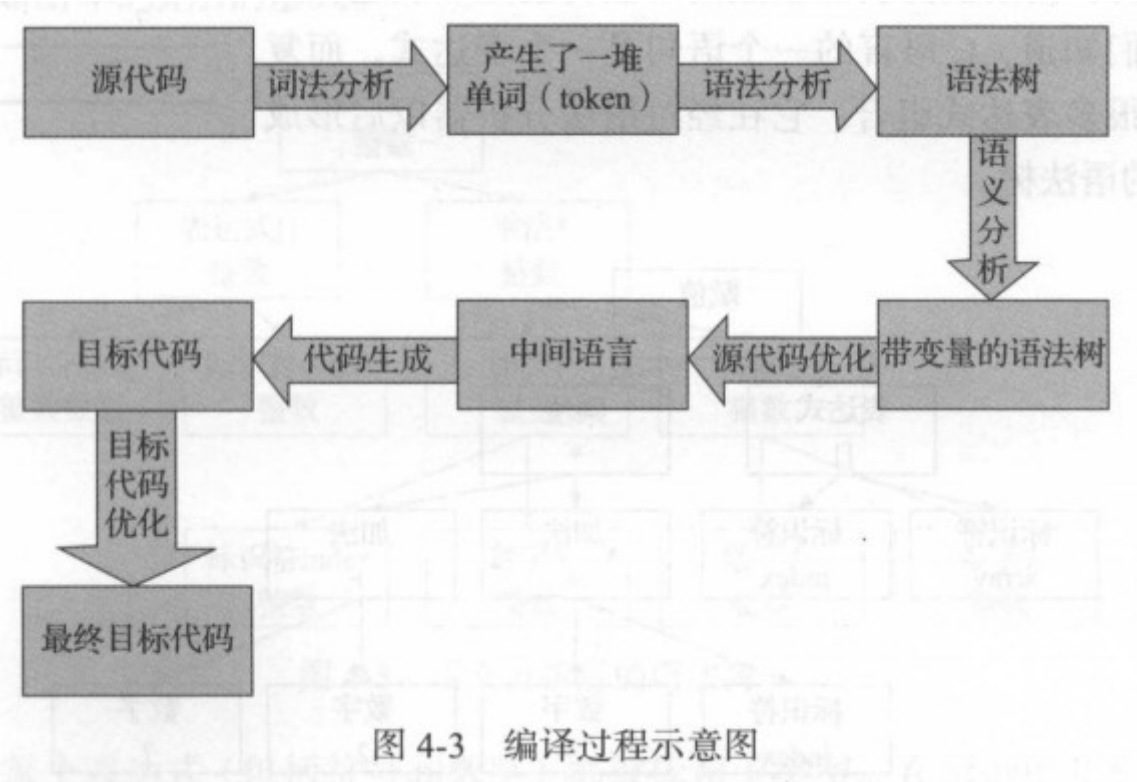


图 4-3　编译过程示意图

下面以一段很简单的C 语义的代码为例子，来详细说明编译的过程 。

```
array[index] = (index+5)*(2+7);
```

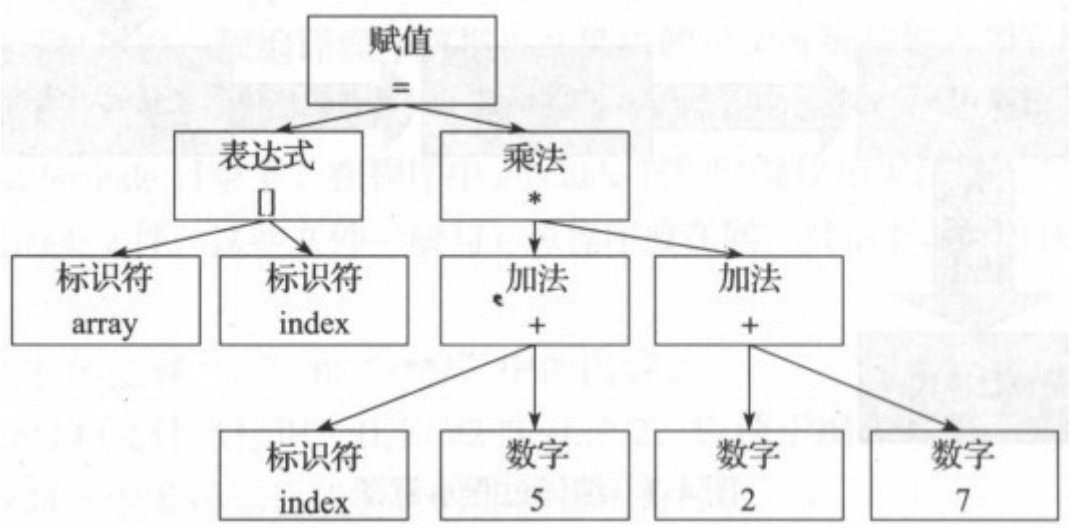
- 词法分析。
  首先源代码程序被输入到扫描器，扫描器的任务很简单，它只是简单地进行词法分析，运用一种类似于有限状态机的算法可以很轻松地将源代码的字符序列分割成一系列的记号 。词法分析产生的记号一般可分为如下几类：关键字、标识符、字面量（包含数字、字符串等）和特殊记号（如加号、等号）。

表 4-1　词法分析结果

单词（token）	类型
array	标识符
[	左方括号
index	标识符
]	右方括号
=	赋值
(	左圆括号
index	标识符
+	加号
5	数字
)	右圆括号
*	乘号
(	左圆括号
2	数字
+	加号
7	数字
)	右圆括号

词法分析结果

- 语法分析 。
  接下来语法分析器将对由扫描器产生的记号进行语法分析，从而产生**语法树**。 整个分析过程采用了**上下文无关文法**的分析手段 。 简单地讲，由语法分析器生成的语法树就是以表达式为节点的树。我们知道，C 语言的一个语句是一个表达式，而复杂的表达式由很多表达式组合。 它在经过语法分析器以后形成如图所示的语法树。



语法树

可以看到，整个语句被看作一个赋值表达式：赋值表达式的左边是一个数组表达式；它的右边是一个乘法表达式；数组表达式又由两个符号表达式组成，等等 。 符号和数字是最小的表达式，它们不是由其他的表达式来组成的 ，所以通常为整个语法树的叶节点 。在语法分析的同时，确定**运算符的优先级和含义。** 比如乘法表达式的优先级比加法高，而圆括号表达式的优先级比乘法高等等

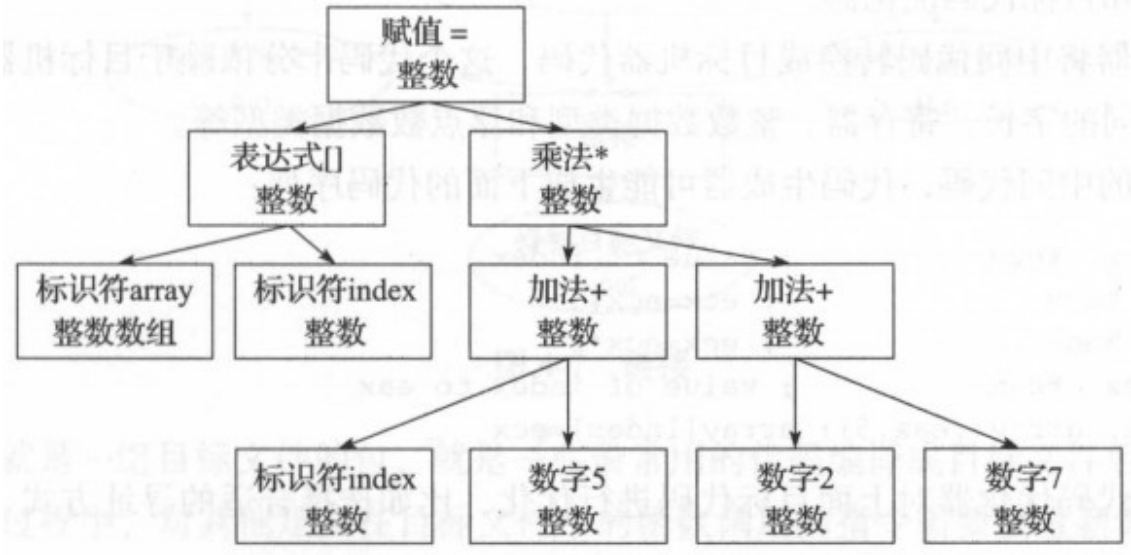
。 如果出现了表达式不合法，比如各种括号不匹配、表达式中缺少操作符等，编译器就会报告语法分析阶段的相关错误。

- 语义分析

  语义分析是由语义分析器完成。语法分析仅仅是完成了对表达式的语法层面的分析，但是它并不了解这个语句是否真正有意义。比如对 C 语言里面两个指针做乘法运算是没有意义的，但是这个语句在语法上是合法的。编译器所能分析的语义是**静态语义**。所谓静态语义是指在编译期间可以确定的语义。与之对应的动态语义就是只有在运行期间才能确定的语义。

  - 静态语义通常包括声明和类型的匹配及类型的转换等。比如当一个浮点型的表达式赋值给一个整型的表达式时，其中隐含了一个浮点型赋值给一个指针的时候，语义分析程序会发现这个类型不匹配，编译器将会报错 。
  - 动态语义一般指在运行期间出现的语义相关的问题，比如将0作为除数是一个运行期语义错误。

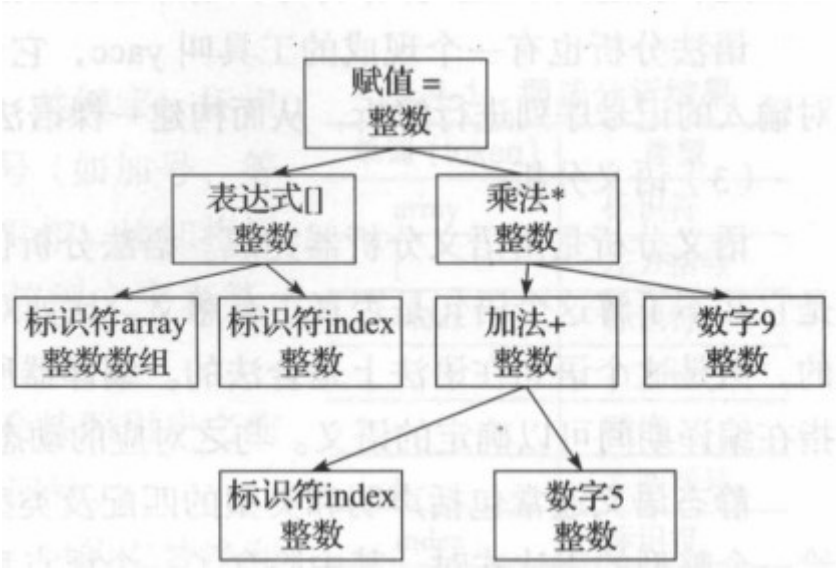
  经过语义分析阶段后，整个语法树的表达式都被标识了类型，如果有些类型需要做隐式转化，语义分析程序会在语法树中插入相应的转换节点。上面描述的语法树在经过语义分析阶段以后成为了如图所示的形式。



语义分析后的语法树

- 中间语言的生成 。现代的编译器有着很多层次的优化，往往在源代码级别会有一个优化过程。 这里所描述的**源码级优化**器在不同编译器中可能会有不同的定义或一些其他差异 。 源代码优化器会在源代码级别进行优化。 在示例中，可以发现，（2+7）这个表达式可以被优化掉，因为它的值在编译期间就可以被确定。经过优化的语法树如图所示 。

> 可以看到（2+7）这个表达式被优化成9 。 其实直接在语法树上进行这类优化比较困难，所以源代码优化器往往将整个语法树转换成中间代码，它是语法树的顺序表示，其实它已经非常接近目标代码了 。 但是中间代码一般跟目标机器和运行时环境是无关的，比如不包含数据的尺寸 、 变量的地址和寄存器的名字等。 中间代码有很多种类型，在不同的编译器中有着不同的形式，比如三地址码等。 优化后的语法树中间代码使得编译器可以被分成前端和后端。 编译器**前端**负责产生**机器无关的中间代码**，编译器**后端**则负责**将中间代码转换成目标机器代码** 。 这样对于一些可以跨平台的编译器而言 ，它们可以针对不同的平台使用同一个前端和针对不同的机器平台的数个后端。



- 目标代码的生成与优化。
  源代码级优化器产生中间代码标志着下面的过程都属于编译器后端。 编译器后端主要包括代码生成器和目标代码优化器。代码生成器将中间代码转换成目标机器代码，这个代码十分依赖于目标机器，因为不同的机器有着不同的字长、寄存器、整数数据类型和浮点数数据类型等。对于示例的中间代码，代码生成器可能生成下面的代码序列：

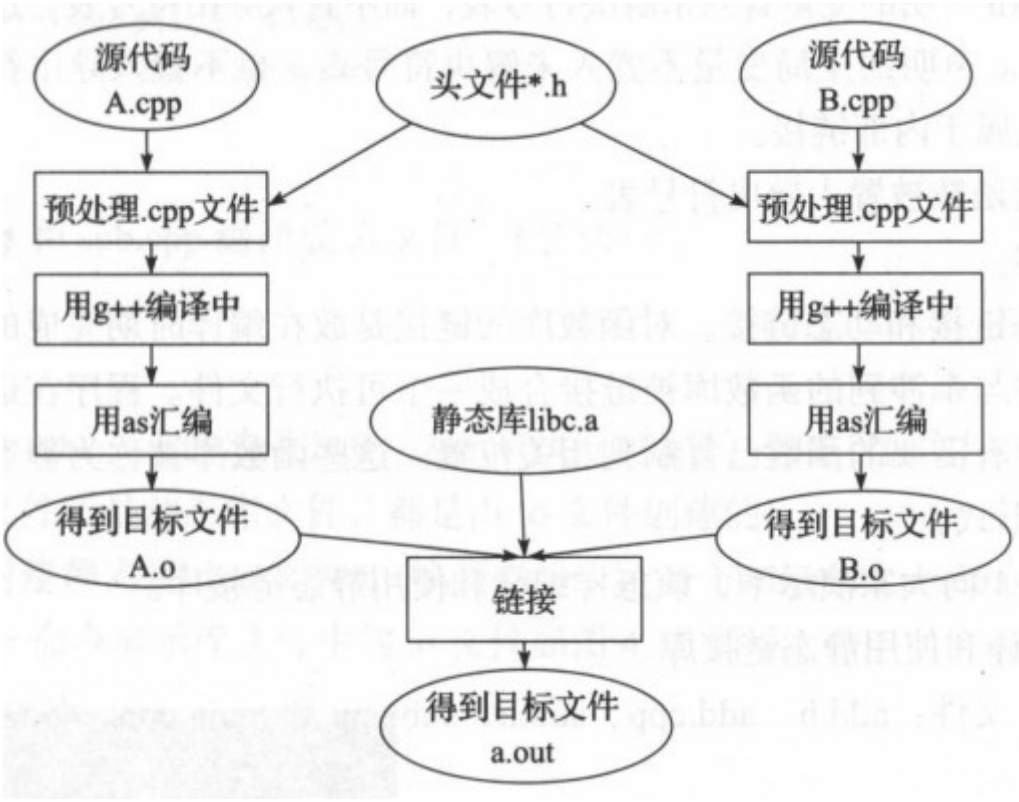
```
movl index, %ecx            ; value of index
addl $5, %ecx               ; ecx = ecx + 5
mull $9, %ecx               ; ecx = ecx * 9
movl index, %eax            ; value of index to eax
movl %ecx, array(,eax,5)    ; array[index] = ecx
```

  最后 目标代码优化器对上面目标代码进行优化，比如选择合适的寻址方式、使用位移来代替乘法等。

> 经过了扫描（词法分析）、语法分析 、 语义分析、源代码优化 、 目标代码生成和目标代码优化，编译器经过这么多步骤，源代码终于被编译成了目标代码 。 但是这个目标代码中有一个问题： index 和 array 的地址还没有确定 。 如果现在把目标代码使用汇编器编译成真正能够在机器上执行的指令 ，那么 index 和 array 的地址从哪里得的呢？如果index 和 array 定义在跟上面的源代码同一个编译单元里面，那么编译器可以为 index 和 array 分配空间，确定它们．的地址，但如果是定义在其他的程序模块中呢？
>
> 事实上，定义其他模块的全局变量和函数在最终运行时的绝对地址都要在最终链接的时候才能确定 。 所以现代的编译器可以将一个源代码文件编译成一个未链接的目标文件，然后由链接器最终将这些目标文件链接起来形成可执行文件。

链接

把每个源代码模块独立地编译，然后按照要将它们“组装”起来，这个组装模块的过程就是链接 。 链接的主要内容就是把各个模块之间相互引用的部分都处理好，使得各个模块之间能够正确的衔接。
但从原理上讲，它的工作无非就是把一些指令对其他符号地址的引用加以修正 。 链接过程主要包括了**地址和空间分配**、**符号决议**和**重定位**等这些步骤。
最基本的静态链接过程如图所示 。 每个模块的源代码文件（如 .c 文件）经过编译器编译成目标文件（如 .o 文件），目标文件和库一起链接形成最终可执行文件。 而最常见的库就是运行时库，它是支持程序运行的基本函数集合 。

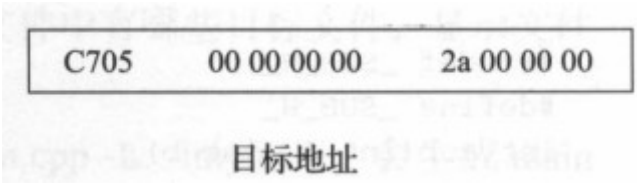


链接

库其实就是一组目标文件的包，就是一些最常用的代码编译成目标文件后打包存放 。在链接过程中，对其他定义在目标文件中的函数调用的指令需要被重新调整，对实用其他定义在其他目标文件的变量来说，也存在同样问题。
结合具体的 CPU 指令来深入了解这个过程 。 假设有个全局变量 var，它在目标文件 A中 。 当要在目标文件 B 里面要访问这个全局变量，如有这么一条指令 ：

```
movl $0x2a, var
```

然后编译目标文件 B ，得到这条指令机器码，如图所示。



由于在编译目标文件 B 的时候，编译器并不知道变量 var 的目标地址，所以编译器在没法确定地址的情况下，将这条 mov 指令的目标地址设为 0 ，等待链接器在将目标文件 A 和 B 链接起来的时候再将其修正。假设 A 和 B 链接完成后，变量 var 的地址确定下来为0x1000，那么链接器将会把这个指令的目标地址部分修改成 0x10000。这个地址修正的过程也叫作**重定位**，每个要被修正的地方叫一个重定位人口 。
每个目标文件除了拥有自己的数据和二进制代码外，还提供了 3 个表

- **未解决符号表**提供了在该编译单元里引用但是定义并不是在本编译单元的符号以及其出现的地址；
- **导出符号表**提供了本编译单元具有定义，并且愿意提供给其他单元使用的符号及其地址；普通变化及其函数被置入导出符号表。
- **地址重定向表**提供了本编译单元所有对自身地址的引用的记录 。

外部链接与内部链接

- **外部链接**：编译器将extern声明的变量置入未解决符号表，而不置入导出符号表。
- **内部链接**：编译器将static声明的全局变量不置入未解决符号表，也不置入导出符号表，因此其他单元无法使用。

静态链接与动态链接

对函数库的链接是放在编译时期完成的是静态链接。 所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。 程序在运行时，与函数库再无瓜葛，因为所有需要的函数已复制到相关位置。 这些函数库被称为静态库，通常文件名为“ libxxx.a”的形式 。