

接下里先来介绍下程序从源文件到可执行文件的各个步骤，

包括预处理、编译、汇编和链接过程:

首先，我们提供一些.c源文件等

然后，这些源文件会通过一系列.c文件的翻译器

1~调用C预处理器cpp(将main.c翻译成ASCII中间文件main.i)

2~调用编译器(cc1)将main.i翻译成ASCII汇编语言文件main.s

3~编译器生成汇编文件，然后汇编器将其翻译，结果生成一个.o文件(main.o等)

接下来，链接器将取到这些.o文件，然后还会加上一些必要的系统目录文件，(include 的文件等)，将这些链接到一起，创建一个单个的可执行文件然后执行

在这里对几个后缀文件进行分类

像main.c和sum.c这些的.c后缀文件被称为源文件

翻译后产生的main.o和sum.o等.o后缀文件为可重定位目标文件，因为这些文件可以 组合在一起形成完全链接的可执行目标文件

至于程序这样一步步的转换的原因有很多，

最明显的一点就是这样可以**模块化**，可以将代码分解为更小的部分，同时也可以提高重用性

另一个点就是**效率**，毕竟在这样的架构下，当我们需要去对一个模块进行修改时，我们不必要去对全部的模块都进行重新编译，只需要对我们修改的模块进行重新编译即可

分析下这个效率：对于一个项目的编译，在第一次编译时，编译器会生成所有编译文件 .o文件，对于这些文件，当我们没有对对应的源文件进行修改时，编译器不会对这些进 行重新翻译，而是一直保留这些.o文件并使用，当且仅当在源文件发生修改时会对其进 行重新翻译，这就优化了效率

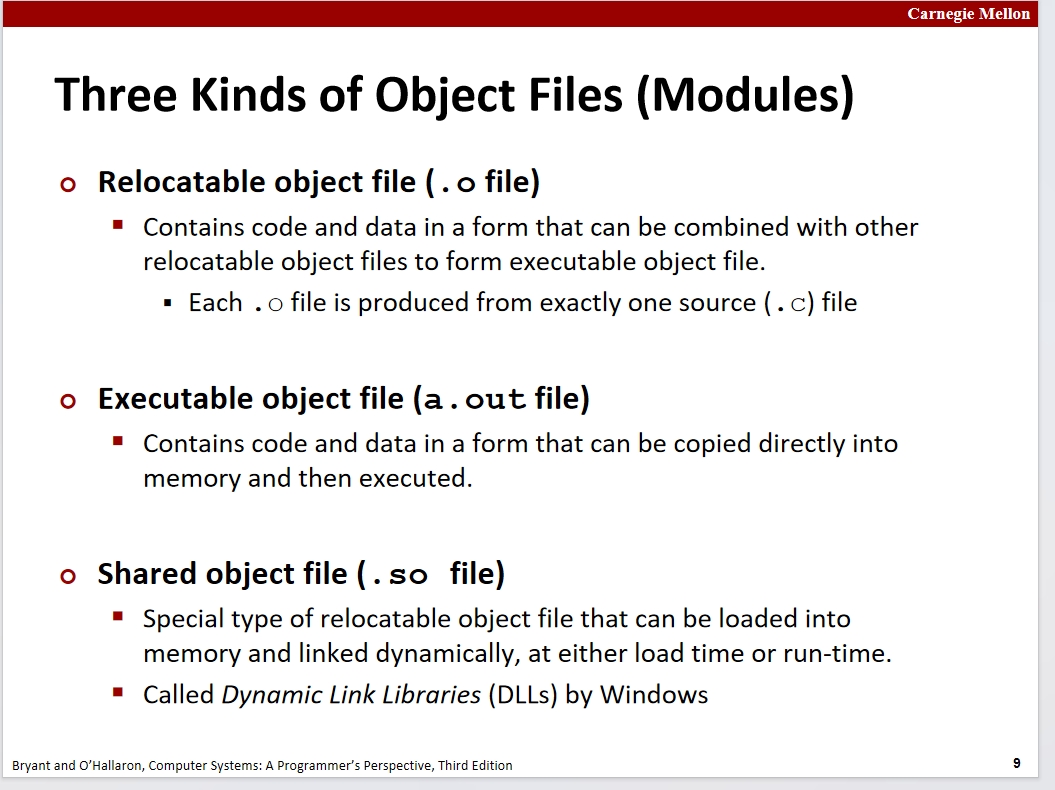
同时，还有就是可扩展性，这种链接性允许我们使用很多其他的源文件，这些就能支持我们一般使用的一些库函数

接下来说一下汇编器对程序的几个处理:  
1~汇编器会将符号定义储存在.o后缀的目标文件中，可能是通过构建一个符号表来储存符号 对应的信息，在这个表中，全是一系列的结构数组，每个结构数组里包含该符号的信息。 在这个数组中，每一个元素都是一个符号及其包含的信息，包括但不限于符号名称，符 号类型，地址或者偏移量，作用域，大小，定义位置等

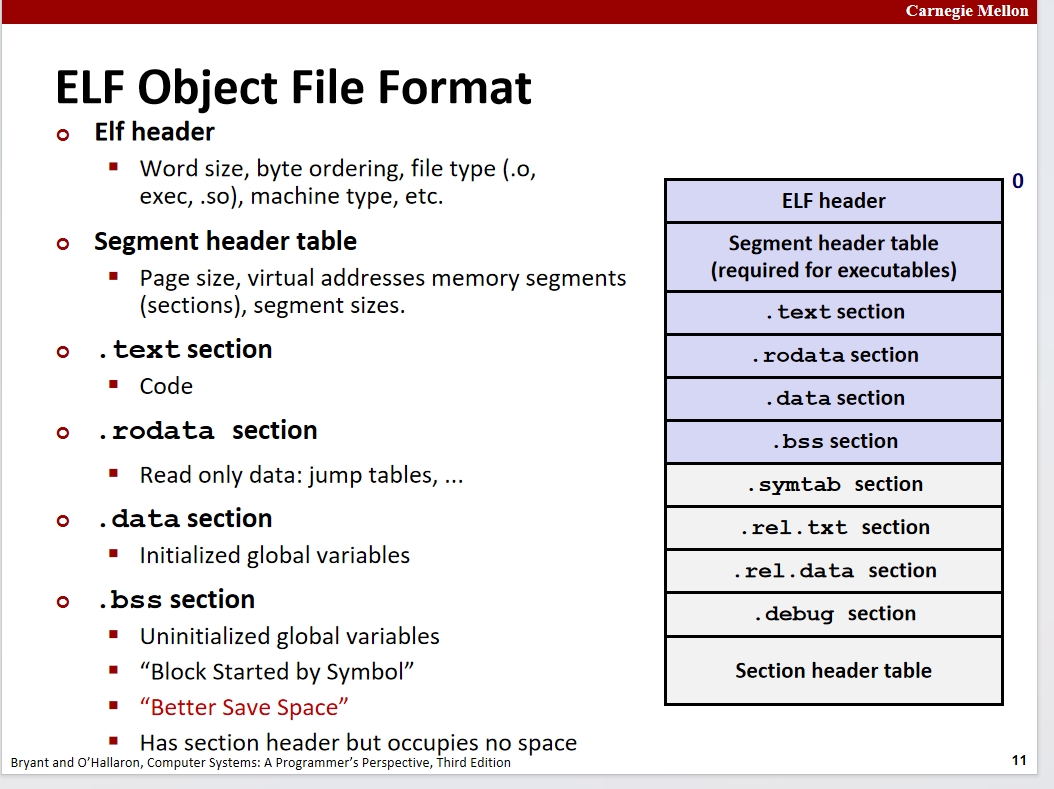
所以链接中的符号解析的意思是，在链接中，将每一个符号引用与一个符号定义相关联

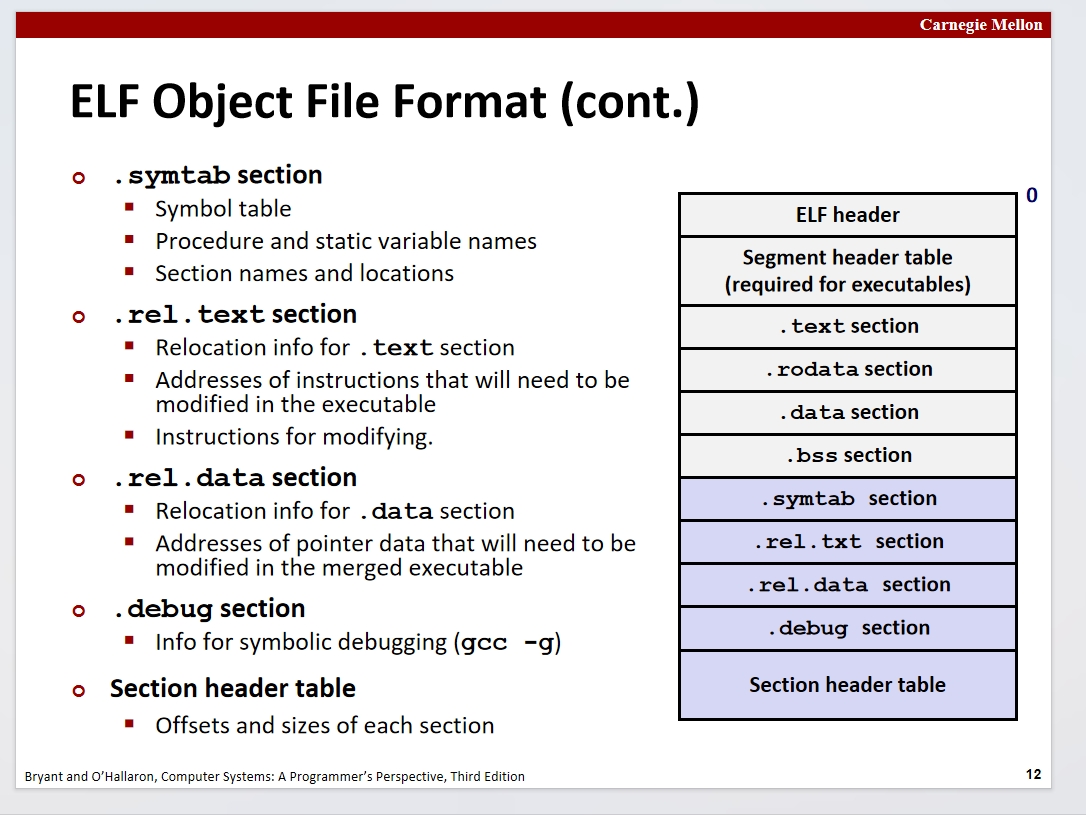
2~接下来会进行第二步，即重定位，在这一步中，编译器会将所有模块合并到一起，进入单 个可执行目标模版中，就可以直接加载和执行。对于这个的理解，可以视作是一个对整 个程序所调用的函数的位置的一次更新。

就比如，你可能在写程序时会用到一些其他文件中的一些库函数，而这个对于这个 文件来说，它的符号表中并没有这个函数的信息，它只能通过一些方式知道这个函数存 不存在，至于对于函数具体的位置的确定，这需要多个源文件间进行确定，对各个文件 的地址表进行查询，更新这些函数的地址，使得之后在文件的调用过程 中能够 被 正确调用。同时需要注意，在重定向前，各个文件所有的地址表中的地址一般是相对于 这个文件的地址，而在重定向的这个截断，编译器会将所 有文件的地址表逐渐进行 合并。在这个过程中，各个函数在当前文件的相对地址会被转换为一个绝对地址以此来 实现之后单个可执行文件的大地址表，使得所有文件的变量的地址都能在一个表中体现

接下来对一些目标文档文件进行分析

对于这几个后缀文件，现在基本目标模块统一用的都是ELF格式的标准格式





接下来分析下二进制头文件的结构

1~首先是一个header ，这个头定义了生成这个文件系统的字节顺序和字的大小， 使用什么样的字节顺序(大端小端)，也就是说这个提供了该文件的基本配置信息

2~接着是一个节头部表，这个只在可执行文件中定义，就比如.exe .out 无后缀等文 件中定义，这个指出所有不同段在内存中的定义，包括但不限于各种变量，函 数，数据结构的地址

3~接着是代码本身的.text部分,这个就是程序的二进制代码

4~再之后是只读数据区(.rodate)这个包括但不限于switch的跳转表，字符串常量和 其他只读的全局数据

关于.text和.rodate的属性:只读，因此无法对这部分数据进行写入，只会由系 统自动生成

5~再之后是.date区域，该区域包含了所有的初始化了的全局变量和静态变量数据的 空间

6~接下来是.bss区域，该区域包含了所有的未初始化了的全局变量金额静态变量的 数据的空间

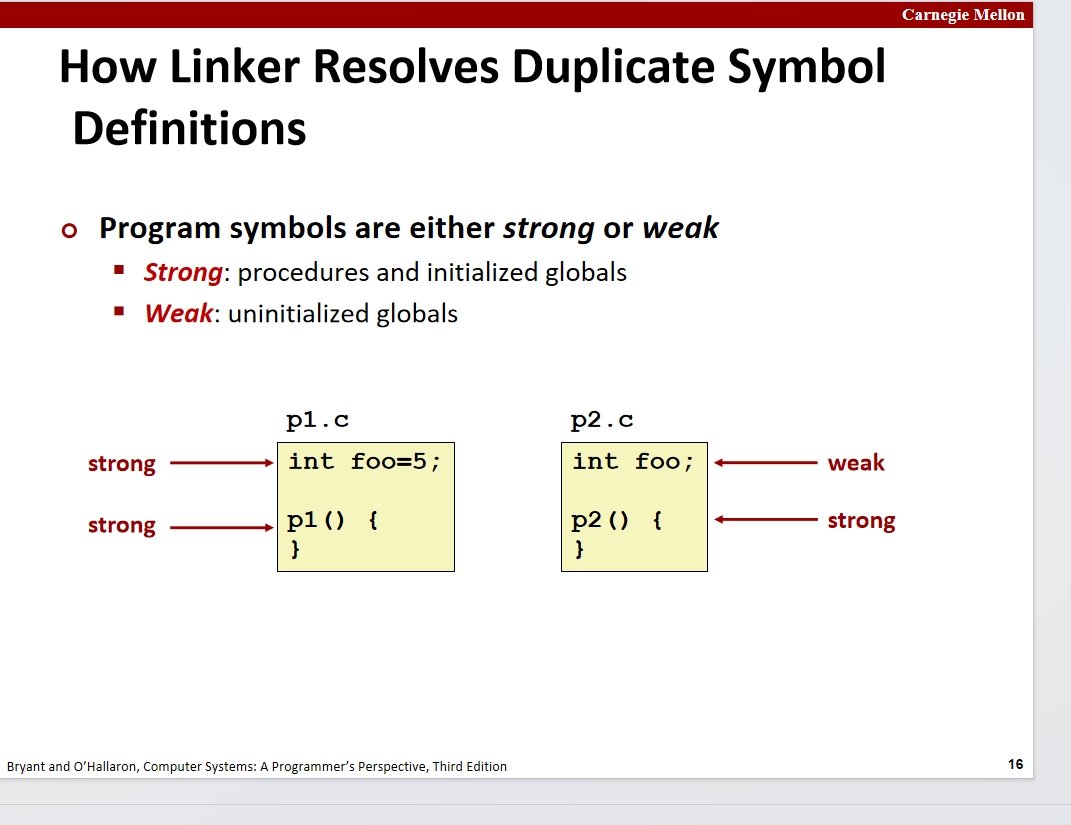
7~.symtab中主要用于储存符号表，包含程序运行时的所有的符号的信息，就是之 前的那个地址表，包含所有源文件的变量，函数及各种杂七杂八的

8~.rel.text .tel.date这俩个主要用于储存一些需要再每次编译时都进行重定位的符号

9~.debug 这个包含了将源代码中的行号与机器代码中的行号相关联的信息

10~Section header table 这个包含了每个源代码文件在这个二进制文件中各个不同 部分的起始位置

接下来分析下一些符号的属性:



这里是指符号具有一种强弱的区分

有一些规定，将函数名称或者已经初始化好的全局变量视为强符号，弱符号是 未初始化的全局变量

链接器在符号强弱上的一些规定:

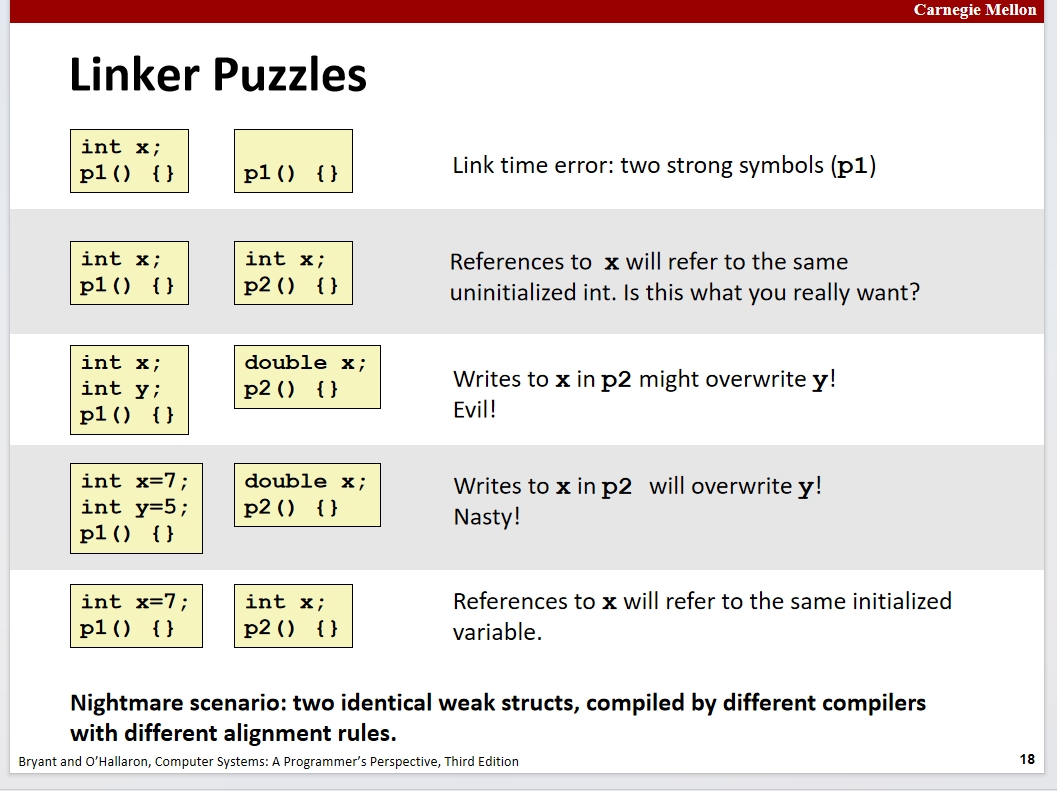
1~不允许出现相同的强符号，也就是说，不允许在多个文件中重复出现相同的函数 名或初始化好的全局变量等，这会导致链接器抛出异常

2~链接器当在多个源文件中遇到一个强符号和多个弱符号的时候，会将弱符号关联 到对应的强符号的符号表上

3~如果有多个弱符号同名，链接器则会从这些弱符号中任意选择一个，这个可能 导致问题，使用 gcc -fno -common能够使得编译器在识别 到多个同名弱符号时 抛出错误

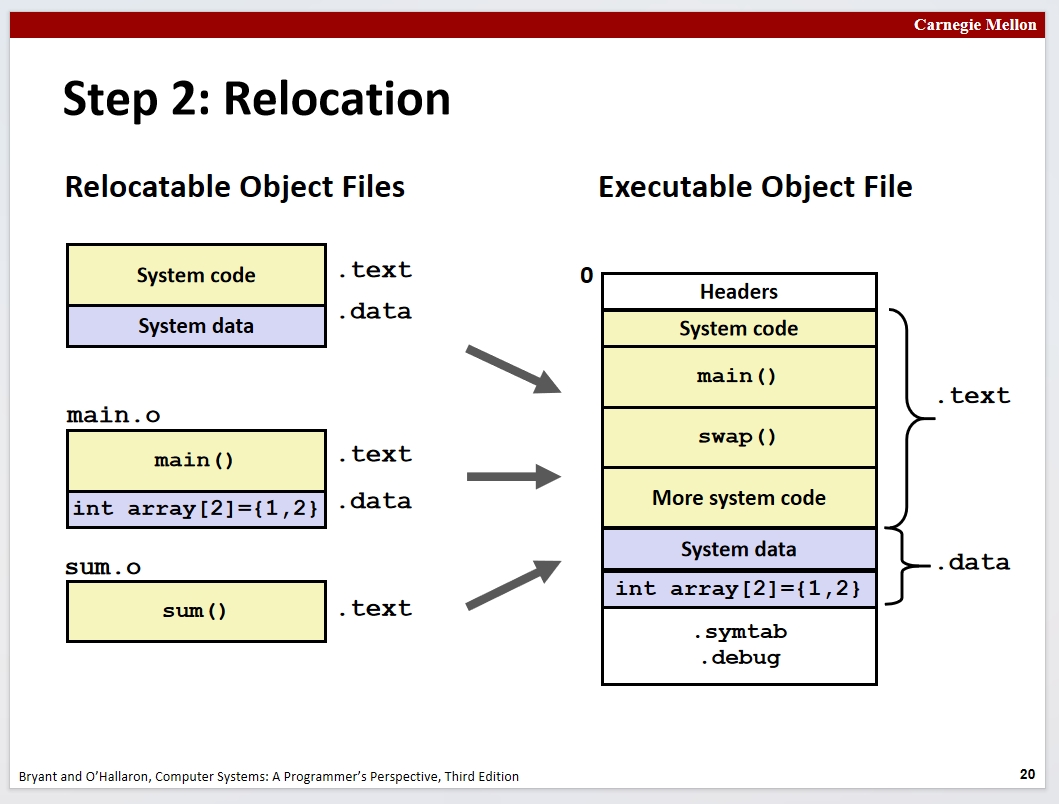
链接器是最应该避免的一种，毕竟这个是你通常无法了解它到底做了什么

这些举出几个容易被忽略的错误，就比如，我们在多个源文件中都声明了一个x变 量且都没有进行初始化，其中存在int型和double型等不同的类型，由于只会链接 一个定义，只会写入一个位置。故这里对一个变量是会对多个源文件中的代码逻辑 产生影响的。而且还可能出现一些额外的情况，



就比如这里的第二块白色部分，如果我们对右边的x进行修改，由于一些原因，左 边源代码中的y其实是会跟在x之后的，由于double是8字节的，故在修改时会修 改8字节的数据，但是这对于左边的数据是不可接受的，当对这个8字节数据进行 修改时，左边的其实会一次性覆盖掉x与y整块的数据，导致一些意料之外的修改， 而这个如果实在关闭对强弱定义的报错的情况下是很难发现这个错误的

需要注意一点，这里假设定义的x和y是定义的全局变量且他们的储存地址是连续 的，不连续也无所谓，这个只是想让我们知道，这种强弱的情况下是可能出现对于 一些不期望的数据的改变的



接下来对程序链接成一个可执行文件进行分析

首先，有一段系统代码system code 这一段代码会在程序运行前后运行，程序运行时，会从lib.c执行启动代码，这个代码中一般设置了程序的执行环境

接着，当链接器重定向.o等目标文件时，会将所有代码都放在每个模版的文本部分处，接着会将这些文件放于可执行对象文件的.text部分中，也就是之前说的代码的二进制文本部分。在这个.text部分中，包括了所有系统代码(包括但不限于用户提供的代码，应用到的系统的库函数，外部的动态库等)，

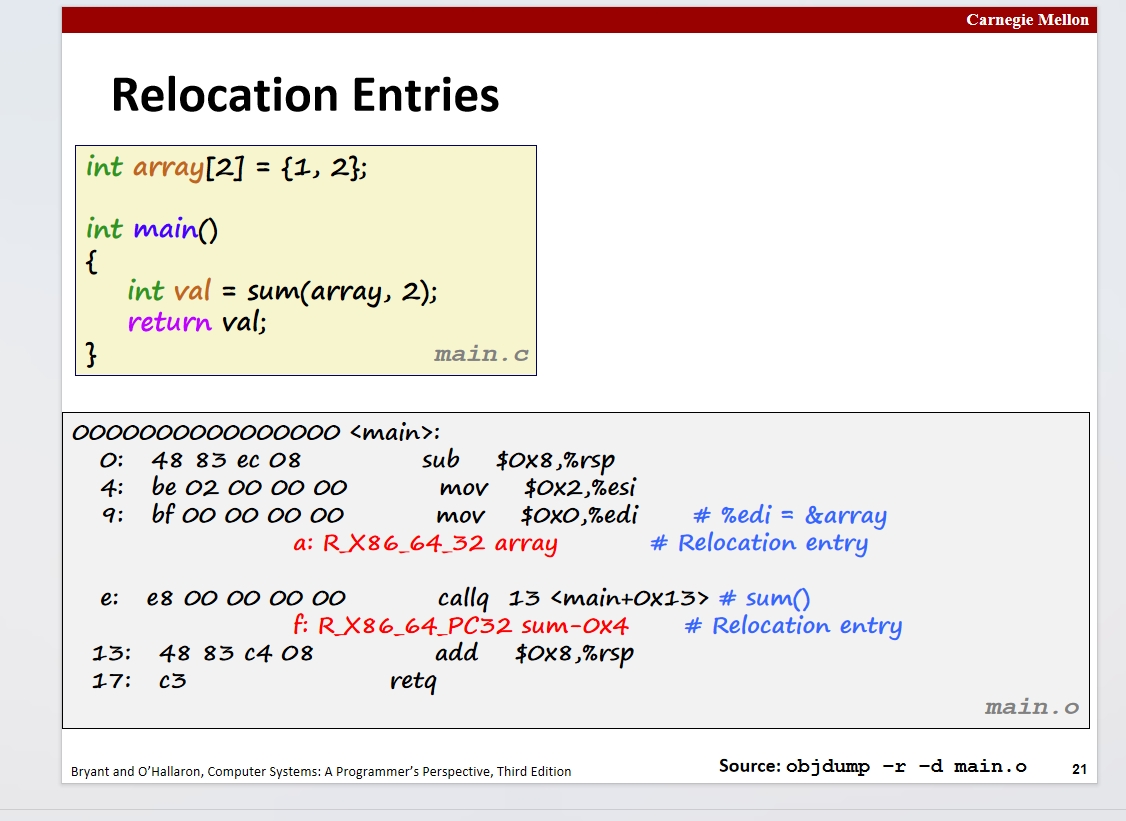
同时，会对所有的数据也采用同样的事情，将来自各个文件的数据放到一起组成一个大的.date(已初始化)块和.bss块(未初始化)

除此之外，还会出现整个可执行文件的符号表和调试信息

而种种这些的实现，需要一个链接器，当程序加载时，需要给这些不同的符号确定一个地址，因此，链接器将会提供一个main函数的绝对地址，程序将从这个绝对地址开始运行，同时也会给每个符号都分配一个绝对地址，正如之前所说，对于相对地址向绝对地址的转换，这是在链接器阶段实现的

但是，需要注意的是，在编译器阶段，编译器并不知道各个符号的具体内部储存，它只关注代码的逻辑，只关注优化代码。它会对代码进行重排等优化，当时不了解这些使用的符号的具体地址，只会记录下这些符号的使用情况等。将这些信息生成符号表，重定位信息和其他辅助信息等供链接器使用。

也就是说，对于一些代码的位置，就比如，函数中可能调用别的函数，但是在编译器处理之后，这些调用的位置是可能发生改变的，编译器会记录下这些改变的情况并传给链接器，链接器会根据这些数据进行适当的调整等



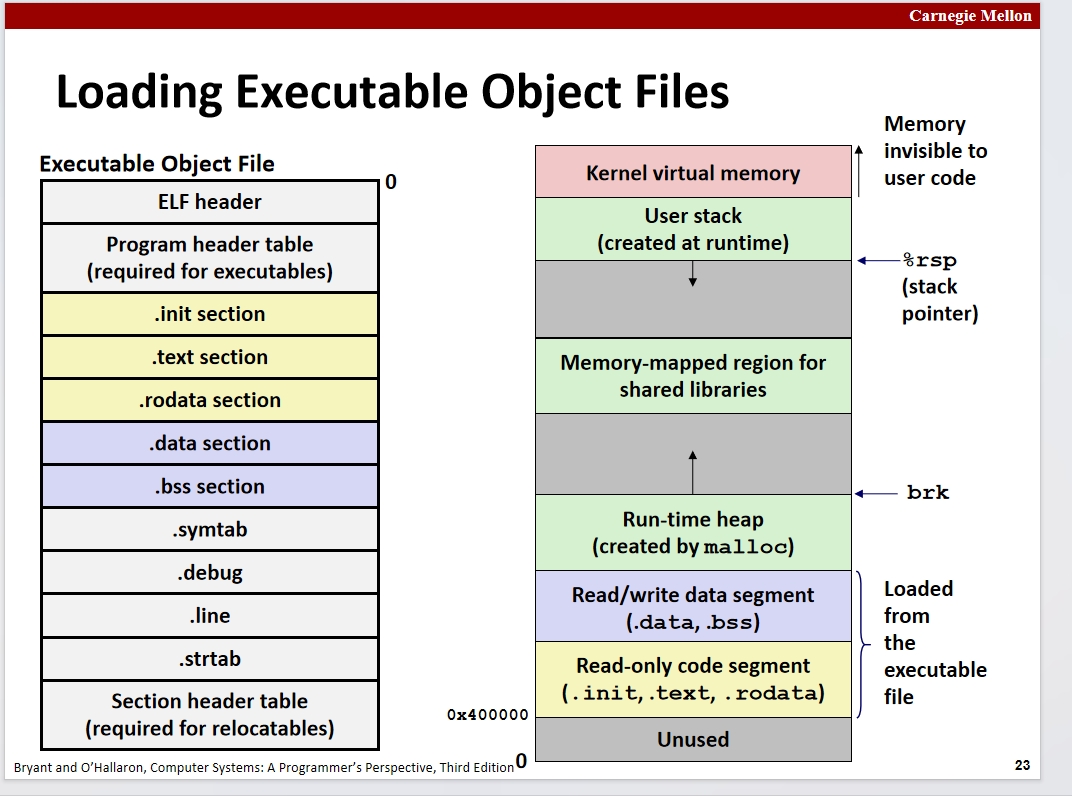
就比如这里，对于编译器生成的代码，它并不知道这里的sum函数所需要的array数组的地址，这是要提供一个具体的绝对地址的，但是这个阶段下并没有生成这个地址。因此，编译器会站在编译器传递这个地址参数时使用一个全0的地址，并在这一行之后添加一行额外的来实现地址的偏移来实现正确地址的传递

就比如这里的 a: R\_X86\_64\_32 array 等俩段红字

都是一个重定位信息，要的就是提醒链接器在链接时对这俩段代码进行地址的替换以传递正确的参数

R\_X86\_64\_32

这里的R表示重定位，X86\_64指明当前是针对X86\_64的重定位类型，PC指程序计数器，32便是重定位的最终位数，至于重定位的情况，需要注意一点，就是每个源文件中的符号表中的地址其实也是相对于本符号表中的地址。那么要保证链接后的还能够使用相对位置来计算绝对位置，就需要保证对于程序来说，这个相对位置是基本不变的，正如之前所说，是有一个东西来进行每个源文件之间的位置的区别的，在符号表中也存在这种以来计算最后链接器需要替换的绝对地址

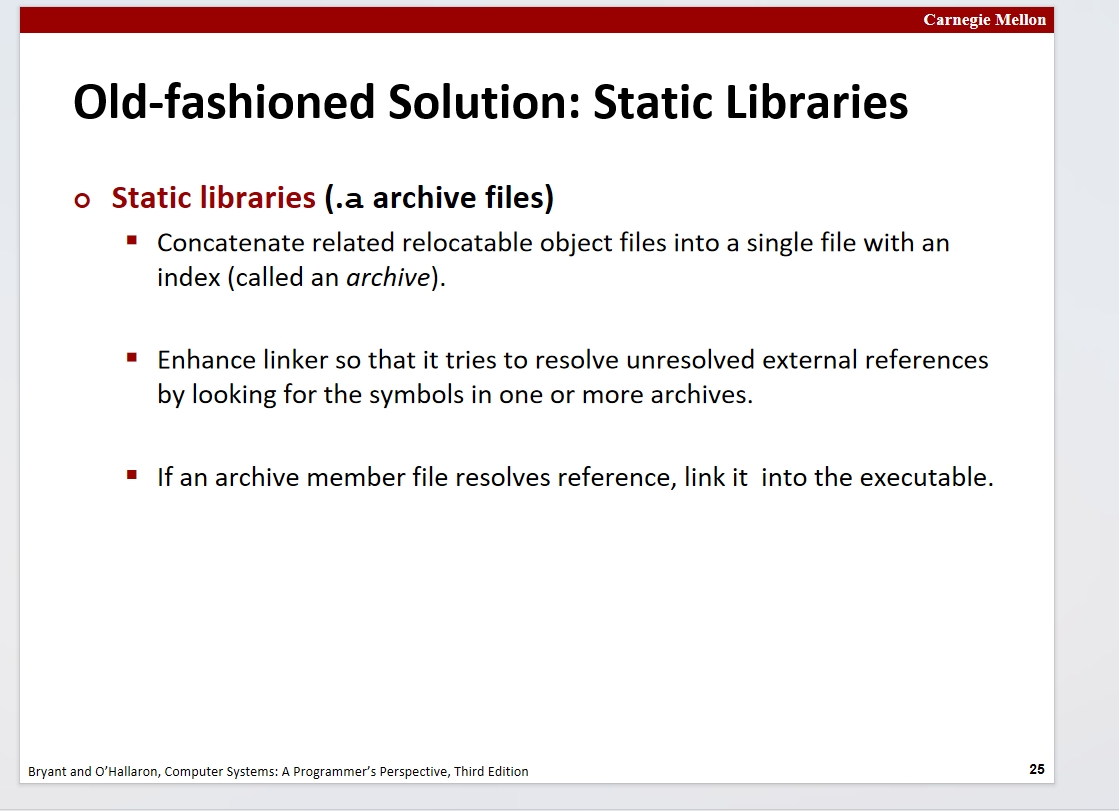


链接器创建一个目标文件后，这个对象文件可以加载代码和数据，并且这个对象文件可以直接加载到内存中并无需进一步修改

在程序启动后，操作系统会先执行.init段中的代码，包括各种初始化等，接下来会将所有的.text和.rodate等也加载到内存中，这样就形成了对应的只读代码段，这段代码包含了对应的各种变量，符号表，跳转表等一系列杂七杂八的信息

接着，.date和.bss部分也会直接复制到内存中，这里面的变量等会被初始化为符号表中的信息

程序的结构如上右图所示

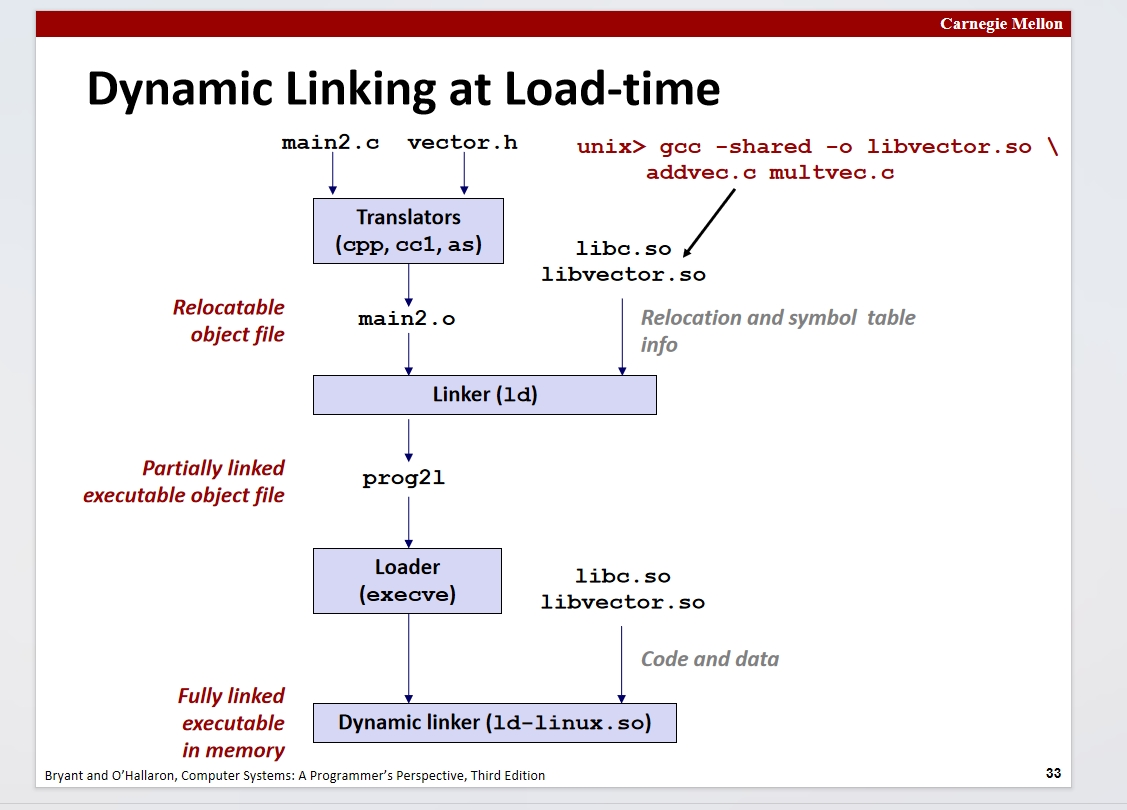


对程序，我们可以使用一些类的库函数，我们对其有一定的理解，这些函数每一个都是一个独立的.o文件，但是事实上我们不需要对每一个使用到的都进行链接，这是得益于一个**静态库技术**

这个将多个.o文件归档放到一个大文件中，然后使用一个内容目录，告诉每个.o文件的偏移量，也就是说，这个技术就是把多个.o文件归档到一起，当我们需要使用这里的部分函数时，我们只需要链接这个静态库，让静态库自己给出我们需要的函数的地址，不需要我们一个个去进行链接

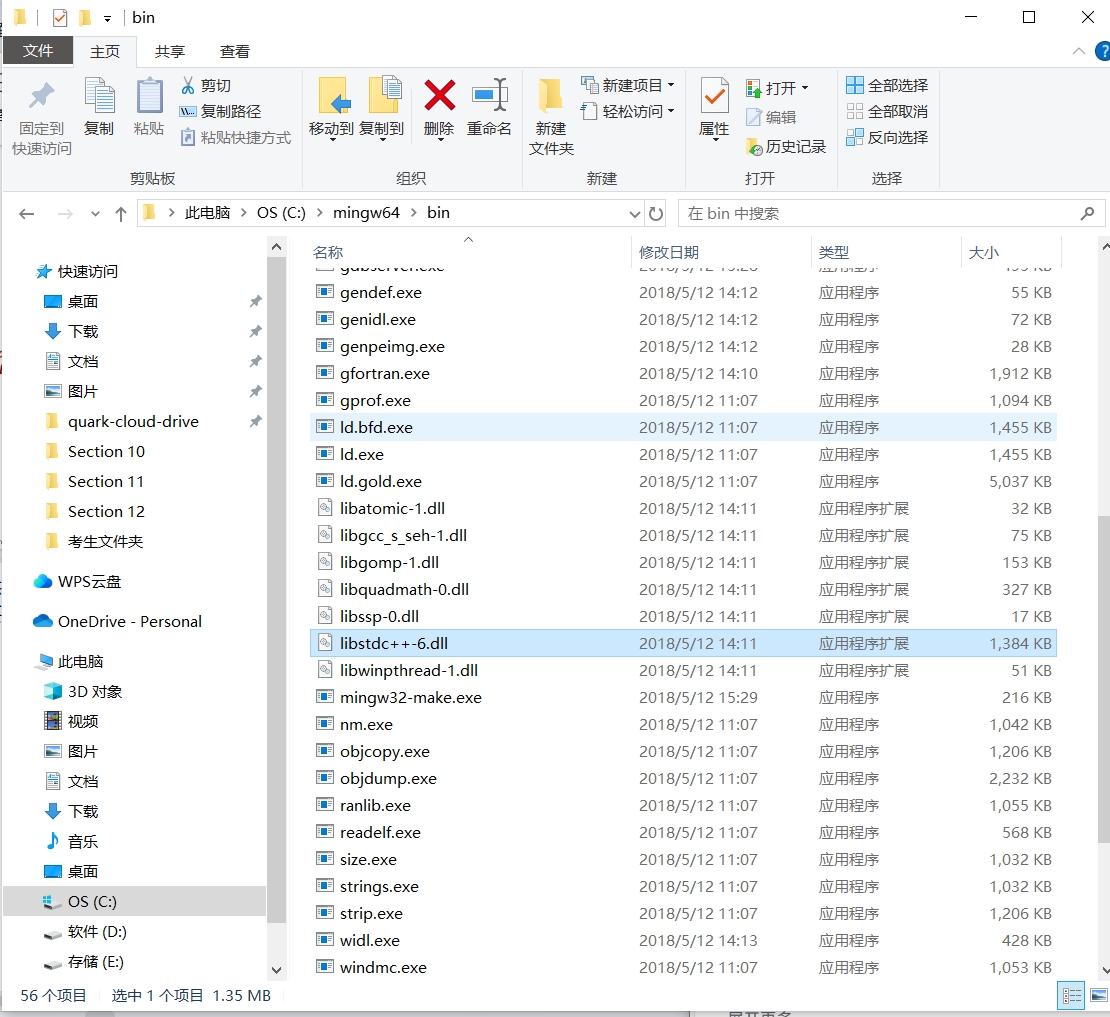
但是静态库有一个缺点，就是对于每个需要的原文件，它都需要拷贝一份副本进去，这是一个耗时且耗空间的事

因此，这里出现了共享库，这个允许程序共有一个库函数，对于这个库函数，只有一份副本且共用在win中，这些以.dll后缀结尾，在linux中，以.so后缀结尾，对于共享库，一些性质基本跟静态库一致，但有一部分需要区分，对于动态库，其每个库函数最多只可能生成一个副本，而对于程序，其最多也就只能使用一个函数的一个副本，这个副本是各个部分共用的



解释下动态库如何工作

首先，需要先将我们自己需要的动态库找到，程序本身也链接了一些动态库



接下来，编译器会将程序自己编译产生的.o目标文件和我们要链接的对应动态库文件传给链接器，这时，链接器并没有实际的复制我们在.o文件中可能使用到的任何动态库中的函数代码，链接器在这个过程中并不对动态库执行任何操作，只是在符号表中进行记录，使得程序知道这个动态库中的函数的存在。在加载程序时需要通过这些来解析程序对这些函数的引用，也就是说，程序会在需要时再加载函数。

所以在.o目标文件和共享库链接完后的知识一个部分链接的文件，这个并没有完全链接起来，我们是无法获取这个阶段的文件的

接下来加载器Loader会获取可执行文件，同时需要对应的共享库文件，这个文件目录是链接器确定记录的

也就是说，对于动态库中的函数，在程序加载之前都是不确定的，这里的加载说的是程序加载到内存中而不是程序使用函数的阶段

再之后，动态链接器就会修复引用回代码中对动态库函数的引用，然后，我们的二进制文件就可以正确执行了

这里解释下inlcude和动态库(静态库)间的联系，首先先明确一点，这俩个之间存在着一点联系。但不多，对于一个动态库(静态库)文件，他给我们提供了一系列的.o文件，而这些.o文件其实又是一系列的函数实现文件。而实际上，我们即使链接上了一个库，编译器也是不了解我们这个库中包含了哪些文件的。库文件中只有实现，而我们要使用我们需要一些声明来实现一个编译器与库间的关系确定。也就是说，我们的include库文件一般包含的都是对于动态库的函数声明，使得编译器在预处理阶段不会报错

**库打桩，启动**

库打桩的目的:截获来自库的函数调用

(有了在这个调用信息你就可以做一些事了，就比如debug等)

这是一个大块，分有多个操作情况，或者说，这个有多种实现库打桩的技术

1~最简单粗暴的，就是直接对原代码进行修改，但是这其实是不符合要求的，因为我们打桩要的就是找到现有的代码中存在的问题，我们直接对原代码进行修改，这个可能会导致一些新的问题

2~通过在源代码增加一些新的调试函数，这些调试函数一般都会调用我们进行打桩的库函数，且这些添加的调试函数中我们一般会添加一系列的调试信息。接着，我们会在添加这些新的调试函数中增加一些宏定义来进行替换，一般都是将我们希望进行打桩的函数宏替换成我们新的调试函数，通过这样，使得原本调用的库函数能够替换为我们现在需要的调试函数。由于这个调试函数中包含了原本的库函数只是多了一些调试信息，故这里在没有改变原有功能的基础上实现的调试

但需要注意的是，这需要对原代码有一定的了解，不会使得宏替换后的函数改变原有的功能

3~还有一种方法，而且这种方法可以避免编译，不过这个需要我们在定位时使用链接时间。需要使用\_wrap\_的前缀来定义我们的调试函数，并在编译时使用选项

-Wall --wrap -Wl

gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl

我们通过使用这个-Wl参数进行打桩

这个参数是指我们会将后面的这些链接器参数传给链接器

--wrap参数是另链接器将对后面函数名的引用都解析为\_\_wrap\_的格式，通过这样来实现在链接阶段对对应函数的替换

4~接下来还有一种操作，通过使用一些dlopen函数参数来实现对应的功能

mallocp = dlsym(RTLD\_NEXT, "malloc");

就比如这里的RTLD\_NEXT参数，这个能够使得我们在查找共享库函数的时候，不在当前打开的共享中进行查找，而是下一个对应的共享库中的进行查找，而这个操作的前提就是我们明确了解并且保证我们接下来的共享库中包含了我们需要的调试函数，这个共享库的顺序可以通过多方面进行保证，由于时间原因，后面再进行补充，可以通过宏定义，编译顺序，调整链接器顺序来实现等