CSC 488S/CSC 2107S Lecture Notes

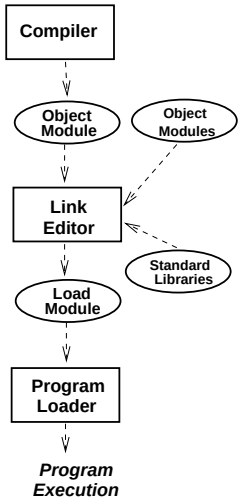
These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the
Winter 2012/2013 term at the University of Toronto

Linking and Loading[a]

An Object module produced by a compiler is linked together with other object modules and standard libraries to form a complete executable program.
Issues dealt with by the linking process

- Resolution of external symbol references
- Relocation of code addresses
- Support for separate compilation
- Support use of standard libraries
- Creation of loadable module

Loading reads a complete executable program (`a.out` or `a.exe`) from disk into memory and starts its execution..



[a] See: John R. Levine, *Linkers & Loaders*, Morgan Kaufmann, 2002

Object Module Information

A typical object module may contain

- Header information. e.g. code size, name of source file, creation date, etc.
- Object code. Binary machine instructions and data.
  Data may include initialized data (e.g. constants) and uninitialized space for static storage.
- Relocation information. Places in the object code that need to be modified when the linker places the object code.
- Symbols
  – **Exported names** defined in the object module.
  – **Imported names** used in the object module
- Debugging information. Information about the object module that is useful to a runtime debugger. e.g. source file coordinates, internal symbol names, data structure definitions.

Linking is a Two Pass Process

Input: a collection object modules and libraries

First Pass

- Scan input files to determine size of all code and data segments.
- Build a symbol table for all imported or exported symbols in all input files
- Resolve all connections between imported symbols and exported symbols
- Maps object code segments to image of output file
- Assign relative addresses in this output block to all symbols

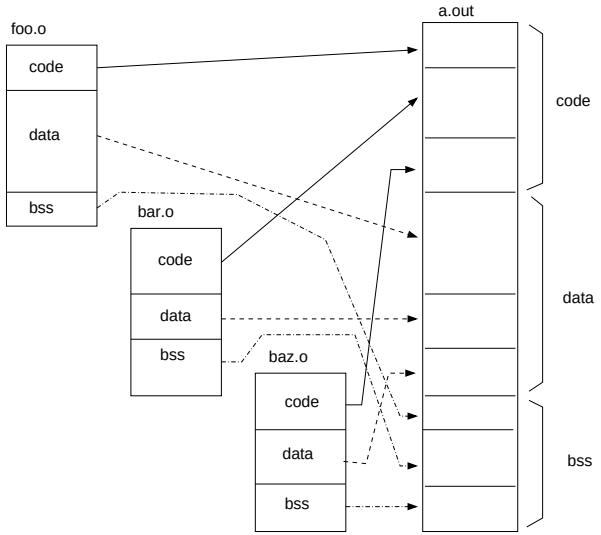
Resolving Symbols

- Every external symbol used in an object module should be resolved to a symbol exported from some other object module or library.
- Unresolved symbol are usually a link time error.
- If multiple instances of a external symbol are available there are several strategies:
  – emit a link time error message
  – pick one symbol with a warning message
  – silently pick one symbol

Second Pass

- Read and relocate object code and data to the output block.
- Replace symbolic addresses with block relative numeric addresses.
- Adjust memory addresses in the object code to reflect new location in the output block.
- Write output file
  – header information
  – relocated code and data segments
  – symbol table information
  – debugging information

Linking Example



Loading

- In most modern systems, the load module can simply be read into a block of memory and then executed. No load time relocation is required.
- To make this possible compilers must generate position independent code e.g. branches are relative to the program counter, data access is via a base register.
- Dynamic linking allows some binding of symbols to library modules to be deferred until runtime.

Dynamic Linking and Loading

- Defer much of the linking process until a program is loaded into memory and starts execution.
- Advantages
  – Dynamically linked shared libraries are easier to create.
  – Dynamic libraries are easier to update without required extensive relinking.
  – Dynamic linking allows runtime loading and unloading of modules.
- Disadvantages
  – Substantial runtime overhead for dynamic linking
  – Dynamic linking redone for every execution.
  – Dynamic link libraries are larger than static libraries due to symbol information.
  – Changes to/unavailability of dynamic libraries may make programs unexecutable.

In Conclusion: User Interfaces for Compilers

- The importance of good *Human Engineering* in designing the interface between the compiler and the user cannot be over emphasized.
- **The compiler should describe problems with a program in terms that the user can understand.**

Good: Syntax error on line 100 of file myProgram.c
The reserved word **if** cannot be followed by the identifier *myVar*

Bad: Illegal symbol pair **if** *identifier* . Parse stack dump:
123 ＜identifier＞
122 **if**
121 ＜statement＞
120 ＜block head＞
...

Really Bad: Syntax error in program. Compilation terminated.

Unacceptable: java.lang.NullPointerException: null
or Segmentation fault - core dump

What the Compiler Should Do

- Determine if the program is correct.
- Describe the statically detectable errors at compile time.
- Translate the source program into an equivalent object program.
- Emit code to detect and describe dynamically detectable errors during program execution.
- Language designer (should) specify the possible errors in a language.
  Note the difference between error, implementation defined and unspecified.
- The language implementor decides between static and dynamic detection of errors.
- All errors *should be* detectable.

Error Detection Tradeoffs

- Static detection of errors may greatly increase
  - Complexity of the compilers internal data structures and algorithms
  - The time required to compile all programs.
- Example: To detect aliasing of variables the compiler needs links between all calls of a routine and the definition of the routine.
  The time to do aliasing detection is quadratic in the size of the program.
- Dynamic detection of errors at runtime may greatly increase
  - The size of the program due to extra chacking code (e.g. array subscript checking).
  - The execution time of the program due to the time required to execute checking code.
- Examples of expensive run time checking:
  - Uninitialized variable checking .
  - Array subscript checking .
  - Dangling and Nil pointer errors .

Compiler Reaction to Errors in Programs

Bad: Compiler Crash or Infinite Loop.

Bad: Java execption stack trace.

Bad: Generate incorrect object program and no error messages.

Poor: Stop the compilation

Good: Recover from the error and continue compilation.
Hard to do well. Error cascades and false errors are problems.

Great: Repair error and continue the compilation
Perfect correction is undecidable.
Use heuristics and systematic strategies.

Errors should be detected as soon as possible.
Localize (in a routine) the production of error messages.
Optionally print error summary at end of compilation
13 Errors detected, Last error on line 219 of file urProg.c

How the Compiler Can Help the User

- Use symbol table to produce information for the programmer
  - Cross reference list for identifier definition and use.
  - Frequency of usage information for identifiers.
  - Diagnostics for possible errors
    Variable/constant declared but never used.
    Variable assigned to but never used.
  - List sizes of data structures. Warn of excessive fill in data structures.
  - List code size and usage information for routines.
- Identify potentially inefficient constructs for the user.
  Statement 415 in yourProg.c generated 600 bytes of code.
  Statement 311 in utility.c implemented by 10 calls to library routines.
- Optionally summarize compiler internal resource usage.
  Used 400 of 511 symbol table entries.
  Provides warning of possible overrun of compiler limits.

- Provide user control over compiler processing options
  Example: `gcc` (see `man gcc` for the gory details).
  Good:
  - allows user control over optimization
  - provides work around for compiler problems

  Bad:
  - can become *very* complicated[a]
    For example gcc has

13 general options	19 C language options
3 language independ options	40 C++ language options
80 warning options	65 debugging options
131 optimization options	hundreds of machine dependent options

  - more than most users will ever use
  - makes software maintenance harder, since compiler options change compiled program

[a] Care to guess what the gcc option `-fmudflap` means?

Compiler Should be Self Diagnostic

- Compiler should be *error imune*, i.e. never crash for *any* possible input.
- Compiler must be programmed to detect *all* violations of compiler internal limits. Do internal consistency checking of compiler data structures.
- Use exception handlers to trap programming errors and shut down gracefully.
  An internal error has occurred. Please file a bug report.
- Optionally print out compiler internal performance statistics
  10,301 identifier lookups performed. 73% were local scope.
  4,219 scanner tokens produced, 3745 nodes in abstract syntax tree.
  143,216 bytes of code generated, 40,219 instructions.
  10.37 seconds in semantic analysis, 21.7 seconds in code generation.
- Optionally provide information (perhaps embedded in the object program) about the options used in the compilation
  Compiled on 2010-03-22 at 14:00:01 using superCompiler version 4.3.21
  Options -noCheckArray -O17 -stuffStructs -fastStrings

Example: `perl -V`

```
Summary of my perl5 (revision 5 version 14 subversion 2) configuration:
  Platform:
    osname=linux, osvers=2.6.42-37-generic, archname=x86_64-linux-gnu-thread-multi
    uname='linux batsu 2.6.42-37-generic #58-ubuntu smp thu jan 24 15:28:10 utc 20
    config_args='-Dusethreads -Duselargefiles -Dccflags=-DDEBIAN -Dcccdlflags=-fPI
    hint=recommended, useposix=true, d_sigaction=define
    useithreads=define, usemultiplicity=define
    useperlio=define, d_sfio=undef, uselargefiles=define, usesocks=undef
    use64bitint=define, use64bitall=define, uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -p
    optimize='-O2 -g',
    cppflags='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -pipe -fsta
    ccversion='', gccversion='4.6.3', gccosandvers=''
    intsize=4, longsize=8, ptrsize=8, doublesize=8, byteorder=12345678
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=16
    ivtype='long', ivsize=8, nvtype='double', nvsize=8, Off_t='off_t', lseeksize=8
    alignbytes=8, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =' -fstack-protector -L/usr/local/lib'
    libpth=/usr/local/lib /lib/x86_64-linux-gnu /lib/../lib /usr/lib/x86_64-linux-g
    libs=-lgdbm -lgdbm_compat -ldb -ldl -lm -lpthread -lc -lcrypt
    perllibs=-ldl -lm -lpthread -lc -lcrypt
    libc=, so=so, useshrplib=true, libperl=libperl.so.5.14.2
    gnulibc_version='2.15'
```

```
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-Wl,-E'
    cccdlflags='-fPIC', lddlflags='-shared -O2 -g -L/usr/local/lib -fstack-protect
Characteristics of this binary (from libperl):
  Compile-time options: MULTIPLICITY PERL_DONT_CREATE_GVSV
                        PERL_IMPLICIT_CONTEXT PERL_MALLOC_WRAP
                        PERL_PRESERVE_IVUV USE_64_BIT_ALL USE_64_BIT_INT
                        USE_ITHREADS USE_LARGE_FILES USE_PERLIO USE_PERL_ATOF
                        USE_REENTRANT_API
  Locally applied patches:
    <DELETED for brevity>
  Built under linux
  Compiled at Mar 18 2013 19:17:55
  @INC:
    /etc/perl
    /usr/local/lib/perl/5.14.2
    /usr/local/share/perl/5.14.2
    /usr/lib/perl5
    /usr/share/perl5
    /usr/lib/perl/5.14
    /usr/share/perl/5.14
    /usr/local/lib/site_perl
```