U N I V E R S I T Y ®

CS 50011: Introduction to Systems II

Lecture 8: Compiling and Linking

Prof. Jeff Turkstra

Lecture 08

- Compiling a program
- Compiler structure
- Static vs. dynamic linking

- Some slides by Prof. Gustavo Rodriguez-Rivera

Program

- File in a particular format containing necessary information to load an application into memory and execute it
  - Often time part of this is split off into the “loader” and libraries
- Programs include:
  - Machine instructions
  - Initialized data
  - List of library dependencies
  - List of memory sections
  - List of values determined at load time

Executable file formats

- Number of formats
  - ELF – Executable Link File
    - Used on most *NIX systems
  - COFF – Common Object File Format
    - Windoze
  - a.out – Used in BSD (Berkeley Standard Distribution) and early UNIX
    - Not usually used anymore
- BSD UNIX and AT&T UNIX are predecessors to modern *NIXes

ELF

- File header
  - Magic number
  - Version
  - Target ABI
  - ISA
  - Entry point
  - Pointers to
    - Program header
    - Section header
  - etc

Program header

- How to create the process image
  - Segments
  - Types
  - Flags
  - File offset
  - Virtual address
  - Size in file
  - Size in memory

Section header

- Type (data, string, notes, etc
- Flags (writable, executable, etc
- Virtual address
- Offset in file image
- Size
- Alignment

- readelf –headers /bin/ls
- objdump -p, -h, -t

Building a program

- Start with source code
  - hello.c
- Prepocessor
- Compiler
- Assembler
- Linker

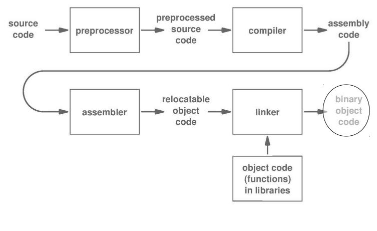


Figure 4.6 The steps used to translate a source program to the binary object code representation used by a processor.

Preprocessor

- When a .c file is compiled, it is first scanned and modified by a preprocessor before being handed to the real compiler
- Finds lines beginning with #, hides them from the compiler, or takes some action
- #include, #define
- #ifdef, #else, #endif

- Can do math
  - #if (FLAG % 4 == 0) || (FLAG == 13)
- Macros
  - #define INC(x) x+1
  - No semi-colon
  - Have to be careful
    - #define ABS(x) x < 0 ? -x : x
    - ABS(B+C)

- Parentheses around substitution variables

  #define ABS(x) ((x) < 0 ? -(x) : (x))

Why macros?

- Run time efficiency
  - No function call overhead
- Passed arguments can be any type
  - #define MAX(x,y) ((x) > (y) ? (x) : (y))
  - Works with ints, floats, doubles, even chars

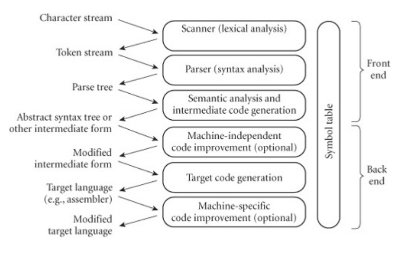
Lots of other tricks

printf(“The date is %s\n”, __DATE__);

- Most preprocessor features are used for large/advanced software development practices

- gcc -E

Compiler?


Character stream
Token stream
Parse tree
Abstract syntax tree or other intermediate form
Modified intermediate form
Target language (e.g., assembler)
Modified target language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis and intermediate code generation
Machine-independent code improvement (optional)
Target code generation
Machine-specific code improvement (optional)
Symbol table
Front end
Back end

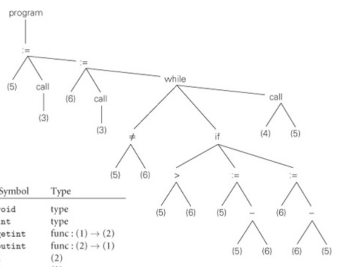
* http://www.cs.montana.edu/~david.watson5/

```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

Compiler?

- Scanning or lexical analysis
  - Groups program into tokens
    - Token = smallest meaningful unit of a program
- Parsing or syntax analysis
  - Create a parse tree
  - Shows how tokens “fit together”
  - Context-free grammar

- Semantic analysis
  - Determines/discovers meaning
  - Builds symbol table
  - Builds syntax tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

- Code generation
  - Traverse symbol table and syntax tree
  - Generate loads, stores, arithmetic ops, tests, branches, etc

- gcc -c
  nm -v

Assembler

- Discussed in architecture lecture
- gcc -S

Libraries

- Libraries are just collections of object files
  - Internal symbols are indexed for fast lookup by the linker
- Searched for symbols that aren't defined in the program
  - Symbol found, pull it into executable (static)
  - Otherwise include a pointer to the file, loaded by loader

Statically linked

- Faster, to a degree
- Portable
- Larger binaries
- Fixed version, no updates

Dynamically linked

- More complexity
- Easy to upgrade libraries
  - Vulnerabilities
- Have to manage versions
- Loader re-links every time program is executed

```
readelf --dynamic /bin/ls
ldd /bin/ls
```

Interpreter

```
readelf --headers /bin/ls
```

Lazy binding

- Binding a function call to a library can be expensive
  - Have to go through code and replace the symbol with its address
- Delay until the call actually takes place
  - Calls stub PLT function
  - Invokes dynamic linker to load the function into memory and obtain real address
    - Rewrites address that the sub code references
    - Only happens once
- Procedure Lookup Table (PLT)

- gcc -o
  nm

Makefile

- Simple way to help organize code compilation

  gcc -o hello hello.c somefunc.c -I.

```
hello: hello.c hellofunc.c
    gcc -o hello hello.c hellofunc.c -I.
```

Or...

```
CC=gcc
CFLAGS=-I.

hello: hello.o hellofunc.o
    $(CC) -o hello hello.o hellofunc.o -I.
```

```
CC=gcc
CFLAGS=-I.
DEPS = hello.h

%.o: %.c $(DEPS)
$(CC) -c -o $@ $< $(CFLAGS)

hello: hello.o hellofunc.o
    gcc -o hello hellomake.o hellofunc.o -I.
```

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
    gcc -o $@ $^ $(CFLAGS)
```

```
▪ IDIR =../include
▪ CC=gcc
▪ CFLAGS=-I$(IDIR)
▪
▪ ODIR=obj
▪ LDIR =../lib
▪
▪ LIBS=-lm
▪
▪ _DEPS = hellomake.h
▪ DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
▪
▪ _OBJ = hellomake.o hellofunc.o
▪ OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
▪
▪
▪ $(ODIR)/%.o: %.c $(DEPS)
▪ $(CC) -c -o $@ $< $(CFLAGS)
▪
▪ hellomake: $(OBJ)
▪ gcc -o $@ $^ $(CFLAGS) $(LIBS)
```

```
.PHONY: clean

clean:
    rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Questions?