UC-SLS Lecuture 17 : In to the Light - C Intro

Contents

- create a directory `mkdir cintro; cd cintro`
- copy examples
- add a `Makefile` to automate assembling and linking
  - we are going run the commands by hand this time to highlight the details
- add our `setup.gdb` to make working in gdb easier
- normally you would want to track everything in git

```
$ ls /home/jovyan/cintro
Makefile   hello.s  misc.h     setup.gdb
csumit1.c  loop.c   myfunc0.c  usecsumit1.S
$
```

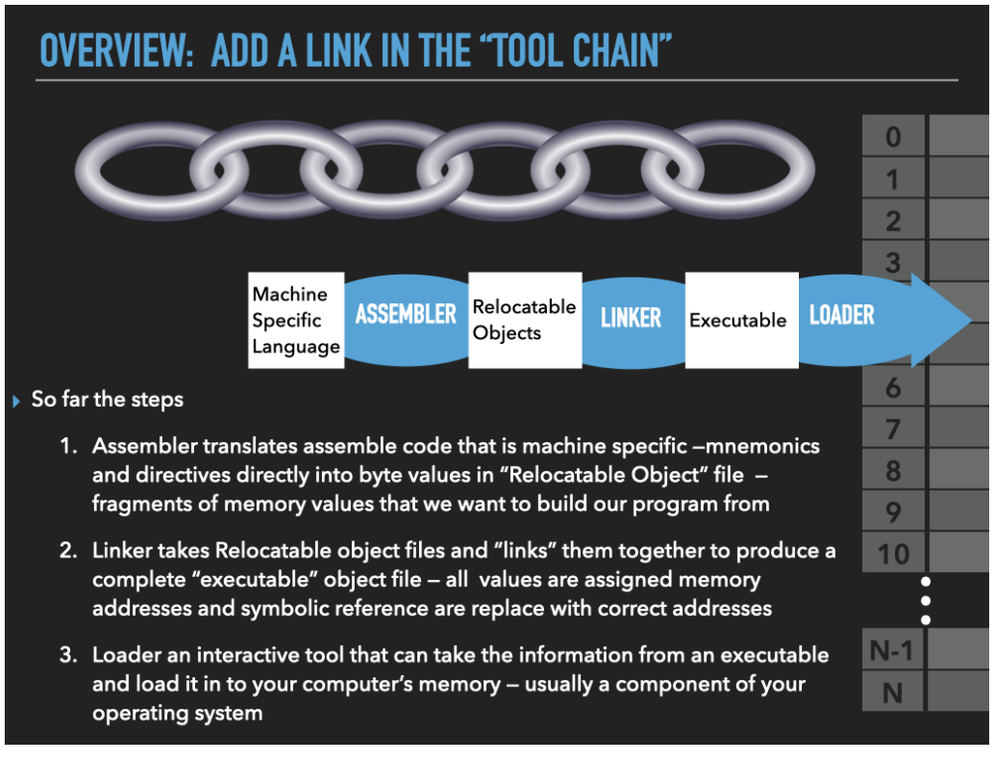
17.1. What are some the downsides of Assembly Programming?

1. The human burden of assembly programming and programmer lock in
2. The code we write is locked to a specific CPU
3. The code we write is locked to a specific OS

17.2. High level languages

- downsides?
- lets look at python "Hello World"
  - How many instructions for our "hello.s" : < 10 instructions
  - `python -c print("Hello World")`: 10's, 100's, 1000's, 10,000, 100,000's, 1,000,000's, > 10,000,000's ???
  - gdb
    - `display /1i $pc`
    - `starti print("Hello World")`
    - `while 1`
    - `  stepi`
    - `end`
  - https://github.com/python/cpython

17.3. The ToolChain

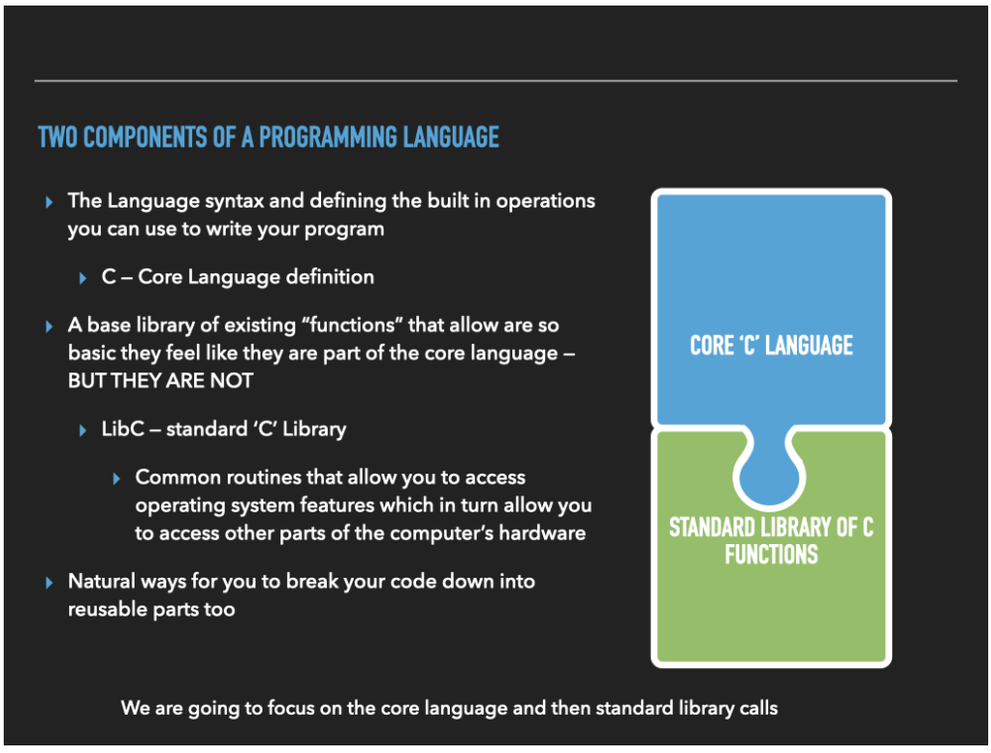


OVERVIEW: ADD A LINK IN THE "TOOL CHAIN"

- The COMPILER –
  - Takes Program Source Files – Description of operations and Data types conforming to a **machine independent** language standard and translates it to the assembly language specific to a particular CPU
- Today the most common base compiled language for a new system is 'C'

OVERVIEW: ADD A LINK IN THE "TOOL CHAIN"

- Most other languages and software is layered on top of this chain either directly or indirectly
  - Operating systems are often written in 'C'
  - Python Interpreter is written in C or C++
  - Java Virtual Machine written in C or C++
  - Other languages have compilers that produce C
- In some sense 'C' has become the "assembly" code for many other pieces of software



17.4. ToolChain in action: A simple C version of sumit

CODE: csumit1.c

```
long long XARRAY[1024];

long long  sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

CODE: usecsumit1.S

```
        .intel_syntax noprefix

        .global _start
_start:
        call sumit
        mov rdi, 0      # rdi = 0 = exit return value
        mov rax, 60     # rax = 60 = exit syscall num
        syscall         # exit(0)
```

17.4.1. Run Assembler on usecsumit1.S

****Use assembler as expected to** `usecsumit1.S` → `usecsumit1.o`

```
as -g  usecsumit1.S -o usecsumit1.o
```

```
$ ls -l usecsumit1.o
-rw-r--r--. 1 jovyan root 2016 Sep 29 16:40 usecsumit1.o
```

17.4.2. Run Compiler on csumit.c

A New Step - compile for `csumit1.c` → `csumit1.s`

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel csumit1.c -o
csumit1.s
```

```
$ ls -l csumit1.s
-rw-r--r--. 1 jovyan root 348 Sep 29 16:40 csumit1.s
```

CODE: csumit1.c

```
long long XARRAY[1024];

long long  sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

CODE: csumit1.s

```
        .file   "csumit1.c"
        .intel_syntax noprefix
        .text
        .globl  sumit
        .type   sumit, @function
sumit:
        xor     r8d, r8d
        xor     eax, eax
.L2:
        add     r8, QWORD PTR XARRAY[0+rax*8]
        inc     rax
        cmp     rax, 10
        jne     .L2
        mov     rax, r8
        ret
        .size   sumit, .-sumit
        .comm   XARRAY,8192,32
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

17.4.3. Run Assembler on csumit1.s

Now same old same old for `csumit1.s` → `csumit1.o`

```
as -g  csumit1.s -o csumit1.o
```

```
$ ls -l csumit1.o
-rw-r--r--. 1 jovyan root 2304 Sep 29 16:40 csumit1.o
```

17.4.4. Linking it all together

Use Linker to create executable `usecsumit1.o` + `csumit1.o` → `usecsumit1`

```
$ ls -l usecsumit1.* csumit1.*
-rw-r--r--. 1 jovyan root  160 Sep 29 16:40 csumit1.c
-rw-r--r--. 1 jovyan root 2304 Sep 29 16:40 csumit1.o
-rw-r--r--. 1 jovyan root  348 Sep 29 16:40 csumit1.s
-rw-r--r--. 1 jovyan root  183 Sep 29 16:40 usecsumit1.S
-rw-r--r--. 1 jovyan root 2016 Sep 29 16:40 usecsumit1.o
```

```
ld -g  usecsumit1.o csumit1.o -o usecsumit1
```

```
$ ls -l usecsumit1* csumit1.*
-rw-r--r--. 1 jovyan root  160 Sep 29 16:40 csumit1.c
-rw-r--r--. 1 jovyan root 2304 Sep 29 16:40 csumit1.o
-rw-r--r--. 1 jovyan root  348 Sep 29 16:40 csumit1.s
-rwxr-xr-x. 1 jovyan root 5920 Sep 29 16:40 usecsumit1
-rw-r--r--. 1 jovyan root  183 Sep 29 16:40 usecsumit1.S
-rw-r--r--. 1 jovyan root 2016 Sep 29 16:40 usecsumit1.o
```

17.4.5. Run our executable

```
$ ./usecsumit1
```

17.4.5.1. Exercises

- **add io to usecsumit1**
  - **read binary input in from stdin into XARRAY**
  - **add a sum memory variable**
  - **after call to sumit move the result in rax into the sum memory variable**
  - **write value of sum memory variable to stdout**

17.5. Remember `objdump` is another useful tool

- **Let's us work directly with object files both relocatable and executables**
- **Let's us see what's inside and where and what the loader is supposed to do**
- **Has similar capabilities to debugger like** `gdb`
  - **but often easer to use when you just want to look at things and not actually debug/run things.**

```
$ objdump -M intel -S -d usecsumit1

usecsumit1:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
        .intel_syntax noprefix

        .global _start
_start:
        call sumit
  401000:       e8 10 00 00 00          call   401015 <sumit>
        mov rdi, 0      # rdi = 0 = exit return value
  401005:       48 c7 c7 00 00 00 00    mov    rdi,0x0
        mov rax, 60     # rax = 60 = exit syscall num
  40100c:       48 c7 c0 3c 00 00 00    mov    rax,0x3c
        syscall         # exit(0)
  401013:       0f 05                   syscall

0000000000401015 <sumit>:
        .intel_syntax noprefix
        .text
        .globl  sumit
        .type   sumit, @function
sumit:
        xor     r8d, r8d
  401015:       45 31 c0                xor    r8d,r8d
        xor     eax, eax
  401018:       31 c0                   xor    eax,eax
.L2:
        add     r8, QWORD PTR XARRAY[0+rax*8]
  40101a:       4c 03 04 c5 00 20 40    add    r8,QWORD PTR [rax*8+0x402000]
  401021:       00
        inc     rax
  401022:       48 ff c0                inc    rax
        cmp     rax, 10
  401025:       48 83 f8 0a             cmp    rax,0xa
        jne     .L2
  401029:       75 ef                   jne    40101a <sumit+0x5>
        mov     rax, r8
  40102b:       4c 89 c0                mov    rax,r8
        ret
  40102e:       c3                      ret
```

17.5.1. OF COURSE THE DEBUGGER IS STILL OUR BEST FRIEND

- **we can do everything we were doing before**
- **examine memory**
- **list assembly source**
- **disassemble**
- **set break points**
- **But now if we allow the assembler and linker to produce debug info `-g` then**
  - **we can work with C source level**
    - **list C source that corresponds to the opcodes**
    - **set break points via C source lines**
    - **examine C variables with the debugger knowing the correct types**
      - **1, 2, 4 or 8 byte types**
      - **pointers vs variables**
      - **signed versus unsigned**
      - **and support for complex heterogeneous types (we will see this later)**

        Debug

- `gcc --static -g -nostartfiles -nostdlib  csumit1.c usecsumit1.S -o sum`
  - **here we let the compiler driver do all the steps for us**
    - **it creates the necessary "intermediary" files (.s and .o files) and removes then when done**
    - **it runs the "compiler", "assembler" and "linker" for us : use `-v` flags to sees this happend**
  - **in our case since we are not using the C library or "runtime" we suppress their use**
    - **rather we want to provide our own `_start`**
    - **we are just using the compiler to avoid writing all our code in assembly**
- **we can now `list sumit`**
- **set breakpoints on C lines `break 8`**
- `disassemble sumit`
- **print and examine C variables**
  - `p i`
  - `p sum`
  - `p XARRAY`
  - `p XARRAY[0]`
- **ask what the type of a variable is**
  - `whatis XARRAY`

In other words our use of a particular memory location is now clear and explicit

‘C’ — AN AUTOMATED ASSEMBLY PROGRAMMER

Dennis MacAlistair Ritchie
(September 9, 1941 – c. October 12, 2011)

Kenneth Lane Thompson
(born February 4, 1943)

Timeline of language development

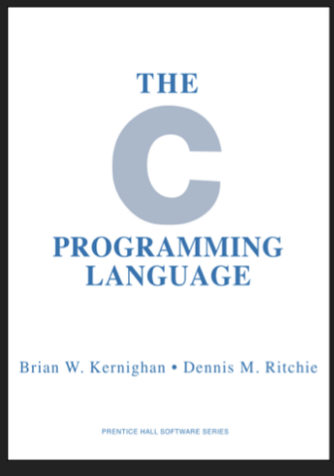
Year	C Standard[9]
1972	Birth
1978	K&R C
1989/1990	ANSI C and ISO C
1999	C99
2011	C11
2017/2018	C18

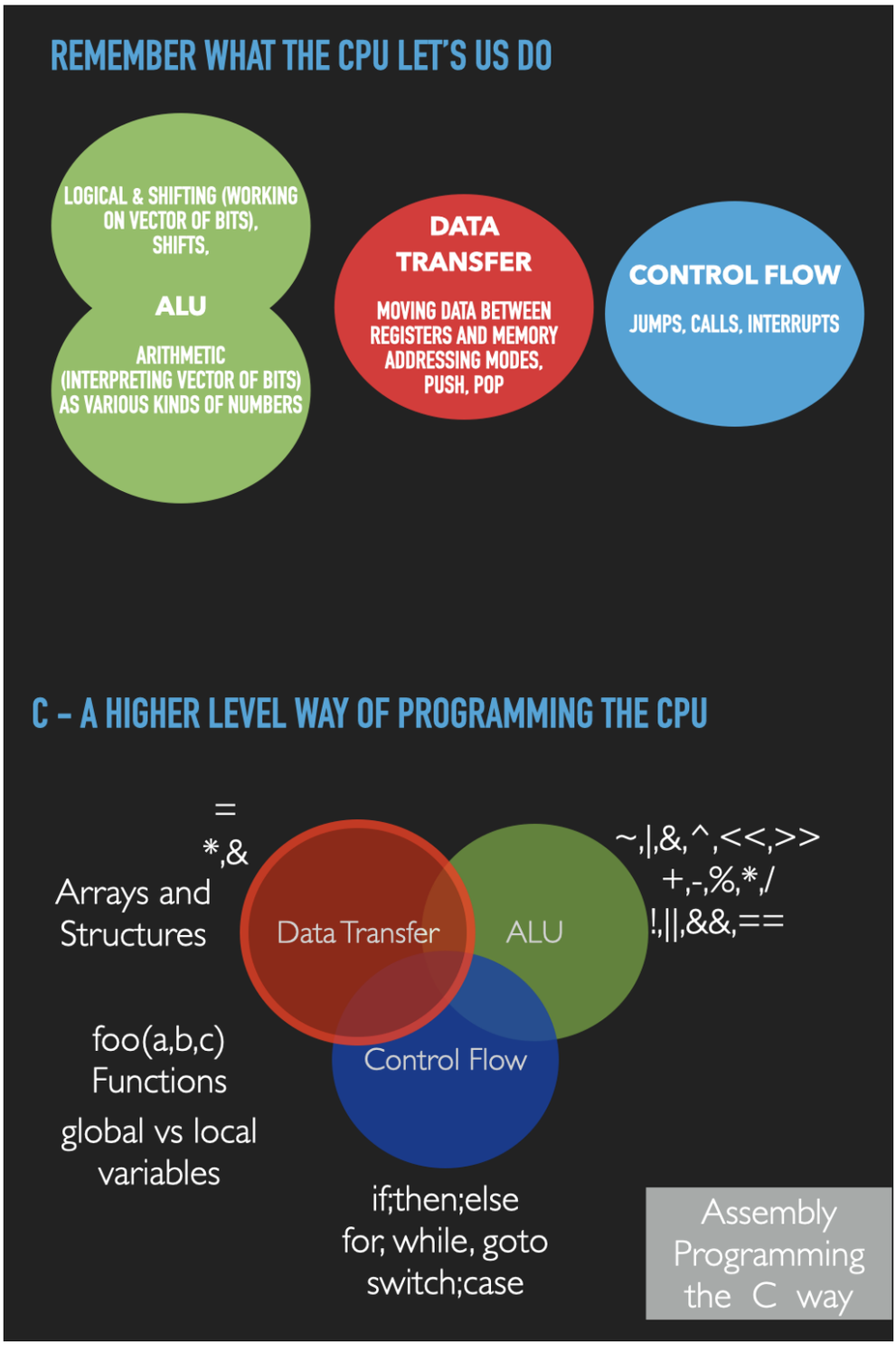
The origin of C is closely tied to the development of the Unix operating system, originally implemented in assembly language on a PDP-7 by Dennis Ritchie and Ken Thompson, incorporating several ideas from colleagues. Eventually, they decided to port the operating system to a PDP-11. The original PDP-11 version of Unix was also developed in assembly language.[10]

At Version 4 Unix, released in November 1973, the Unix kernel was extensively re-implemented in C.[10] By this time, the C language had acquired some powerful features such as `struct` types.

LINUX A DOMINATE UNIX BASE CHILD IS STILL WRITTEN IN C AND ASSEMBLY

Many later languages have borrowed directly or indirectly from C, including C++, C#, Unix's C shell, D, Go, Java, JavaScript (including transpilers), Limbo, LPC, Objective-C, Perl, PHP, Python, Rust, Swift, Verilog and SystemVerilog (hardware description languages).[5] These languages have drawn many of their control structures and other basic features from C. Most of them (Python being a dramatic exception) also express highly similar syntax to C, and they tend to combine the recognizable expression and statement syntax of C with underlying type systems, data models, and semantics that can be radically different.





17.6. Before we can get really get going

17.6.1. Our particular compiler : GCC

https://gcc.gnu.org/onlinedocs/gcc/index.html#SEC_Contents

- **Example of one of the "C" Standards : https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf**
- **is the back bone of the GNU Linux software environment**
- **however it is by no means the only C compiler tool chain**
  - **llvm**
  - **xlc**
  - **Microsoft's C compiler**
  - **and many more**

17.6.2. Controlling the compiler with its command line options

- **The C compiler is a very sophisticated program and has many options that control the assembly code it creates**
- **we are going to use options so that make it easier to read the code it creates**
  - **normally the code it creates does not target human readability**
  - **It usually includes a lot of extra directives to provide the debugger and other tools with more information**
  - **and by default wants to keep compilation fast**

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

17.6.2.1. Optimization level –0

- **by default if no optimization level is given to `gcc` the compiler suite we are using**

- **produces code that is suited for use and manipulation in the debugger**
  - **this code sacrifices efficiency so that it is easy to work at the "C" source level**
- **we will use `-Os` to set the optimization level to `s` which**
  - **does many optimizations but tries to keep the code "small"**

17.6.2.2. Turning off features that we don't need

- **We also going turn off some features designed to exploit features of the Intel processor that make the code more secure**
  - **`-fcf-protection=none`**
- **We are going to turn off certain debugging directives**
  - **`-fno-asynchronous-unwind-tables`**
- **We are going to turn off support for dynamic (load/runtime) relocation and linking**
  - **`-fno-pic` : turn off generation of position independent code**
  - **`-static` : force static linking**

17.6.2.3. We are going to force ourselves to write good code making all warning errors

- **`-Werror`**
- **In general you should never have warning in your code - it is a sign that you don't know what you are doing**

17.6.2.4. Explicitly produce assembly files

To have the compiler driver only preprocess and compile to produce .s files

- **`-S`**

17.6.2.5. Generate intel syntax assembly code

To have it generate intel syntax

- **`-masm=intel`**

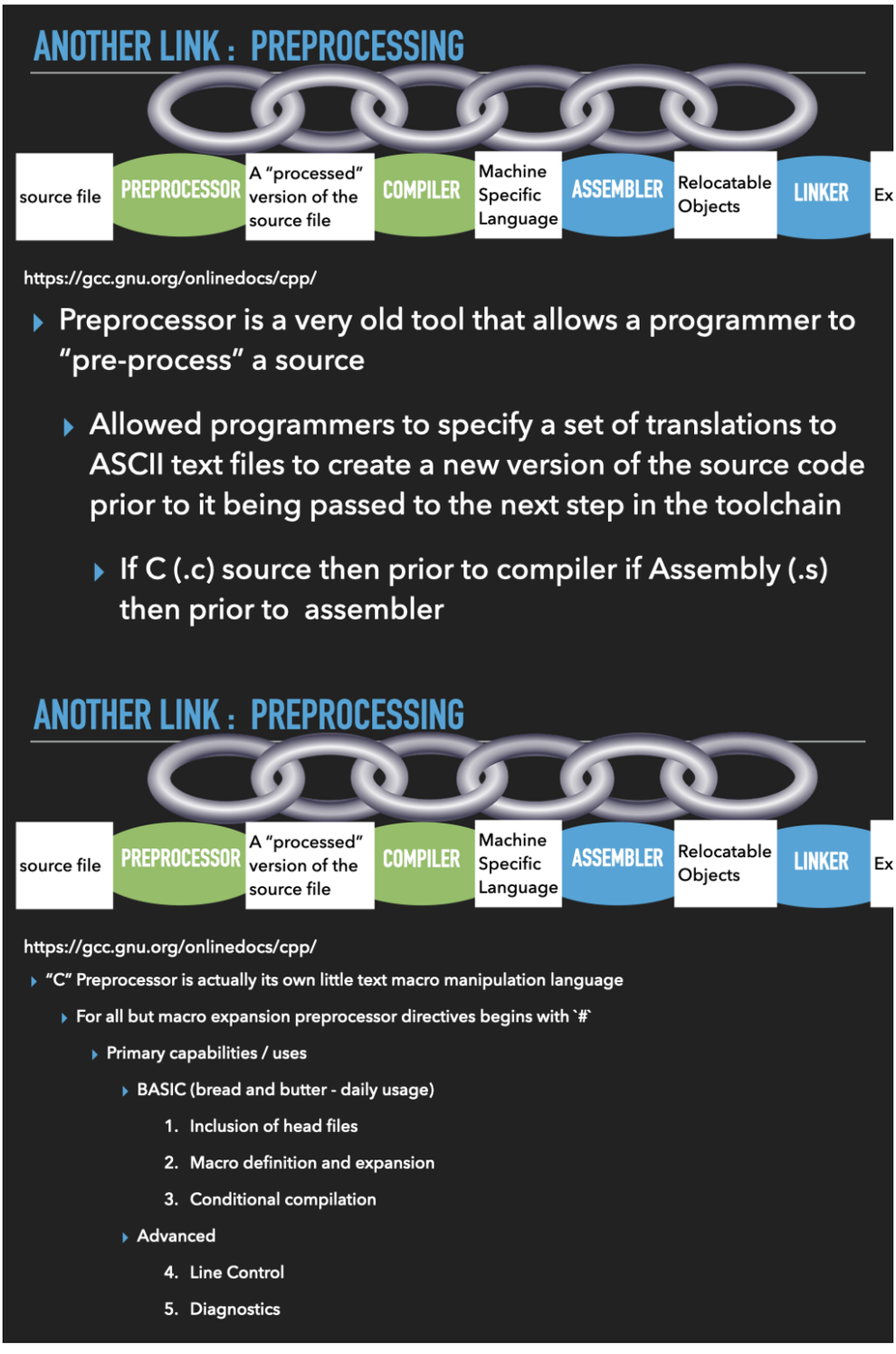
17.6.2.6. Eg.

```
gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel myfunc0.c -o
myfunc0.s
```

17.7. Compiler Driver vs Compiler and "Building"

- **The term compiler is used in multiple ways**
  1. **component of the tool chain that translates C into assembly**
     - **or whatever language it was built for (eg. Fortran, etc)**
  2. **The master command of the the tool chain that**
     - **a meta command that knowns how to invoke each of the components of the tool chain**
     - **passed a default set of options to the components**
       - **you can override these**
     - **default behavior is to try and run all steps of the toolchain to try and create and executable from the files specified**
       - **eg. `gcc my1.c my2.c myasm.S neuralnet.o -o myexe`**
       - **if any errors occurs it will stop and report them from which ever component failed**
       - **it creates all intermediary files as temporaries and removes them once done**
         - **eg. it creates `.s` and `.o` files as it needs too and deletes them when it is done**
         - **you can override this behavior**
           - **`-E` only pre-process**
           - **`-S` pre-process and compile to produce `.s`**
           - **`-c` pre-process, compile and assemble to produce `.o`**
  3. **The most common use is to separated out creating `.o` from linking into an executable eg.**
     - **Your program is composed from a mixture of source files:**
       - **my1.c, my2.c, myasm.S**
       - **for these you would separately "compile" each to produce a corresponding .o**
         1. **`gcc -c my1.c -o my1.o` (runs pre-processor, compiler, and assembler)**
         2. **`gcc -c my2.c -o my2.o` (runs pre-processor, compiler, and assembler)**
         3. **`gcc -c myasm.S -o myasm.o` (runs pre-processor and assembler)**
     - **Your program uses an existing object file from a library that someone gave you**
       - **neuralnet.o**
     - **Now you as a separate step you link all the object files together to produce your binary**
       - **`gcc my1.o my2.o myasm.o neuralnet.o -o myexe`**
     - **You would automate all of this with an Makefile so that when you want to "build" your executable**
       - **make will**
         1. **Only run the necessary steps "compile" steps depending on what source files are newer than their corresponding `.o`**
         2. **Then it will re-link all the `.o`, updated ones and existing ones, to produce a new version of the executable**

17.8. Another Link: The C Preprocessor

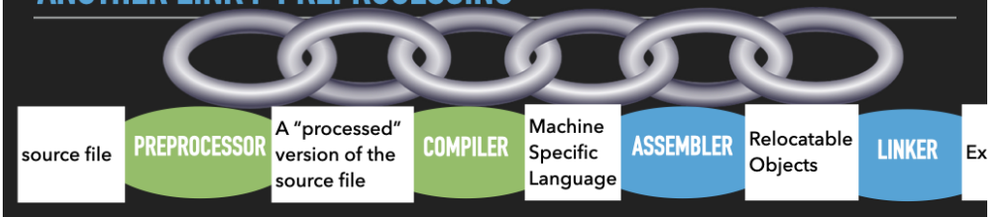


ANOTHER LINK : PREPROCESSING

https://gcc.gnu.org/onlinedocs/cpp/

1. Inclusion of head files
   - "These are files of declarations that can be substituted into your program."
     - substitute the contents of specified file into current file at this point
       - `#include <file>` or `#include "file"` - differs in where to look for the specified file

ANOTHER LINK : PREPROCESSING



https://gcc.gnu.org/onlinedocs/cpp/

2. Macro definition and expansion
   - "You can define *macros*, which are abbreviations for arbitrary fragments of C code. The preprocessor will replace the macros with their definitions throughout the program. Some macros are automatically defined for you."

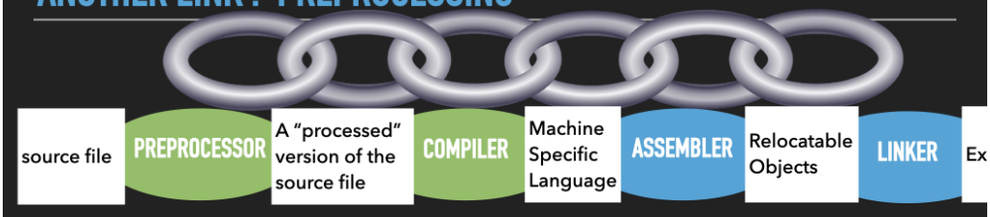
ANOTHER LINK : PREPROCESSING

https://gcc.gnu.org/onlinedocs/cpp/

2. Macro definition and expansion

- `#define DEBUG` - DEBUG is now a defined macro that substitutes to nothing - any place in source file where the string DEBUG occurs it will be substituted with nothing
- #define LEN 10 - LEN is now a defined macro that substitutes to the string 10
- #define GETELEMENT(i) (i>0 && i<LEN) ? Array[i] : exit(-1)
  - Any place that GETELEMENT(value) appears in the source the macro will expand exactly to the string with value substituted for the macro parameter i
  - Very powerful but be careful don't abuse
  - Often its is better to just write real functions and let the compiler take care of things
  - However really useful for debugging code

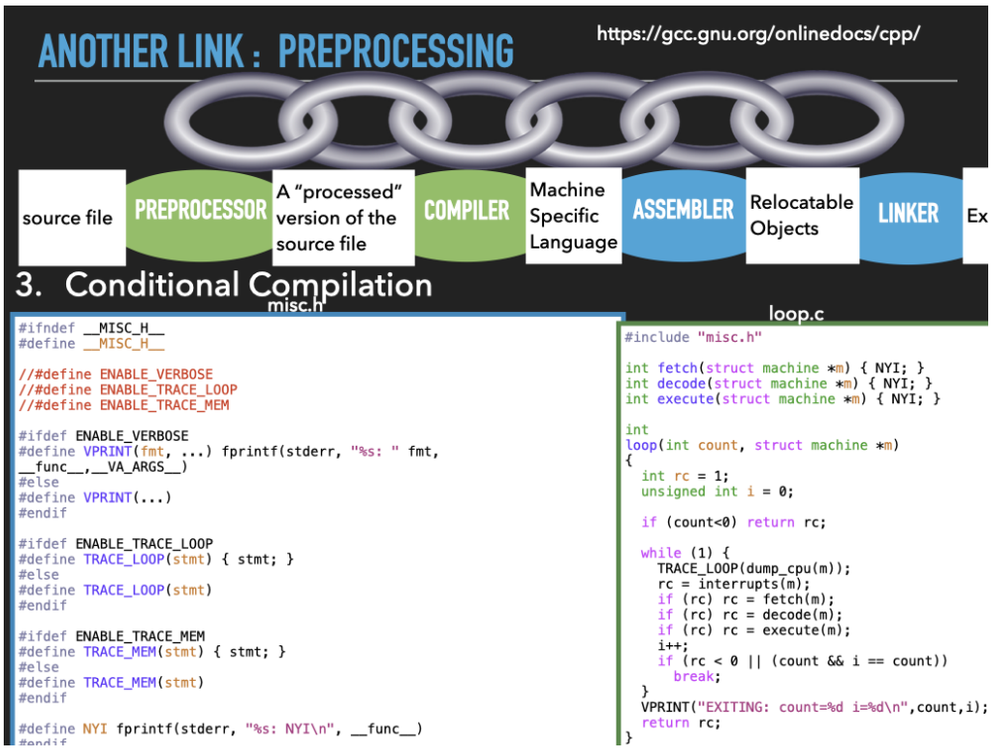
ANOTHER LINK : PREPROCESSING



https://gcc.gnu.org/onlinedocs/cpp/

3. Conditional Compilation

- "You can include or exclude parts of the program according to various conditions"
  - "A *conditional* is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special `defined` operator."



17.8.1. Preprocessor in action

CODE: misc.h

```
#ifndef __MISC_H__
#define __MISC_H__

//#define ENABLE_VERBOSE
//#define ENABLE_TRACE_LOOP
//#define ENABLE_TRACE_MEM

#ifdef ENABLE_VERBOSE
#define VPRINT(fmt, ...) fprintf(stderr, "%s: " fmt, __func__,__VA_ARGS__)
#else
#define VPRINT(...)
#endif

#ifdef ENABLE_TRACE_LOOP
#define TRACE_LOOP(stmt) { stmt; }
#else
#define TRACE_LOOP(stmt)
#endif

#ifdef ENABLE_TRACE_MEM
#define TRACE_MEM(stmt) { stmt; }
#else
#define TRACE_MEM(stmt)
#endif

#define NYI fprintf(stderr, "%s: NYI\n", __func__)
#endif
```

CODE: loop.c

```
#include "misc.h"

int fetch(struct machine *m) { NYI; }
int decode(struct machine *m) { NYI; }
int execute(struct machine *m) { NYI; }

int
loop(int count, struct machine *m)
{
  int rc = 1;
  unsigned int i = 0;

  if (count<0) return rc;

  while (1) {
    TRACE_LOOP(dump_cpu(m));
    rc = interrupts(m);
    if (rc) rc = fetch(m);
    if (rc) rc = decode(m);
    if (rc) rc = execute(m);
    i++;
    if (rc < 0 || (count && i == count))
      break;
  }
  VPRINT("EXITING: count=%d i=%d\n",count,i);
  return rc;
}
```

```
$  gcc -E loop.c
# 1 "loop.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "loop.c"
# 1 "misc.h" 1
# 2 "loop.c" 2

int fetch(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int decode(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int execute(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }

int
loop(int count, struct machine *m)
{
  int rc = 1;
  unsigned int i = 0;

  if (count<0) return rc;

  while (1) {
    ;
    rc = interrupts(m);
    if (rc) rc = fetch(m);
    if (rc) rc = decode(m);
    if (rc) rc = execute(m);
    i++;
    if (rc < 0 || (count && i == count))
      break;
  }
  ;
  return rc;
}
```

- **In some sense the preprocessor lets us "craft" our code into our own creature that can be customized and controlled**
  - **We can now change our code by modifying and defining macros**
    1. **Edit "misc.h"**
    2. **or use ability to define and undefine macros via gcc command line options**
       - **https://gcc.gnu.org/onlinedocs/gcc/Preprocessor-Options.html**

```
-D name
Predefine name as a macro, with definition 1.

-D name=definition
The contents of definition are tokenized and processed as if they appeared during translation phase three in a
'#define' directive. In particular, the definition is truncated by embedded newline characters.

If you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax
to protect characters such as spaces that have a meaning in the shell syntax.

If you wish to define a function-like macro on the command line, write its argument list with surrounding parentheses
before the equals sign (if any). Parentheses are meaningful to most shells, so you should quote the option. With sh and
csh, -D'name(args…)=definition' works.

-D and -U options are processed in the order they are given on the command line. All -imacros file and -include file
options are processed after all -D and -U options.

-U name
Cancel any previous definition of name, either built in or provided with a -D option.
```

```
$  gcc -D ENABLE_TRACE_LOOP -E loop.c
# 1 "loop.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "loop.c"
# 1 "misc.h" 1
# 2 "loop.c" 2

int fetch(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int decode(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int execute(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }

int
loop(int count, struct machine *m)
{
  int rc = 1;
  unsigned int i = 0;

  if (count<0) return rc;

  while (1) {
    { dump_cpu(m); };
    rc = interrupts(m);
    if (rc) rc = fetch(m);
    if (rc) rc = decode(m);
    if (rc) rc = execute(m);
    i++;
    if (rc < 0 || (count && i == count))
      break;
  }
  ;
  return rc;
}
```

```
$  gcc -D ENABLE_VERBOSE -E loop.c
# 1 "loop.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "loop.c"
# 1 "misc.h" 1
# 2 "loop.c" 2

int fetch(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int decode(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int execute(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }

int
loop(int count, struct machine *m)
{
  int rc = 1;
  unsigned int i = 0;

  if (count<0) return rc;

  while (1) {
    ;
    rc = interrupts(m);
    if (rc) rc = fetch(m);
    if (rc) rc = decode(m);
    if (rc) rc = execute(m);
    i++;
    if (rc < 0 || (count && i == count))
      break;
  }
  fprintf(stderr, "%s: " "EXITING: count=%d i=%d\n", __func__,count,i);
  return rc;
}
```

```
$  gcc -D ENABLE_VERBOSE -D ENABLE_TRACE_LOOP -E loop.c
# 1 "loop.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "loop.c"
# 1 "misc.h" 1
# 2 "loop.c" 2

int fetch(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int decode(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }
int execute(struct machine *m) { fprintf(stderr, "%s: NYI\n", __func__); }

int
loop(int count, struct machine *m)
{
  int rc = 1;
  unsigned int i = 0;

  if (count<0) return rc;

  while (1) {
    { dump_cpu(m); };
    rc = interrupts(m);
    if (rc) rc = fetch(m);
    if (rc) rc = decode(m);
    if (rc) rc = execute(m);
    i++;
    if (rc < 0 || (count && i == count))
      break;
  }
  fprintf(stderr, "%s: " "EXITING: count=%d i=%d\n", __func__,count,i);
  return rc;
}
```

By Jonathan Appavoo