The C preprocessor

- The C compiler performs *Macro expansion and directive handling*
  - Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros
- Example:
  - Specify how to expand macros with the DEBUG variable
  - gcc -D option (e.g. gcc –D DEBUG program.c)

```
#ifdef DEBUG
 #define DPRINT(s) fprintf(stderr,”%s\n”,s)
 #else
 #define DPRINT(s)
 #endif
```

The C preprocessor

- The C compiler performs *macro expansion* and *directive handling*
  - Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros
- We will issue the following gcc command
    - **“gcc –D DEBUG fnptr.c”**
    - **We run the executable and compare with the executable produced without the –D option**
- **We will next issue the following command with the –E option to see the output of the preprocessor**
    - **“gcc –D DEBUG fnptr.c –E”**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifdef DEBUG
 #define DPRINT(s) fprintf(stderr,"%s\n",s)
 #else
 #define DPRINT(s)
 #endif



union VAL {int i_val; float f_val;};
struct list {
     union VAL v;
     struct list * next;
  };

typedef struct list List;
```

```
void init_i(union VAL *v){
DPRINT("In function init_i");
v->i_val = 0;
}
void init_f(union VAL *v) {
DPRINT("In function init_i");
v->f_val = 1.0;
}
List *makeGenList (int n, void
(*f)(union VAL * )) {
    List * l, *l1 = NULL; int i;
    for (i = 0; i < n; i++) {
        l = (List*) malloc(sizeof(List));
        (*f)(&(l->v) );
        l->next = l1;
        l1 = l;
    };
    return l;
}
```

Make files

- A makefile is a script for the command "**make**"
- A make file specifies a sequence of commands to be executed (subjected to conditions), to produce a document
  - E.g. to produce an executable binary code
  - Other examples include production of PDF documents, … …
- The "make" command invokes a program that maintains a make library that keeps the time stamps and modification status

A simple makefile

(from GNU make tutorial)

Default goal

A target

prerequisites

Recipe

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
         gcc -o edit main.o kbd.o command.o
display.o \
                   insert.o search.o files.o utils.o

   main.o : main.c defs.h
         gcc -c main.c
   kbd.o : kbd.c defs.h command.h
         gcc -c kbd.c
   command.o : command.c defs.h
command.h
        gcc -c command.c
   display.o : display.c defs.h buffer.h
         gcc -c display.c
```

```
insert.o : insert.c defs.h buffer.h
         gcc -c insert.c
   search.o : search.c defs.h buffer.h
         gcc -c search.c
   files.o : files.c defs.h buffer.h command.h
         gcc -c files.c
   utils.o : utils.c defs.h
         gcc -c utils.c
.PHONY: clean
   clean :
         rm edit main.o kbd.o command.o
display.o \
           insert.o search.o files.o utils.o
```

Empty prerequisite

Nobody's prerequisite
Must be made by "make clean"

Use variables in makefile

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o


edit : $(objects)
          cc -o edit $(objects)
```

Using implicit rules to simplify makefile

```
objects = main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o

  edit : $(objects)
      cc -o edit $(objects)

  main.o : defs.h
  kbd.o : defs.h command.h
  command.o : defs.h command.h
  display.o : defs.h buffer.h
  insert.o : defs.h buffer.h
  search.o : defs.h buffer.h
  files.o : defs.h buffer.h command.h
  utils.o : defs.h

  .PHONY : clean
  clean :
      rm edit $(objects)
```

Some recipes and prerequisites can be deduced by make

GDB: Gnu’s symbolic debugger

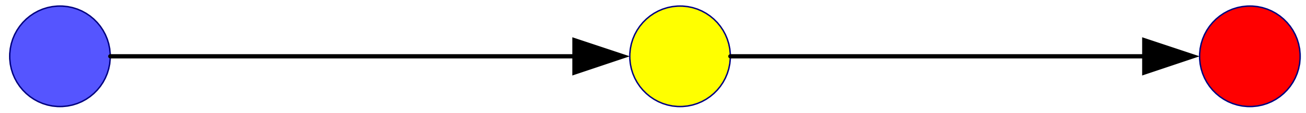
- Debugging can be performed at the assembly code level (i.e. dealing with instructions)
  - When we do not have the source code to recompile, that’s the only option we may have
- At the source level, for C, we can use the –g option when running the GNU C compiler
  - This is for the compiler to insert symbolic information,
    - i.e. information about the source code

  into the executable code such that the execution state can be displayed in terms of the source code

- Next, we enter the gdb environment for debugging the executable program by the command “**gdb <executable file name>**”
- Once in gdb, we can start executing the program by
  - Typing “**run**”, which may be followed by the command arguments if the executable expects any
  - Often, we can set up “break points” such that the program automatically suspends execution at a certain program location

- GDB is a good tool also for understanding the run-time organization of the program
  - How the functions are related to each other in the memory
  - How the program variables are allocated
- We will introduce the most basic gdb commands
  - Students can explore new commands on their own once they get quite familiar with the basic commands

Fault-Error-Failure Model



Fault

Root cause,
e.g. faulty code

Error

Start to misbehave,
intermediate erroneous states,
e.g. erroneous return value

Failure

Affect result visible to users
e.g. undesired output

Lapie'95

Program failures

- There are two kinds of program failures
  - The program aborts, e.g. "**seg fault**"
  - We find the program producing incorrect result, e.g. the print out shows wrong result
  - Before a **programming error** (or "fault") leads to a **failure** at run time, the program may go through a sequence **incorrect states**
  - It is often useful to detect such incorrect states early, by inserting "**assertions**" in the program.
- We first illustrate the use of GDB to debug a program that aborts

- When the program aborts, the debugger will have the failure location (i.e. the instruction location) and the program state (i.e. the values of the program variables and registers) recorded
  - We can inspect the program states
    - This includes the entire stack and the global variables
    - Registers
  - We can also rerun the program and inspect how the program reaches the program state at the time of failure

An example from Midterm1 (strwrite.c)

```
#include <stdio.h>

void xOutChar(char* str){ *str = 'X'; }
int main(){
 char* str = "Hello World";
 char* temp = str + 5;
 xOutChar(temp);
 printf("%s", str);
}
```

a.out

```
Segmentation fault
```

gdb a.out

```
Program received signal SIGSEGV,
Segmentation fault.
0x0000000000400548 in xOutChar
(str=0x400681 " World")
    at strwrite.c:3
3       void xOutChar(char* str){ *str = 'X'; }
```

Understanding “segmentation fault”

- The OS partition the physical address space into a few segments
- The compiler marks on every .o file different *sections,* e.g. .text section, .data section, .ro-data section (read-only data), etc
- The linker combines these sections together and put in the executable binary (e.g. the a.out file)
- The loader, after loading the executable binary to the memory for execution, will set the permission attribute for each section (or segment)
  - disabling “write” for read-only segments
  - disabling “execute” for non-code segments

The memory layout display

- The System Programming course should go into details of linker, loader, etc
  - Here we present some basics to help debugging
- The Unix "readelf" command can show us the memory map of an executable binary (in ELF format, which is the default on our Linux machines), or an object file
- We now examine the output generated by
- "readelf –a strwrite"

Excerpt of *readelf* display

- Entry point address: 0x400450

```
Section Headers:
  [Nr] Name   Type        Address               Size                 Flags
  [0]          NULL       0000000000000000      0000000000000000
[13] .text     PROGBITS   0000000000400450      0000000000000218     AX
[15] .rodata PROGBITS     0000000000400678      0000000000000013     A
[16] .eh_frame_hdr PROGBITS 000000000040068c  000000000000002c     A
[24] .data     PROGBITS   0000000000601010      0000000000000010     WA
```

W (write), A (alloc), X (execute)

0x0000000000400548 in xOutChar **in .text**
(str=0x400681 " World") **in .rodata**

Section to Segment mapping:

- 00
- 01 .interp
- 02 .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
- 03 .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
- 04 .dynamic
- 05 .note.ABI-tag
- 06 .eh_frame_hdr
- 07
- 08 .ctors .dtors .jcr .dynamic .got
- 09

Section	**Description**	
Text (.section .text)	Contain code (instructions). Contain the _start label.	shared among every process running the same binary
Read-Only Data (.section .rodata)	Contains pre-initialized constants.	Constants and string literals
Read-Write Data (.section .data)	Contains pre-initialized variables.	
BSS (.section .bss)	Contains un-initialized global and static data.	

The “**where**” command

- Shows the location (instruction location and the corresponding source code line) of the abort site
- Shows the call chains, including the source code lines of the function calls

```
(gdb) where
#0  0x0000000000400548 in xOutChar
(str=0x400681 " World")
   at strwrite.c:3
#1  0x0000000000400572 in main () at
strwrite.c:7
(gdb)
```

Infinite loop or infinite recursion (infinite.c)

```
include <stdio.h>
int main(){
 int i, j=0;

 for (i=0; i < 1000; i++)
{
       j += i;
       i = 0;
 }

 printf("%d\n", j);
}
```

```
^C
Program received signal SIGINT,
Interrupt.
0x000000000040056c in main () at
infinite.c:6
6       for (i=0; i < 1000; i++) {
(gdb) print i
$1 = 1
(gdb) (gdb) print j
$2 = 1670244998
(gdb) (gdb) step
7               j += i;
(gdb) step
8               i = 0;
```

Example: find the middle number

if x < y < z, m = y

```
#include <stdio.h>
int main () {
int x, y, z, m;
  scanf("%d%d%d\n", &x, &y, &z);
  m = x;
    /* fault-I: should be m = z */
  if(y < z)     /* if_1 */
     if(x < y)  /* if_2 */
        m = y;
     else if(x < z)     /* if_3 */
        m = y;
          /* fault-II: should be m = x */
  else
     if(x > y)  /* if_4 */
      m = y;
   else if(x > z)       /* if_5 */
      m = x;
 printf("Middle number is : %d\n",m);
}
```

Example from "Effective Fault Localization Based on Minimum Debugging Frontier Set", *ACM Conference on Code Generation amd Optimization, 2013, By*

- Feng Li   Wei Huo   Congming Chen   Lujie Zhong   Xiaobing Feng
- Institute of Computing Technology, Chinese Academy of Sciences
-
- Zhiyuan Li
- Purdue University

Breakpoints for debugging

- For the example, we input 1, 4, 8, get output 4 (ok)
- But with input 8, 4, 1, we get output 1 (not ok)
- W/o going into the analysis of the logic, we use the example to illustrate how to set breakpoints and step through the execution

```
(gdb) break main
Breakpoint 1 at 0x400594: file cgo.c, line 4.
 (gdb) run
Starting program: /home/ci/Teach/240/Examples/GDB/cgo 10 1 7
Breakpoint 2, main () at cgo.c:4
4       scanf("%d%d%d\n", &x, &y, &z);
(gdb) next
8 4 1
5         m = x;
(gdb) n
7         if(y < z)     /* if_1 */
(gdb) break cgo.c:18
Breakpoint 2 at 0x400605: file cgo.c, line 18.
(gdb) n
Breakpoint 2, main () at cgo.c:18
18       printf("Middle number is : %d\n",m);
(gdb) print m
$2 = 8
```

Examples of common errors

Null pointer dereference
Use after free
Double free
Array indexing errors
Mismatched array new/delete
Potential stack/heap overrun
Return pointers to local variables
Logically inconsistent code
Uninitialized variables
Invalid use of negative values
Under allocations of dynamic data
Memory leaks
File handle leaks
Unhandled return codes

Assertions

```
int main() {
 int a, b, c;

 a = 10;
 b = some_function_computes_something();

 c = a/b;
 return 0;
}
```

Assertions

```
int main() {
 int a, b, c;

 a = 10;
 b = 1/a;
 assert(b!=0);
 c = a/b;
 return 0;
}
```

Assertions

- Used to help specify programs and to reason about correctness
- precondition
  - an assertion placed at the beginning of a section of code
  - determines the set of states under which the code is expected to be executed
- postcondition
  - placed at the end
  - describes the expected state at the end of execution.
- #include <assert.h>

```
assert (predicate);
```

How can this code fail?

```
  #include <stdio.h>
#define MAX 10
char* f(char s[]);

int main() {
  char str[MAX];
  char *ptr=f(str);
  printf("%c\n", *ptr);
  return 0;
}
```

How can this code fail?

```
#include <stdio.h>
#include <assert.h>
#define MAX 10
char* f(char s[]);

int main() {
  char str[MAX];
  char *ptr=f(str);
  assert(ptr!=NULL);
  printf("%c\n", *ptr);
}
char *f(char s[]) {
  return NULL;
}
```