# CSC / CPE 357

Systems Programming

# The GNU Debugger (gdb)

- A debugger for several languages, including C and C++
- Allows you to inspect what a program is doing as it executes.
- Errors like segmentation faults may be easier to resolve with the help of gdb
- Online manual: http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

# Compiling for GDB

Must compile with the `-g` option to enable debugging support (which gdb needs):

```
gcc [other flags] -g <source files> -o <output file>
```

# Starting gdb

- From the UNIX/Linux shell, prefix an executable name with `gdb` to start a debugging session:

- `gdb ./a.out`

- This starts the `gdb` shell, which has history, tab completion, arrow keys to recall commands, and a built in `help` command

# Running Your Code

- To run the program, use:
- `(gdb) run`
- If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
- If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error.

# Breakpoints

- Breakpoints can be used to pause your running program at a designated point.

- The command "`break`" sets a breakpoint at a specified file-line pair

- `(gdb) break program.c:23`
  - Sets a breakpoint at line 23, of `program.c`. When the running program reaches that location, gdb will pause and prompt you for another command.
- `(gdb) break my_function`
  - Break any time the function `my_function` is called

# Next vs Step

On a line of code that has a function call, `next` will treat the function call as a single line of code, while `step` will run each line within the function.

The `next` command:

```
(gdb)
11      my_function();
(gdb) next
12 }
```

The `step` command:

```
(gdb)
11      my_function();
(gdb) step
my_function() at program.c:5
5       return 0;
(gdb)
```

# Printing / Watching Variables

- The `print` command prints the value of the variable specified, and `print/x` prints the value in hexadecimal:
  - ```
    (gdb) print i
    (gdb) print/x i
    ```

- **watchpoints** pause the program whenever a watched variable's value is modified:
  - `(gdb) watch i`
  - Whenever `i`'s value is modified, gdb will pause and print out the old and new values
  - Subject to variable *scope* at the time you create watchpoint (eg. if more than one variable `i` declared in your code)
    ```

# Examining Memory

The command `x` allows you to examine the current contents of memory

- `x addr`
  - Example: `x 0x54320`
- `x/nfu addr`
  - `n`, the repeat count (how much memory to display)
  - `f`, the display format (default is x for hexadecimal, also: **`s`**tring, **`i`**nstructions, etc.)
  - `u`, the unit size (**`b`**ytes, **`h`**alf-words; two bytes, **`w`**ords; four bytes, **`g`**iant bytes, eight bytes)
  - Examples:
  - `x/3uh 0x54320` - Display three halfwords (`h`) of memory, formatted as unsigned decimal integers (`u`), starting at address `0x54320`
  - `x/4xw $sp` - Print the four words (`w`) of memory above the stack pointer (`$sp`)

# Several Useful gdb Commands

- `list` - display lines of code above and below the line the program is stopped at
- `backtrace` - display stack trace of the function calls that lead to an error (should remind you of Java exceptions)
- `where` - stacktrace (nested function calls) at any stage in execution
- `finish` - runs until the current function is finished
- `delete` - deletes a specified breakpoint
- `info breakpoints` - shows information about all declared breakpoints

# The C Preprocessor

- C provides certain language facilities by means of a **preprocessor**
  - Separate first step in compilation

- The two most frequently used features:
  - `#include` - include the contents of a file during compilation,
  - `#define` - replace a token by an arbitrary sequence of characters.

- Other features include conditional compilation and macros with arguments.

# File Inclusion

- To include the contents of a separate file:
  - `#include "filename"` - file in same directory as source file
  - `#include "directory/filename"` - file in directory relative to source file
  - `#include <filename>` - use files from system-defined locations

- Included file may itself contain `#include` lines

- When an included file is changed, all files that depend on it must be recompiled.

# Macro Substitution

- `#define name replacement text`
  - All occurrences of `name` will be replaced by the replacement text.
  - Multi-line definition supported with `\` at the end of each line
- Replacement active from point of definition to the end of the source file being compiled.
- Definition may use previous definitions
- Substitutions are made only for "tokens," and do not take place within quoted strings.
  - For example:

```
#define SIZE 100
printf("SIZE is %d\n", %p); // no substitution
#define NEWSIZE 150  // no substitution of SIZE
```

# Macros with Arguments

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Any use of `max` expands before compilation (avoids overhead of function call)

```
x = max(p+q, r+s); // expands to: x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Caution:

```
max(i++, j++)  // what is the result?
```

```
#define square(x) x * x
square(z+1)    // expansion?
```

# String Concatenation

# prefix causes expansion into a quoted string. Example:

```
#define dprint(expr) printf(#expr " = %g\n", expr)


dprint(x/y)
```

expands to:

```
    printf("x/y" " = &g\n", x/y);
```

Strings are concatenated automatically, end result is:

```
    printf("x/y = &g\n", x/y);
```

# Conditional Preprocessing

Preprocessing can be controlled with `#if` / `#else` conditions

For example, to make sure that the contents of a file `hdr.h` are included only once:

```
#if !defined(HDR)
#define HDR

/* contents of hdr.h go here */

#endif


// shorthand for first line: #ifndef HDR
```

# Multi-File Compilation

- As projects grow, it can be very useful to split declarations and code into separate files
  - Clear project structure
  - Re-use declarations of shared functions

- Example:
  - `program.c` - main function
  - `helper_code.c` - helper functions
  - `program.h` - declarations

# Multi-File Compilation

- Compile by listing all `.c` files on `gcc` command line:
  - `gcc -o program program.c helper_code.c`
  - (`.h` files are automatically located and included by the compiler)
- Tedious
- Time-consuming
  - All code is recompiled, even if changes affect only some components

# `make`

- UNIX/Linux systems commonly use the `make` build utility
- Make tracks changes to files and dependencies between files to simplify and speed up compile process
- Relies on a `Makefile`
  - Text file that specifies rules used to build code

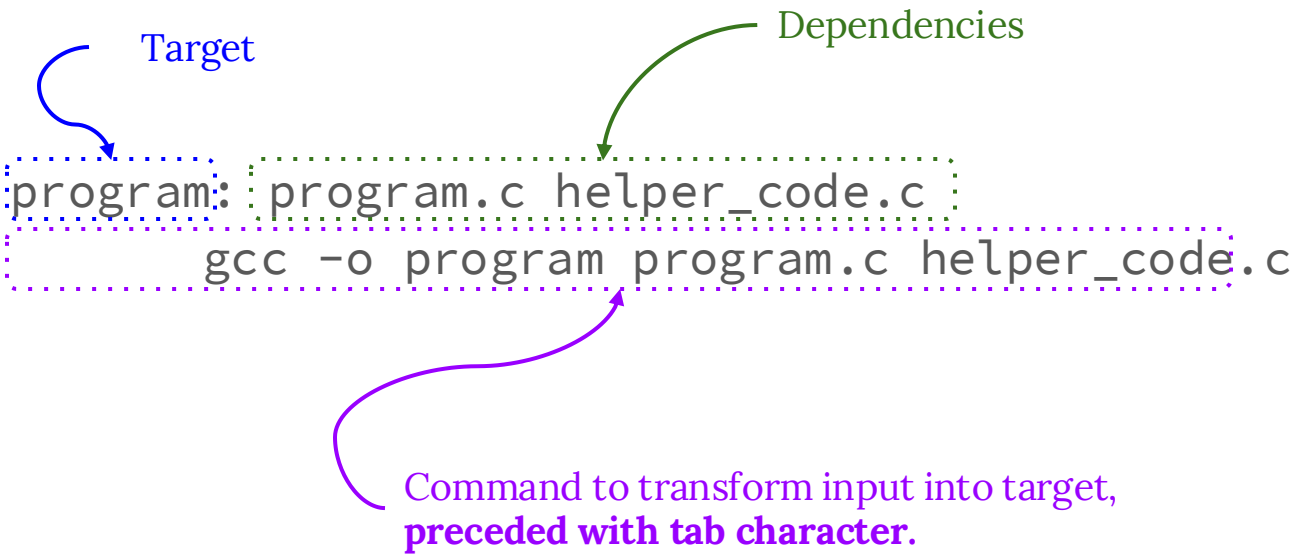
# Makefile

- A Makefile consists of a sequence of rules.

- Each rule specifies:
  - a target
  - enumeration of files or other targets on which the target depends
  - commands which define how to transform (ie. compile) the components into the target

# Makefile

- Example `Makefile`:



# Compiler Options (gcc)

- Thus far, we have mainly used `gcc` in its "default" mode
  - `-g` to compile for debugging
- Several other options we will use:

| | |
|---|---|
| `-o out_file` | control the name of the compiled binary file (versus default of a.out) |
| `-Wall` | turn on all the most commonly-used compiler warnings |
| `-ansi -pedantic` | report warnings about violations of the ANSI/ISO standard |
| `-std=gnu99` | use the GNU / ISO C99 language standard |

```
gcc -o hello -Wall -ansi -pedantic -std=gnu99 hello.c
```