CSCI [4|6]730:
A C Refresher or Introduction

Hello Word!

`~/ctutorial/`

How to learn C?

In addition to syntax you need to learn:

- **the Tools.**
- **the Libraries.**
- **And the Documentation (how to access)**

- **Practice on cluster nodes on nike.**
  - » **vcf0 – vcf5 on nike.cs.uga.edu**

```
{vcf5:maria:85} ssh vcf4.cs.uga.edu
Are you sure you want to continue connecting (yes/no)? Yes
maria@vcf4.cs.uga.edu's password:
```


Diving In: A Simple C Program
1-hello-word.c

```
/* header files go up here -- specifies headers needed for routines */
/* note that C comments are enclosed within a slash and a star, and may
   wrap over lines */
// but if you use the latest gcc, two slashes  to end of line will
// work too, like C++
#include <stdio.h>  /* prototypes processed by cpp */

// a comment about comments single line doesn't always work
/*    multiline comment should work everywhere …
*/

/* main returns an integer */
int main( int argc, char *argv[] )
{
printf( "hello, world\n" );
return(0);   /* returns 0 by conventions indicates all went well */
}
```

declarations

functions()

main()

Diving In: A Simple C Program
1-hello-word.c

```
/* header files go up here -- specifies headers needed for routines */
/* note that C comments are enclosed within a slash
and a star, and may wrap over lines */
// but if you use the latest gcc, two slashes will work too, like C++
#include <stdio.h>  /* prototypes processed by cpp */

/* main returns an integer */
int main( int argc, char *argv[] )
{
/* printf is our output function; by default it writes to standard out */
/* printf returns an integer, but we ignore it here */
/*1 [stout]   >& redirect stout and stderr */
/* >& /dev/null - suppress all output */ /*(cat f1 > myout) >& myerror */

printf( "hello, world\n" );
/* return 0 by conventions indicates all went well */
return(0);
}
```

declarations

functions()

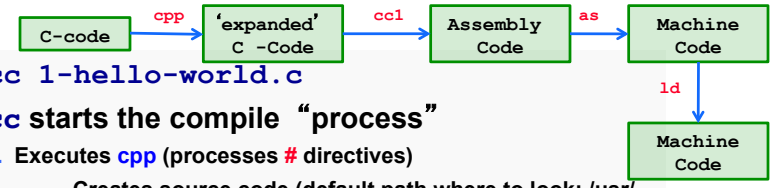
main()

*.c File Name

- Naming the program (e.g., 1-hello-world.c, main.c)
  - » Arbitrary – Not Like in Java where file name is connected with file content (class name).
  - » Constraint: Need to end with a *.c'
- Coming up: compiling (4.9.0)
  - » /usr/bin/gcc
  - » gcc –v

```
{nike:maria:21} gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/local/gcc/4.9.0/libexec/gcc/x86_64-unknown-linux-gnu/4.9.0/lto-wrapper
Target: x86_64-unknown-linux-gnu
Configured with: ../gcc-4.9.0/configure --prefix=/usr/local/gcc/4.9.0 --enable-languages=c,c++
Thread model: posix
gcc version 4.9.0 (GCC)
```


How to Compile and Run a C-program: 1-hello-world.c



- gcc 1-hello-world.c
- gcc starts the compile "process"
  1. Executes cpp (processes # directives)
     - Creates source code (default path where to look: /usr/include). Expands the code.
  2. Compilation (cc1): gcc -print-prog-name=cc1
     - Transforms C code to assembly code (.s code)
  3. Assembler (as) runs
     - Transforms assembly code to machine code
  4. Linker (ld) runs
     - Links code together to create the final executable
- ./a.out
  - » int main(int argc, char *argv[])

1

argv[1]=NULL (end)

- cpp 1-hello-world.c > 1-hello-world.cpp
- cc1 1-hello-world.cpp
  - » Creates: 1-hello-world.s
- as -o 1-hello-world.o 1-hello-world.s
- ld tricky to use for linking.
  - » gcc 1-hello-world.o -o 1-hello-world
  - » ld 1-hello-world.o -lc —entry main -l /lib/ld-linux.so.2

Compile Command line & 'flags'

- prompt> gcc -o first first.c # -o lets you specify the executable name
- prompt> gcc -Wall first.c # -Wall gives much better warnings
- prompt> gcc -g first.c # use -g to enable debugging with gdb
- prompt> gcc -O first.c # use -O to turn on optimization
- strip.

find . -name first-print

Linking Libraries

- Example: `fork()` requires a library, namely the C-library. The C library is *automatically* linked, so all we need then is :
  - » The 'including' the right `#include` file "<>", -i, -I to to find the prototype of the function (return type, date types of parameters).
  - » How to find out:
    - `man fork` `//[unistd.h]`
  - » CAVEAT: the controversial and dreaded `LD_LIBRARY_PATH`
  - » `(C11)`
  - » May fix (e.g., readline) problems

Lets say that again….

- fork() requires the C-library (`clib`). The C library is *automatically* linked in, so all we need then is :
  - » How do you know what to include?
  - » `man fork`
  - » BUT – Wait a minute why a library - Fork is a system call! [a request of 'service' by the OS from the application]
    - – `C library provides`
      - `C –wrappers for all system calls – which simply traps into the OS`
    - – The 'real' system call in Linux e.g., `is sys_fork()`

Other Libraries: The Math Library (e.g., sqrt(), pow())

- gcc [flag ...] file ... -lm [library ...]
- #include <math.h>
  - » In /usr/lib
  - » Statically linked .a (compile time)
    - – Combines code (copies) directly into executable
  - » Dynamically linked shared library .so (run time)
    - – Smaller code base (can be shared by multiple processes)
    - – A reference and only links when needed, smaller code base (some work), hooks in code triggers the run time system to load in the library, only when needed
  - » /usr/libm.a & /usr/libm.so (different location per installation)
  - » Link editor searches for library in a certain order.
  - » -lm directory path include) and –L(directory path)

Multiple Files (`hw.c, helper.c Makefile2`)

```
prompt> gcc –o hw hw.c helper.c -lm
```

Problem: Remake everything (2 programs here) every time, even if the change is only in hw.c

Approach: Separate 2 step compilation process that only re-compiles source files that

have been modified

- Create object files then link *.o files
- Then link these files into an executable

Separate Compilation

```
# note that we are using -Wall for
   warnings and -O for optimization
prompt> gcc -Wall -O -c hw.c
prompt> gcc -Wall -O -c helper.c
prompt> gcc -o hw hw.o helper.o -lm
```

- -c flag produces an object file
  - Machine level code (not executable)
  - Need to link to make an executable

```
prompt> gcc -o hw hw.c helper.c -lm
```

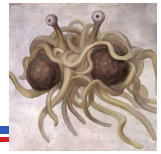

Make & Makefiles

- Make make things easier to handle the compilation process.

```
target: prerequisite1 prerequisite2
  command1
  command2
```

- Target usually the name of executable of
  1) the program file (e.g., helloworld) or
  2) the action (e.g., clean).
- command1/command2 … preceeded by a tab \t

Make – Makefiles (be aware of the dreaded white space phenomena)



```
hw: hw.o helper.o
   gcc -o hw hw.o helper.o -lm
hw.o: hw.c
  gcc -O -Wall -c hw.c
helper.o: helper.c
  gcc -O -Wall -c helper.c
clean:
  rm -f hw.o helper.o hw
```

- vi – enables you to see white spaces by ('escape' then) :list or :set list

What is going on here?

```
hw: hw.o helper.o
    gcc -o hw hw.o helper.o -lm
hw.o: hw.c
  gcc -O -Wall -c hw.c
helper.o: helper.c
  gcc -O -Wall -c helper.c
clean:
  rm -f hw.o helper.o hw
```

- Goes to target hw (first target) need the prerequisites
- Check them in turn (according to times stamp) and see if they need to be re-made

Make macros

- Make facilitates macros (short cuts):
  - » CC = gcc
  - » OBJECTS = data.o main.o
  - » Project1: $(OBJECTS)
- Examples of Special macros
  - » CC, CFLAGS (compiler, and compiler flags)
  - » $@ short cut for full name of current target

```
%.o : %.c
      $(CC) –c –o $@ $(CFLAGS)
```


Advice Getting Started with Makefile

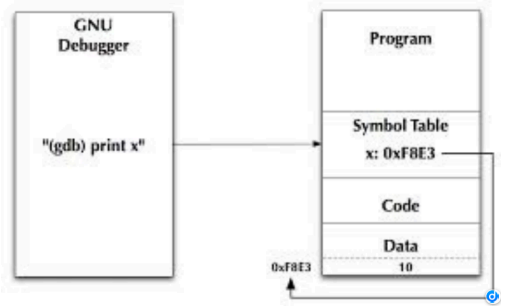
- Start by using a working Makefile (a template), then edit the file how you like it.
- Be aware of white spaces
- Be aware of white spaces
- Be aware of white spaces

Debugger

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

- The debugger: it is another program
- Gnu Debugger (GDB)
- `gcc -g -o program program.c`
  - » `-g "make the executable debuggable"`
- `-g'ing gcc.`
  - » Symbol Table
    - – List of "names" of identifier is accessible to the debugger
      - type, scope, and location of identifiers (e.g., variables and functions)
  - » Prevent the compiler from re-arranging (optimizing) the code

Debugger



- Visualization

UNIX Tools: Examining Obj Files

- nm - list symbols from object file
  - » http://linux.about.com/library/cmd/blcmdl1_nm.htm
  - » nm –a symboltable
- objdump – more detailed information
  - » objdump –syms program
- readelf – anoher aproaach

Debugging

```
#include <stdio.h>
struct Data {
int x;
};
int main( int argc, char *argv[] )
{
struct Data *p = NULL;
printf("%d\n", p->x);
}
```


Debugging

- gcc -g -o 3-buggy 3-buggy.c
- {nike:maria:428} 3-buggy
- Segmentation Fault(coredump)
- gdb 3-buggy
  - » run
  - » print p
  - » break main
- https://www.gnu.org/software/gdb/documentation/
- http://sourceware.org/gdb/current/onlinedocs/gdb.html

gdb debug a running program

```
{vcf5:maria:5} 4-longprogram &
{vcf5:maria:5} ps –f | grep maria
maria  3520  3497  0 10:06 pts/1    00:00:00 4-longprogram
maria  3521  3497  5 10:07 pts/1    00:00:00 ps –ef
{vcf5:maria:7} gdb 4-longprogram 3520
(gdb) list
(gdb) next
Single stepping until exit from function __nanosleep_nocancel,
3        for (i=0; i < 10000; i++)
(gdb) print i
$1 = 34
(gdb) next
15              a = a+i; \(gdb) print i
$2 = 36
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb)
```

GDB

- `(gdb) help`
- `Help running`
- `Help files`
- `Help breakpoints`

- `Run, step, next`

A Note : Integrated Development Environments (IDE)

- **Debugger, Editor, Compilation**
- **Eclipse - http://www.eclipse.org/callisto/c-dev.php**
- **Microsoft Visual Studios**

Documentation: Oh Man

- **man XXX**
- **man -k**

The *Ultimate* C Reference Guides

- ***"The C book" or the "K & R Book":***
  - ***The C Programming Language, by Brian Kernighan and Dennis Ritchie (thin, concise and all you really need…)***
- ***The GDB Booklet***
  - ***Debugging with GDB: The GNU Source-Level Debugger, by Richard M. Stallman, Roland H. Pesch***
    - ***http://sourceware.org/gdb/current/onlinedocs/gdb.html***
- ***The Unix System Programming Book***
  - ***Advanced Programming in the UNIX Environment, by W. Richard Stevens***
  - ***There is a later edition…***