Intro to C Programming – Lecture 1

- Introduction to C
- Writing C Programs
- Our First C Program

What is C?

- Dennis Ritchie–AT&T Bell Laboratories–1972
  - 16-bit DEC PDP-11 Computer (right)
- Widely used today
  - extends to newer system architectures
  - efficiency/performance
  - low-level access

Features of C

C features:

- Few keywords
- Structures, unions – compound data types
- Pointers – memory, arrays
- External standard library – I/O, other facilities
- Compiles to native code
- Macro preprocessor

Versions of C

Evolved over the years:

- 1972 – C invented
- 1978 – *The C Programming Language* published; first specification of language
- 1989 – C89 standard (known as ANSI C or Standard C)
- 1990 – ANSI C adopted by ISO, known as C90
- 1999 – C99 standard
  - mostly backward-compatible
  - not completely implemented in many compilers
- 2007 – work on new C standard C1X announced

In this course: ANSI/ISO C (C89/C90)

What is C used for?

Systems programming:

- OSes: like Linux, Windows
- microcontrollers: automobiles and airplanes
- embedded processors: phones, ipods, microwaves, etc.
- DSP processors: digital audio and TV systems
- ...

C vs. related languages

- More recent derivatives: C++, Objective C, C#
- Influenced: Java, Go, Python (quite different)
- C lacks:
  - exceptions
  - range-checking
  - garbage collection
  - object-oriented programming
  - polymorphism
  - ...
- Low-level language ⇒ faster code (usually)

Warning: low-level language!

Inherently unsafe:

- No range checking
- Limited type safety at compile time
- No type checking at runtime

Perfectly legal to shoot yourself in the foot.

Handle with care.

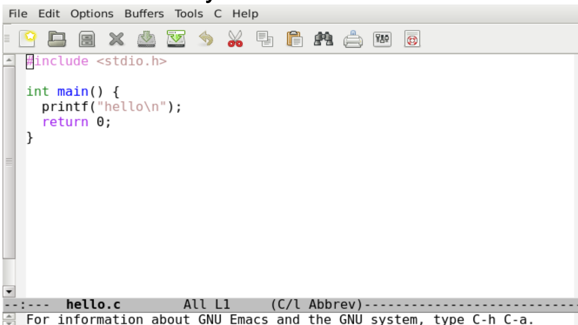
- Always run in a debugger like `gdb` (more later. . .)
- Never run as `root`
- Never run as Administrator

Lecture 1

- Introduction to C
- Writing C Programs
- Our First C Program

Editing C code

- `.c` extension
- Editable directly



- More later. . .

Compile with GCC

- `gcc` (included with most Linux distributions): compiler
- Run `gcc`:

```
prompt% gcc -Wall infilename.c -o
outfilename
```

- `-Wall` enables most compiler warnings
- More complicated forms exist
  - multiple source files
  - auxiliary directories
  - optimization, linking
- Embed debugging info and disable optimization:

```
prompt% gcc -g -O0 -Wall infilename.c -o
outfilename
```

Debugging

```
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/justin/hello...done.
(gdb) list
1       #include <stdio.h>
2
3       int main() {
4         printf("hello\n");
5         return 0;
6       }
(gdb) br 3
Breakpoint 1 at 0x80483ed: file hello.c, line 3.
(gdb) run
Starting program: /home/justin/hello

Breakpoint 1, main () at hello.c:4
4         printf("hello\n");
(gdb) 
```

Figure: `gdb`: command-line debugger

Using gdb

Some useful commands:

- `break` *`linenumber`* – create breakpoint at specified line
- `break` *`file`*`:`*`linenumber`* – create breakpoint at line in file
- `run` – run program
- `c` – continue execution
- `next` – execute next line
- `step` – execute next line or step into function
- `quit` – quit `gdb`
- `print` *`expression`* – print current value of the specified expression
- `help` *`command`* – in-program help
- `list` – list source lines

The IDE – all-in-one solution

- Popular IDEs: Eclipse (CDT), Microsoft Visual C++ (Express Edition), KDevelop, Xcode, . . .
- Integrated editor with compiler, debugger
- Very convenient for larger programs

Lecture 1

- Introduction to C
- Writing C Programs
- Our First C Program

Hello, UNM-LA students

- In style of “Hello, world!”
- `.c` file structure
- Syntax: comments, macros, basic declarations
- The `main()` function and function structure
- Expressions, order-of-operations
- Basic console I/O (`puts()`, etc.)

Structure of a `.c` file

```
/* Begin with comments about file contents */

Insert #include statements and preprocessor
definitions

Function prototypes and variable declarations

Define main() function
{
  Function body
}

Define other function
{
  Function body
}
⋮
```

Comments

- Comments: `/* this is a simple comment */`
- Can span multiple lines

```
/* This comment
   spans
   multiple lines */
```

- Completely ignored by compiler
- Can appear almost anywhere

```
/* hello.c -- our first C program

   Created by Daniel Weller, 01/11/2010 */
```

The `#include` macro

- Header files: constants, functions, other declarations
- **#include** <stdio.h> – read the contents of the *header file* `stdio.h`
- `stdio.h`: standard I/O functions for console, files

```
/* hello.c -- our first C program

   Created by Daniel Weller, 01/11/2010 */

#include <stdio.h> /* basic I/O facilities */
```

More about header files

- `stdio.h` – part of the C Standard Library
  - other important header files: `ctype.h`, `math.h`, `stdlib.h`, `string.h`, `time.h`
  - For the ugly details: visit `http://www.unix.org/single_unix_specification/` (registration required)
- Included files must be on *include path*
  - `-I`*`directory`* with `gcc`: specify additional include directories
  - standard include directories assumed by default
- **#include** `"stdio.h"` – searches `./` for `stdio.h` first

Declaring variables

- Must declare variables before use
- Variable declaration:

```
int n;
float phi;
```

- `int` -integer data type
- `float` -floating-point data type
- Many other types (more next lecture. . .)

Initializing variables

- Uninitialized, variable assumes a default value
- Variables initialized via assignment operator:

```
n = 3;
```

- Can also initialize at declaration:

```
float phi = 1.6180339887;
```

- Can declare/initialize multiple variables at once:

```
int a, b, c =0, d=4;
```

Arithmetic expressions

Suppose `x` and `y` are variables

- `x+y`, `x-y`, `x*y`, `x/y`, `x%y`: binary arithmetic
- A simple statement:
  `y = x+3*x/(y-4);`
- Numeric literals like `3` or `4` valid in expressions
- Semicolon ends statement (not newline)
- `x += y`, `x -= y`, `x *= y`, `x /= y`, `x %= y`: arithmetic and assignment

Order of operations

- Order of operations:

Operator	Evaluation direction
+, - (sign)	right-to-left
*, /, %	left-to-right
+, -	left-to-right
=, +=, -=, *=, /=, %=	right-to-left

- Use parentheses to override order of evaluation

Order of operations

Assume `x` $=2.0$ and `y` $=6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   ```
   float z = x+3*x/(y-4);  →  float z = x+3*x/2.0;
   ```

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   ```
   float z = x+3*x/(y-4); → float z = x+3*x/2.0;
   ```

2. Evaluate multiplies and divides, from left-to-right

   ```
   float z = x+3*x/2.0;→ float z = x+6.0/2.0; → float z = x+3.0;
   ```

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` → `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` → `float z = x+6.0/2.0;` → `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` → `float z = 5.0;`

Order of operations

Assume $\mathtt{x} = 2.0$ and $\mathtt{y} = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` → `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` → `float z = x+6.0/2.0;` → `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` → `float z = 5.0;`

4. Perform initialization with assignment

   Now, $\mathtt{z} = 5.0$.

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` → `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` → `float z = x+6.0/2.0;` → `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` → `float z = 5.0;`

4. Perform initialization with assignment

   Now, $z = 5.0$.

How do I insert parentheses to get $z = 4.0$?

Order of operations

Assume $x = 2.0$ and $y = 6.0$. Evaluate the statement

```
float z = x+3*x/(y-4);
```

1. Evaluate expression in parentheses

   `float z = x+3*x/(y-4);` → `float z = x+3*x/2.0;`

2. Evaluate multiplies and divides, from left-to-right

   `float z = x+3*x/2.0;` → `float z = x+6.0/2.0;` → `float z = x+3.0;`

3. Evaluate addition

   `float z = x+3.0;` → `float z = 5.0;`

4. Perform initialization with assignment

   Now, $z = 5.0$.

How do I insert parentheses to get $z = 4.0$?

```
float z = (x+3*x)/(y-4);
```

Function prototypes

- Functions also must be declared before use
- Declaration called *function prototype*
- Function prototypes:

  `int factorial (int );` or `int factorial (int n);`

- Prototypes for many common functions in header files for C Standard Library

Function prototypes

- General form:
  `return_type function_name(arg1,arg2,...);`
- Arguments: local variables, values passed from caller
- Return value: single value returned to caller when function exits
- `void` – signifies no return value/arguments

```
int rand(void);
```

The `main()` function

- `main()`: entry point for C program
- Simplest version: no inputs, outputs 0 when successful, and nonzero to signal some error

```
int main(void);
```

- Two-argument form of `main()`: access command-line arguments

```
int main(int argc, char **argv);
```

- More on the `char **argv` notation later...

Function definitions

```
Function declaration
{
  declare variables;
  program statements;
}
```

- Must match prototype (if there is one)
  - variable names don't have to match
  - no semicolon at end
- Curly braces define a *block* – region of code
  - Variables declared in a block exist only in that block
- Variable declarations before any other statements

Our main() function

```c
/* The main() function */
int main ( void ) /* entry point */
{
  /* write message to console */
  puts ( "hello, UNMLA students" );

  return 0; /* exit (0 => success) */
}
```

- `puts()`: output text to console window (stdout) and end the line
- String literal: written surrounded by double quotes
- **return** 0;
  exits the function, returning value 0 to caller

Alternative `main()` function

- Alternatively, store the string in a variable first:

```c
int main(void) /* entry point */
{
  const char msg[] = "hello, UNMLA students";

  /* write message to console */
  puts(msg);
```

- `const` keyword: qualifies variable as constant
- `char`: data type representing a single character; written in quotes: `'a'`, `'3'`, `'n'`
- `const char msg[]`: a constant array of characters

More about strings

- Strings stored as character array
- Null-terminated (last character in array is `'\0'` null)
  - Not written explicitly in string literals
- Special characters specified using \ (escape character):
  - `\\` – backslash, `\'` – apostrophe, `\"` – quotation mark
  - `\b`, `\t`, `\r`, `\n` – backspace, tab, carriage return, linefeed
  - *ooo*, \x*hh* – octal and hexadecimal ASCII character codes, *e.g.* `\x41` – `'A'`, `\060` – `'0'`

Console I/O

- stdout, stdin: console output and input streams
- `puts(`*`string`*`)`: print string to stdout
- `putchar(`*`char`*`)`: print character to stdout
- *`char`* `= getchar()`: return character from stdin
- *`string`* `= gets(`*`string`*`)`: read line from stdin into string
- Many others

Preprocessor macros

- Preprocessor macros begin with `#` character

  #include <stdio.h>

- **#define** msg `"hello, UNMLA students"`

  defines *msg* as `“hello, UNMLA students”` throughout source file

- many constants specified this way

Defining expression macros

- **#define** can take arguments and be treated like a function

  ```
  #define add3(x,y,z) ((x)+(y)+(z))
  ```
- parentheses ensure order of operations
- compiler performs inline replacement; not suitable for recursion

Conditional preprocessor macros

- **#if** , **#ifdef**, **#ifndef**, **#else**, **#elif** , **#endif**
  conditional preprocessor macros, can control which lines are compiled
  - evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
  - the `gcc` option `-D`*`name`*`=`*`value`* sets a preprocessor define that can be used
  - Used in header files to ensure declarations happen only once

Conditional preprocessor macros

- **#pragma**
  preprocessor directive
- **#error**, **#warning**
  trigger a custom compiler error/warning
- **#undef** msg
  remove the definition of `msg` at compile time

Compiling our code

After we save our code, we run `gcc`:

```
prompt% gcc -g -O0 -Wall hello.c -o
hello.exe
```

Assuming that we have made no errors, our compiling is complete.

Running our code

Or, in `gdb`,

```
prompt% gdb hello.exe
⋮
Reading symbols from hello.exe...done.
(gdb) run
Starting program:  hello.exe
hello, UNMLA students

Program exited normally.
(gdb) quit
prompt%
```

Summary

Topics covered:

- How to edit, compile, and debug C programs
- C programming fundamentals:
  - comments
  - preprocessor macros, including `#include`
  - the `main()` function
  - declaring and initializing variables, scope
  - using `puts()` – calling a function and passing an argument
  - returning from a function

MIT OpenCourseWare
http://ocw.mit.edu

6.087 Practical Programming in C
IAP 2010

For information about citing these materials or our Terms of Use,visit: http://ocw.mit.edu/terms.