University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

March 20, 2001

Introduction to C

History

- C was originally developed by Brian Kernighan and Dennis Ritchie to write UNIX (1973).
- Intended for use by expert programmers to write complex systems.
- Difficult for novices to learn.
- Very powerful – lots of rope to hang yourself.
- Very close to assembly language (at least of the machines of that era).
- OS's of that era often written in assembly.

Slide 14–1

History. . .

- Need to do "low level" things in OS that your average application doesn't.
- Trades programming power for speed and flexibility.
- No surprises.
- Understanding assembly will help you greatly with C.
- 1st C standard was K&R, 1978.
- Standardized by ANSI committee in 1989. They formalized extensions that had developed and added a few. We will learn ANSI C.

Slide 14–2

Hello World

- Assume the following in the contents of the file `hello.c`, created using your favorite text editor:

```
1   #include <stdio.h>
2
3   void
4   main(void)
5   {
6     printf("Hello World\n");
7   }
```

- Line numbers are for discussion only, not really in file.

Slide 14–3

Hello World...

- To create an executable (`a.out`) from `hello.c` it must be compiled into machine code. There are four steps in the compilation of a C program on UNIX, each handled by a different program:

cpp: C pre-processor. Converts C source into C source, e.g. `hello.c` into `hello.i`.

cc1: C compiler. Converts C source into assembly language, e.g. `hello.i` into `hello.s`.

as: Assembler. Converts assembly code into machine code, e.g. `hello.s` into `hello.o`.

Slide 14–4

Hello World...

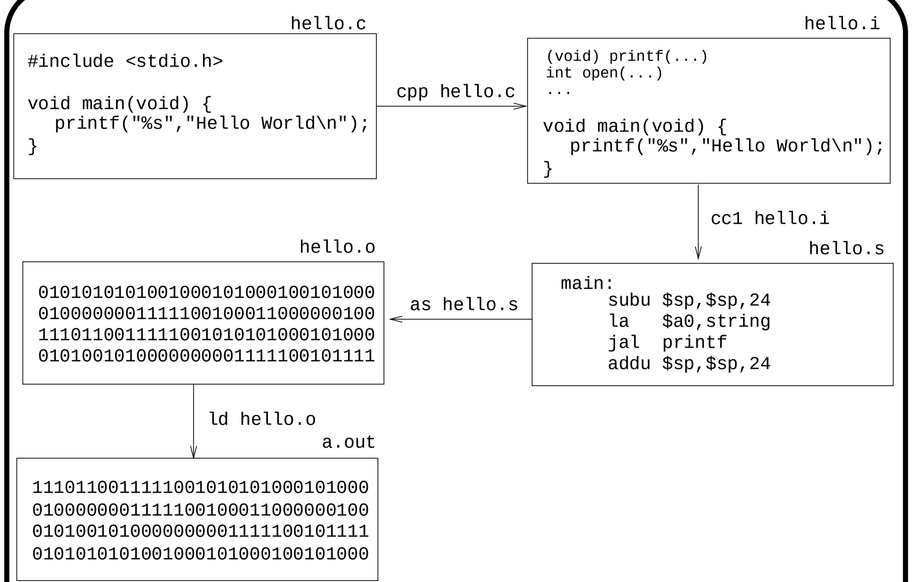
ld: Linker/Loader. Converts machine code into executable program, e.g. `hello.o` into `a.out`.

- The user typically doesn't invoke these four separately. Instead the program `cc` (`gcc` is GNU's version) runs the four automatically, e.g.

```
$ gcc hello.c
```

produces `a.out`.

Slide 14–5



Slide 14–6

C Preprocessor

- The C preprocessor (`cpp`) is a program that preprocesses a C source file before it is given to the C compiler.
- The preprocessor's job is to remove comments and apply preprocessor directives that modify the source code.
- Preprocessor directives begin with ⌜`#`⌝, such as the directive ⌜`#include <stdio.h>`⌝ in `hello.c`.

Slide 14–7

C Preprocessor...

Some popular ones are:

#include <file> The preprocessor searches the system directories (e.g. `/usr/include`) for a file named **file** and replaces this line with its contents.

#define word rest-of-line Replaces word with rest-of-line thoroughout the rest of the source file.

#if expression ··· **#else** ··· **#endif** If expression is non-zero, the lines up to the `#else` are included, otherwise the lines between the `#else` and the `#endif` are included.

Slide 14–8

C Preprocessor...

- The C preprocessor is a very powerful tool. You can use it to include other files, do macro expansion, and perform conditional text inclusion. The C compiler doesn't handle any of these functions.

1. Do not define complicated macros using `#define`. Macros are difficult to debug.
2. Do not use conditional text inclusion, except perhaps to define macros in a header file.
3. Use ⌜`#include "foo.h"`⌝ to include header files in the current directory, ⌜`#include <foo.h>`⌝ for system files.

Slide 14–9

C Preprocessor Examples

```
#define ARRAY_SIZE 1000
char str[ARRAY_SIZE];

#define MAX(x,y) ((x) > (y) ? (x) : (y))
int max_num;
max_num = MAX(i,j);

#define DEBUG 1

#ifdef DEBUG
   printf("got here!")
#endif

#if defined(DEBUG)
   #define Debug(x) printf(x)
#else
   #define Debug(x)
#endif

Debug(("got here!"));
```

Slide 14–10

C Syntax

- Syntax is how the characters and words of a program must be put together. It is like the spelling and grammar of the programming language. Semantics is what the program means, i.e. what it does.
- A syntax error means the program can't be compiled.
- gcc will produce a (cryptic) error message.
- A semantic error means the program doesn't produce the correct result.

Slide 14–11

C Syntax...

C syntax is not line-oriented. This means that C treats newline characters (carriage returns) the same as a space. These two programs are identical:

```
#include<stdio.h>void main(void){printf("Hello World\n");}
```

```
#include <stdio.h>void
main(
void)
{printf("Hello World\n"
); }
```

Spaces, tabs, and newlines are known as **whitespace**, and the compiler treats them all as spaces.

Slide 14–12

C Functions

- A C program is a collection of functions. There are no procedures in C; everything is a function, although some functions don't return a value.
- When a C program begins running, execution starts in the function `main`.
- A function definition starts with a function header, which tells us the
  name of the function, its return type, and its parameters:

  `void main(void)`

Slide 14–13

`void main(void)`

1. First is the type of the return value. In this case the function doesn't return a value. In C this is expressed using the type `void`.
2. The function's name is `main`.
3. Following the function name is a comma-separated list of the formal parameters for the function. Each element of the list consists of the parameter's type and name, separated by whitespace.
4. If main took parameters it might look like this:

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

Slide 14–14

C Functions...

- Following the function header is the function body, surrounded by braces:

```
{
        declarations
        statements
}
```

Once again, C doesn't care about lines. You can put this all on one line if you like, and indent it any way you want (or not at all).

Slide 14–15

1. Definitions define types, e.g. a new type of structure.
2. Declarations declare variables and functions. Statements are the instructions that do the work. Statements must be separated by semicolons ';'.
3. The brace-delimited body is a form of compound statement or block. A block is syntactically equivalent to a single statement, except it doesn't have to be followed by a semicolon.
4. The "hello world" program has no definitions, no declarations, and has one statement, a call to the printf function on line 6.

```
printf("Hello World\n");
```

Slide 14–16

`printf`

- `printf` is used to print things to the terminal, in this case the character array `"Hello World"`.
  1. The newline character '\n' causes the terminal to perform a carriage-return to the next line.
  2. When the compiler sees the literal character array (string) `"Hello World"` it allocates space for it in memory, and passes its address to printf.
  3. `printf`'s arguments can be complicated. We'll look at `printf` in more detail later.

Slide 14–17

Compiling C

- We'll be using the `gcc` compiler, the free C/C++ compiler from GNU.
- To compile `hello.c` do

  ```
  $ gcc -o hello hello.c
  ```
- This creates a file `hello` which can then be executed:

  ```
  $ hello
  ```
- Sometimes you will want to optimize your code. If so, compile with the `-O` flag:

  ```
  $ gcc -O -o hello hello.c
  ```

Slide 14–18

Various flags to the `gcc` compiler will halt compilation after a certain stage. Looking at the output after each stage can be interesting, and sometimes helpful in identifying compiler bugs.

gcc -E hello.c The preprocessed C source code is sent to standard output.

gcc -S hello.c The assembly code produced by the compiler is in `hello.s`.

gcc -c hello.c Compile the source files, but do not link. The resulting object code is in `hello.o`.

gcc -v hello.c Print the commands executed to run the stages of compilation.

Slide 14–19

Debugging C

- Often you will want to debug your code. If so, compile with the `-g` flag:

```
$ gcc -g -o hello hello.c
```

- You can debug using `gdb`, or `ddd` (if you're running X-windows):

```
$ gdb hello
$ ddd hello
```

Slide 14–20

.h and .c files

- A program's code is normally stored in a file that ends in `.c`.
- Often there are a number of `definitions` that you wish to share between several `.c` files. These are put in a `.h` file. Here's `globals.h`:

```
#define SIZE 10
typedef myType int
```

- In any `.c` file in my program I can then include `globals.h`:

```
#include "globals.h"
```

Slide 14–21

Makefiles

- When you have more than one C module (file) that needs to be compiled, and there's a special order in which they need to be compiled, you need to create a makefile.
- Here's an example program consisting of two files:

```
// hello.c
#include "msg.h"
int main(void) {printf(MESSAGE);}

// msg.h
#define MESSAGE "Hello World!"
```

Slide 14–22

Makefiles. . .

- Here's the makefile:

```
hello: hello.o
   gcc -o hello hello.o
hello.c: hello.c msg.h
   gcc -o hello.o -c hello.c
```

- When I type `make` the right commands to build the program will be issued:

```
$ make
gcc -o hello.o -c hello.c
gcc -o hello hello.o
```

Slide 14–23

Makefiles. . .

- Whenever you change one of the source files, just type `make` again:

```
$ touch msg.h
$ make
gcc -o hello.o -c hello.c
gcc -o hello hello.o
```

Slide 14–24

Makefiles. . .

- The rule

```
hello: hello.o
   gcc -o hello hello.o
```

says:

> “when `hello.o` is newer than `hello`, it’s time to create a new version of `hello`. The command to do this is `gcc -o hello hello.o`.”

- Note, the first character in the command line must be a `TAB`.

Slide 14–25

Makefiles. . .

- You can have more than one *target* in a makefile:

```
love: love.c msg.h
   gcc -o love love.c
war: war.c msg.h
   gcc -o war war.c
```

- The commands

```
$ make love
$ make war
```

will then create the two programs `love` and `war`, respectively.

Slide 14–26

Tools: cb

- `cb` will take an ugly program

```
int main(void) {printf("hello world!");}
```

and make it beautiful

```
$ cb -s -j hello1.c
int main(void)
{
   printf("hello world!");
}
```

Slide 14–27

Tools: lint

- lint will warn you of potential problems with your code:

```
$ lint hello1.c
implicitly declared to return int
    (1) printf

function falls off bottom without returning value
    (1) main

function returns value which is always ignored
    printf
```

Slide 14–28

Example I

- Here's a small example C program. You're not supposed to understand this yet.
- The program is compiled and executed like this (`-lm` means to link in the `math` library):

```
$ gcc -lm sine.c
$ a.out
```

```
Value of PI = 3.141593

angle      Sine
  0        0.000000
  10       0.173648
  ....      .....
 360       -0.000000
```

Slide 14–29

```
/* Michel Vallieres 1995 */
#include <stdio.h>
#include <math.h>
int main() {
    int    angle_degree;
    double angle_radian, pi, value;
    printf ("Compute a table of the sine function\n\n");
    pi = 4.0*atan(1.0);
    printf(" Value of PI = %f \n\n", pi);
    printf(" angle     Sine \n");
    angle_degree=0;
    while ( angle_degree <= 360 ) {
       angle_radian = pi * angle_degree/180.0;
       value = sin(angle_radian);
       printf(" %3d      %f \n ", angle_degree, value);
       angle_degree = angle_degree + 10;
}}
```

Slide 14–30

Debugging with GDB

HELP What commands are there?

HELP command Describe a particular command.

quit Exit from `gdb`.

BACKTRACE displays of stack trace of current procedures and parameters

BREAK sets a breakpoint

CONT resumes execution after a breakpoint

DEFINE builds a user alias

DELETE, CLEAR removes a breakpoint or trace event

HELP accesses on-line help

INFO BREAK lists all breakpoints events

Slide 14–31

Debugging with GDB

INFO LOCALS lists the names and values of all local variables

LIST displays the current source code file

NEXT, STEP suspends execution after a number of lines of code have executed

PRINT displays the values of variables and expressions

RUN start your program

SET changes the values of variables

SHELL passes a command to the interactive shell process for execution

WATCH sets a trace event

Displaying Source Code

list display the next ten lines of source code.

list line1 display ten lines of source code, starting at **line1**

list line1, line2 display the source code between **line1** and **line2**.

list function list the source code of a particular **function**.

Displaying Values

info locals displays the name and values of all local variables in the currently active procedure.

print expression displays values of expressions.

set variable = expression alter the contents of a variable.

Breakpoints

break linenumber set breakpoints by line numbers, relative to the beginning of the source code file.

break function stop execution at the beginning of a given function.

break offset set a breakpoint at offset lines from current step.

break if condition stop execution whenever this condition is true.

info break display all current events.

delete event remove a breakpoint having a particular event identifier.

clear line remove the breakpoint associated with **line**.

Tracing Execution

watch function print source code lines when the program reaches a specific function.

watch variable display the values of a variable when its value changes.

watch expression display the values of an expression when its value changes.

Using the watch command will slow the execution of the program substantially

Running your Program

run arguments will start execution of the program.

cont will resume execution of the program after it has been stopped.

step step through the next source line.

next step through the next source line. If the line is a function call, stop at the line after the call.

The Program Stack

backtrace display the call stack.

frame number go to frame number **number**.

up go to the frame above the current one.

down go to the frame below the current one.

Example I

```
#include <stdio.h>
static int statvar=89;
struct nomstruct{
        int l;
        char nom[12];
        char *prenom;
} varstruct, *ptrstruct;
int divzero (){
        int i,j,k;
        i=5;
        j=0;
        k = i/j;
        return k;
}
main(){ int mainvar = 7;
        ptrstruct = &varstruct;
        varstruct.nom[0] = 'Z';
        statvar = 63;
        varstruct.nom[1] = '\0';
        varstruct.prenom = "valerie";
        printf("%d \n",divzero());
}
```

Example II

```
#include <stdio.h>
int main(void)
{
  int i=0;
  float x, y, z=0.0;

  x = 3.14159;
  y = x * x + 3.0;

  for (i = 0; i < 100; i++)  {
      x += i + 1;
      y /= .987 * i + 1;
  }

  z = x + y;

  return 0;

}
```

Slide 14–40

Control Constructs

- C has pretty much the same control constructs as Java:

`=`	Assignment
`/* */`	Comments
`//`	Comments
`while (`*expr*`)` *stat*	While-loop
`for(i=0;i<n;i++)` *stat*	For-loop. Note that `i` cannot be declared in the header.
`break`	Break out of a loop or switch.

Slide 14–41

`return` *expr*	Return value from function
`L:`	Declare a label.
`goto L`	Goto a label.
`if (`*expr*`)` *stat* `else if (`*expr*`)` *stat* `else` *stat*	If-statement.

`switch(`*expr*`) {`
`case 1:` *stat*`; break;`
`case 2:` *stat*`; break;`
`default:` *stat*`; break;`
`}`
	Switch-statement.

Slide 14–42

Operators

`()`	Function call.
`[]`	Array index.
`.`	Structure access.
`->`	Structure access through pointer.
`x++ x--`	Increment/decrement and return *previous* value.
`++x --x`	Increment/decrement and return *new* value.
`!x`	Logical negation ($!0 \Rightarrow 1, !1 \Rightarrow 0$).
`~x`	Bit-wise not.

Slide 14–43

`*x`	Pointer dereference (what `x` points to).
`&x`	Address-of `x`.
`sizeof(x)`	Size (in bytes) of `x`.
`(T)x`	Cast `x` to type `T`.
`x*y x/y x%y`	Multiplication, division, modulus
`x+y x-y`	Addition, subtraction
`x<<y x>>y`	Shift `x` `y` bits to the left/right.
`x<y x<=y x>y x>=y`	Compare `x` and `y`. Return `1` for TRUE and `0` for FALSE.
`== !=`	Equality test.
`x&y x|y`	Bitwise `and` and `or`.

Slide 14–44

`x^y`	Bitwise `xor`.
`x&&y`	Short-circuit `and`.
`x||y`	Short-circuit `or`.
`x?y:z`	`if (x) y else z`.
`x=y`	Assignment.
`x+=y x-=y x*=y x/=y x%=y x>>=y x<<=y x&=y x|=y x^=y`	Augmented assignment ($x+=y \equiv x=x+y$).
`x,y`	Evaluate `x` then `y`, return `y`.

Slide 14–45

Constants

`0x12ab`	A hexadecimal constant.
`01237`	An octal constant (prefixed by `0`).
`34L`	A long constant integer.
`3.14, 10., .01, 123e4, 123.456e7`	Floating point (`double`) constants.
`'A', '.', '%'`	The ASCII value of the character constant.
`"apple"`	A string constant.

Slide 14–46

Constants. . .

`\n`	A "newline" character.
`\b`	A backspace.
`\r`	A carriage return (without a line feed).
`\'`	A single quote (e.g. in a character constant).
`\"`	A double quote (e.g. in a string constant).
`\\`	A single backslash

Slide 14–47

Bit-operations

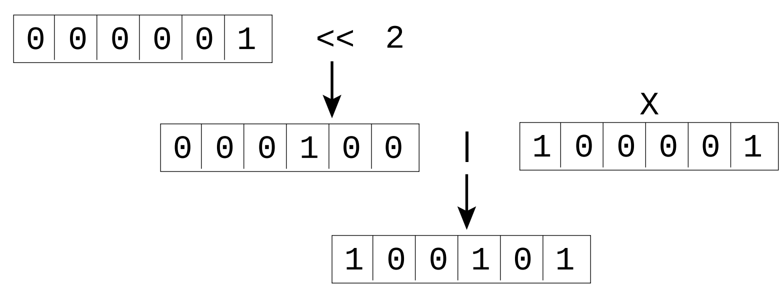
- C provides several operators for manipulating the individual bits of a value:

Operator	Meaning
`&`	bitwise AND
`|`	bitwise OR
`^`	bitwise XOR
`~`	one's complement
`<<`	left-shift
`>>`	right-shift

Slide 14–48

Bit-operations: `x |= (1<<n)`

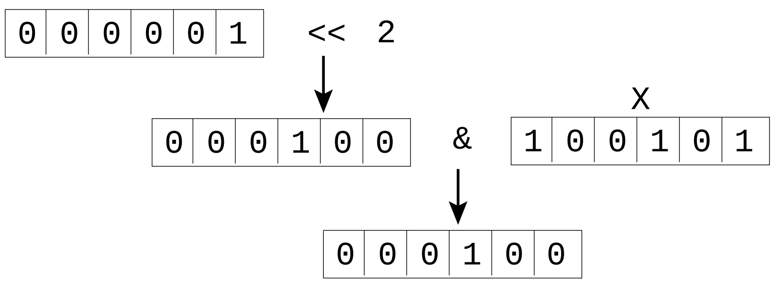
- Set bit **n** in variable **x**. `(1<<n)` shifts 1 left by **n** bits. The result is OR'ed into **x**.



Slide 14–49

Bit-operations: `x & (1<<n)`

- Test bit **n** in variable **x**. `(1<<n)` creates a value with the appropriate bit set by shifting 1 left by **n** bits. It is AND'ed with **x** to see if that bit is set in **x**.



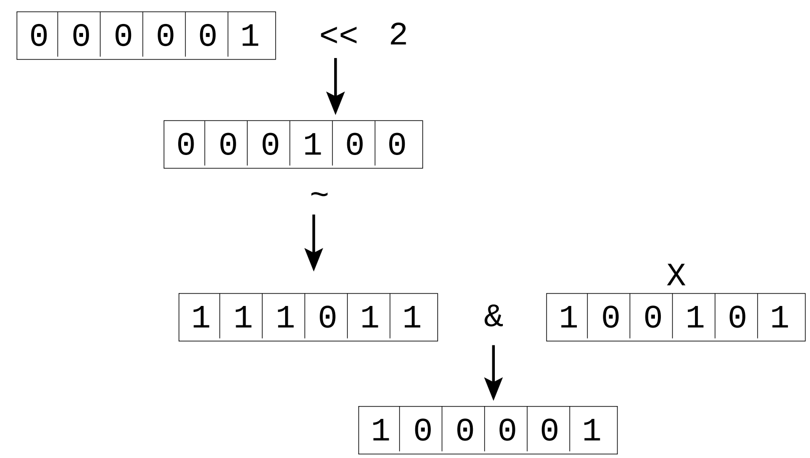
Slide 14–50

Bit-operations: `x & ~(1<<n)`

- Clear bit **n** in variable **x**.
- `(1<<n)` creates a value with the appropriate bit set by shifting 1 left by **n** bits.
- The one's complement operation '~' flips all the bits in the value, resulting in a value with every bit but the **n**'th set.
- It is AND'ed with **x** to clear the **n**'th bit but leave the rest unchanged.

Slide 14–51

Bit-operations: `x & ~(1<<n)`



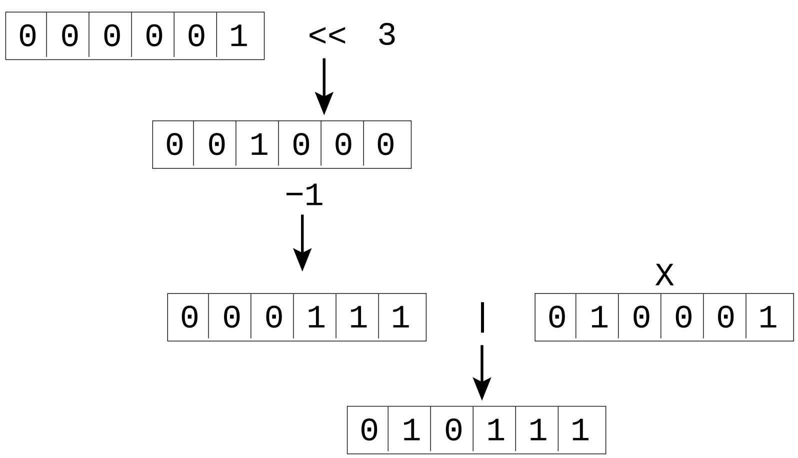
Slide 14–52

Bit-operations: `x |= (1<<(n+1))-1`

- Suppose we want to set the low-order 3 bits.
- We have to OR **`x`** with the value that has only these bits set.
- 111_2 is $1000_2 - 1$.
- So we have to shift 1 left by 4 bits, subtract 1, and OR it into **`x`**. In general we shift left by **`n+1`** bits.

Slide 14–53

Bit-operations: `x |= (1<<(n+1))-1`



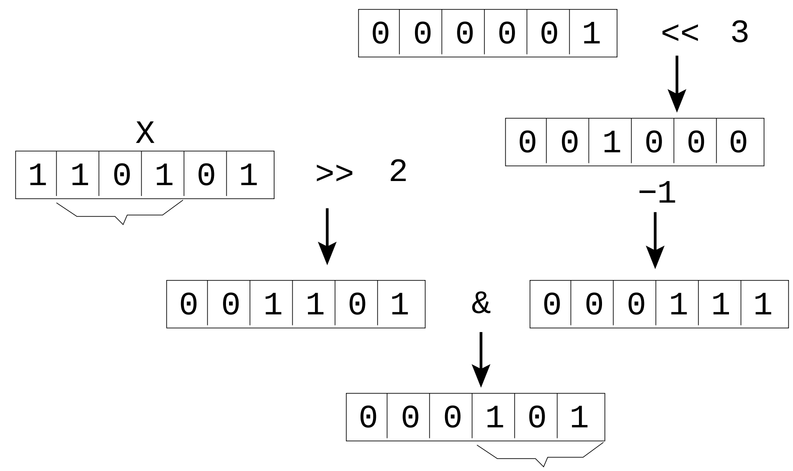
Slide 14–54

Bit-operations: `(x>>p)&((1<<(n+1))-1)`

- Suppose we want to extract n bits from **`x`**, starting at position p.
- We create a mask with all ones in the lower n bits the same way as before: shift 1 left by $n + 1$ bits and subtract 1.
- Next, we shift **`x`** right by p bits.
- Finally, we AND the mask and **`(x>>p)`** to strip out any extra high-order bits.

Slide 14–55

Bit-operations: `(x>>2)&((1<<(3+1))-1)`



Slide 14–56

Readings and References

- *Back to C*, by Don Waugaman, `http://www.cs.arizona.edu/classes/cs340/fall00/misc/back_to_c.ps`.
- *ANSI C for Programmers on UNIX Systems*, by Tim Love, `http://www.cs.arizona.edu/classes/cs340/fall00/misc/teaching_C.ps`.
- Various tutorials on C: `http://www.1001tutorials.com/c/index.shtml`
- A `make` tutorial: `http://www.geocities.com/SiliconValley/Lakes/2688/Make/index.html`.

Slide 14–57

Readings and Reference...

- A `gdb` tutorial: `http://www.info.ucl.ac.be/etudiants/outils/cornell/gdb/`.
- C snippets: `http://www.snippets.org/`.
- Ten Gotchas of the C language: `http://www.andromeda.com/people/ddyer/topten.html`.
- Steve Summit's Introductory C course `http://www.eskimo.com/~scs/cclass/cclass.html`.

Slide 14–58