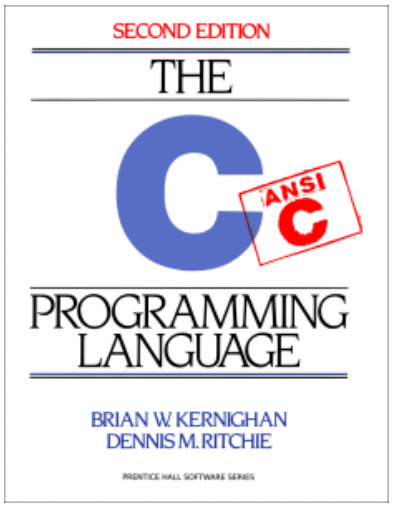
CS 241
Data Organization
Introduction to C

Brooke Chenoweth

University of New Mexico

Fall 2014

Read Kernighan & Richie



- 1.5 Character Input and Output
- 1.6 Arrays
- 1.7 Functions
- 1.8 Function Arguments - call by value

History of C

- C was originally developed by Brian Kernighan and Dennis Ritchie to write UNIX (1973), as a followup to "B", 1970.
- Intended for use by expert programmers to write complex systems.
- Complex for novices to learn.
- Very powerful – lots of rope to hang yourself.
- Very close to assembly language (of machines of that era).
- OS's of that era often written in assembly.

History cont...

- Need to do "low level" things in OS that your average application doesn't.
- Trades programming power for speed and flexibility.
- 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.

Why use C?

- Professionally used language C/C++
- Compact language, does not change (unlike Java and C++)
- Used in many higher level courses like: Networking, Operating Systems, Compilers, ...
- Often no need involve graphics (usually slows things down) - Original unix didn't have much of graphic stuff, so added on later.

An example C program

```
/* Small C program example */
#include <stdio.h>
int main ( void )
{
   int numTrucks = 0;
   for (numTrucks = 5; numTrucks >= 0; numTrucks--)
   {
      printf("Trucks left in depot: %2d\n", numTrucks);
   }
   return 0;
}
```

Code saved in file: `dispatch.c`

Compiling and running

```
$> gcc dispatch.c
$> ./a.out
Trucks left in depot:  5
Trucks left in depot:  4
Trucks left in depot:  3
Trucks left in depot:  2
Trucks left in depot:  1
Trucks left in depot:  0
```

What's the difference between a C program running on your computer and a Java program?

Java vs. C

On the following pages a number of comparisons between Java and C will be presented in the following format:

Java version here...

```
String str = "I am Java";
System.out.println ( str );
```

C version here...

```
char *str = "I am not Java!";
printf ( "\%s\n", str );
```

Compilation and Running

C code must be compiled to *native* machine code in order to run on a computer.

- Compile:
  `$> javac SourceFile.java`
  (From source to byte code)
- Run:
  `$> java SourceFile`
  Run byte code on VM

- Compile:
  `$> gcc SourceFile.c`
  (From src to machine code)
- Run:
  `$> ./a.out`
  Execute machine code

Another example program

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

```
#include <stdio.h>

void main ( void )
{
  printf ( "Hello World\n" );
}
```

Compiling a C Program

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

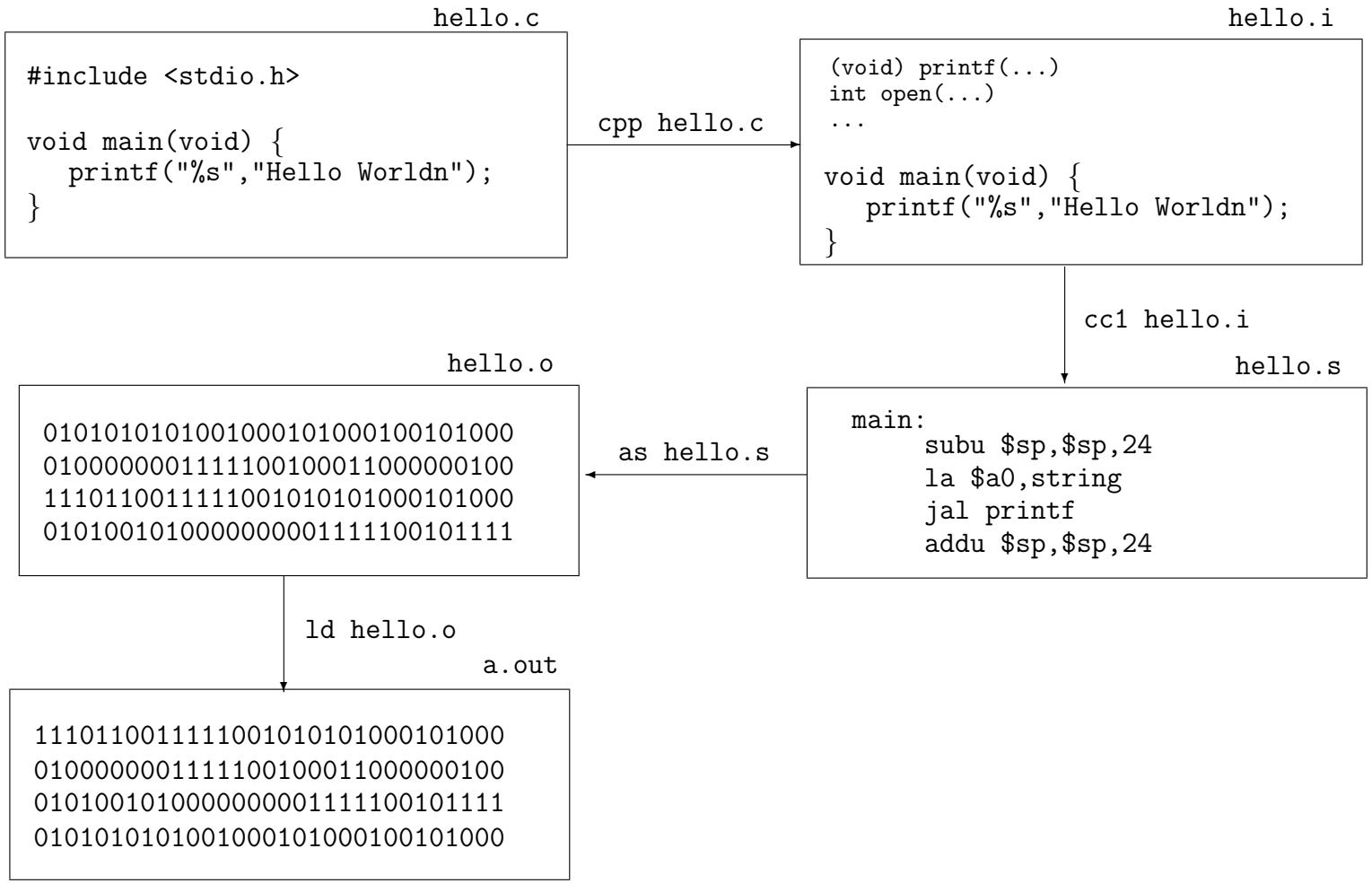
Compiling a C program

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`.

C Program Compilation Process



Data types

Java primitive types:

- `boolean`
- `char`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `String`

C corresponding types:

- `int`
- `char`
- `char`
- `short int`
- `int`
- `long int`
- `float`
- `double`
- `char*`

Data Type Modifiers

- Data types in C have a number of modifiers that can be applied to them at declaration time (and typecast time).
- `short` – Works on `int`, half size `int`
- `long` – Works on `int`, `long int`, `double`
- `unsigned` allows for cardinal numbers only
- `signed` allows for both positive and negative numbers
- `char` is `unsigned` by default
- Other integer types are signed by default

Working with data types

- Technically any variable of any type is "just a chunk of memory", and in C can be treated as such.
- Therefore, it's ok to typecast almost anything to anything – Note, this is not always a good thing, but can come in handy some times.
- Bitwise operations are allowed on all types (if properly typecast)

Defining Variables

- Variable declarations are done similarly to how it's done in Java

In Java:

```
int x;

int y = 5;

String s = "Hello";

double [][] matr;
```

In C:

```
int x;

int y = 5;

char* s = "Hello";

double matr[12][12];
```

- **Note!** Variables in C do not get default values assigned to them! Always initialize all your variables to avoid problems. (Experience talking)

Constants

Constants from a C perspective are constant values entered *in your source code* – can be determined by compile time.

Type	**Example constant**
`int`	`123` or `0x7B` (hex)
`long int`	`123L` or `0173L` (octal)
`unsigned int`	`123U`
`char`	`'x'`
`char*`	"Hello"

See K&R, Chap 2.3 for more info

Constants through the C Preprocessor

Some example definitions:

```
#define MAXVALUE 255
```

```
#define PI 3.14159265
```

The C pre-processor will make a textual substitution for every occurence of the words `MAXVALUE` and `PI` in your source code *before!* compilation takes place.

Aside: C preprocessor definitions can be passed to the compiler reducing the need to edit the source code for a constant change!

The 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`.

The 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.

The C preprocessor

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

1. Avoid defining complicated macros using `#define`.
   Macros are difficult to debug.
2. Avoid 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.

Preprocessor Example

```
#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!"));
```

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\n");}
```

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

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

Functions

In Java: *methods* In C: *functions*

- A C program is a collection of functions. C is a "flat" language. All functions are "at the same level". No objects (as in java).
- Some functions can have a `void` return type, meaning they don't return a value.
- A function definition starts with the function header, that tells us the name of the function, its return type, and its parameters:

```
void main(void)
```

`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)
```

C Functions

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

```
{
        declarations;
        statements;
}
```

Again, C doesn't care about lines. It's possible to put this all on one line to create an illegible mess... *However... if your programs aren't properly indented they will not receive many points when we grade them!*

C Functions...

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 no need to end with 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 (on an earlier slide).

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

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.

More on the C compiler

- We'll be using the `gcc` compiler, GNU's free compiler.
- To compile `hello.c` do
  `$> gcc -o hello hello.c`
- This creates a file `hello` which can then be executed:
  `$> hello`
- Use the `-O` flag to optimize your program:
  `$> gcc -O -o hello hello.c`
- To get more feedback from the c compiler, use:
  `$> gcc -Wall -o hello hello.c`

More on the C compiler

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.

Programs and `.c` and `.h` 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"
```

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 <stdio.h>
#include "msg.h"
int main(void) {printf(MESSAGE);}

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

Makefiles...

- Here's the makefile:

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

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
```

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`.

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.

Control Constructs

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

`/* */`	Comments
`//`	Comments (not in ANSI C!)
`while (<expr>)` `  <statement>`	While-loop
`for(i=0;i<n;i++)` `  statement`	For-loop. Note - Can't declare i in header

`if(<expr>)`
`  <statement>`
`else`
`  <statement>`
	If-Else. (Else is optional)

`break`	Break out of a loop or switch.

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.

Operators...

`*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`: `1` is true and `0` is false.
`== !=`	Equality test.

Operators...

`x&y x|y`	Bitwise `and` and `or`.
`x^y`	Bitwise `xor`.
`x&&y`	Short-circuit (logical) `and`.
`x||y`	Short-circuit (logical) `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 ≡ x=x+y)

`x,y`	Evaluate `x` then `y`, return `y`.

More constant examples

`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. (Note the single quotes)
`"apple"`	A string constant.

More 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

printf function

```
printf("Name %s, Num=%d, pi %10.2f","bob",123,3.14);
```

Output:

```
Name bob, Num=123, pi              3.14
```

printf format specifiers:

%s	string (null terminated char array)
%c	single char
%d	signed decimal int
%f	float
%10.2f	float with at least 10 spaces, 2 decimal places.
%lf	double

printf function: %d

%d: format placeholder that prints an `int` as a signed decimal number.

```
#include <stdio.h>
void main(void)
{
   int x = 512;
   printf("x=%d\n", x);
   printf("[%2d]\n", x);
   printf("[%6d]\n", x);
   printf("[%-6d]\n", x);
   printf("[-%6d]\n", x);
}
```

Output:

```
x=512
[512]
[   512]
[512   ]
[-   512]
```

printf function: %f

```c
#include <stdio.h>
void main(void)
{
   float x = 3.14159265358979238;
   double z = 3.14159265358979238;
   printf("x=%f\n", x);
   printf("z=%f\n", z);
   printf("x=%20.18f\n", x);
   printf("z=%20.18f\n", z);
}
```

Output:

```
x=3.141593
z=3.141593
x=3.141592741012573242
z=3.141592653589793116
```

Significant Figures

Using `/usr/bin/gcc` on moons.unm.edu, a float has 7 *significant figures*.
Significant figures are not the same as decimal places.

```
float x = 1.0/30000.0;
float z = 10000.0/3.0;
printf("x=%.7f\n", x);
printf("x=%.11f\n", x);
printf("x=%.15f\n", x);
printf("z=%f\n", z);
```

Output:

```
x=0.0000333
x=0.00003333333
x=0.000033333333704
z=3333.333252
```

printf function: %e

%e: Format placeholder that prints a float or double in *Scientific Notation*.

```
#include <stdio.h>
void main(void)
{
   float x = 1.0/30000.0;
   float y = x/10000;
   float z = 10000.0/3.0;
   printf("x=%e\n", x);
   printf("y=%e\n", y);
   printf("z=%e\n", z);
   printf("x=%.2e\n", x);
}
```

Output:

```
x=3.333333e-05
y=3.333333e-09
z=3.333333e+03
x=3.33e-05
```

Casting int to float

```
1  #include <stdio.h>
2  void main(void)
3  {
4     int a = 2;
5     int b = 3;
6     float c = a/b;
7     float x = (float)a / (float)b;
8     printf("c=%f x=%f\n", c, x);
9     printf("c=%3.0f x=%3.0f\n", c, x);
10 }
```

Output:

```
c=0.000000 x=0.666667
c=  0 x=  1
```

Line 6: Integer division, then cast to float.

Line 7: Cast to float, then floating point division.

Keyword: sizeof

```
#include <stdio.h>
void main(void)
{
   printf("char=%lu bits\n", sizeof(char)*8);
   printf("short=%lu bits\n", sizeof(short)*8);
   printf("int=%lu bits\n", sizeof(int)*8);
   printf("long=%lu bits\n", sizeof(long)*8);
   printf("long long=%lu bits\n", sizeof(long long)*8)
}
```

Output on moons.cs.unm.edu:

```
char=8 bits
short=16 bits
int=32 bits
long=64 bits
long long=64 bits
```

`lu` stands for Unsigned Long. On some machines, each of these types has a different size. On others, int = long = 16 bits.

printf function: %c

```
#include <stdio.h>
void main(void)
{
   char x = 'A';
   char y = 'B';
   printf("The ASCII code for %c is %d\n", x, x);
   printf("The ASCII code for %c is %d\n", y, y);
   y++;
   printf("The ASCII code for %c is %d\n", y, y);
}
```

Output:

```
The ASCII code for A is 65
The ASCII code for B is 66
The ASCII code for C is 67
```

ASCII Character Codes (Printable)

32		46	.	60	<	74	J	88	X	102	f	116	t
33	!	47	/	61	=	75	K	89	Y	103	g	117	u
34	”	48	0	62	>	76	L	90	Z	104	h	118	v
35	#	49	1	63	?	77	M	91	[	105	i	119	w
36	$	50	2	64	@	78	N	92	\	106	j	120	x
37	%	51	3	65	A	79	O	93	]	107	k	121	y
38	&	52	4	66	B	80	P	94	^	108	l	122	z
39	’	53	5	67	C	81	Q	95	_	109	m	123	{
40	(	54	6	68	D	82	R	96	‘	110	n	124	—
41	)	55	7	69	E	83	S	97	a	111	o	125	}
42	*	56	8	70	F	84	T	98	b	112	p		
43	+	57	9	71	G	85	U	99	c	113	q		
44	,	58	:	72	H	86	V	100	d	114	r		
45	–	59	;	73	I	87	W	101	e	115	s		

Logical operators

```c
#include <stdio.h>
void main(void)
{
   int a = 5;
   int b = 2;
   int c = 7;

   printf("%d\n", a + b < c);
   printf("%d\n", a + b == c);
   printf("%d\n", a - b == c);
   printf("%d\n", a - b != c);
}
```

Output:

```
0
1
0
1
```

While loop

```
1  #include <stdio.h>
2  void main(void)
3  {
4     int x = 1;
5
6     while (x < 200)
7     {
8        printf("[%d]", x);
9        x = x * 2;
10    }
11    printf("\n");
12 }
```

For loop

```
 1 #include <stdio.h>
 2 void main(void)
 3 {
 4    float lower = 50;
 5    float upper = 75;
 6    float step = 15;
 7    float f;
 8
 9    for (f = lower; f <= upper; f = f + step)
10    {
11       printf("%4.1f\n", f);
12    }
13 }
```

for and while

```
 1  int i;
 2  for (i=0; i<8; i++)
 3  {
 4    printf("[%d: %d] ", i, i%4);
 5  }
 6  printf("\n");
 7
 8  i=0;
 9  while (i<8)
10  {
11    printf("[%d: %d] ", i, i%4);
12    i++;
13  }
14  printf("\n");
```

Find the Syntax Error

```
1  #include <stdio.h>
2
3  #define LOWER 0
4  #define UPPER = 300
5
6  void main(void)
7  {
8     int f = LOWER;
9
10    while (f <= UPPER)
11    {
12       printf("%d\n", f);
13       f = f + 15;
14    }
15 }
```

- On which line will the compiler report an error?
- How would you edit the file to fix it?

If, Else If, and Else

```
#include <stdio.h>
void main(void)
{
   char c = getchar();
   if (c == 'c')
   { printf("Club\n");
   }
   else if (c == 'd')
   { printf("Diamond\n");
   }
   else if (c == 'h')
   { printf("Heart\n");
   }
   else
   { printf("Spade\n");
   }
}
```

- getchar reads one character from standard input stream (keyboard).
- What does this program do?
- What is a likely logic error?

Spot the error

```
#include <stdio.h>
void main(void)
{
   int grade = 87;

   if (grade > 90);
   {
      printf("You get an A\n");
   }

   if (grade < 60);
   {
      printf("You fail\n");
   }
}
```

Output:

```
You get an A
You fail
```

Spot the error

```c
#include <stdio.h>
void main(void)
{
   int grade = 87;

   if (grade > 90)
      printf("Congratulations\n");
      printf("You get an A\n");

   if (grade < 60)
   {
      printf("You fail\n");
   }
}
```

Output:

```
You get an A
```