Intro to C, part 1 - Basic Syntax

Context

The **C language** was created by Dennis Ritchie at Bell Labs in the early 1970s. It was designed to improve on the **B language** (created in 1969), which was a simplified version of **BCPL** (Basic Combined Programming Language, 1967). Brian Kernighan and Dennis Ritchie literally wrote the book on C, often referred to as "K&R":

The C Programming Language (1st ed, 1978; 2nd ed, 1988)

The first program in the book prints "hello, world", now a tradition in coding examples.
C is notable for several reasons:

- It was designed to be **cross-platform**, and run on different types of hardware.
- It was designed to be powerful and to produce **efficient machine code**.
- Thus, it was used to write operating systems (including Unix), compilers, debuggers, device drivers, and software for embedded systems.
- Its syntax and concepts were used in other languages, such as C++, Java, and C#.

This activity explores some key features of C that differ from other languages that you might be familiar with.

Sample C Program (for section A)

```c
#include <stdio.h>
#define TEST_VALS { -1, 0, 1, 2, 10, 100 }

/* function to estimate pi using given number of terms */
double estimatePi(int terms) {
  if (terms <= 0) return 0.0;   // ignore bad inputs
  double numer = -1.0, denom = 0.0, pi = 0.0;

  for (int i=0; i<terms; i++) { // loop through terms
    numer *= -1;
    denom  = 2*i + 1;
    pi    += numer / denom;
  }

  return 4*pi;
}

/* function to test estPi() */
void testEstimatePi() {
  int vals[] = TEST_VALS;

  for (int i=0; i<6; i++) {
    printf( "%3d %f\n", vals[i], estimatePi( vals[i] ) );
  }
}

/* function to interact with user */
void interact() {
  int terms = 1;
  printf("Welcome to the Pi Estimator!\n");

  while (terms > 0) {
    printf("How many terms do you want (0 to quit)? ");
    scanf("%d", &terms);
    printf("Pi with %d terms is %f\n", terms,  estimatePi(terms));
  }

  printf("Goodbye!");
}

/* main function, called when the program runs */
void main() {
  testEstimatePi();
  interact();
}
```

A. Simple C Program (~5 - 10 min)	

1. Use the sample C code provided to answer the questions below.

a.	In what line is `TEST_VALS` defined?	
b.	In what line is `TEST_VALS` used?	
c.	What characters mark … **comments**?	
d.	… character **strings**?	
e.	… the start and the end of a **block** of code?	
f.	How many functions are defined in the code? List their names.	
g.	What is the return type for `estimatePi()`?	
h.	What are the input parameter(s) for `estimatePi()`?	
i.	What keyword marks a function with no return value?	
j.	What function actually prints text?	
k.	Languages structures that control the order of code are called **control structures** (e.g. `if, for, while`). Which control structures look like Java or C#?	

2. Describe how you would change this program to add a test for 1000 terms. For each change specify the line number and describe what the change should be.

B. Preprocessor Directives (~ 10 min)	start time:

Lines that start with `#` change the C code **before** the compiler scans and parses it.
Thus, such lines are called **preprocessor directives**. Two common directives are:

`#include` (copy) the contents of another file into the current code.
Usually, such files declare useful functions and other information that should be near the head (top) of a file, so they are called **header files**, and have a `.h` extension.
(Thus, this is similar to `import` in Java and `using` in C#.)
A `.h` file in quotes ("") refers to local or program-specific information,
and a `.h` file in angle brackets (<>) refers to a standard C library. For example:

- `stdio.h` declares standard functions for input & output.
- `stdlib.h` declares other standard utility functions.

`#define` a **macro** that can be used in the file. Wherever the macro name appears in the code, it is replaced by the macro body. For example:

- `#define TEST_VALS { -1, 0, 1 }`
- `#define MIN(X,Y)  ( (X) < (Y) ? (X) : (Y) )`

1. Use the information above to write C directives that:

a.	Include `math.h` (a header for C library of math functions)	
b.	Define PI to be 3.14159	
c.	Define SUM3 to take 3 values and return their sum.	

2. A macro might look like a function, but does not always act like one.
For a **function**, the computer needs extra time and memory for every function call, pass values to the function, create local variables, run the function, and return any results.
For a **macro**, the preprocessor replaces the name with the body before the program runs, arguments are only evaluated at runtime, and there is no runtime cost for a function call.

Which approach (function or macro):

a.	evaluates arguments before passing them to the body?	
b.	uses extra time and memory?	
c.	is likely to be faster?	

3. (Optional) This macro sums the squares of its inputs:

```
      #define sumsqr(X,Y) (X*X + Y*Y)
```

The table below lists C statements, the result of expanding the macro, and the value of c.
If `sumsqr()` was a **function**, each statement will set `c` to 25,
but since `sumsqr()` is a **macro**, some statements produce incorrect values.
Complete the table to show the macro result and the value of c.
Do not add parentheses or other code that are not in the statement or macro.

	C Statement(s)	Macro Result	value of c

a.	`a=3, b=4,`
	`c = sumsqr(a,b  );`
		`(a*a + b*b)`
		`⇒ (3*3 + 4*4)`	

b.	`a=3, b=2,`
	`c = sumsqr(a,b+b);`
			17

c.	`a=2, b=3,`
	`c = sumsqr(a+a,b);`		

d.	`a=2, b=3,`
	`c = sumsqr(a+1,b+1);`		

4. (Optional) Describe the general issue that causes the incorrect values in the table above,
and show an improved `sumsqr()` macro to avoid this issue.

C. Checking for Errors (~ 8min)

```
01 #include <stdio.h>
02
03 int get(int array[], int size, int index) {
04   if ( /* INCOMPLETE */ ) return 0;
05   else                    return array[index];
06 }
07
08 void main() {
09   int data[5] = { 97,98,99,100,101 };
10   printf( "line 1: %d %d \n", data[ 0], data[4] );
11   printf( "line 2: %d %d \n", data[-1], data[5] );
12 }
```

1. Use the sample code above to answer the questions below.

a.	What would line 10 print?	
b.	Why would the array references in line 11 cause **error messages** in Java and C#?	
c.	What code should replace **INCOMPLETE** so that `get()` will check for these errors, and return 0 if a problem is found?	
d.	If every call to `get()` requires 10 assembly language instructions, how many instructions would it take to `get()` 1000 values in an array?	

2. Recall that C was developed in the 1970s to produce efficient machine code.
Explain why the designers might have decided that C would **not** check for such errors.

3. Would you prefer to use a language that checks for these errors or not? Explain.

D. Text Input & Output (I/O) (~15 min)	

	C Source Code	**Expected Output**
01	`printf("This is a test.\n");`	`This is a test.`
02	`printf("%c\t%c\n", 'A', 'Z');`	`A       Z`
03	`printf(    "%d+%d=%d\n"  , 2,2,4);`	`2+2=4`
04	`printf(  "%4d+%4d=%4d\n" , 2,2,4);`	`   2+   2=   4`
05	`printf("%02d+%02d=%02d\n", 2,2,4);`	`02+02=04`
06	`printf(    "%f\n", 3.14159);`	`3.141590`
07	`printf( "%7.3f\n", 3.14159);`	`  3.142`
08	`printf("%07.2f\n", 3.14159);`	`0003.14`
09	`printf("%s\n" , "This is a test.");`	`This is a test.`
10	`printf("%%\t%%\n");`	`%       %`

1. In C, a common and flexible way to **print output** is with `printf()`,
which prints to the **standard output** (also called **stdout**), usually the terminal window.
The first argument specifies how to print later arguments, and is called a **format string**.
For each later argument, the format string has a **conversion specification**.
Use the C code and its output above to answer the questions below.

a.	What specifies a **tab**?	
b.	What specifies a **newline**?	
c.	What character starts each conversion specification?	
d.	What specifies that a **character** be printed?	
e.	What specifies that a **string** be printed?	
f.	What specifies that a **decimal integer** be printed with at least 4 characters?	
g.	What does %% produce?	

2. The conversion specification for floating point numbers can contain 1 or 2 integers separated by a period.

a.	What does the 1st integer do?	
b.	What does the 2nd integer do?	
c.	What does a leading zero do?	

3. `printf()` returns the **number of characters** that are printed.

a.	What does `printf()` return in line 03 above?	
b.	What does `printf()` return in line 08 above?	

	C Source Code	**Expected Input**
01	`char a,b,c; int d; float f; char s[10];`	
02	`scanf("%d", &d);`	`1234`
03	`scanf("%f", &f);`	`23.345`
04	`scanf("%c %c %c", &a, &b, &c);`	`x y z`
05	`scanf("%s",  s);`	`hello world`
06	`printf("c=%c d=%d f=%f s=%s\n",c,d,f,s);`	
07	`// prints: z 1234 23.345 hello`	

4. In C, a common and flexible way to **read input** is with `scanf()`,
which reads from the **standard input** (also called **stdin**), usually the terminal window.
The first argument is a format string that specifies how to scan values for later arguments.
The information for each argument is called a **conversion specification**.
Use the C code and expected input above to answer the questions below.

a.	What characters specify that a character should be read?	
b.	What characters specify that a string of characters is read?	
c.	What character usually comes before each later arguments? (You will learn what this means in Part II.)	

5. `scanf()` returns the **number of values** (not characters) that are scanned.

a.	What does `scanf()` return in line 03 above?	
b.	What does `scanf()` return in line 08 above?	

```
int  printf(                const char* format, ... );
int fprintf( FILE* stream, const char* format, ... );
int sprintf( char* buffer, const char* format, ... );
int   scanf(                const char* format, ... );
int  fscanf( FILE* stream, const char* format, ... );
int  sscanf( char* buffer, const char* format, ... );
```

6. C has other I/O functions that are similar to `printf()` and `scanf()`
but use a **file** or a **string** instead of stdout or stdin.

a.	Which function would you use to **print** to a **file**?	
b.	What do you think the `s` in `sprintf` stands for?	
c.	Which function would you use to **read** from a **string**?	
d.	Could this function be called several times using the same string?	

7. We don't always know the exact format of input that a program should read.
Explain why it is often easier to scan a full line as a string, and then scan that string.

E. Try Yourself (~ 10 min)	

1. Write a C program that reads in 10 integers from standard input. You are guaranteed that the values provided to the program are integers within the int range.
The program should determine and display the smallest, largest and average value for the collection of the ten numbers.
You should write and compile the program outside of this document.

F. Conclusions

For more information, read documentation on C and the library functions.
K&R is still a useful reference, and a great example of a concise language introduction.
On Linux, type `man` (for manual) and the function or topic, e.g. `man printf`
There are also web sites with documentation for the C libraries.

Remember: *With great power comes great responsibility.*
With C, we can do things that would be difficult or impossible in other languages,
but that means we must be even more careful to write code that is clear and easy to read.

1986 Winning Entry, International Obfuscated C Code Contest

by Eric Marshall

```
                                                   extern int
                                                       errno
                                                         ;char
                                                            grrr
                             ;main(                           r,
  argv, argc )            int    argc                           ,
   r        ;           char *argv[];{int                     P( );
#define x  int i,       j,cc[4];printf("      choo choo\n"     ) ;
x  ;if    (P(  !        i              )     |  cc[  !      j ]
&  P(j    )>2  ?        j              :     i  ){*  argv[i++ +!-i]
;              for    (i=              0;;    i++                   );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```