ALGONQUIN COLLEGE

MODULE 2 : GETTING STARTED - C FUNDAMENTALS

Professor : Dave Houtman

Office: T323

Office Hrs: Monday 15:30 – 16:00

Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

Example 01 : 'Hello World'

Program Description:

Print "Hello World" out to the console

Note: 'Hello World' is often the first program one encounters in programming books in any language. More than just a dumb exercise, "Hello World" demonstrates that the programmer is capable of implementing the three minimum requirements essential for any piece of code. It shows that:

(1) the programmer can interface to and communicate with the hardware,
(2) the code can be successfully executed on that hardware, and
(3) the code can send a message to an output device to report on its status.

Trivia:

a. The first published example of "Hello World" in a computer book is found in K&R.
b. See http://en.wikipedia.org/wiki/List_of_Hello_world_program_examples for a list of 'Hello World' in *109* different programming languages.

Example 01: 'Hello World' Output

```
houtmad@ubuntu:~/examples$ Ex1
Hello World!
houtmad@ubuntu:~/examples$
```

Example 01: 'Hello World'

```
// This program outputs 'Hello World' to the screen

#include <stdio.h>

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

K&R 9

2.1 Comments

```
//    This program outputs 'Hello World' to the screen
```

- C permits two standard ways to add comments (also found in Java). Use the double slashes '//' to comment out single lines:

```
// This is called a line comment.  You can use
// as many as you want, but each line must
// start with a //.  Note that this format is
// not available in earlier versions of C
// prior to C99 (the 1999 ANSI/IEEE standard)
// and may trigger a warning depending on which
// version of the gcc compiler you are using.
```

2.1 Comments

```
/*    This program outputs 'Hello World' to the screen  */
```

- Alternately, you can use the `/*… */` format to add a whole block of comments. This format is also useful for commenting out blocks of code during debugging. This form of coding is allowed in all versions of C.

```
/*
   This is called a block comment. Everything
   following the slash-star is treated as a
   comment, up to the closing star-slash.  This
   is part of the original C standard and should
   not cause problems with any C compiler.
*/
```

K&R 12

2.1 Comments

- Note that you can use JavaDocs style comments in C without triggering an error, i.e.

```
/** A JavaDoc-style comment */
```

Format allowed, but JavaDoc-style comments not possible in C

but only because this looks just like a `/*…*/` comment to the C compiler. There may eventually be C IDEs that provide full JavaDocs support, but this isn't a standard feature in C or C++; Javadoc isn't supported at the present time.

- Does Eclipse support C/C++? Yes. See http://www.math.ucla.edu/~anderson/UsingEclipseCPP/, or http://www3.ntu.edu.sg/home/ehchua/programming/howto/eclipsecpp_howto.html for details. Note that Eclipse support for C/C++ is not as mature as its Java support. (Note: We will not be using Eclipse in this course.)

2.2 The C Preprocessor: CPP directives

```
// This program outputs 'Hello World' to the screen

#include <stdio.h>

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

K&R 74

K&R 190

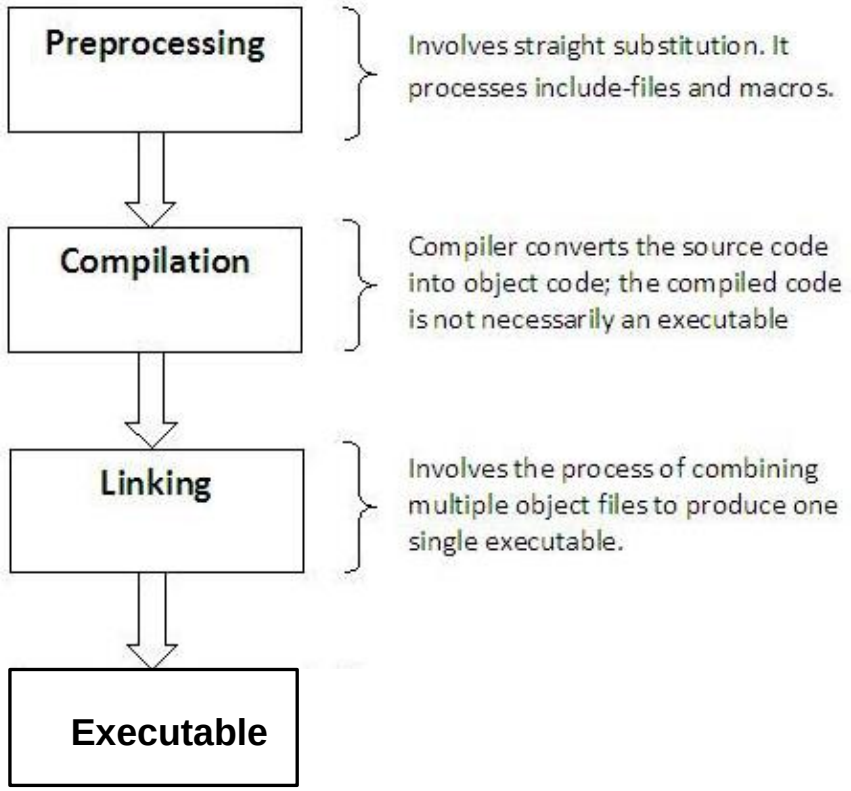
- **`#include`** is a ***C Preprocessor*** (***CPP***) ***Directive*** that tells the compiler to insert the contents of a library directly into the code *prior* to actual compilation—in this case, the **`stdio.h`** library. The **`stdio.h`** library contains the code for the `printf()` function, discussed shortly.

K&R 200

- As a crude first approximation, assume that **`#include`** in C does the same thing as **`import`** in Java—it provides access to a library of code.

2.2 The C Preprocessor: CPP directives

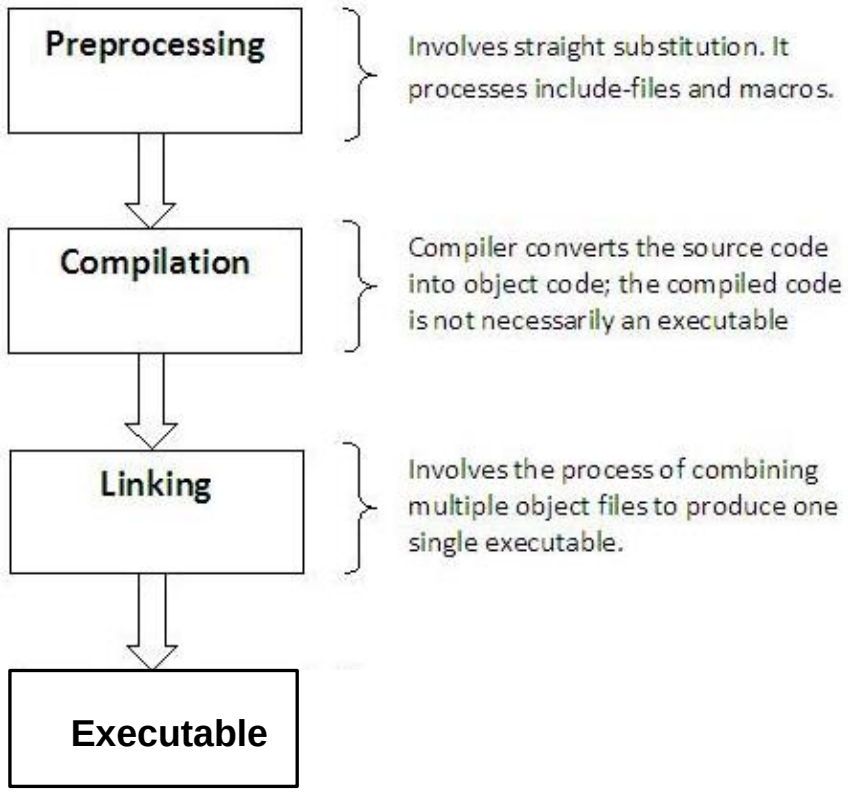
- C is a compiled language, and as such, it goes through a series of steps before an executable file is created; these steps are shown at right. While you have control over these features in most modern languages, they are often hidden by the IDE, which effectively automates the entire process of compilation. As with most things in C, your control over compilation/linking is far more direct than in most modern languages.



2.2 The C Preprocessor: CPP directives

- The first step in the process which eventually produces an executable file is ***preprocessing***. The preprocessor is the component of `gcc` that looks through your C code *prior* to compilation, checks for anything that starts with a # —used to signal all CPP directives—and if it's a valid command, performs the desired task.

K&R 188



2.2 The C Preprocessor: CPP directives

- CPP directives *are not part of the C Language* itself, but rather constitute a very small, separate, auxiliary language, designed to control the compilation process i.e. you determine which parts of the code will be compiled and which will not.

- Because they are *not* C language commands, CPP directives are *not* terminated by a ';'

- The CPP recognizes a standard set of twelve directives (to be discussed later); additionally, different compilers may include their own directives unique to a particular OS, platform, or compiler.

2.2 The C Preprocessor: CPP directives

- *Prior* to *actual* compilation of the C code, the CPP expands out and implements each preprocessor directive. For the code in Example 01, the first step would be to insert the actual contents of `stdio.h`:

So while the programmer sees this…	…the preprocessor substitutes this:

`#include <stdio.h>`
 	
`...`
	`#ifndef _STDIO_H`
	
	`#if !defined __need_FILE && !defined __need___FILE`
	`# define _STDIO_H 1`
	`# include <features.h>`
	
	`__BEGIN_DECLS`
	
	`# define __need_size_t`
	`# define __need_NULL`
	`# include <stddef.h>`
	
	`# include <bits/types.h>`
	`struct _IO_FILE;`
	`...// plus a thousand more lines of code`

Note that included files may contain other CPP directives (including still more `#includes`), which must be resolved **recursively** prior to actual compilation.

2.3 The C Preprocessor: `#include`

```
#include <stdio.h>
```

- As in Java, libraries may be either ***system libraries***, which are prepackaged with the language, or ***user-defined libraries***.

1) In C, ***system libraries*** are indicated by angular brackets, <>. For example, the CPP directive:

```
#include <math.h>
```

says: "look for the file **`math.h`** *in the usual place*". For most Linux variants, this means looking in the folder **`usr/include`**

2) ***User-defined libraries*** are indicated by double quotes. For example:

```
#include "myMathFtns.h"
```

means: "look in the current directory for the file **`myMathFtns.h`**, and then check the system **`$PATH`** for other possible locations."

2.4 main()

```
// This program outputs 'Hello World' to the screen

#include <stdio.h>

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

- As in Java, the **`main()`** function represents the starting point for program execution. Unlike Java, in each program there can only be one **`main()`** method.
- Note that there's really nothing special about `main()`; in most ways, its just another function. The only major differences are that (1) it is identified by the compiler as the starting point for program execution, and (2) it takes, and returns, certain standardized parameters.

2.4 main()

- In C, the full version of **main()** typically takes the form:

```
int main(int argc, char *argv[]){
```

This is C's version of

```
public static void main(String[] args)
```

in Java. Note that in Java, the argument of the function,

```
String[] args
```

replaces C's **int argc, char *argv[]**. As stated earlier, objects relieve you of much of the 'heavy lifting' that was previously done by pointers.

2.4 main()

- As with the Java version, there is some flexibility in this format. In particular, you can, in some cases, replace the stuff in brackets with `void`, giving just:

```
int main(void){...}
```

In some C compilers, the word `void` can often be dropped altogether, and you may be able to get away with just:

```
int main(){...}
```

2.4 main()

- As in Java, `main()` has a default integer return type, which is used to signal successful completion of the program to the operating system or calling program. In principle, Example #1 should return an integral value, using, at very least,

```
return 0;
```

But most C compilers are not this restrictive; you can omit the return type from `main()` with only a warning. But good practice dictates that you should use a proper return type.

2.5 stdlib.h()

- The system library **stdlib.h** contains a number of important system constants, including:

EXIT_SUCCESS

and

EXIT_FAILURE

These words are defined as integer values (equal to 0 and 1 respectively) that indicate the completion state of your code at end of execution. Thus you would use

```
return EXIT_SUCCESS;
```

or

```
return EXIT_FAILURE;
```

depending on whether the program exits successfully or not.

Example 01 Revisited

```
/*****************************************
  Declare include files
 *****************************************/

#include <stdio.h>
#include <stdlib.h>

/*****************************************
  Start main()
 *****************************************/

int main( void ){
      printf ("Hello World!\n");          // output message
      return EXIT_SUCCESS;
}
```

2.5 stdlib.h()

Note the inclusion of `stdlib.h`, in this program:

```
#include <stdlib.h>
```

K&R 208

`stdlib.h` is a large library that contains not just useful constants like `EXIT_SUCCESS` and `EXIT_FAILURE`, but a broad range of useful functions as well; you'll be seeing this library a lot. In brief, its functions belong to six groups:

1. Integer math
2. Algorithms, including a `rand()` function to generate random numbers
3. Text conversion, to cast chars to numbers, e.g. `atoi()`
4. Multibyte conversions (which need not concern us)
5. Storage allocation – a big topic to be discussed later on
6. Environmental factors – allows the program to talk to its environment

2.6 printf()

```
// This program outputs 'Hello World' to the screen

#include <stdio.h>

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

- **`printf()`** is a function defined in the **`stdio.h`** library.
- Since there are no classes in C, *there's no such thing as* *`System.out.println()`*. But **`printf()`** is (for now, for our purposes) the C equivalent.

2.6 printf()

- **`printf()`** is a standard function found in the **`stdio.h`** library which is used for outputting text strings to the command line. The simplest possible output has the form:

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

- In C, "`Hello World`" is passed to `printf()` as a constant string.

  Well, not really.

  There's no such thing as 'a string' in C (a string, after all, is an object). What is actually passed to `printf()` is the address of the first letter in the string. So what `printf()` actually receives in this case is *not* the string `"Hello World"`, but the *address* of the "H" at the location where the string is stored in memory.

2.6 printf()

Internally, during compilation, the C compiler looks at your string of characters and appends the `'\0'` escape character on to the end of it, to indicate *where the string terminates*. So what `printf()` *actually* gets passed during execution is an **address** that points to the *first* of twelve characters

```
'H' 'e' 'l' 'l' 'o' ' ' 'W' 'o' 'r' 'l' 'd' '\0'
```

which it prints out consecutively until it finds the termination character (the `'\0'`, which is not printed).

K&R 31

2.7 Character Constants

- As in Java, C uses escape characters to insert special directives or formatting information into the output. These should already be familiar to you:

Escape char	Operation
`\n`	Inserts a newline into the output stream.
`\t`	Inserts a tab into the output.
`\r`	Return; positions the cursor back to the start of the current line. Does not add newline.
`\\`	Insert a backslash into the output stream.
`\" or \'`	Insert double or single quotes.
`\0`	The null character; this is inserted at the end of a character string in C to signal string termination

K&R 37 K&R 155

2.7 Character Constants

Again, C deals with this internally by treating the appending the escape character(s) prior to the `'\0'`. Hence

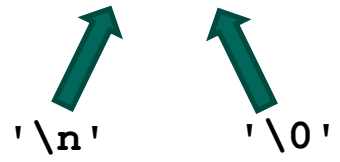
```
"Hello\n"
```

is internally converted to

```
'H' 'e' 'l' 'l' 'o' '\n' '\0'
```

Or, to translate this into its hexadecimal equivalent (using an ASCII table), this is:

```
48 65 6C 6C 6F 0A 00
```



2.7 Character Constants

- In gcc, double quotes are used with strings

```
"Hello\n"
```

whereas single quotes are associated with individual characters.

```
'H' 'e' 'l' 'l' 'o' '\n' '\0'
```

The distinction is important, since a character is a primitive data type, whereas a string is an array of characters (and therefore must be dealt with using pointers). This usage may be somewhat compiler dependent, but in gcc, you need to be aware of this.

- Other languages do not distinguish between "…" and '…'. For example, in JavaScript, strings are signaled by "…" or '…', as long as you use the same symbol to open and close the quote.

2.7 Character Constants

- if you don't insert the newline escape sequence (`'\n'`) after the string, like this

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

then, in the UNIX/Linux world, your output will be followed immediately by the command line prompt, like this:

```
Hello World$
```

System prompt for next command

rather than

```
Hello World
$
```

\n forces new line prior to system prompt

which you'd expect.

2.8 Formatting Characters

- As with Java, output is controlled by formatting characters, which always begin with a `%`. These characters, in the form `%Z`, act as placeholders for the values contained in variables that you want to display. Here's a short list of the most useful formats for this course:

Format char	Prints argument as	Format char	Prints argument as
`%c`	A character	`%e,%E`	A floating-point number e.g. 7.12e+00
`%d,%i`	An integer	`%f`	A floating-point number e.g 7.12
`%u`	An unsigned integer	`%s`	A string
`%lu`	A long unsigned integer	`%lf`	A double
`%x,%X`	An unsigned hexadecimal character	`%p`	A pointer to a void
		`%%`	Writes out a single %

Source: *A Book on C, 4e*, A. Kelley, I. Pohl. Addison Wesley, 1991. pg. 494

K&R 125

2.8 Formatting Characters

- Additional formatting is possible by inserting certain special characters between the % and the formatting character. Some of these possibilities are shown in the table below. Assume the following variables as arguments:

```
c = 'A',s[] = "Blue Moon",i = 123, double x = 0.123456
```

Format	Argument	Printed as:	Action performecd
%2c	c	" A"	Allow for width 2
%-3c	c	"A "	Width 3, left shifted
%.6s	s	"Blue M"	Precision: 6 spaces
%-11.8	s	"Blue Moo "	Width 11 allowed, precision 8, left shifted
%05d	i	"00123"	5 spaces allowed, 0 padded
%10.5f	x	" 0.12346	Width 10, precision 5
%-12.5e	x	"1.23457e-01 "	Left shifted, precision 5

Source: *A Book on C, 4e*, A. Kelley, I. Pohl. Addison Wesley, 1991. pg. 498

K&R 15

2.8 Formatting Characters

- Using a slightly more sophisticated example, we can use the formatting characters as place holders for other information. For example,

```
printf("Hello %s in %d\n", "World", 2014);
```

prints out

```
Hello World in 2014
```

Note that for each formatting character (`%Z`) in the initial string, there are an equal number of arguments, separated by commas, in the remainder of the parameter list passed to `printf()` – two in the above example: `"World"` and `2014`. Furthermore, the data type of each of the arguments in the remainder of the parameter list must conform to the type specified by its corresponding formatting character, thus

```
printf("Hello %s in %d\n", "World", 2014);
```

2.8 Formatting Characters

- There are many other ways to do the previous example, each equivalent to the next:

```
printf("Hello World in 2014\n");

printf("%s", "Hello World in 2014\n");

printf("%s\n", "Hello World in 2014");

printf("%s%d\n", "Hello World in ", 2014);

printf("%s %s %s %d\n","Hello","World","in",2014);

printf("%s%c%c %d\n", "Hello World ",'i','n',2014);
```

K&R 11

Notes on `printf()`

- C formats strings in a fashion similar to Java—but unlike C++ (which uses `cin` and `cout` streams for its I/O). But most of the rules for outputting strings should look familiar to you from your Java experience.
- In Java, there's more than one way to print out information; so too in C. While we are not limited strictly to `printf()`, this is the most basic function for outputting information to the screen.
- The '`f`' in `printf()` stands for 'formatted', since the command uses `%d`, `%s`, etc. to format its output.
- `printf()` is, in fact, one of a family of console output functions available for use…
- …however, there is no such thing as `println()` or `prints()` in C.

(See http://cboard.cprogramming.com/c-programming/147435-writing-println-code.html)

Notes on `printf()`

- You can use multiple strings inside `printf()`, without the need for continuation characters to tie the strings together. For example:

```
printf("Hello World!" "Have a nice day, eh?" "Cheers\n");
```

*Whitespace between strings is the equivalent to **concatenation**.*

- Additionally, `printf()` ignores any carriage returns passed to the function:

```
printf("This is one very long sentence, "
"written over several lines with carriage returns "
"all over the place "
"but printf ignores all the extra stuff and "
"prints out just what's written.\n");
```

```
houtmad@ubuntu:~/examples$ Ex1v2
This is one very long sentence, written over several lines with carriage returns
 all over the place but printf ignores all the extra stuff and prints out just w
hat's written.
houtmad@ubuntu:~/examples$
```

Questions

1. Identify six compile-time errors in the following C code sample, and indicate how you would fix them:

```
#include "stdio.h";

int main(){
  printf('%s%d\n', "Hello World in", "2014")
  return (EXIT_SUCCESS);
}
```

2. Assume the following `printf()` statement appears in an otherwise correctly-formatted `main()` function. What is the output? How would you correct this to display the true intent of the code?

```
printf("%s", "C strings are terminated with \0\n");
```

3. True or false: A `printf()` statement always takes as many arguments/parameters as it has formatting characters (i.e. `%s`, `%d`, etc.)

Example 02 : Data type, size, and range

Program Description:

Most programming languages require that the programmer declare variables for information storage and manipulation during program execution. Such variables rely on the concept of ***data type***, which determines the size (the number of bits/bytes) and ***range*** (of values) of the data that can be stored.

C contains six primitive data types: `char`, `short`, `int`, `long`, `float` and `double`. Additionally, the first four can be ***signed*** or ***unsigned***. Write a program that shows the size of these data types in bytes, along with the range of values they can hold.

Example 02: Data type, size, and range output

```
houtmad@ubuntu:~/examples$ Ex2v3
The size and range of each data type is:

        char:           1 byte(s), from -128 to 127
        unsigned char:  1 byte(s), from 0 to 255
        short:          2 byte(s), from -32768 to 32767
        unsigned short: 2 byte(s), from 0 to 65535
        int:            4 byte(s), from -2147483648 to 2147483647
        unsigned int:   4 byte(s), from 0 to 4294967295
        long:           4 byte(s), from -2147483648 to 2147483647
        unsigned long:  4 byte(s), from 0 to 4294967295
        float:          4 byte(s), from 1.175494e-38 to 3.402823e+38
        double:         8 byte(s), from 2.225074e-308 to 1.797693e+308

houtmad@ubuntu:~/examples$
```

2.9 The `sizeof()` operator

K&R 165

- Every C compiler contains an ***operator*** called **`sizeof()`**. This useful operation—the only built-in "function" in the language—returns the size of a data type in bytes. For example `sizeof(int)` typically returns 4, `sizeof(float)` returns 4, `sizeof(unsigned short)` returns 2, etc.

  This is useful when your code has to run on multiple hardware platforms, and you need to know, for example, if an integer occupies 2 bytes or 4 bytes, and is the memory address size 16 bits, 32 bits or 64 bits?

  So

```
printf("Size of an int is %u bytes\n", sizeof(int));
```

  prints out

```
Size of an int is 4 bytes
```

  on a PC whose C compiler uses 4 bytes for integer data types.

2.9 The `sizeof()` operator

- **`sizeof()`** returns an unsigned integer on 32-bit PCs and therefore requires the **`%u`** format specifier for print out to the console. On 64-bit architectures, you'll need to use the **`%lu`** specifier instead. (There's more to say on this subject later. In fact, C solves this problem for you);

  For the purpose of this course, we will assume a 32-bit architecture in every example. Therefore, `sizeof()` always returns an unsigned int (i.e. 4 bytes). If you are using a 64-bit architecture, you'll need to change **`%u`** to **`%lu`** in order to print out the value returned by `sizeof()`.

  Therefore, as a first approximation of our Example 02 code, we might write the following:

Example 02: Data type, size, and range

```
#include <stdio.h>
#include <stdlib.h>

int main(void){

    printf("The size of the data types "
           "for this compiler are:\n\n");
    printf("\tchar:    %u byte(s)\n", sizeof(char));
    printf("\tshort:   %u byte(s)\n", sizeof(short));
    printf("\tint:     %u byte(s)\n", sizeof(int));
    printf("\tlong:    %u byte(s)\n", sizeof(long));
    printf("\tfloat:   %u byte(s)\n", sizeof(float));
    printf("\tdouble:  %u byte(s)\n", sizeof(double));
    printf("\n");
    return EXIT_SUCCESS;

}
```

Example 02: Data type, size, and range

…and the output (using the current gcc compiler) is:

```
houtmad@ubuntu:~/examples$ Ex2
The size of the data types for this compiler are:

        char:    1 byte(s)
        short:   2 byte(s)
        int:     4 byte(s)
        long:    4 byte(s)
        float:   4 byte(s)
        double:  8 byte(s)

houtmad@ubuntu:~/examples$
```

2.9 The `sizeof()` operator

- C++ supports the **`sizeof()`** operator; Java doesn't. Java is—theoretically at least—platform independent. Consider the following:
  - If an integer is *always* 4 bytes long (regardless of the hardware it runs on), you never need to use a `sizeof()` operator to find out how many bytes it occupies on a particular architecture: *its always the same in Java*.
  - If you use objects, you never need to know whether the platform your code runs on uses 32-bit addresses or 64-bit addresses—only the JVM for that machine cares.
  - Furthermore, since memory management is handled automatically by the JVM, you'll rarely need to know exactly how much memory to create—hence no need to know its size in bytes.
  - In C, you may need to know how much space needs to be freed up at any point in time, while in Java, garbage collection handles that particularly problematic detail automatically. Java knows; C doesn't.

  So platform independence makes `sizeof()` obsolete in Java—although whether the language *should* have this operator *is* a point of contention.

2.10 C data types

- Since Java contains all of C's data types, the following table should look familiar:

Value	Data Type	Size (bits)	Size (bytes)	Used for
character	**char**	8	1	ANSI character set
integer	**short**	16	2	Integer values
integer	**int**	16	4	Integer values
integer	**long**	32	4	Large Integers
decimal	**float**	32	4	Floating-point values
decimal	**double**	64	8	Very large floating-point values

As stated earlier, the C standard is 'flexible' in its interpretation of data type sizes. C was written to run on different hardware platforms with different memory requirements. So the above values, while part of the C standard, should be taken as minimum guidelines rather than fixed values.

2.10 C data types

- In actual practice, there is no difference between `shorts` and `ints` in C, whether `signed` or `unsigned`. (However, see http://stackoverflow.com/questions/12279060/difference-between-short-int-and-int-in-c for a useful discussion of the subject.)
- Depending on its implementation, a C compiler may offer **`long long`** or **`long double`** data types. These are non-standard types that are HW dependent and are therefore not generally portable between platforms.
- While there are 'officially' six primitive data types in C (shown above), it is useful to imagine two more 'unofficial' data types, to be discussed later:

Value	Data Type	Size (bits)	Used for
N/A	**`* (pointer)`**	(depends on HW)	Stores an address in memory
N/A	**`void`**	N/A	Indicates the absence of type

2.10 C data types

K&R 35

- There are *long forms* and *short forms* of data type declarations which, again, may be somewhat compiler dependent. So you may see the following:

Short form	Long form	C compiler interpretation
`short`	`short int`	`signed short int`
`unsigned short`	`unsigned short int`	`unsigned short int`
`int`	`signed int`	`signed int`
`unsigned`	`unsigned int`	`unsigned int`
`long`	`long int`	`signed long int`
`unsigned long`	`unsigned long int`	`unsigned long int`

So by default, any data type that is missing `signed` or `unsigned` is assumed to be the `signed`—it stores + and – values by default.

Note also that the default data type is assumed to be an `int`. So if you say `unsigned` and don't follow it with a data type, you've declared an `unsigned int`. Generally, this use of `unsigned` is considered bad practice, although it still shows up frequently in old code.

NOTE: C, C++, and Java Data Types

- C lacks Java's `boolean` data type; there's no such thing as `true` or `false`, unless you specifically define these values in code;
- C's `char` type is only one byte wide, and so can only hold—at most—the extended ASCII character set, 256 characters in all. C++ has an additional type, called `wchar_t`, designed to support the Unicode character set.
- When a C program strictly conforms to the ANSI standard, C's data types are *not* the same size as Java's. For example, Java's `int`s are 4 bytes, ANSI C's are 2; Java's `char`s are 2 bytes (capable of holding UNICODE values), while C's `char`s occupy 1 byte (for ASCII); and a Java `long` value occupies twice as many bytes as C's version (which is 4 bytes, not 8);
- In Java, all data types are `signed` by default (i.e. any type can hold either positive or negative values). In C, data types are automatically signed, but they may be declared as `unsigned`. An unsigned value has roughly double the positive range of a signed value

2.11 Data type range

- The following table indicates the variety of combinations of signed and unsigned data types, along with their typical ranges

un/signed	C Data Type	Byte Size	Range
unsigned	**char**	**1**	0 to +255
signed	**char**	**1**	-128 to +127
unsigned	**short**	**2**	0 to +65535
signed	**short**	**2**	-32768 to +32767
unsigned	**int**	**2**	0 to 65535
signed	**int**	**2**	-32768 to +32767
unsigned	**long**	**4**	0 to +4294967295
signed	**long**	**4**	-2147483648 to +2147283647
	float	**4**	–3.4028235E+38 to –1.40129845e–45 1.4012985e–45 to 3.4028235E+38
	double	**8**	–1.7976931348623157E+308 to –4.94065645841246544e–324 4.94065645841246544e–324 to 1.7976931348623157E+308

2.11 Data type range

- The size of C data types is compiler dependent. The only guarantee dictated by the C standard is that

```
sizeof(char) = 1

sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)

sizeof(signed) = sizeof(unsigned) = sizeof(int)

sizeof(float) <= sizeof(double) <= sizeof(long double)
```

Source: *A Book on C, 4e*, A. Kelley, I. Pohl. Addison Wesley, 1991. pg. 123

Therefore, nothing prevents a C compiler from having `shorts`, `ints`, and `longs` that are each 8 bytes in length.

2.11 Data type range

- the files `limits.h` and `float.h` contain predefined constants for integer and floating point types that indicate the limits for each data type allowed by the C compiler. These constants are typically named **`CHAR_MIN`**, **`CHAR_MAX`**, **`SHRT_MIN`**, **`SHRT_MAX`**, …**`FLT_MIN`**, **`FLT_MAX`**. For the integer values, these exist in both signed and unsigned versions.

K&R 213

- These values are fixed using the **`#define`** CPP:

```
#define CHAR_MIN -128
#define CHAR_MAX 127
#define UCHAR_MAX 255
...
```

which we cover in the next section

Example #2 revisited

- We can use the values defined in `float.h` and `limits.h` to output the range allowed by each data type. For example, for the floating point type:

```
#include <stdio.h>
#include <float.h>

int main(void){

   printf("The size of the float data type is ");
   printf("%u byte(s)\n  ", sizeof(float));
   printf("from %e\t to %e\n", FLT_MIN, FLT_MAX);

}
```

```
houtmad@ubuntu:~/examples$ Ex2v2
The size of the float data type is 4 byte(s) from 1.175494e-38 to 3.402823e+38
houtmad@ubuntu:~/examples$
```

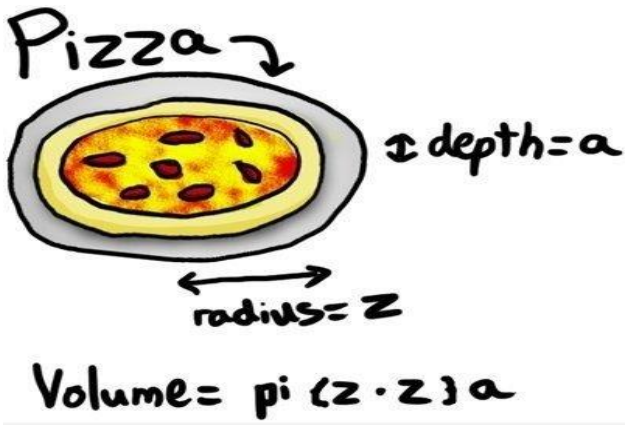
Example #2 revisited

- Finally, we can put all the pieces together to create the program that gives the desired output:

```
#include <stdio.h>
#include <limits.h>
#include <float.h>

main () {

  printf ("The size and range of each data type is:\n\n");
  printf ("\tchar:\t\t%d byte(s), from %d to %d \n",
      sizeof(char          ), CHAR_MIN, CHAR_MAX  );
  printf ("\tunsigned char:\t%d byte(s), from %d to %d \n",
      sizeof(unsigned char ), 0       , UCHAR_MAX );
  printf ("\tshort:\t\t%d byte(s), from %hi to %hi \n",
      sizeof(short         ), SHRT_MIN, SHRT_MAX  );
   ...
```

(The complete program listing is given in Ex2.c)

Example 03 : Pizza slice calorie calculator



Program Description:

This program calculates the number of calories in a slice of pizza based on the diameter of the pizza (in inches, input by the user) along with a number of predetermined constants, such as the average number of calories per gram of plain pizza (cheese and dough only).

The total pizza volume is approximated by a cylinder as:

$$\text{volume} = \pi * (\text{diameter}/2)^2 * \text{height}$$

The mass is obtained by multiplying this volume by the "pizza dough density" factor (taken to be 0.35 g/cm3); the total number of calories in the entire pizza is just the mass times the calories per gram of pizza (assumed to be 2.86 cal/g). Finally, we divide the result by the number of slices per pizza (assumed equal to 12) to get the total number of calories per slice.

Example 03: Pizza slice calorie counter

```
houtmad@ubuntu:~/examples$ Ex3
To calculate the calories in a pizza slice
enter the diameter of the pizza in inches: 14
The number of calories in a slice of 14 inch pizza is 149.1
houtmad@ubuntu:~/examples$
```

Example 03: Pizza slice calorie counter

```
#include <stdio.h>
#include <stdlib.h>
#define PIZZA_HEIGHT          1.8
#define DOUGH_DENSITY         0.35
#define CALORIES_PER_GRAM     2.86
#define SLICES_PER_PIZZA      12
#define PI                    3.1415927
#define CM_PER_INCH           2.54

int main(void){
   float diameter = 0, radius, volume, mass, calories_per_slice;
   printf("To calculate the calories in a pizza slice\n");

   printf("enter the diameter of the pizza in inches: ");
   scanf("%f", &diameter);                    // NOTE:diameter in inches

   radius = (diameter * CM_PER_INCH)/2.00;  // NOTE:radius in cm
   volume = PI * radius * radius * PIZZA_HEIGHT;
   mass = volume * DOUGH_DENSITY;

   calories_per_slice = (mass * CALORIES_PER_GRAM)/SLICES_PER_PIZZA;
   printf("The number of calories in a slice of %d inch pizza"
      " is %-5.1f\n", (int)diameter, calories_per_slice);
}
```

2.12 Using `#define` to create constants

```
#define PIZZA_HEIGHT    1.8
```

- The **`#define`** CPP performs textual substitution on your code prior to compilation. It serves *two* purposes in C:

K&R 17

  - it can be used to assign fixed values prior to compilation, setting the value at the end of the statement equal to the string literal that precedes it. Thus

```
#define CHAR_MIN -128
```

K&R 189

    says: wherever **`CHAR_MIN`** appears in the code, insert the value -128 *prior* to compilation. i.e. **`#define`** can be used to create pre-compile-time constants;

  - It can also be used to build functional components out of simpler elements, i.e. to create macros—coming up shortly

2.12 Using `#define` to create constants

- Here's another example of a **`#define`** in practical use:

```
#define ERRMSG "****Error %d: Web site %s.****\n"
```

This could then be used, for example, in:

```
printf(ERRMSG, 401, "unauthorized");
printf(ERRMSG, 404, "not found");
```

This would print out:

```
****Error 401: Web site unauthorized.****
****Error 404: Web site not found.****
```

So `#define` is effective not just for creating literal constants and macros: it can be used to provide consistent output across your application.

2.12 Using `#define` to create constants

- In programming jargon, this usage, in creating something which is not strictly required for the proper execution of the program, but which simplifies reading the code (sometimes enormously), is known as ***syntactic sugar***. Note that some syntactic sugar is necessary in most code, since the alternative is often far less comprehensible.

2.12 Using `#define` to create constants

- As seen in example #2, **`#define`** can be used to declare values prior to compilation which remain fixed forever after—constants, essentially.

```
#define PIZZA_HEIGHT                1.8
#define DOUGH_DENSITY               0.35
#define CALORIES_PER_GRAM           2.86
#define SLICES_PER_PIZZA            12
#define PI                          3.1415927
#define CM_PER_INCH                 2.54
```

2.13 Adding User-declared .h files using #include

- In programs with a lengthy list of defines, it is useful to place these declarations in a separate, user-defined .h file. For example, we might put all pizza-related information in a file called **`pizzainfo.h`**

```
// Basic Pizza Logistics
#define PIZZA_HEIGHT                    1.8
#define DOUGH_DENSITY                   0.35
#define CALORIES_PER_GRAM               2.86
#define SLICES_PER_PIZZA                12

// Toppings - Added calorie content per slice
#define ANCHOVIES_ADDED                 0.82
#define EXTRA_CHEESE_ADDED              1.08
#define TOMATOES_ADDED                  0.09
#define PEPPERONI_ADDED                 0.83
#define GREEN_PEPPERS_ADDED             0.42
#define RED_PEPPERS_ADDED               0.41
#define ONIONS_ADDED                    0.28
```

2.13 Adding User-declared .h files using #include

- This allows our original program to become more simplified:

```
#include <stdio.h>
#include <stdlib.h>
#include "pizzainfo.h"

#define PI                  3.1415927
#define CM_PER_INCH         2.54

int main(void){
   float diameter = 0, radius, volume, mass ...
   //etc
```

Note the use of the "" to indicate a user-defined library. Also recall that **`pizzainfo.h`** needs to be in either the same directory as your .c program, or it must be somewhere in the `$PATH`.

Finally, note that unused `#define`s do not increase the size of the executable file in any way, since they are…unused.

2.14 Using `const` to create constants

- C offers a second way to declare constants, using the keyword **`const`**. Substituting

K&R 36

```
const float PI = 3.1415927;
```

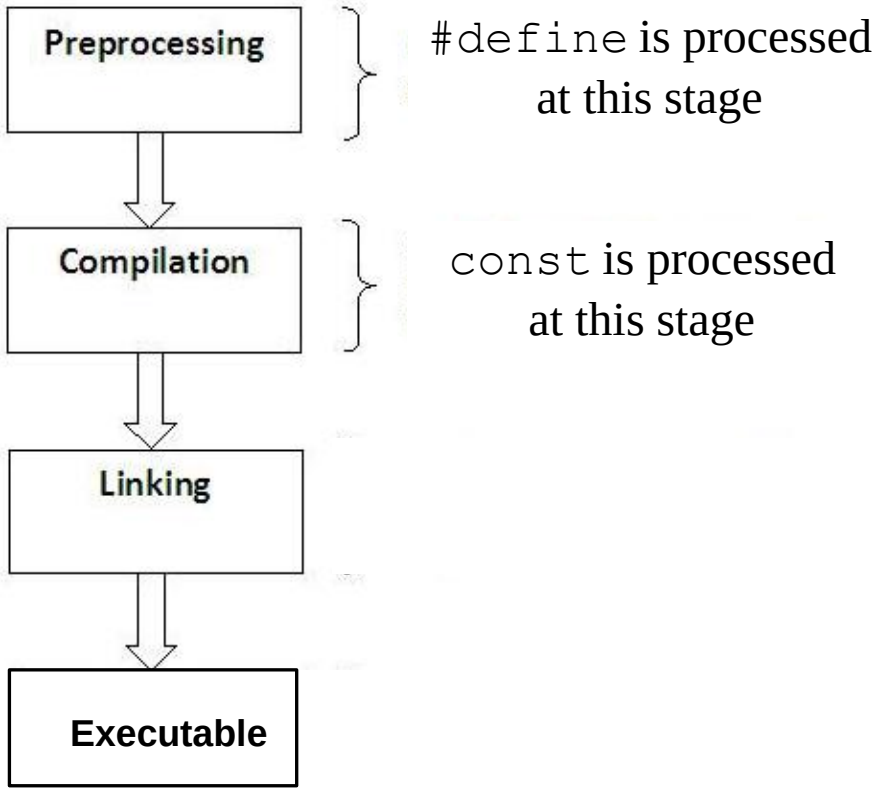
for

```
#define PI 3.1415927
```

gives exactly the same results.

2.14 Using `const` to create constants

So `#define` and `const` have essentially the same effect on our program—they create constant values—but affect different parts of the program in different ways during the compilation process.



2.14 Using `const` to create constants

- `const` is used to indicate to the compiler that a value will not change during execution. For example:

```
const float PI = 3.1415927;
```

  sets the value **`PI`** equal to 3.1415927 and indicates that the value should be *treated* as fixed throughout execution. Any attempt to change **`PI`** in code will flag an error.

- Note that the definition above involves the actual allocation of memory—you need to reserve 4 bytes to hold the floating point value **`PI`**. No such memory is allocated by the `#define` statement—it's straight substitution.

- The chief advantage of using `const` is that, when used properly, it allows the compiler to optimize your code. When you indicate that a value will not be changed during execution, this helps the C compiler to optimize your code, speeding execution.

2.14 Using `const` to create constants

- Unfortunately, the suggestion that `const` *guarantees* that a value *cannot* be changed is not strictly true. Like any declaration, it's a *statement of intent* on the part of the programmer. By using pointers, it is possible to change this value by writing into the memory where **`PI`** is located. A better term to reflect what `const` is supposed to do is `"readonly"`, i.e. the programmer intends only to read from this variable's memory address, never to write something new to it.
- Nonetheless, `const` has its uses, things for which you can't use `#define`, as we'll see later when we deal with pointers.

2.15 Declaring variables

```
float diameter = 0, radius, volume, mass, calories_per_slice;
```

- The rules for variable names are the same as in Java:
  - The first character can be any letter, either uppercase or lowercase, or an underscore. But it should not be a number. (However, this rule *is* compiler dependent. For example, gcc allows a number at the start of an identifier.)
  - After the first character, subsequent characters can be any letter, any number, or an underscore.
  - Aside from the underscore, special characters, like `&`,`*`,`-`,`~`,`%` etc. are prohibited in identifiers.

2.15 Declaring variables

- While permitted, it is generally inadvisable to start a variable name with an underscore; identifiers used in libraries are typically prefixed with '_' while certain predefined constants are prefixed with '_ _'. In either case you should avoid starting a function, variable or constant with '_' since it might already be reserved for use by the C compiler.

  For example, `printf()` is represented internally by the function `_Printf()`.

- Names are ***case sensitive***, thus `xyz` is different from `xYz`.

- You cannot use reserved words as identifiers; so `for`, `if`, and `while` are not allowed as variable/function names. Of course, the previous rule means that nothing prevents you using `foR`, `IF`, or `While` as names…but this should generally be avoided.

- In Microsoft documentation, variables are often prefixed with one or sometimes two characters (like i, c, d or f) to indicate the data type (e.g. int, char, double, or float). This is called ***Hungarian notation***; it is widely used.

2.15 Declaring variables

- Because of its inherent flexibility, some C compilers allow special characters to be used in identifiers (such as the $ character). But you shouldn't assume this is a standard feature of C—it is the exception, rather than the rule.

- According to ANSI C specifications, distinct variable names can have at least 31 characters. This is a minimum; depending on the implementation, some compilers may allow many more—`gcc` allows 63 chars max. However, some caution is in order here. Some compilers may *allow* X characters, but internally truncate a name to (X-N) characters, leading to potential naming conflicts. For example, if a compiler allows 63 characters in a variable name but uses only 32 internally, then the following declarations will trigger an error:

Potential naming conflict

```
int thisIsAVeryLongVariableNameOfTypeInt;
int thisIsAVeryLongVariableNameOfTypeIntWithSomeExtraChars;
```

since both get truncated to **`thisIsAVeryLongVariableNameOfTyp`**

2.16 Keywords

K&R 154

- The following words are reserved in one or more implementations of C; *they cannot be used as identifiers.*

`asm`[1]	`default`	`float`	`register`	`switch`
`auto`	`do`	`for`	`return`	`typedef`
`break`	`double`	`fortran`[2]	`short`	`union`
`case`	`else`	`goto`[3]	`signed`	`unsigned`
`char`	`entry`[2]	`if`	`sizeof`	`void`
`const`	`enum`	`int`	`static`	`volatile`
`continue`	`extern`	`long`	`struct`	`while`

1. Traditionally reserved, but not found in ANSI C, `asm` is nonetheless frequently used in modern compilers, e.g. Microsoft C, Visual C++, and Borland C; it is used for inserting assembly language into programs.
2. Reserved in the original C, but never used in modern ANSI C; `entry` and `fortran` are essentially obsolete keywords.
3. Legal, but it is considered bad form to use this keyword in all but a few very rare cases. Use this at your own risk, since its use guarantees that you will certainly be mocked and ostracized by your peers, even when it is used appropriately.

Source (with modifications): *C: A Reference Manual 3e*, S. Harbison, G. Steele. Prentice Hall, 1991. pg. 16

2.17 Making multiple declarations on the same line

```
float diameter = 0, radius, volume, mass, calories_per_slice;
```

- C allows the programmer to declare multiple variables on the same line and initialize them simultaneously. The above code is equivalent to the following separate declarations:

K&R 38

K&R 72

```
float diameter = 0;
float radius;
float volume;
float mass;
float calories_per_slice;
```

- This ability extends to other data types, as we shall see later, including arrays, structures, pointers to data types, and even forward declarations to functions.

2.18 scanf()

K&R 128

```
scanf("%f", &radius);
```

- **scanf()** is C's version of `Scanner()`—it allows simple input on formatted data.
- In Java, you instantiate a `Scanner` object—perhaps called `input`—and then read in data of a certain type using, e.g., `input.nextInt()`. **scanf()** works in much the same way, except that here you specify the data type using `%d`, `%e`, `%f`, `%u`,…etc. (rather than `nextInt`, `nextFloat`, `nextLong`…)
- Since the user enters a carriage return to enter a value, you do not need to use `\n` when using **scanf()**; this feature is provided by your input.

2.18 scanf()

- You *cannot* use the width, precision, and alignment formats like

```
printf("%3.2d" radius);
```

in `scanf()`. The line:

```
scanf("%3.2d", &radius );
```

flags an error.

`scanf()` does not allow specialized formatting on input

- Just as `printf("Hello")` passes the *address* of the first character in the string argument to the function, `scanf()` requires the address where the data *is to be stored*. The '`&`' in front of the variable name has the effect of passing the *address* of that variable to `scanf()`, as required. Learn to read the ampersand '`&`' (when it appears before a variable) as: "the address of".

2.18 scanf()

- Note that if you use **radius** rather than **&radius** in this function, then you are passing the *contents* of **radius** to `scanf()`, rather than the *address* of `radius`. Since `scanf()` expects to see an address and **radius** *contains* a floating point type, this will trigger an error: **&radius** is an address; **radius** is not—it's what's located *at* that address.
- This rule only applies to the six primitive data types: `char`, `short`, `int`, `long`, `float` and `double`; it does ***not*** apply to reference data types like strings and arrays, which are automatically referred to by their address.
- In Java, there's more than one way to read in and print out information; so too in C. While we are not limited strictly to `printf()` and `scanf()`, these are the two basic functions for outputting and inputting values into our code.
- Like `printf()`, `scanf()` is one member of a family of functions designed to read in data.

2.19 Using `math.h`

- As a final note, as with `EXIT_SUCCESS`, `EXIT_FAILURE`, `CHAR_MIN` and `CHAR_MAX`, the value of π is predefined in a C library. `math.h` contains the line

`#define M_PI 3.14159265358979323846`

and you should use this rather than define your own version of π. i.e.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pizzainfo.h"

#define CM_PER_INCH          2.54

int main(void){
   float diameter = 0, radius, volume, mass, calories_per_slice;
   ...
   volume = M_PI * radius * radius * PIZZA_HEIGHT;
   ...
}
```

2.19 Using `math.h`

K&R 207

Note that while most of the regular mathematics operations `+`, `-`, `*`, `/` and `%` are native to C, almost all of the important math functions are located in the `math.h` library. (The exception is the `rand()` function, listed at bottom)

Function	Operation
`sqrt()`	Take the square root of a double and return a double
`pow()`	Takes in two doubles and raises the first to the power of the second, returning a double
`exp()`	Takes the exponent of the double value input and returns a double, i.e. e^x
`log()`	Takes the natural log, *ln*, of the value entered and returns a double
`sin()`	Takes the sine as a double and returns a double
`cos()`	Takes the cosine as a double and returns a double
`tan()`	Takes the tangent as a double and returns a double
`rand()`	Returns a random number between 0 and `RAND_MAX`; NOTE: in ANSI C this is located in `<stdlib.h>`, not `<math.h>`

Review - Example 03: Pizza slice calorie counter

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pizzainfo.h"

#define CM_PER_INCH   2.54

int main(void){
   float diameter = 0, radius, volume, mass, calories_per_slice;
   printf("To calculate the calories in a pizza slice\n");

   printf("enter the diameter of the pizza in inches: ");
   scanf("%f", &diameter);

   radius = (diameter * CM_PER_INCH)/2.00;
   volume = M_PI * radius * radius * PIZZA_HEIGHT;
   mass = volume * DOUGH_DENSITY;

   calories_per_slice = (mass * CALORIES_PER_GRAM)/SLICES_PER_PIZZA;
   printf("The number of calories in a slice of %d inch pizza"
       " is %-5.1f\n", (int)diameter, calories_per_slice);
}
```

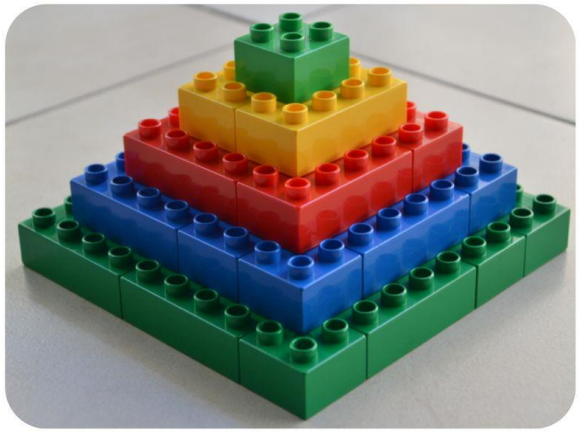
Questions

1. Does the following code fragment compile properly? If not, then explain why. If so, then explain its effect.

```
#define GBs 4;
...
const int number_of_gigabytes = GBs;
printf("There are %d gigabytes\n", GBs);
```

2. Assume question 1 above compiles correctly (make any corrections you feel are necessary). If you swap the two lines shown, will the program still compile properly?

Example 04 : Count the number of blocks in a Pyramid



Program Description:

Assume a three-dimensional pyramid consists a stack of equal-sized cubes, with the width of each layer increasing by one block on each level of the structure. Therefore, the first (top) layer consists of 1 block; the second layer adds $2^2 = 4$ blocks, for a total of 5 blocks; the next layer adds $3^2 = 9$ for a total of 14 blocks; and so on. This program calculates the answer to the question: What will be the total number of blocks in the pyramid given its height (in blocks)?

Example 04 : Volume of a Pyramid

```
houtmad@ubuntu:~/examples$ Ex4
Enter pyramid height in blocks: 2
The total volume of a pyramid of height 2 is 5 blocks
houtmad@ubuntu:~/examples$ Ex4
Enter pyramid height in blocks: 3
The total volume of a pyramid of height 3 is 14 blocks
houtmad@ubuntu:~/examples$ Ex4
Enter pyramid height in blocks: 4
The total volume of a pyramid of height 4 is 30 blocks
houtmad@ubuntu:~/examples$ Ex4
Enter pyramid height in blocks: 60
The total volume of a pyramid of height 60 is 73810 blocks
houtmad@ubuntu:~/examples$
```

Example 04 : Volume of a Pyramid

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main (void){

   int hPyr = 0, hCtr, totVol = 0;

   printf("Enter pyramid height in blocks: ");
   scanf("%d", &hPyr);
   assert(hPyr > 0);

   for (hCtr = 1; hCtr <= hPyr; hCtr++)
      totVol += hCtr * hCtr;

   printf("The total volume of a pyramid of height "
      "%d is %d blocks\n", hPyr, totVol);

   return EXIT_SUCCESS;
}
```

2.20 assert()

K&R 210

```
assert(hPyr > 0);
```

- The C library <assert.h> contains a simple error-checking function called assert() —the forerunner of throw, try and catch in C++ and JavaScript. assert() is a primitive form of an ***exception handler***. It's format is:

```
assert(if_false);
```

i.e. assert is triggered when the expression in brackets is *not* true. If the assertion is false, then the system will print out an error message and abort the program. Hence

```
assert(hPyr > 0);
```

says: "flag an error if hPyr is *not* a positive number"; if the assertion is true, assert does nothing, and execution proceeds as normal.

2.21 for loops

K&R 53

```
for (hCtr = 1; hCtr <= hPyr; hCtr++)
```

- A `for` loop in C looks and acts just like a `for` loop in Java (and C++). Each `for` loop contains three complete statements, separated by semicolons:

```
for(initialization; comparison; increment){…}
```

where:

`initialization` allows one or more variables to be initialized
`comparison` determines whether the loop continues or not
`increment` allows one or more variables to be incr/decremented

Commas can be used to separate expressions within the `for` loop, e.g.:

```
for(x=3, y=4; ((y < 3) || (x > 4)); y++, x--)
```

You can omit any or all of the three statements, eg. **`for(;;)`** is allowed

2.21 `for` loops

K&R 53

- Historically, the loop control variable (LCV, `hCtr` in the above case) was declared separately in C programs, prior to the `for` loop itself. As of ANSI C99, you can declare the LCV inside the initialization part of the `for` statement:

```
for (int hCtr = 1; hCtr <= hPyr; hCtr++)
```

If your compiler is not set up to use the C99 standard, the newer `for` loop will cause the compiler to throw an error. To remove the error, specify **`-std=c99`** as an option in `gcc`.

- The enhanced `for` loop found in Java does not exist in C; it exists in C++ but only in later versions of that language.

2.22 Operators and expressions

K&R 39

```
totVol += hCtr * hCtr;
```

- All of the operators and expressions common to Java are also found in C—with the exception of Java's string operator (+, for concatenation).

Operator Type	Purpose
Arithmetic	Take variables and literals as their operands and return a single value, e.g. using +, -, /, *, %, ++, --.
Assignment	Set one value equal to another[1]
Comparison	Compare two values and return true (1) or false (0)
Logical	Operate on Boolean expressions and return true (1) or false (0)
Bitwise	Similar to Logical operators, except that they operate on each bit separately

1. shortform operators, like the += operator used above (they are sometimes called compound operators), are a mixture of arithmetic and assignment operators combined into one expression.

2.22 Operators and expressions

- Whenever a sequence of operations is performed, the language needs to have a built-in order of precedence – a set of rules that determine in which order the operations are preformed. Higher precedence operations are performed first. For arithmetical expressions this order is:

K&R 48

++ and -- (prefix)	
Parentheses	`( )`
Division/Multiplication/Modulus	`/ * %`
Addition/Subtraction	`+ -`
Assignment and shortform Operators	`=, +=, -=, etc.`
++ and -- (postfix)	

- Note: the order of precedence is probably more important to know in C than in Java, since C makes more use of (fairly) compact, dense expressions. The two most important things to remember are:

(1) everything in parentheses gets looked at first (except prefix), and
(2) the assignment ("=") happens after everything else (except postfix).

2.23 Shortform Operators and Expressions

K&R 46

- All C-based languages, including C++, Java and JavaScript, have 'shortcut' operators that frequently allow for less verbose output than writing mathematical expression in the 'normal' way. The following are all valid shortform operators

```
num1 += 6;          // addition    : num1 = num1 + 6;
num2 %= 12;         // modulus     : num2 = num2 % 12;
num3 &= 0x0FFF;     // bitwise AND: num3 = num3 & 0x0FFFh;
num4 |= 0xAAAA;     // bitwise OR : num4 = num4 | 0xAAAAh;
num5 ^= 0x03;       // bitwise XOR: num5 = num5 ^ 0x03h;
num6 <<= 2;         // left bitshift: num6 = num6 << 2;
                    // same thing as  num6 *= 4  …Why?
```

Notes:

(1) shortform operators have the same precedence as =, i.e. the shortform operation gets done last (except for postfix).

(2) <= is a comparison; <<= is an assignment.

(3) the modulus operator % is extremely useful for generating a series of repeating numbers that 'loop'—without the use of a loop structure.

2.24 Note: The `scanf()` 'bug' (it's a *feature*)

- Because of the way **`scanf()`** works, it contains a curious 'feature' which makes its use problematic for beginners (and frequently, for experts as well). The problem looks like this:

This works:

```
int myFirstInt, mySecondInt;
printf("Enter an integer\n");
scanf("%d", &myFirstInt);    // success
```

If we wish to read in additional integers using `scanf()`, we simply add:

```
printf("Enter another integer\n");
scanf("%d", &mySecondInt);   // no problem
```

2.24 Note: The `scanf()` 'bug' (it's a *feature*)

- However, if we repeat the same process using characters instead of integers, we run into problems:

The first call to `scanf()` works:

```
char myFirstChar, mySecondChar;
printf("Enter a character\n");
scanf("%c", &myFirstChar);   // success
```

But if we wish to read in additional characters using `scanf()`, the operation hangs:

```
printf("Enter another character\n");
scanf("%c", &mySecondChar);  // fails
```

The `scanf()` 'bug'

2.24 Note: The `scanf()` 'bug' (it's a *feature*)

- There are several solutions—to be discussed later on—but for now the best solution is to include the line

```
fflush(stdin);
```

after `scanf()` whenever it is used to read in multiple characters one at a time. For example:

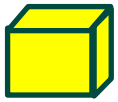
```
int myFirstChar = 0, mySecondChar = 0;
printf("Enter a character\n");
scanf("%c\n", &myFirstChar);    // success
fflush(stdin);

printf("Enter another character\n")
scanf("%c\n", &mySecondChar);   // more success
fflush(stdin);
```

Review - Example 04 : Volume of a Pyramid

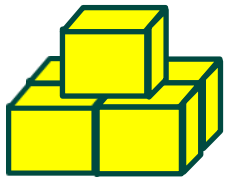
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main (void){

   int hPyr = 0, hCtr, totVol = 0;

   printf("Enter pyramid height in blocks: ");
   scanf("%d", &hPyr);
   assert(hPyr > 0);

   for (hCtr = 1; hCtr <= hPyr; hCtr++)
      totVol += hCtr * hCtr;

   printf("The total volume of a pyramid of height "
      "%d is %d blocks\n", hPyr, totVol);

   return EXIT_SUCCESS;
}
```

Programming Challenges

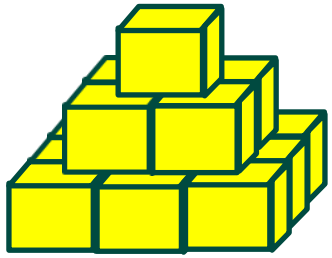
Following Example 04, write a program that calculates the exposed surface area of a pyramid of height N (in blocks), assuming each block is a cube with each face having an area of one unit, while the base of the pyramid is unexposed. For example, a pyramid of height 1 will have 5 exposed faces:



A pyramid of height 2 will have 17 exposed faces, minus the area of the top cube covering the second layer; hence it has a total surface area of 16:



A pyramid of height 3 will have an effective surface area equivalent to 33 exposed faces, and so on:



…write a program to perform the calculation for a pyramid N blocks high

Example 05 : The high/low number guessing game

Program Description:

The computer generates a random number between 1 and 100; the user guesses this number and is told his guess is 'high' or 'low', until the correct number is entered. The total number of attempts is reported back to the user prior to exit.

Example 05: The High/Low Guessing Game output

```
houtmad@ubuntu:~/examples$ Ex5
Enter a number between 1 and 100: 50
Your guess was low; try again: 75
Your guess was low; try again: 87
Your guess was high; try again: 81
Your guess was low; try again: 84
Your guess was correct! It took 5 attempts
houtmad@ubuntu:~/examples$
```

Example 05 : The High/Low Guessing Game

```
#include <stdio.h>
#include <stdlib.h>
#define MAXNUM 100
#define THEN_ITS(hilo) printf("Your guess was "#hilo" ");

int main (void){
   unsigned int target, guess, attempts = 0;
   target = ((double)rand()/RAND_MAX) * MAXNUM;

   printf("Enter a number between 1 and %d:", MAXNUM);

   do {
      scanf("%d", &guess);
      attempts++;
      if (guess > target) THEN_ITS(high; try again:)
      else if (guess < target) THEN_ITS(low; try again:)
      else THEN_ITS(correct!)
   } while (guess != target);

   printf("It took %d attempts\n", attempts);
   return EXIT_SUCCESS;
}
```

2.25 Macro creation using #define

```
#define THEN_ITS(hilo) printf("Your guess was "#hilo" ");
```

- As before, whenever the CPP finds a directive in the form:

`#define X Y`

then wherever it finds the identifier **`X`** in your code, it substitutes **`Y`**, where **`X`** and **`Y`** may each consist of any string of characters (however, X may not contain spaces, since the space indicates the end of X and the start of Y.) In the macro variation of **`#define`**, whenever **`X`** contains parenthesis, the contents of these are taken as one or more parameters to be inserted into the string **`Y`** prior to compilation. For example, given

```
#define SQUARE(x) x*x
```

K&R 75

then whenever `SQUARE(…)` appears in code, the square of the value in parentheses will be inserted.

2.25 Macro creation using #define

- Such definitions can be used in expressions as well. For example, using

```
#define SQUARE(x) x*x
```

again, if we then have, in our `main()` program, the comparison:

```
if (SQUARE(3) + SQUARE(4) == SQUARE(5)) {
    ...
}
```

Then the preprocessor expands this out to:

```
if (3*3 + 4*4 == 5*5) {
    ...
}
```

And, since the expression in parentheses is true, the code inside the braces will be executed.

2.25 Macro creation using #define

- Because `#define` uses direct substitution, we can use one `#define` inside another. For example, in the following:

```
#define SQUARE(a) a*a
#define SQR_HYPOT(x, y) SQUARE(x) + SQUARE(y)
square_of_hypotenuse = SQR_HYPOT(3, 4);
```

The second line is expanded out to:

```
#define SQR_HYPOT(x, y) x*x + y*y
```

based on the definition in the first line. The second line is then used in the third line, being expanded out, prior to compilation, to

```
square_of_hypotenuse = 3*3 + 4*4;
```

2.25 Macro creation using #define

- `#define` must be used with some caution. Consider the following definition. Can you spot the error? What does `SQR_HYPOT(3,4)` get expanded out to?

Incorrect use of #define

```
#define SQUARE(a) a*a
#define SQR_HYPOT (x, y) SQUARE(x) + SQUARE(y)
square_of_hypotenuse = SQR_HYPOT(3, 4);
```

2.25 Macro creation using #define

- Similarly,

```
#define PRINTOUT(str) printf("Printout is: <<%s>>", str)
```

says: "wherever **`PRINTOUT(…)`** occurs in code, insert the character(s) in the parentheses and then insert the rest of the definition

```
printf("Printout is: <<%s>>", …)
```

prior to actual compilation."

This form of **`#define`** allows you to create short, inline functions—***macros***—for inclusion in your programs prior to compilation.

2.25 Macro creation using #define

macro *n*. …**2.** In programming languages, such as C or assembly language, a name that defines a set of instructions that are substituted for the macro name wherever the name appears in a program (a process called *macro expansion*) when the program is compiled or assembled. Macros are similar to functions in that they can take arguments and in that they are calls to lengthier sets of instructions. Unlike functions, macros are replaced by the actual instructions they represent when the program is prepared for execution; function instructions are copied into a program only once.

—Microsoft Computer Dictionary 4th ed. (1999)

NOTE: "*Macro expansion is a tricky operation, fraught with nasty corner cases and situations that render what you thought was a nifty way to optimize the preprocessor's expansion algorithm wrong in quite subtle ways.*"

— https://gcc.gnu.org/onlinedocs/cppinternals/Macro-Expansion.html

2.25 Macro creation using #define

- So for example, if we then call, in our `main()` program,

```
PRINTOUT("Apple");
```

then the string **`"Apple"`** takes the place of **`str`** in the latter portion of the **`#define`** statement. So the preprocessor expands our macro out to:

```
printf("Printout is: <<%s>>", "Apple");
```

And the output, following compilation and execution, will be:

```
Printout is: <<Apple>>
```

2.25 Macro creation using #define

- When the **#** symbol appears *before* the macro variable name (i.e. **`str`** in the above example), the inserted word or value is treated as a ***string literal*** by wrapping double quotes around the word in the final output. For example, if the previous macro was written

```
#define PRINTOUT(str) printf("Printout is: <<"#str">>")
```

(Note groupings of parenthesis)

then `PRINTOUT(Apple);`

(without the quotes) causes the string "**`Apple`**" to be inserted (with quotes around it) wherever **`#str`** appears in the final part of the macro. So the preprocessor expands our macro out to:

```
printf("Printout is: <<""Apple"">>");
```

and because `printf()` ignores multiple string parameters and concatenates them into one continuous string, the result is the same as before:

```
Printout is: <<Apple>>
```

Example 02 Revisited

- When code is formatted almost identically several times in a program (with slight changes), `#define` can be used to shorten up the code considerably and simplify the program.
- Consider, as an example, the code from Example 02. The repetitive parts are highlighted in bold:

```
/* … includes go here   */

int main(void){
    printf("The size of the data types "
           "for this compiler are:\n\n");
    printf("\tchar:    %u byte(s)\n", sizeof(char));
    printf("\tshort:   %u byte(s)\n", sizeof(short));
    printf("\tint:     %u byte(s)\n", sizeof(int));
    printf("\tlong:    %u byte(s)\n", sizeof(long));
    printf("\tfloat:   %u byte(s)\n", sizeof(float));
    printf("\tdouble:  %u byte(s)\n", sizeof(double));
    printf("\n");
    return EXIT_SUCCESS;
}
```

Example 02 Revisited

- We case use `#define` to create a macro to handle each line of the form:

```
    printf("\tsomedatatype: \t%u byte(s)\n", sizeof(somedatatype));
```

```
/*  Declare include files, etc.  */

#define TYPEINFO(type) printf("\t"#type":\t%u \
                                    bytes(s)\n", sizeof(type))

int main(void){
    printf("The size of the data types "
           "for this compiler are:\n\n");
    TYPEINFO(char);
    TYPEINFO(short);
    TYPEINFO(int);
    TYPEINFO(long);
    TYPEINFO(float);
    TYPEINFO(double);
    printf("\n");
}
```

2.25 Macro creation using #define

- So when you wish to have a value (either a word or number) printed out in a string, put the # symbol in front of the variable name; this says: "this is to be treated as a string literal". When you wish to insert a value into a function and have it treated as a true variable, drop the # symbol and just use the variable name by itself.

Finally, in:

```
#define TYPEINFO(type) printf("\t"#type":\t%u \
                                    bytes(s)\n", sizeof(type))
```

what does the '\' at the end of the first line do? It's the continuation character; it says: "to be continued on the next line".

We could write this on a single line without the '\' as

```
#define TYPEINFO(type) printf("\t"#type":\t%u bytes(s)\n", sizeof(type))
```

...but this becomes a bit unwieldy.

2.25 Macro creation using #define

- So when we expand the last line, it says that

```
TYPEINFO(type)
```

stands for

```
printf("\t"#type":\t%u bytes(s)\n", sizeof(type))
```

When we use **`TYPEINFO()`** in our program, for example as in:

```
TYPEINFO(char);
```

then when **`char`** is substituted, this gets expanded to

```
printf("\t"#char":\t%u bytes(s)\n", sizeof(char))
```

which then becomes, effectively

```
printf("\tchar:\t%u bytes(s)\n", sizeof(char))
```

—our original statement.

2.26 Using `rand()`

```
((double)rand()/RAND_MAX) * MAXNUM;
```

- The `rand()` function is found in `stdlib.h`. This function returns an integral value between 0 and `RAND_MAX`, which is guaranteed to be a number not less than 32767 (but which varies depending on the compiler used; in gcc `RAND_MAX` is 2147483647 on a 32-bit platform. Therefore, you should always use the value determined by the compiler, stored in `RAND_MAX`).

K&R 136

- The result of the calculation in parentheses generates a number between 0 and 1. Therefore, multiplying this value by `MAXNUM` generates a number between 0 and `MAXNUM`.

2.26 Using `rand()`

- Computers are designed to avoid randomness at all costs; therefore, designing an algorithm that generates true randomness is virtually impossible. Pseudo-random number generating algorithms must invariably be seeded each time they are called with a truly random value, otherwise they will always produce the same set of values in the same sequence. Time is most often used for this purpose. Thus the above code should be modified to:

```
srand(time(NULL));
target = ((double)rand()/RAND_MAX) * MAXNUM;
```

where **`srand()`** is a function that seeds the **`rand()`** with the current time from the **`time()`** function, which is found in the **`<time.h>`** library.

K&R 211

2.27 Type conversion

```
((double)rand()/RAND_MAX)  * MAXNUM;
```

- Note the effect of the `(double)` cast on this statement. If we remove the cast, the expression in parentheses becomes

```
rand()/RAND_MAX
```

Probable unintended rounding error

If you divide an `int` by a larger `int`, the result is also an `int`. Since `rand()` *almost* always generates a number less than `RAND_MAX`, the result is 0: fractional numbers are rounded down to the next integer.

- Variables have both a type and a value; whenever an operation is done on two variables, ***type conversion*** rules apply, particularly when two different data types are used in the same operation.

K&R 159

2.27 Type conversion

- In general, type conversions may be implicit—they occur automatically—or explicit. For example, in Example 03, when we entered the diameter of the pizza

```
float diameter…
scanf("%f", &diameter);
```

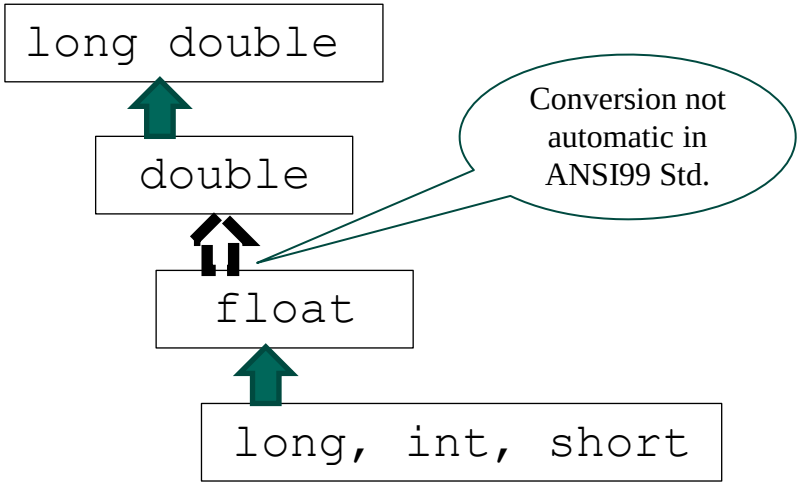
we input an integer value—say 14. The compiler recognizes this and converts the value to 14.0, a floating point value, as specified by the type.

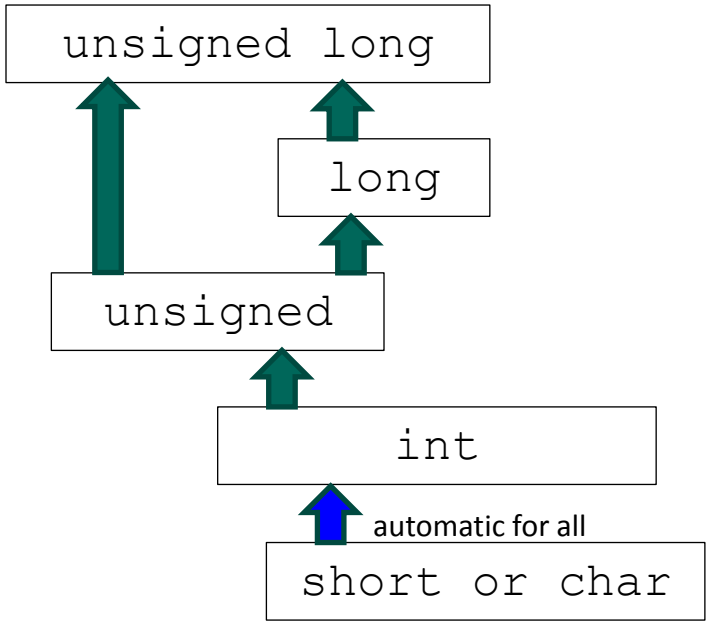
- Type conversions may be performed programmatically, at the request of the programmer, for example when a cast is deliberately applied, as in:

```
printf("The number of calories in a slice of %d inch pizza"
       " is %-5.1f\n", (int)diameter, calories_per_slice);
```

2.27 Type conversion

- A set of common rules, known as 'Usual Arithmetic Conversions', automatically convert data types whenever common math operations (i.e. +, -, *, /, %, =) are employed. For any such expression, smaller-sized data types are promoted 'up' to the next largest data type in the expression according to the following rules:





K&R 40, 42

2.27 Type conversion

- Notice how this chart works in the following examples

  For the examples shown in the table, assume the following declarations

```
char c;   short s;   int i;     long l;     unsigned int u;
unsigned long ul;   float f;   double d;    long double = ld;
```

Expression	Result Type	Expression	Result Type
`c – s/i`	int	`u * 7 - i`	unsigned int
`u * 2.0 – i`	double	`f * 7 - i`	float
`c + 3`	int	`7 * s * ul`	unsigned long
`c + 5.0`	double	`ld + c`	long double
`d + s`	double	`u - ul`	unsigned long
`2 * i/l`	long	`u - l`	*system-dependent*

From: *A Book on C*, A. Kelley and I. Pohl, Addison Wesley, 1998, pg. 133

2.27 Type conversion

- Note that `shorts` and `chars` are automatically converted in any expression to `ints` (i.e. even if you're adding two `shorts`, you end up with an `int`, unless you explicitly cast to a `short`) This is called ***integer promotion***. For example:

K&R 157

```
#include <stdio.h>

int main(void){

    char c = 'a';    // ascii value of 'a' is 97
    short s = 3;

    printf("Size of s = %u\n", sizeof(s));
    printf("Size of c = %u\n", sizeof(c));
    printf("Size of s + c = %u\n", sizeof(s + c));
    printf("Value of s + c = %d\n", s + c);
    printf("Character equivalent of s + c = %c\n", s + c);
}
```

```
Size of s = 2
Size of c = 1
Size of s + c = 4
Value of s + c = 100
Character eqivalent of s + c = d
```

2.27 Type conversion

- As seen above, a character is internally promoted to its integer value whenever an operation is performed on it—even though its original data type is still a character. This can lead to some subtle and confusing errors. For example, in the following example,

```
char c = 'A';
printf("Size of c is %u\n", sizeof(c));
printf("Size of \'A\' is %u\n", sizeof('A'));
```

the type of the character 'A' is in fact an `int`, not a `char`, as you'd expect.

```
Size of c is 1
Size of 'A' is 4
```

2.27 Type conversion

- If one operand is of type `long` and the other is `unsigned int`, then if a `long` type can hold the `unsigned` value, the conversion is to `long`; otherwise both operands may be converted to `unsigned long`. Hence, there are *two* possibilities for the type conversion depending on the possible outcomes.

2.27 Type conversion

- Of course, the safest way to make a conversion is to cast it directly, using an explicit cast:

```
int myInt= 3;
float myFloat;
myFloat = (float) myInt;     // myFloat = 3.0
```

Note that:

1. Cast operators have the same precedence as any other unary operator (i.e. they act on one item only). Thus in

```
(float) myInt + 7;
```

the cast gets done first. So this is the equivalent of:

```
((float) myInt) + 7;
```

and *after* that the rules of automatic conversion apply.

K&R 42

K&R 165

2.27 Type conversion

2. Casting from a smaller data type to a larger data type is rarely a problem. ***Narrowing*** a cast from, say an `long` to an `int` is problematic, and will often generate at least a warning from the compiler:

```
int myInt = 40000;  // Note: assumed unsigned
signed short   mySShortA = (short) myInt;  //won't fit
unsigned short myUShortB = (unsigned short) myInt; //fits
unsigned short myUShortC = (unsigned short) (2 * myInt);
                                            //won't fit
```

3. When a decimal type value is converted to an integer type, everything after the decimal is dropped, i.e.

```
int main(){
    float f1 = 1.99999, f2 = 2.444444, f3 = 3.14;
    int tot = f1 + f2 + f3;     // = 7.584443
    printf("%i\n", tot);        // Output is 7
}
```

2.27 Type conversion

- Returning to the original statement in Example 05:

```
target = ((double)rand()/RAND_MAX) * MAXNUM;
```

`rand()` returns a `long` (in gcc), which gets cast to a `double`. At this point, automatic type conversion ensures that the entire calculation will be 'cast up' to the `double` data type—allowing for decimal values. In the final stage, the value on the right hand side of the equation—still a `double`—must be implicitly converted to an integer to fit into the integer `target` on the left side of the equation.

Note that casts must be used with some caution: you need to understand the range and type of data values (signed or unsigned) that will be assigned to your variables in order for the cast to work reliably. As one popular programmer's joke has it:

"A cast is so-named because
it allows something broken to limp along."

Review - Example 05 : The High/Low Guessing Game

```
#include <stdio.h>
#include <stdlib.h>
#define MAXNUM 100
#define THEN_ITS(hilo) printf("Your guess was "#hilo" ");

int main (void){
   unsigned int target, guess, attempts = 0;
   target = ((double)rand()/RAND_MAX) * MAXNUM;

   printf("Enter a number between 1 and %d:", MAXNUM);

   do {
      scanf("%d", &guess);
      attempts++;
      if (guess > target) THEN_ITS(high; try again:)
      else if (guess < target) THEN_ITS(low; try again:)
      else THEN_ITS(correct!)
   } while (guess != target);

   printf("It took %d attempts\n", attempts);
   return EXIT_SUCCESS;
}
```

Questions

1. A compiler error results at the second `#define` statement below. Why?

```
#define SQUARE(x) x*x                    // Find x^2
#define FOURTH(y) SQUARE(SQUARE(y)) // Find y^4
```

2. A compiler error results at `PRINTOUT`. Why?

```
#define PROUT(pr) printf("Printout is: <%s>", pr)
PROUT("Print something out");
```

3. This one compiles okay, but won't give the output you expect. Why?

```
#define SQUARE(a) a*a
int x = 3, y = 2, z;
Z = SQUARE(x + y);
```

NOTE: Results are very much compiler dependent. `gcc`, in fact ignores comments added to `#defines`, where other compilers would flag an error or abort compilation.

Questions

4. Assume MAXNUM is defined as 100 in the following example, and assume, for simplicity, that RAND_MAX equals 10,000. For the loop:

```
int found=0;
for (int numCtr = 0; numCtr < RAND_MAX; numCtr++){
   target = (rand()/RAND_MAX) * MAXNUM;
   if (target==GUESS) found++;
}
```

a. Assume GUESS equals 50; what is the value of found when the loop exits?
b. Assume GUESS equals 0; what is the value of found when the loop exits?
c. Assume GUESS equals MAXNUM (100); what is the value of found when the loop exits?
d. Assume the first line inside the for loop is changed to:

```
target = ((double)rand()/RAND_MAX) * MAXNUM;
```

and assume GUESS equals MAXNUM. What is the value of found when the loop exits? For all values other than MAXNUM, what does found equal (on average)?

Questions

5. For any value of `MAXNUM` in the code

```
target = ((double)rand()/RAND_MAX) * MAXNUM;
```

how many distinct numbers will be generated for the value of `target`?

6. What happens if, in calculating the value of `target` in question 5 above, you use instead

```
target = ((MAXNUM * rand())/RAND_MAX);
```

What, if anything happens to:

a. the number of distinct values output
b. the distribution (or frequency) of those numbers
c. potential compile- or run-time problems with the code

Write a short program with this code in it, and run tests to check your assumptions.

Questions

7. Consider the following program, which is designed to test for random numbers by generating fifty separate trials of random numbers, with each trial generating ten numbers:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXNUM  100

int main(void){
   int result;

   for (int i = 0; i < 50 ; i++){
      srand(time(NULL));
      printf("Output for trial %d is:  ", i);

      for (int j = 0; j < 10; j++){
         result = ((double)rand()/RAND_MAX) * MAXNUM;
         printf("\t%d", result);
      }
   printf("\n");
   }
}
```

Questions

Here's the output from this program. Notice that each outer loop generates the same set of 10 digits each time, even though `srand()` is being used to seed the random number generator at the start of each trial. How do you account for these results?

```
Output for trial 0 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 1 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 2 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 3 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 4 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 5 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 6 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 7 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 8 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 9 is:      99    57    34    52    78     6    19    46    35    53
Output for trial 10 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 11 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 12 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 13 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 14 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 15 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 16 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 17 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 18 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 19 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 20 is:     99    57    34    52    78     6    19    46    35    53
Output for trial 21 is:     99    57    34    52    78     6    19    46    35    53
```

Example 06 : The number of days to date

Program Description:

Electronics devices (and other products) are frequently stamped with the day of the year on which they were manufactured, to allow bad batches to be easily identified. The number on the chip above, 04404, indicates it was manufactured on the 44th day of 2004, i.e. Feb 13, 2004.

The current program prompts the user to enter the current day, month, and year, and returns the number of days in the year up to that date.

Example 06 : Number of days to date

```
houtmad@ubuntu:~/examples$ Ex6
Enter the day (1, 2, ...30, 31): 31
Enter the month (Jan=1, Feb=2, etc.): 12
Enter the year (2012, 2013, 2014...): 2014
31/12/2014/ is day #365

houtmad@ubuntu:~/examples$ Ex6
Enter the day (1, 2, ...30, 31): 1
Enter the month (Jan=1, Feb=2, etc.): 3
Enter the year (2012, 2013, 2014...): 2014
1/3/2014/ is day #60

houtmad@ubuntu:~/examples$ Ex6
Enter the day (1, 2, ...30, 31): 1
Enter the month (Jan=1, Feb=2, etc.): 3
Enter the year (2012, 2013, 2014...): 2012
1/3/2012/ is day #61

houtmad@ubuntu:~/examples$
```

Example 06 : Number of days to date

```
#include <stdio.h>
#include <stdlib.h>
#define FEB 2

int main (void){
   unsigned int dy, mth, yr, isLeapYr, mthCtr, totDays = 0;
   unsigned int monthSz[] = {31,28,31,30,31,30,31,31,30,31,30,31};

   printf("Enter the day (1, 2, ...30, 31): ");
   scanf("%d", &dy);
   printf("Enter the month (Jan=1, Feb=2, etc.): ");
   scanf("%d", &mth);
   printf("Enter the year (2012, 2013, 2014...): ");
   scanf("%d", &yr);
   isLeapYr = ((yr%4==0) && !((yr%100==0) && !(yr%400==0)));
   for (mthCtr = 0; mthCtr < mth-1; mthCtr++)
      totDays += monthSz[mthCtr];
   totDays += dy;
   if (isLeapYr && (mth > FEB)) totDays++;

   printf("%d/%d/%d is day #%d\n\n", dy, mth, yr, totDays);
}
```

2.28 One-dimesional arrays

```
unsigned int monthSz[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

An array is a collection of data that can be accessed by an integer ***index***. As in Java, to declare an array declaration of undetermined size you'd use:

data_type array_name[size];

For example:

```
int mthSz[12];
```

K&R 23

As in Java, there is a shortcut method for setting the size of the array, shown in the code at top. In the above example, there are twelve integers (called ***initializers***) listed inside the braces. Hence the default size of the array is set to 12, even though it isn't specified specifically.

2.28 One-dimesional arrays

You can use both methods simultaneously:

```
int monthLength[12] = {31,28,31,30,31,30,31,31,31,30,31};
```

but note that if the number of initializers on the RHS is less than the number indicated in square brackets on the LHS, C will fill in the remaining elements with 0's. This can be dangerous, since responsibility rests with the user to ensure that the number of elements in braces matches the number of initializers.

For example, the above declaration/assignment actually has fewer than 12 initializers on the RHS. So the contents of the array are actually:

```
{31,28,31,30,31,30,31,31,31,30,31,0};
```

The same potential problem exists in Java.

2.28 One-dimesional arrays

Character arrays may be initialized in two ways:

```
char ch[] = "xyz";
```

automatically initializes the array `ch[]` to the string `"xyz"`, *plus* the `'\0'` character appended on to the end. Thus the above statement is identical to:

```
char ch[] = {'x', 'y', 'z', '\0'};
```

Despite appearances, it's worth noting that the C compiler treats these two statements somewhat differently—but with identical results. The double quotes "..." indicate that the starting address of the string is returned to the variable `ch`.

```
char ch[] = "xyz";
```

So `ch` *points to* the memory location containing the `"x"` in `"xyz"`.

2.28 One-dimesional arrays

In the second version, the single quotes '...' around each letter indicate that the numerical (i.e. ASCII) value is to be stored in memory. Thus

```
char ch[] = {'x', 'y', 'z', '\0'};
```

is identical to

```
char ch[] = {122, 123, 122, 0};
```

where the `char` data type says: these numbers are to be used as characters. The braces `{}` indicate this is an array, and returns the address of its first value, so `ch` will store an address, just as it did before.

Alternately, this could be written as

```
char ch[] = {0x78, 0x79, 0x7A, 0x0};
```

where the `0x` notation indicates that the values are represented in hexadecimal.

2.28 One-dimesional arrays

- A common way to initialize an array with the same value is to use the following format:

```
char firstname[14] = {'\0'};
```

This populates the array `firstname` with fourteen `'\0'` character constants.

2.28 One-dimesional arrays

- As with non-array declarations, like `unsigned int dy`,
  - the data type reflects the data you intend to store in the array, along with the total size occupied in memory
  - the name of the array follows the same rules as for variable declarations
- Array declarations are reference data types, and hence can alternately be referenced by pointers in C—a subject to be covered in Module 5
- As in Java, arrays in C are zero-based. This causes occasional problems, and adjustments must be made to account for the expected offset. For example, in the code for Example 06 the user naturally enters '1' for Jan, '2' for Feb, etc., even though January corresponds to the 0^{th} element of the array.

2.28 One-dimesional arrays

- As with Java, C uses brackets [ ] for the array declaration, but braces { } for assigning specific values to the array, separated by commas, hence

```
int monthSz[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

- Uninitialized arrays must be given a size: there's no such thing as

```
int Ar[];
```

Array is unsized

- The value returned in the element of an array is specified using its index, so

```
monthSz[0]
```

returns 31.

2.29 Boolean Logic

```
isLeapYr = (yr%4==0) && !((yr%100==0) && !(yr%400==0));
```

Logical operators in C operate on true and false values according to the same rules of ***Boolean logic*** found in Java. There are four standard operators

Symbol	Operation
&&	Boolean AND – if the two operands are both non-zero, the result is 1
||	Boolean OR (or inclusive OR) – if either of the two operands are non-zero, the result is 1
^	Boolean XOR – if the two operands are the same, the result is 0, else 1
!	Boolean NOT – the value becomes its opposite, a true value becomes false, and a false value becomes true

K&R 168

2.29 Boolean Logic: Review

Logical operators in detail

AND

&&	true	false
true	true	false
false	false	false

OR

||	true	false
true	true	true
false	true	false

XOR

^	true	false
true	false	true
false	true	false

NOT

	!
true	false
false	true

2.29 Boolean Logic

```
isLeapYr = (yr%4==0) && !((yr%100==0) && !(yr%400==0));
```

Leap years occur whenever:

- The year is divisible by 4, unless
- The year is divisible by 100, unless
- The year is divisible by 400

Here's another way to put it (courtesy of Wikipedia):

```
if year is divisible by 400 then
   is_leap_year
else if year is divisible by 100 then
   not_leap_year
else if year is divisible by 4 then
   is_leap_year
else
   not_leap_year
```

2.29 Boolean Logic

Unfortunately, neither definition helps simplify the translation from a written algorithm into actual C code. Oftentimes its useful to run a few 'virtual' test cases to make sure an algorithm fully makes sense prior to coding it:

Year	isLeapYear?	Year	isLeapYear?
2014	No. Not divisible by 4	**2100**	No. Divisible by 100 but not by 400
2012	Yes. Divisible by 4	**2000**	Yes. Divisible by 100 but also by 400

As a first approximation, assume *every* year evenly divisible by 4 is a leap year. We can use the modulus operator to determine if a number is evenly divisible by 4 by writing

```
isLeapYr = (yr%4==0)
```

So if the value of yr is 2000, 2004, 2008, 2012, etc. its value mod 4 is equal to zero—it's a leap year.

2.29 Boolean Logic

However, years divisible by 100 are excluded. Hence our code becomes:

```
isLeapYr = (yr%4==0) && !(yr%100==0)
```

K&R 39

The final part of the algorithm says that the '100-year-exception' is overridden every 400 years. Hence the *exception* to the 4-year-rule occurs not quite *every* hundred years, but only when the centenary is *not* evenly divisible by 400. Hence we can code the *exception* as:

```
(yr%100==0) && !(yr%400==0)
```

Inserting this into the expression above gives

```
isLeapYr = (yr%4==0) && !((yr%100==0) && !(yr%400==0))
```

This reads, from left to right: "a leap year occurs whenever the year is evenly divisible by 4 and *not* divisible by 100, unless it is divisible by 400."

2.29 Boolean Logic

- In C, `&&` corresponds to the AND operation on *Boolean values*, whereas `&` signals a *bitwise* AND. The same formality occurs with || and |, which correspond to Boolean OR and bitwise OR.

Note: In Java, `&&` and `&` act on Boolean operands in almost exactly the same way, except that `&&` short-circuits, while `&` does not. In C, you don't have a choice: Boolean operators in C, like `&&` and `||`, *always* short-circuit the evaluation of their operands.

2.29 Short-circuiting expressions

- Short-circuiting* uses the fact that logical expressions need not be carried to completion if one of the input values of an AND or OR expression is known to be true or false. Recall that:

AND

&&	true	false
true	true	false
false	false	false

OR

||	true	false
true	true	true
false	true	false

*short-circuit evaluation is also sometimes referred to as 'lazy evaluation'.

2.29 Short-circuiting expressions

- For an AND operation, if either of the inputs is false then the result must be false. Therefore, if we get a `false` in the first expression, we don't need to evaluate the second expression. So

(`false` **AND** …doesn't matter…) → `false`

&&	true	false
true	true	false
false	false	false

- For an OR operation, if one of the inputs is `true` then the result must be true. Again, we don't need to go any farther to know that the overall result will be `true`. So

(`true` **OR** …doesn't matter…) → `true`

||	true	false
true	true	true
false	true	false

2.29 Short-circuiting expressions

- C does *not* contain the literal values `true` and `false`, which are found in Java. Instead, whenever the result of a conditional expression is true, it returns a 1; when false, a 0. Thus, is Example 04, in:

```
assert(hPyr > 0);
```

`assert()` is a function that takes as a parameter, not a Boolean `true` or `false`, but an integer. This integer results from the expression:

```
hPyr > 0
```

which returns the value 1 if the expression is true, 0 otherwise.

2.29 Short-circuiting expressions

- Similarly, in Example 05, in the code

```
do {
    ...
    if (guess > target) THEN_ITS(high; try again:)
    else if (guess < target) THEN_ITS(low; try again:)
    else THEN_ITS(correct!)
} while (guess != target);
```

the code in bold face resolves to a 1 or 0 depending on whether the expression is true or false.

2.29 Short-circuiting expressions

- Similarly, any expression that results in a nonzero value can be treated as true in a conditional expression, while 0 is treated as false. Thus the code:

```
while(1){     // is always 'true'; loop goes forever
   …
}
```

repeats forever, unless the loop contains a `break` or `return` statement or an `exit()` command.

- This explains why, when you use '=' instead of '==' in an expression like

```
for (int ctr = 0; ctr = 10; ctr++){
  ...
}
```

you only loop once. The loop control variable **`ctr`** gets set equal to 10 after the first loop. Since **`ctr`** is now 'true' (i.e. non-zero), the conditional part of the loop is satisfied, and the loop terminates.

2.30 Declaring constant arrays

- Since our array remains constant throughout program execution—the number of days in the month never change– we can make our code tighter by making the array constant:

```
const unsigned int monthSz[] =
        {31,28,31,30,31,30,31,31,30,31,30,31};
```

This tells the compiler that the contents of the array will not change during executing, thus potentially speeding execution; it does not, however, guarantee that the contents of the array *cannot* be changed during execution.

Review - Example 06 : Number of days to date

```
#include <stdio.h>
#include <stdlib.h>
#define FEB 2

int main (void){
   unsigned int dy, mth, yr, isLeapYr, mthCtr, totDays = 0;
   const unsigned int monthSz[] =
                     {31,28,31,30,31,30,31,31,30,31,30,31};

   printf("Enter the day (1, 2, ...30, 31): ");
   scanf("%d", &dy);
   printf("Enter the month (Jan=1, Feb=2, etc.): ");
   scanf("%d", &mth);
   printf("Enter the year (2012, 2013, 2014...): ");
   scanf("%d", &yr);
   isLeapYr = ((yr%4==0) && !((yr%100==0) && !(yr%400==0)));
   for (mthCtr = 0; mthCtr < mth-1; mthCtr++)
      totDays += monthSz[mthCtr];
   totDays += dy;
   if (isLeapYr && (mth > FEB)) totDays++;

   printf("%d/%d/%d is day #%d\n\n", dy, mth, yr, totDays);
}
```

Questions

1. Given:

```
int a = 0, b = 0, c = 1, d = -2, e = 3, z;
```

what is the value of `z` following the execution of each line of code?:

```
z = (a==b);
z = (a=b);
z = (a==--c);
z = ((a=d)==(a==d));
z = ((c==d)==(c=d));
z = ((d/e)<0);
z = (c==e*(c/e));
```

Questions

2. Use short-circuiting to explain why the expression

```
isLeapYr = (yr%4==0) && !((yr%100==0) && !(yr%400==0));
```

will *generally* execute faster than

```
isLeapYr = !((yr%100==0) && !(yr%400==0)) && (yr%4==0);
```

even though both execute exactly the same operation and give the same results

Questions

3. Consider the following code fragment:

```
char response;
printf("Do you want to exit? Enter Y or y.  "
    "To continue, press N or n\n");
scanf("%c", &response);
if (response=='Y' || 'y'){
   printf("I'm outta here\n");
   exit();
}
else
   printf("Okay, let's continue\n");
```

When this code is run, the program *always* prints "`I'm outta here`" and exits. The line "`Okay, let's continue`" is never displayed. Explain why.

Questions

4. *DeMorgan's law* can be used to simplify complex logical statements. It says that, for any two Boolean values, `A` and `B`

```
NOT(A) AND NOT(B) = NOT(A OR B)
NOT(A) OR NOT(B) = NOT(A AND B)
```

Put in C code, this becomes

```
!A && !B = !(A || B)
!A || !B = !(A && B)
```

Use DeMorgan's law to show that the expression

```
isLeapYr = (yr%4==0) && !((yr%100==0) && !(yr%400==0));
```

is equivalent to:

```
isLeapYr = !((yr%4) || ((yr%400) && !(yr%100)));
```

(cont'd)

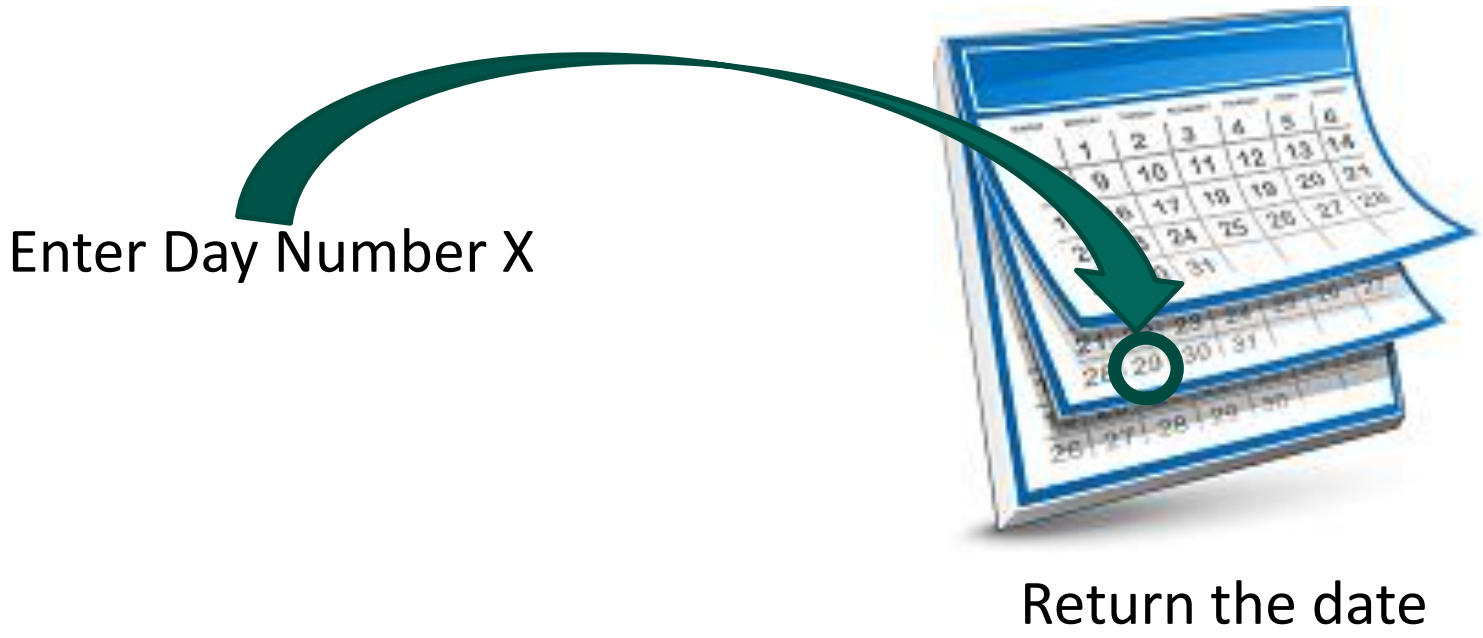
Questions

Note that C code can often be reduced to the point where it becomes less comprehensible, and even incomprehensible; the last exercise is one such case. You *might* use this code to squeeze out a few extra bytes of memory in an embedded system with little RAM, or to execute the program slightly faster on a slow processor. But you probably wouldn't use this 'improved, compacted' code otherwise, since it is far less understandable than the code it replaced in Example 06. Remember:

"Everything should be made as simple as possible,
but not simpler"

Programming Challenges

Write a program that performs the opposite operation as the one given in Example 06, i.e. given the day of the year and the year itself, return the date.



Example 07 : Telephone character to number converter

Program Description:

The modern telephone keypad allows companies to effectively replace phone numbers with characters, allowing users to dial easily-memorized words rather than phone numbers. For example, Apple uses 1-800-my-apple to help you purchase a new computer or component; when its time to discard it, you can call 1-800-got-junk.

This program allows users to enter a phone number containing non-digit characters and output these as phone numbers using the standard convention shown on the keypad above.

Example 07 : Telephone number converter

```
houtmad@ubuntu:~/examples$ Ex7
Enter telephone number: 1-800-my-apple
The number you entered is: 1-800-69-27753

houtmad@ubuntu:~/examples$ Ex7
Enter telephone number: 1-800-got-junk
The number you entered is: 1-800-468-5865

houtmad@ubuntu:~/examples$ Ex7
Enter telephone number: 1-800-OCanada
The number you entered is: 1-800-6226232

houtmad@ubuntu:~/examples$ Ex7
Enter telephone number: 1-800-+++-0000
The number you entered is: 1-800-Bad value entered
houtmad@ubuntu:~/examples$
```

Example 07 : Telephone number converter

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define PHNUMARRAYSZ 14

int main (void){
   char phoneNum[PHNUMARRAYSZ] = {'\0'}, cPhNum;
   int numCtr = 0;

   printf("Enter telephone number: ");
   scanf("%s", phoneNum);
   printf("The number you entered is: ");
   while ((numCtr < PHNUMARRAYSZ) && phoneNum[numCtr]) {

      cPhNum = phoneNum[numCtr];
      if ((isdigit(cPhNum)) || (cPhNum == '-'))
         printf("%c",cPhNum);
      else if (isalpha(cPhNum)){
         cPhNum = toupper(cPhNum);
         switch (cPhNum) {
            case 'A': case 'B': case 'C':
               printf("2"); break;
            ...
```

Example 07 : Telephone number converter

```
            case 'D': case 'E': case 'F':
               printf("3"); break;
            case 'G': case 'H': case 'I':
               printf("4"); break;
            case 'J': case 'K': case 'L':
               printf("5"); break;
            case 'M': case 'N': case 'O':
               printf("6"); break;
            case 'P': case 'Q': case 'R': case 'S':
               printf("7");break;
            case 'T': case 'U': case 'V':
               printf("8"); break;
            case 'W': case 'X': case 'Y': case 'Z':
               printf("9");break;
            }
         }
      else {
         printf("Bad value entered\n"); exit(EXIT_SUCCESS);
      } numCtr++;
   } printf("\n\n");
   return(EXIT_SUCCESS);
}
```

2.31 The ctype.h library

```
if ((isdigit(cPhNum)) || (cPhNum == '-'))
   printf("%c",cPhNum);
else if (isalpha(cPhNum)){
   cPhNum = toupper(cPhNum);
```

The library **ctype.h** contains a number of functions designed to test and convert char data types. Shown below are most of **ctype**'s functions.

K&R 206

function	operation	function	operation
isalnum	Tests for alphanumeric char	**isspace**	Tests for ' '
isalpha	Tests for alphabetic char	**isupper**	Tests for upper case char
iscntrl	Tests for control char	**isxdigit**	Tests for hexadecimal char
isdigit	Tests for a digit char (0-9)	**tolower**	Converts char to lower case
islower	Tests for lower case char	**toupper**	Converts char to upper case
ispunct	Tests for punctuation		

2.32 switch()

K&R 52

The `switch()` statement is the equivalent to a series of `if…else if` statements—but somewhat more elegant. It has the structure:

```
switch(somevalue){
   case num1:    //execute this if somevalue==num1
      // do something here
      break;
   case num2:    //execute this if somevalue==num2
      // do something else here
      break;
   //etc.
   default://do this if none of the other cases apply
      // catch anything not already covered
      break;  // not really necessary
}
```

2.32 switch()

`switch()` has some limitations:

- The data type that you 'switch' on (a `char` in this example), must resolve to an integral data type i.e. `char`, `short`, `int` or `long` (Java is more restricted: you can't switch on a `long` data type.) You cannot switch on a string in either language (although `switch` statements in other languages sometimes allow more flexibility.)

- You can't use a logic expression inside `switch`; you can't write:

```
switch(grade){
   case (grade > 90):
      printf("Bravo!");
      break;
   ...
```

Can't select case based on a Boolean result

- For many simple situations, it is faster and more convenient to use an array rather than a `switch` statement to do the same job.

2.32 switch()

- If you don't use `break` inside each `case`, you 'fall through' to the next `case` statement (which is sometimes desirable, as seen in Example 07.) And yes, this is technically known as "fallthrough."

- while not strictly required, it is considered good practice to include a `default` statement

K&R 53

2.33 Using `break` & `continue`

Both loop structures (like `while`, `do` and `for`) and decision structures (like `if`, `if…else`, and `switch`) may be interrupted before their normal termination condition is reached by using either `break` or `continue`. The former allows execution to jump *out* of the current code block and resume execution after the closing brace. For example, say we wish to read in characters until the a space is reached—in other words, to the end of a word. We could write:

K&R 57

```
// Assume string has been entered into array paragraph[]
for (charCtr = 0; charCtr < AR_MAX; charCtr++){
        ch = paragraph[charCtr];  //enter chars until space
        if ch == ' '
            break;      // end of word; exit loop and resume
                        // execution after the end of the
                        // current code block
        // otherwise do something else here
}           // …and exit the block
```

2.33 Using `break` & `continue`

The `continue` statement ceases execution of the current iteration and jumps *back* to the start of the block being executed, continuing on as if the latest iteration through the block had already been completed. For example:

```
// Assume strings have been entered into array paragraph[]
for (charCtr = 0; charCtr < AR_MAX; charCtr++){
    char ch = paragraph[charCtr];  //go until CR
    if ((ch == '\n') || (ch == '\r'))
       continue;  // skip the current iteration
                  // and go to the next
                  // iteration of the loop
                  // at the start of the block
    else
       // do other things here
}
```

Review - Example 07 : Telephone number converter

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define PHNUMARRAYSZ 14

int main (void){
   char phoneNum[PHNUMARRAYSZ] = {'\0'}, cPhNum;
   int numCtr = 0;

   printf("Enter telephone number: ");
   scanf("%s", phoneNum);
   printf("The number you entered is: ");
   while ((numCtr < PHNUMARRAYSZ) && phoneNum[numCtr]) {

      cPhNum = phoneNum[numCtr];
      if ((isdigit(cPhNum)) || (cPhNum == '-'))
         printf("%c",cPhNum);
      else if (isalpha(cPhNum)){
         cPhNum = toupper(cPhNum);
         switch (cPhNum) {
            case 'A': case 'B': case 'C':
               printf("2"); break;
            ...
```

Example 07 : Telephone number converter

```
            case 'D': case 'E': case 'F':
               printf("3"); break;
            case 'G': case 'H': case 'I':
               printf("4"); break;
            case 'J': case 'K': case 'L':
               printf("5"); break;
            case 'M': case 'N': case 'O':
               printf("6"); break;
            case 'P': case 'Q': case 'R': case 'S':
               printf("7");break;
            case 'T': case 'U': case 'V':
               printf("8"); break;
            case 'W': case 'X': case 'Y': case 'Z':
               printf("9");break;
            }
         }
      else {
         printf("Bad value entered\n"); exit(EXIT_SUCCESS);
      } numCtr++;
   } printf("\n\n");
   return(EXIT_SUCCESS);
}
```

Questions

1. Each of `ctype`'s functions take an integer as an argument and returns an integer. Explain why integers are used as both the input *and* output of these functions.

2. Assume the `while` loop in Example 07 has been implemented using a `for` loop instead, like this:

```
for (numCtr = 0; numCtr < PHNUMARRAYSZ; numCtr++){
   if (phoneNum[numCtr] == '\0') break;
   …
}
```

   Implement the second line inside the `for()` statement so that it executes in exactly the same way. Does this improve the readability of the code? Or does it make readability worse?

Programming Challenges

1. Example 07 uses a `switch` statement to 'translate' between alphabetical characters to numerical characters using the lines:

```
switch (cPhNum) {
   case 'A': case 'B': case 'C':
      printf("2"); break;
   case 'D': case 'E': case 'F':
      printf("3"); break;  //etc…
```

   Replace this code with an array of `int`s that performs exactly the same function. Is one version preferable to the other? Which version would you expect to execute faster? Which takes the least space? Which is more readable? Which more upgradeable?

   Hint: See **K&R 41** on how to convert an alphabetic character to its numerical value in the alphabet, i.e. 'A' = 0, 'B' = 1, 'C' = 2, etc.

Programming Challenges

2. Rewrite Example 07 to implement the following features:

   a) The digital output is stored (or buffered) in an array of 11 `int`s prior to output. If the format is incorrect (e.g. not enough digits in the output, or non-alphanumeric characters in the input) then an error message is printed; but otherwise no part of the input is displayed. (Note: This feature was overlooked in Example 07; hence we get output like "1-800-Bad value entered" when it should just say: "Bad value entered")

   b) The output should be formatted correctly with '-' at the appropriate locations, e.g. 1-800-622-6232

3. Using the `switch` statement, write a calculator program in which the user enters one integer, enters the mathematical operation (+, -, *, /, %), enters the second integer, and finally presses = sign. Note that your program should print out accurate results when the division operation is requested, even thought the program uses only integers. For example, the calculation 3/5 should yield 0.6, not 0.

Programming Challenges

4. Modify the calculator program to handle floating point numbers. Now modify the program further to allow the user to output results of varying precision. For example, the output of 2/3 with six digits of precision is 0.666666.

5. Each strand of DNA is composed of *only* four nucleic acids—Adenine (`A`), Guanine (`G`), Cytosine (`C`), and Thymine (`T`)—strung together like this:

   ```
   3'-ACGCCTTATCGGACTATTTGCCACTCAGCTAC-5'
   ```

   where 3' and 5' are used to signal the ends of the strand (according to certain molecular conventions). The details need not concern us, except for the fact that we always write DNA single strands starting with the 3' end and finishing at the 5' end.

   (con't)

Programming Challenges

In a DNA double helix, the nucleic acids are always paired up, so that `A` is always paired with `T`, and `C` with `G`, but with the 3' and 5' ends reversed, like this:

```
3'-ACGCCTTATCGGACTATTTGCCACTCAGCTAC-5'
5'-TGCGGAATAGCCTGATAAACGGTGAGTCGATG-3'
```

Given a single-strand of DNA as input (entered as a string *with or without* the 3'- / -5' notation), output the complementary strand, starting at the 3' end and finishing with the 5' end, as per convention. So, for example, the compliment of the top strand given above would be:

```
3'-GTAGCTGAGTGGCAAATAGTCCGATAAGGCGT-5'
```

Your code should flag an error if something other than an 'A', 'T', 'C', or 'G' is entered.

Example 08 : Change back from a vending machine

Program Description:

Vending machines must calculate the change returned to the purchaser of a snack after an initial amount has been fed into the machine and the selection has been made. In the early days, this feat was performed mechanically; now, a single chip controls the calculation.

This program allows the user to enter the total amount of the currency entered into a vending machine in dollars and cents, and then, assuming the price of the item selected is fixed at $2.25, returns the total number of nickels, dimes, quarters, loonies, and twonies. For simplicity, it is assumed that all amounts entered have been rounded to the nearest nickel (since pennies no longer exist).

Example 08 : Change back from a vending machine

```
houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 5.00
The change back from 5.00 is: 2.75
1 X 2.00
3 X 0.25

houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 4.15
The change back from 4.15 is: 1.90
1 X 1.00
3 X 0.25
1 X 0.10
1 X 0.05

houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 2.65
The change back from 2.65 is: 0.40
1 X 0.25
1 X 0.10
1 X 0.05

houtmad@ubuntu:~/examples$
```

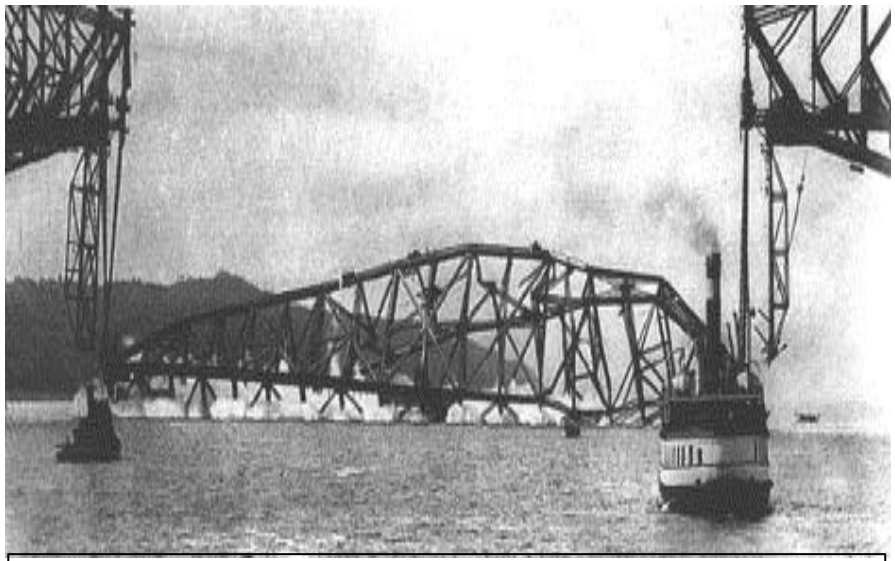
2.34 Software Design: General Steps

- Short programs can oftentimes be written using a quick top-down approach, breaking the problem into smaller units until each component performs a single function—"Everything should be made as simple as possible…". Larger programs may require more detailed analysis, including, in the world of objects, UML. In any case, most programs require *some* planning; only the simplest of programs can be written 'off the top of your head', and even then, you can expect to see a number of bugs.
- C Program development follows the usual set of guidelines for software development:

1. Describe the problem to be solved (e.g. as outlined on the cover page for each example in this module);
2. Create one or more algorithms that outline the steps that will need to be followed; this often takes the form of pseudocode/PDL or a flow chart;
3. Desk check your PDL/flowchart for correctness;
4. Develop your testing regime as you go;
5. Code the program;
6. Test the program thoroughly;
7. Maintain and update the code

Notes on software design: Overview

- Software development is an engineering discipline. Like bridges, buildings, and airplanes, large programs require considerable planning long before actual deployment. Failure to anticipate potential problems can be catastrophic.



PONT DE QUEBEC BRIDGE COLLAPSE, 1907

De HAVILLAND *COMET* CRASH, 1954

Notes on software design: Overview

- Planning also includes addressing issues such as design layout, ergonomics, and accessibility. Even if your software works *exactly* as planned, it means nothing if it doesn't address real and potential problems, particularly if those problems could have catastrophic consequences

THREE MILE ISLAND CONTROL ROOM, 1979

Notes on software design: Overview

- Finally, even well-designed systems can suffer catastrophic collapse under unforeseen circumstances. Therefore, testing with real data under carefully controlled conditions is essential.

TACOMA NARROWS SUSPENSION BRIDGE COLLAPSE, 1940

BLACK MONDAY FINANCIAL COLLAPSE, 1987

See: http://www.thebubblebubble.com/1987-crash/

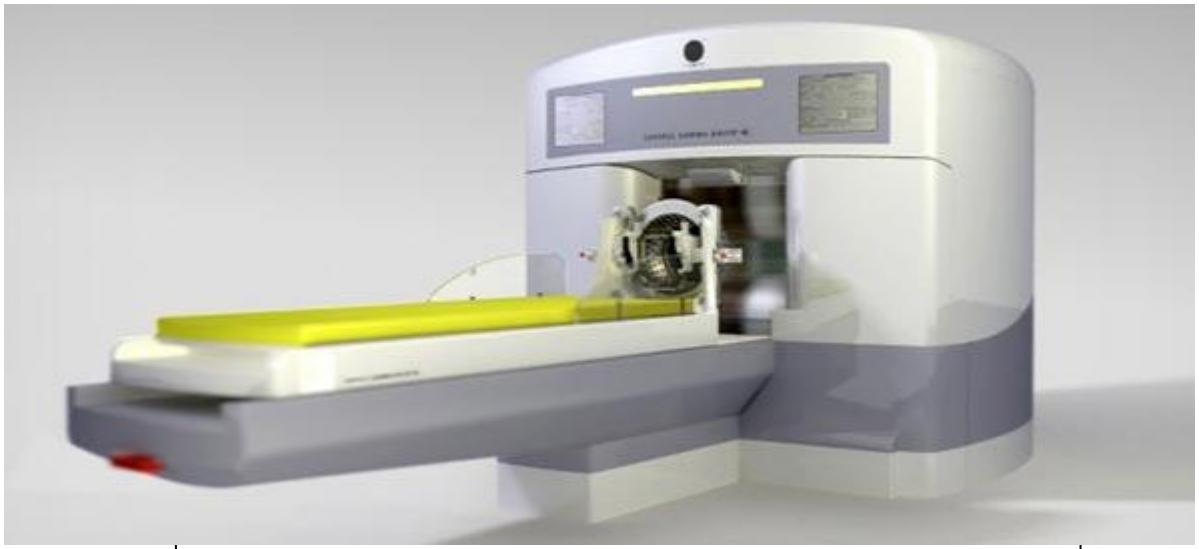
Notes on software design: Overview

- Success is never guaranteed. NASA has one of the best coding shops on the planet. Despite having a reported ten software testers for every programmer, this has not prevented software disasters, like the loss of the Mars Climate Orbiter in 1999.

MARS CLIMATE ORBITER, 1999

Notes on software design: Overview

- One of the best-studied examples of bad software design—which led to five deaths as well as life-long incapacity and pain for a number of victims—is that of the Therac-25® medical accelerator, built by AECL in the 1980s. The inexperience of one of the device's programmers was held to be a contributing factor. The repercussions of this incident continue to reverberate to this day: medical software must be built to very high standards, which came into effect as a direct result of this disaster (which was entirely preventable).



THERAC-25® MEDICAL ACCELERATOR, 1985

See: http://www.ccnr.org/fatal_dose.html, http://en.wikipedia.org/wiki/Therac-25

Notes on software design: Overview

- Worldwide, poor software design and planning continues to cost billions of dollars annually.

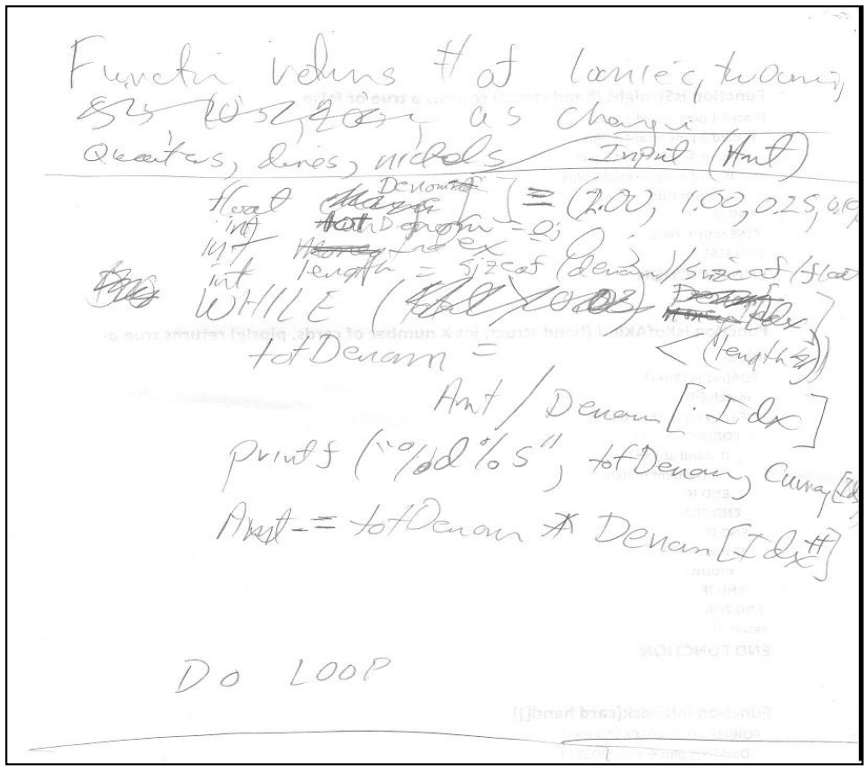
OBAMACARE ROLLOUT, 2013

2.34 Software Design: PDL / Pseudocode

- Pseudocode, or Program Description Language (PDL) is an artificial language useful for sketching out program ideas without regard to any particular language, variable naming restrictions, or syntax. It allows the programmer to concentrate on the problem at hand free from the constraints that would be imposed if they were operating in a particular programming language.

  Shown at right is the first draft of this program, just after I had the idea.

  (Yes, my version of pseudocode looks a lot like C.)



2.34 Software Design: PDL / Pseudocode

- While PDL has a 'free form' feel to it, you should nonetheless apply the same terms consistently. The following 'glossary' indicates the syntax of (what might be called) 'Algonquin Pseudocode.'

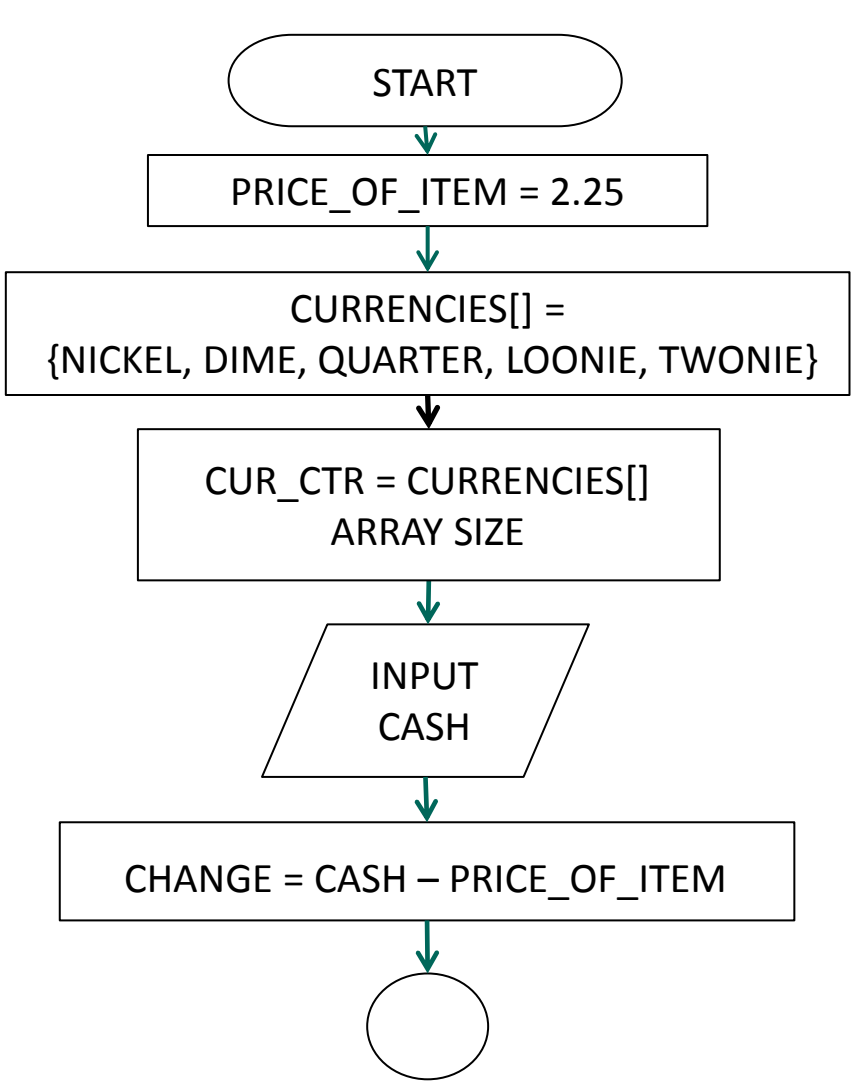
Input/Output	Decision Statement	Loop Statement
GET (i.e. input)	IF…ENDIF	FOR…ENDFOR
PUT (i.e. output)	IF…ELSE…ENDIF	WHILE…ENDWHILE
		DO…WHILE

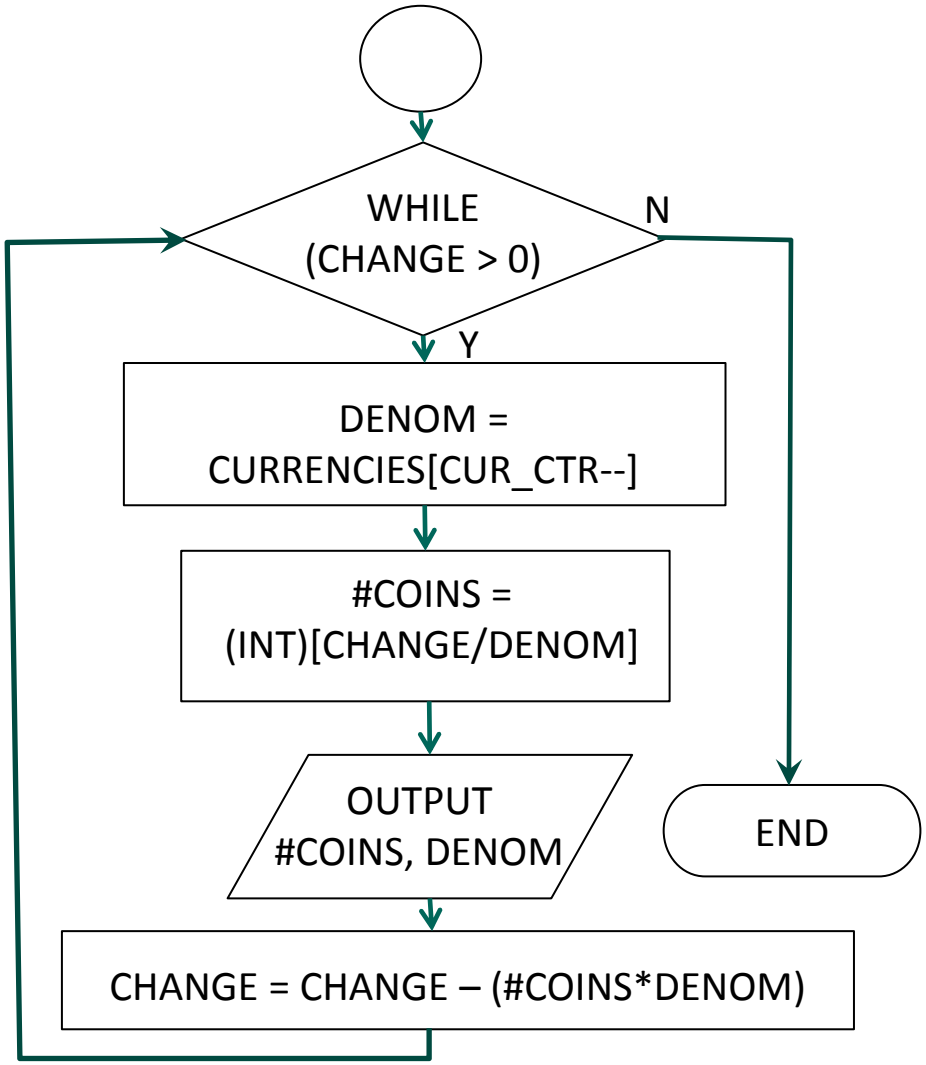
Assignment	Comparison	Logical Operators
← (i.e. <--)	EQUAL	AND
Mathematical Operators	NOT EQUAL	OR XOR
+ - / * MOD SIN() etc.	>= <= > <	NOT

2.34 Software Design: PDL / Pseudocode

```
GET PRICE_OF_ITEM
GET DENOMS_ARRAY(NICKELS, DIMES,…)
ARCTR <- SIZEOF(DENOMS_ARRAY)

PUT "ENTER CASH AMOUNT"
GET CASH

IF (CASH < PRICE_OF_ITEM)
   PUT "ERROR: INSUFFICIENT FUNDS"
   EXIT
ELSE
   CHANGE <- CASH – PRICE_OF_ITEM
   WHILE (CHANGE > 0)
      CURRENT_DENOMINATION <- DENOMS_ARRAY(ARCTR)
      #COINS <- (INT PORTION)CHANGE/CURRENT_DENOMINATION
      CHANGE <- CHANGE – (#COINS * CURRENT_DENOMINATION)
      PUT "#COINS x CURRENT_DENOMINATION"
      ARCTR <- ARCTR – 1
   END WHILE
END IF

END
```

2.34 Software Design: flowcharts





Example 08 : Change back from a vending machine

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "currency.h"

#define PRICE_OF_ITEM 2.25
#define ARSZ(array, type) (sizeof(array)/sizeof(type))

int main (void){
   const float units[]={NICKEL, DIME, QUARTER, LOONIE, TWONIE};
   float cash, change;
   int numberOfCoins, arCtr = (ARSZ(units, float))-1;

   printf("Enter the total cash entered as payment as $.cc ");
   scanf("%f", &cash);
   assert(cash >= PRICE_OF_ITEM);
   change = cash - PRICE_OF_ITEM;

   // at this point we've calculated the amount of change
   // to return to the user.  Now to figure out the twonies,
   // loonies, quarters, dimes and nickels...
   //                                          (con't)
```

Example 07 : Change back from a vending machine

```
// Reminder: Declarations from the previous slide:
//
//   const float units[]={NICKEL, DIME, QUARTER, etc.};
//   float cash, change;
//   int numberOfCoins, arCtr = (ARSZ(double, units))-1 (=4);

 printf("The change back from %2.2f is: %2.2f\n", cash, change);

   while ((change >= 0.0) && (arCtr >= 0)){
      numberOfCoins = change/units[arCtr];
      if (numberOfCoins){
         printf("%d X %2.2f\n", numberOfCoins, units[arCtr]);
         change -= numberOfCoins * units[arCtr];
      }
      --arCtr;
   }
   printf("\n");
   return EXIT_SUCCESS;
}
```

Run-time Error!

2.34 Software Design: Testing

- Programs must be tested thoroughly to ensure they function reliably. This often involves the use of a test plan, like the one shown below:

Description	Number students	Grades	Average/Result
normal	3	100, 90, 80	90
normal	1	45	45
Boundary – number students <= 0	-1, 0		Invalid message …reenter number of students
Boundary – grade = 0	2	0, 10	5
Boundary grade < 0	2	-4	Invalid message reenter grade
Boundary – grade = 100	5	100, 100, 100, 100, 100	100
Boundary – grade > 100			Invalid message reenter grade

2.34 Software Design: Testing

- The best person to design a test plan is usually the programmer who wrote the code, since they will know where the weaknesses in the program reside. Therefore, you should think about your testing regime *while* you are writing your code, not afterwards, and test all reasonable situations.
- Note however, that programmers may have an innate bias in *wanting* to believe their code works correctly, testing only for the situations they believe have a reasonable probability of occurrence, often missing evidence to the contrary. Software testers are employed exactly because they provide a perspective independent from that of the programmer. Generally speaking, programmers and testers see things very differently:
  - Programmers like to believe that their code works correctly, unless proven otherwise;
  - Software testers believe that all software has bugs, and work hard at finding them. (Since software testing is an entry-level position, the surest road to career advancement is to take down a few programmers…this, in turn, keeps the programmers sharp.)

2.34 Software Design: Testing

- The following output shows that there's a problem in Example08: it has a subtle bug that only careful testing reveals. Can you spot the bug in the original code?

```
houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 3.65
The change back from 3.65 is: 1.40
1 X 1.00
1 X 0.25
1 X 0.10
1 X 0.05

houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 3.60
The change back from 3.60 is: 1.35
1 X 1.00
1 X 0.25
1 X 0.05

houtmad@ubuntu:~/examples$ Ex8
Enter the total cash entered as payment as $.cc 2.35
The change back from 2.35 is: 0.10
1 X 0.05

houtmad@ubuntu:~/examples$
```

Questions

1. Find the bug in Example 08. Note that this problem is not limited to C, but occurs C++, Java, and other languages as well.
2. Does converting the data types used in Example08 from `floats` to `doubles` solve the problem?
3. Rewrite Example8 using integral values, i.e. convert all inputs from floating point values like '5.00' to cents e.g. 500.

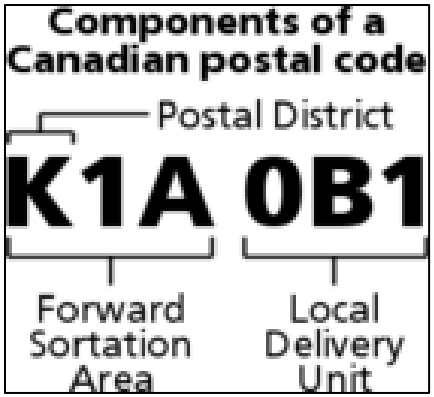
Module 2 Programming Challenges

Write the program indicated in each of the following program descriptions. Each program should take not more than 30 lines of code. For each program:

a. *Think* about what is being asked before you begin doing anything else; be sure you understand the details;
b. *Look* for patterns that repeat; these will simplify the coding;
c. *Anticipate* the possibility that the code will need to be expanded; how might your code need to be modified some day?
d. *Map out* the program with pseudocode/PDL or a flowchart before you sit down to write a single line of code;
e. *Desk check* your pseudocode for the flow of execution;
f. *Choose* suitable test values that will help reveal potential problems with the code;
g. …and *only then* begin to write your code;
h. *Run the tests* and repeat steps *f-g* until the program works reliably under all possible circumstances.

Programming Challenges

Postal Code Validator



Program Description:

Canadian postal codes consist of a sequence of six characters in the form ANA NAN, where A is an alphabetical character, and N is a numerical character. This program checks for the validity of any postal code entered by the user by (1) reading in the postal code as a string seven characters in length (assume a space always exists at the 4th location of the string); (2) looping through `isalpha()` and `isdigit()` three times to validate the code; and (3) prints out an error message like "Not a valid digit at position 5 this postal code" if a mistake is found

Programming Challenges

Scrabble® Word Value

Program Description:

In the game of Scrabble®, players are given a number of random letters and asked to make words out of them. Each word scores certain points according to the letters used, whose values are given in the following array:

```
const unsigned int values[26] =
      {1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,10,1,1,1,1,4,4,8,4,10};
```

This program reads in a scrabble word and calculates the equivalent word value (ignoring any special scrabble conventions, like 'double' and 'triple' word scores.)

Hint: See **K&R 41** for a tip on converting from 'A', 'B', 'C',… to 0, 1, 2…, which will help you access the numerical values in the above array.

Programming Challenges

The Memory Game

Program Description:

This program (1) generates ten non-repeating, randomly-sequenced integers between 0 and 9, (2) displays them for 10 seconds, (3) blanks the console, and then (4) prompts the user to enter the values in the order just displayed. If the order is correct, the program offers a congratulatory message, otherwise something like "bzzzt…wrong answer"—and then exits the guessing loop. The program then offers the user the chance to play again, or quit.

Helpful suggestions:

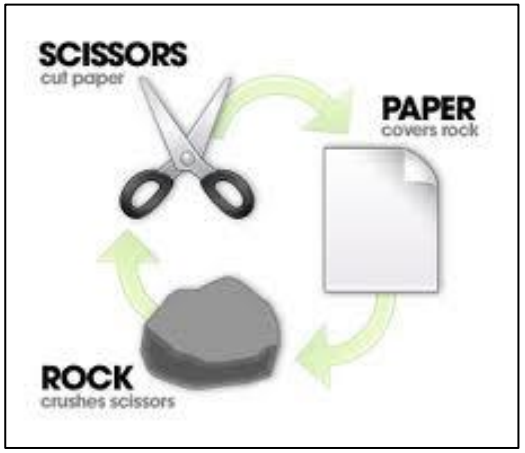
(1) Create an array of 10 integers, with each array element storing its own index (i.e. `ar[0] = 0`, `ar[1] = 1`, etc.). For each index, generate a random integer between 0 and 9, and swap the value stored at the current index with the value stored at the randomly generated index; this should create a shuffled array contianing 10 unique integers

(2) To pause program execution for 10 seconds, use `system(sleep(10));`

(3) To programmatically clear the screen after the 10 second delay, use `system("clear");`

Programming Challenges

Rock, Paper, Scissors



Program Description:

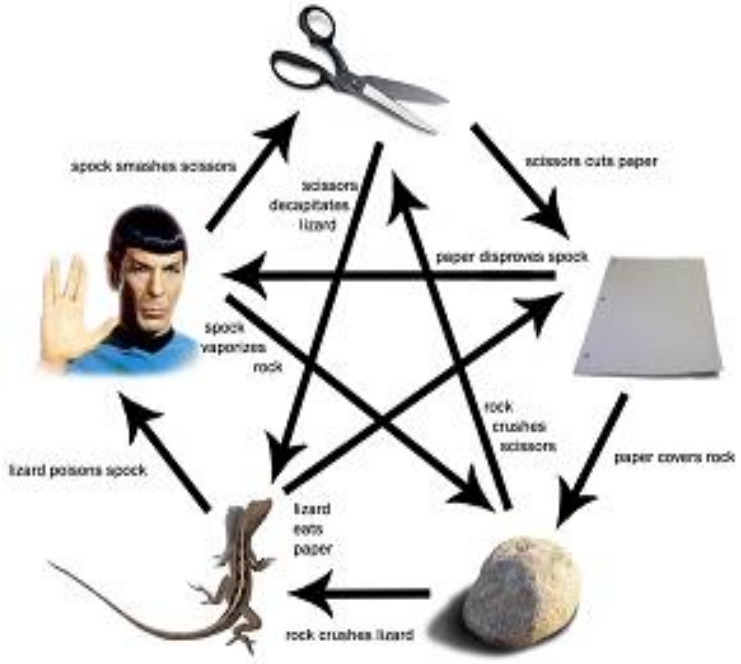
In this program the user enters a character that represents rock (R), paper (P), or scissors (S), the computer responds by generating its own random 'guess', and the winner of the contest is announced. The game proceeds until the user presses a key other than R, P or S, at which time the computer announces the final results i.e. wins, losses and ties, and exits.

Again, the rules of the game are:

a. Rock crushes scissors
b. Scissors cut paper
c. Paper covers rock

Programming Challenges

Rock, Paper, Scissors, Lizard, Spock



Program Description:

This is a variation on the Rock, Paper, Scissors game seen in the previous challenge. Here, there are five choices rather than three, and the rules are somewhat more complicated (see diagram above, or see visit http://www.youtube.com/watch?v=x5Q6-wMx-K8 for a quick summary). Aside from these two details, the overall logic remains the same. Note that, at this stage, using `switch` or multiple `if…else` statements for each possibility is unwieldy; it becomes imperative to find a mathematical algorithm that simplifies the underlying logic of the game in order to code this program efficiently.

Suggestion: for inputs, use 'L' (for lizard) and 'V' (for Vulcan, since 'S' is already taken)

Programming Challenges

Note: Variations on The Rock, Paper, Scissors, Lizard, Spock Game

Several variations of this game exist. Can your program be extended to cover other possibilities, without a major overhaul of the code?