CM 10227/50258: Lecture 3

Dr. Rachid Hourizi and Dr. Michael Wright

October 18, 2016

Last Week

- More Methods
- Conditionals
- Recursion

This Week

- Iteration
- Collections: Arrays and Strings

Resources

- General help on C
  - The C Book - `http://publications.gbdirect.co.uk/c_book/`
  - The Library has books on learning C
- More help with this course
  - Moodle `http://moodle.bath.ac.uk/course/view.php?id=30475`
- Online C IDE
  - `https://www.codechef.com/ide`
  - Remember to select C as the language you are coding in...

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs (Fridays 17:15 to 19:15 in CB 5.13)
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)
  - The Drop in Sessions (booked 20 min appointments from Wednesday @ 11:15-13:05 EB0.7)

- If you are finding the pace a little slow on the other hand, you can now sign up to
    - The Advanced Programming Labs
    - More information on Moodle...

    - Programming Competition: 1st meeting on Thursday

Questions?

Back to C

- We have discussed functions ...
- ... and how they eliminate repetitive code
- ... and simplify code by grouping complex set of statements behind a single command

```c
#include <stdio.h>

int main(void)
{
    int num1 = 7;
    int num2 = 3;
    int result = remainder_of(num1 ,num2);
    print_input(result);
    return 0;
}

void print_input(int i)
{
    printf("%d\n",i);
}

int remainder_of(int i, int j)
{
    return (i%j);
}
```

Back to C

- We have discussed the use of recursion as a means to execute a block of code multiple times

```
int  fibonacci(int n)
{
    if(n == 0)
    {
        return  0;
    }
    else if(n == 1)
    {
        return  1;
    }
    else
    {
        return  fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Back to C

- This week, we will look at an alternative approach to achieving repetition within your programs
- This alternative approach is **iteration**

Iteration

- A process wherein a set of instructions are repeated
- ... in a sequence
- ... a specified number of times
- ... or until a condition is met.

Iteration

- Iteration is the repetition of a process in a computer program, usually done with the help of loops
- C provides a number of these loop statements.
- We will look at the while-loop first

- The standard form of a C while statement is as follows:

```
while (condition is true)
{
    statement 1;
    statement 2;
    statement n;
}
```

```
#include <stdio.h>

main()
{
    int count;
    count = 0;
    while(count < 10)
    {
        printf("hello\n");
        count = count + 1;
    }
    return(0);
}
```

- The flow of execution for a WHILE statement is as follows:
  - Evaluate the condition, yielding TRUE or FALSE.
  - If the condition is FALSE, exit the WHILE statement and continue execution at the next statement.
  - If the condition is TRUE, execute each of the statements in the body and then go back to step 1.
- Notice that if the condition is FALSE the first time through the loop, the statements inside the loop are never executed.
- To avoid an infinite loop, make sure that the condition eventually becomes FALSE

- C also provides a variant of the while loop:
  - do ... while
- do ... while is very similar to the while loop ...
- ... but the condition is checked **after** the first execution of the body statements
- so (in contrast to the while loop) even if the condition is false, the first time through the loop, the statements in the body are still executed once.

- The standard form of a C do . . . while statement is as follows:

```
do
{
    statement 1;
    statement 2;
    statement n;
}
while (condition is true);
```

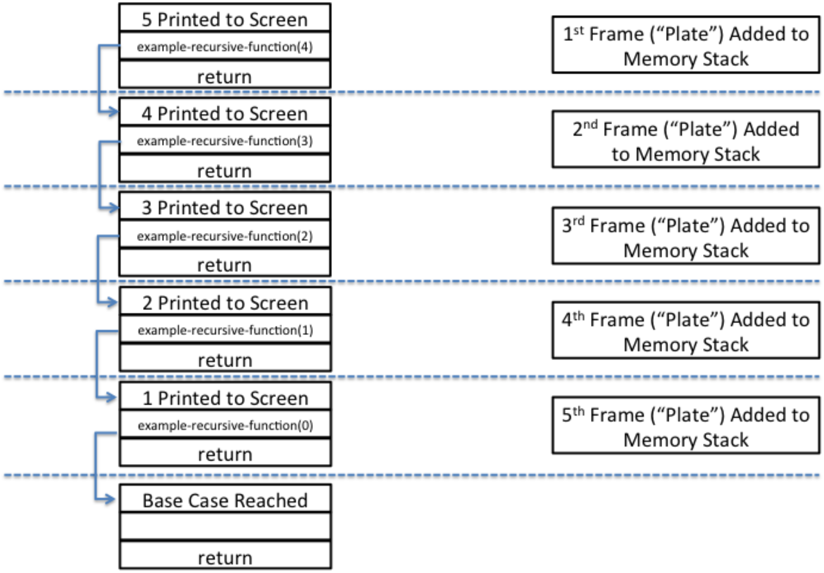
```c
#include <stdio.h>

main()
{
    int count;
    count = 0;
    do
    {
        printf("hello\n");
        count = count + 1;
    }
    while(count < 10);
    return(0);
}
```

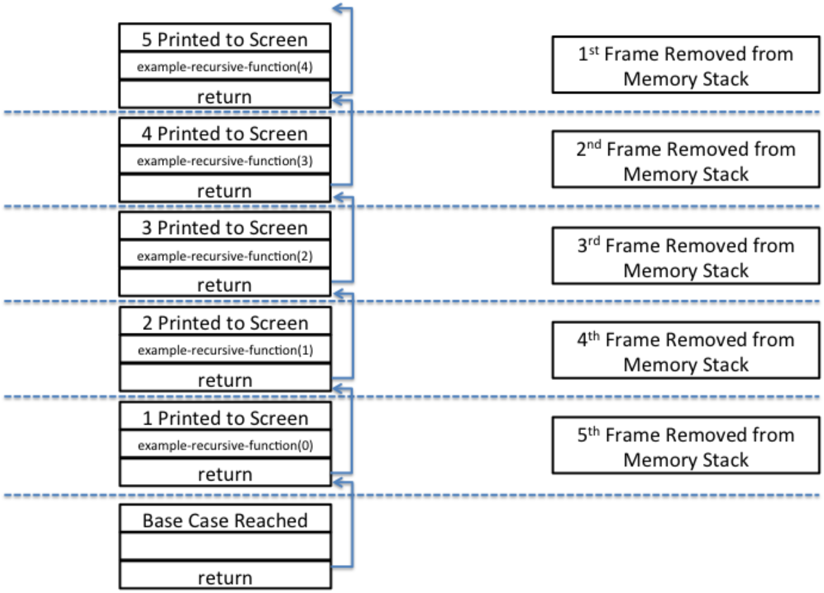
- The flow of execution for a do . . . while statement is as follows:
  - Execute each of the statements in the body
  - Evaluate the condition, yielding TRUE or FALSE.
  - If the condition is FALSE, exit the DO . . . WHILE statement and continue execution at the next statement.
  - If the condition is TRUE then go back to step 1.
- To avoid an infinite loop, make sure that the condition eventually becomes false
- Even if the condition is false the first time through the loop, the statements in the body are still executed once.

Iteration

- Compared to recursion...
- ... Iteration if often faster because we don't need to maintain the stack

```
// recursive fibonacci
int  fibonacci(int n)
{
    if(n == 0)
    {
        return  0;
    }
    else if(n == 1)
    {
        return  1;
    }
    else
    {
        return  fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```


5 Printed to Screen
example-recursive-function(4)
return
4 Printed to Screen
example-recursive-function(3)
return
3 Printed to Screen
example-recursive-function(2)
return
2 Printed to Screen
example-recursive-function(1)
return
1 Printed to Screen
example-recursive-function(0)
return
Base Case Reached
return
1st Frame ("Plate") Added to Memory Stack
2nd Frame ("Plate") Added to Memory Stack
3rd Frame ("Plate") Added to Memory Stack
4th Frame ("Plate") Added to Memory Stack
5th Frame ("Plate") Added to Memory Stack


5 Printed to Screen
example-recursive-function(4)
return
4 Printed to Screen
example-recursive-function(3)
return
3 Printed to Screen
example-recursive-function(2)
return
2 Printed to Screen
example-recursive-function(1)
return
1 Printed to Screen
example-recursive-function(0)
return
Base Case Reached
return
1st Frame Removed from Memory Stack
2nd Frame Removed from Memory Stack
3rd Frame Removed from Memory Stack
4th Frame Removed from Memory Stack
5th Frame Removed from Memory Stack

```
// iterative fibonacci
int fibonacci(int n) {
    int prev = 0;
    int curr = 1;
    int next;
    int count = 2;
    while(count <= n){
        next = prev+curr;
        prev = curr;
        curr = next;
        count = count + 1;
    }
    return curr;
}
```

More Examples of a WHILE Loop

- One of the first uses of computers was to generate mathematical tables, such as log tables.
- Table generation is a good example of iteration.
- Over the following slides, we will develop code that prints out mathematical tables
- We will take an incremental development approach (proposed in the lecture on incremental development) i.e.
  - start with a main function that produces a simple output (such as a single print statement)
  - develop code gradually by adding lines
  - when it works extract it into a function

- We can start with a simple main() function that calls a second create_tables() function
- In this first version, the create_tables() function does nothing but print out some placeholder text

```c
#include <stdio.h>

main()
{
    create_tables();
}

void create_tables(void)
{
    int count;
    count = 1;
    while(count < 10)
    {
        printf("placeholder\n");
        count = count + 1;
    }
}
```

- We can then add a simple iteration to print out a table of logarithms
- (base e by default)

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

main()
{
    create_tables();
}

void create_tables(void)
{
    int count;
    count = 1;
    while(count < 10)
    {
        double l = log(count);
        printf("%d␣\t␣%f␣\n",count,l);
        count = count + 1;
    }
}
```

```
$ gcc -o example_code example_code.c

$ ./example_code
1         0.000000
2         0.693147
3         1.098612
4         1.386294
5         1.609438
6         1.791759
7         1.945910
8         2.079442
9         2.197225
```

Another Example...

- The code examples on the previous slides show the utility of iteration when producing one-dimensional tables
- Iteration is also useful when producing two-dimensional tables
- A two-dimensional table is a table where you choose a row and a column and read the value at the intersection.
- A multiplication table is a good example.

- Lets print a multiplication table for the values from 1 to 6.
- Once again, we will build the code that produces this multiplication table step by step
- i.e. we will use incremental development

- We can start with a simple loop that prints the multiples of 2 all on one line:

```c
#include <stdio.h>

main()
{
    create_tables();
}

void create_tables(void)
{
    int i;
    i = 1;
    while(i <= 6)
    {
        int mult = i * 2;
        printf("%d\t",mult);
        i = i + 1;
    }
}
```

- Where:
  - i is a counter, or loop variable.
  - mult is the calculation want to print
- The output of this program is:

```
$ gcc -o example_code example_code.c

$ ./example_code
2       4       6       8       10      12
```

- The next step is to encapsulate and generalise.

Encapsulation and Generalisation

- Encapsulation usually means wrapping a piece of code in a function.
  - E.g. giving the name is_even() to a series of statements that test whether an integer is even rather than writing out each of those statements every time you need to run that test
- Generalisation means taking something specific and making it more general
  - e.g. starting with code that prints multiples of 2,
  - And changing it so that it prints multiples of any integer.

- Here's the previous loop as a function generalised to print multiples of any number

```c
#include <stdio.h>

...

void print_multiples(int n)
{
    int i = 1;
    while(i <= 6)
    {
        printf("%d\t",i*n);
        i = i + 1;
    }
    printf("\n");
}
```

- print_multiples(3) outputs:

```
$ ./example_code
3        6        9        12       15       18
```

- print_multiples(4) outputs:

```
$ ./example_code
4        8        12       16       20       24
```

- To print an entire multiplication table, we wrap (encapsulate) calls to print_multiples in a loop:

```
#include <stdio.h>

main()
{
    int count = 1;
    while(count < 6)
    {
        print_multiples(count);
        count = count + 1;
    }
}

...
```

- We can now wrap (encapsulate) that encapsulation in another function

```
#include <stdio.h>

main()
{
   print_tables();
}

void print_tables()
{
   int count = 1;
   while(count < 6)
   {
       print_multiples(count);
       count = count + 1;
   }
}

...
```

- To generalise print_tables, add a parameter

```
#include <stdio.h>

...

void print_tables(int high)
{
   int count = 1;
   while(count < high)
   {
       print_multiples(count);
       count = count + 1;
   }
}

...
```

- If printTable is called with the argument 7, we get the following output

```
$ ./example_code
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

```
#include <stdio.h>

main()
{
   print_tables(7);
}

void print_tables(int high)
{
   int count = 1;
   while(count < high)
   {
       print_multiples(count);
       count = count + 1;
   }
}

...
```

```c
...

void print_multiples(int n)
{
    int i = 1;
    while(i <= 6)
    {
        printf("%d\t",i*n);
        i = i + 1;
    }
    printf("\n");
}
```

Next Steps

- The code we have written so far prints the results of our calculations to the standard out
- However, we might want to save these results for future use
- One way to do this in C is by using an Array
- Arrays are a common data structure in all programming languages

Arrays

- You can recognise an array in a C program by the appearance of square brackets [ ]
- An array is a container that holds a fixed number of values of a single type e.g.
  - 9 ints e.g. [1, 2, 300, 4, 500, 6, 7, 800, 9]
  - 4 chars e.g. [a, z, e, g]

- Each item within an array is called an **element**
- Each element is referred to by its numerical **index**
  - The first element in an array is referred to as **[0]**
  - The second element is referred to as **[1]**
  - The third element is referred to as **[2]**
  - and so on ...

- In C a one dimensional array (a series of values) is declared

```
int an_array[7];
```

- Note: You have to allocate memory for the number of elements that you want to exist in that array before you can store data within it

- Any expression with type **int** can be used as an index.

```
an_array[7-5];
```

- Other data types may **not** be used as an index

```
an_array[2.0];    /* ERROR */
```

- If you refer to an element that does not exist, you will get an error.

```
int an_array[5];
an_array[20] = 21;    /* ERROR */
```

- Correct referencing is, perhaps more easily understood through use of an example
- The following code sets up an array of integers ...
- ... and assigns a value to each element in the array

```
#include <stdio.h>

main()
{
    int myarray[5];
    myarray[0] = 1;
    myarray[1] = 2;
    myarray[2] = 3;
    myarray[3] = 4;
    myarray[4] = 5;
}
```

- We can now print data held in the array as we would any other variable

```c
printf("%d\t",myarray[2]);
```

- We can also assign the value held in an array element to another variable

```c
int y = myarray[2];
printf("%d\t",y);
```

- In C you can also declare two-dimensional (e.g. 2x2) arrays

```
#include <stdio.h>

main()
{
    int myarray[2][2];
    myarray[0][0] = 1;
    myarray[0][1] = 2;
    myarray[1][0] = 3;
    myarray[1][1] = 4;
}
```

- We now have all of the C language structures that we need to store the output of print_tables in a two-dimensional array
- Actually writing that code will be one of your lab exercises so I will not go through the answer in these slides

- Be aware that this last step in our incremental development exercise is not as easy as it looks
- You may find it helpful to reconsider the number of methods that you implement when developing this storage function
- Remember to take an incremental approach to building your programs
- That is, start with a program that works and then extend it
- If you get stuck, ask the tutors for help!

Strings

- In the first two weeks of the course, we have looked at different ways to describe data
- More specifically, we have seen the importance in C of describing the type of the data that we store and use.
  - e.g. ints, chars, floats, doubles and booleans
- Assigning types
  - Helps the compiler to know how much memory to allocate and
  - Can also be used to limit the computations on each piece of data to those that are relevant to data of each type

- In the previous section of these slides, we also looked at collecting data in an Array
- For the remainder of this week and the first part of next, we will look the ways in which C uses Arrays to underpin another very common data type: Strings

- Strings in C are stored as an array of chars
- More specifically, a string (in C) is a one dimensional array of chars terminated by a null character ('\0')
- So a String containing the single word “Hello” is actually a six char array (5 letters and a null character).

- A variable containing the String “Hello” can be declared and initialised in multiple ways

- The long version mirrors the ways in which we populated arrays earlier

```
char greeting[6];

greeting[0] = 'H';
greeting[1] = 'e';
greeting[2] = 'l';
greeting[3] = 'l';
greeting[4] = 'o';
greeting[5] = '\0';
```

- A slightly shorter version is as follows:

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- In C, you can also declare and initialise an Array of characters with a String literal (a special case):

```
char greeting[] = "Hello";
```

- Note in this case that the compiler adds the null character ('\0') for you.

- Be careful when using this special case, however since neither of the two following approaches is permitted:

```
char greeting[];
greeting =  Hello ;            //syntax error

char greeting2[12];
greeting2 =  Hello Again ; //also a syntax error
```

- An aside: Just as Arrays are stored as Arrays of chars in C,
- chars themselves are actually stored as as unsigned integers
- i.e. an int code is used to represent each char e.g.
- 97 is the code used to represent the char 'a'
- 113 is the code used to represent the char 'q'
- 65 is the code used to represent the char 'A'

- NOTE

  : Upper case characters are associated with different codes than their lower case equivalents
- NOTE

  : and special characters like '&' also have codes associated with them (38)
- more detail on the American Standard Codes for Information Interchange (ASCII) can be found online
- e.g. http://web.cs.mun.ca/ michael/c/ascii-table.html

- One implication of the way in which C stores chars is that we can also declare and initialise the String "Hello" as follows

```
char greeting[6];

greeting[0] = 72;
greeting[1] = 101;
greeting[2] = 108;
greeting[3] = 108;
greeting[4] = 111;
greeting[5] = 0;
```

- Once we have created Strings use any of the approaches above, we can print them using printf() as before e.g.:

```
#include <stdio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   printf("Greeting␣message:␣%s\n", greeting );

}
```

```
$ Greeting message: Hello
```

- Note the use of a new format specifier, %s for Strings

- Just like the other C data types that we have looked at, strings can be...
- ... assigned to variables

```
char my_string[] = "hello␣world";
```

- ... passed to methods as parameters

```
void print_string(char a_string[])
{
    printf(a_string);
}
```

- We can manipulate individual chars in a String, just as we manipulated the individual integers in the multiplication arrays introduced earlier:

```c
#include <stdio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    greeting[1]='o';
    greeting[2]='w';
    greeting[3]='d';
    greeting[1]='y';
   printf("Greeting␣message:␣%s\n", greeting );
    printf("First␣letter␣is:␣%c\n", greeting[0]);
}
```

```
Greeting message: Howdy
First letter is: H
```

- In addition to the standard array functions, including string.h in a C program provides support for a wide range of functions that manipulate null-terminated strings e.g.:
    - strcpy(s1,s2) copies string s2 into s1
    - strcat(s1,s2) Concatenates (adds) string s2 onto the send of string s1
    - strlen(s1) returns the length of string s1 as an integer

- The following example makes use of few of the above-mentioned functions:

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

int main ()
{
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   strcpy(str3, str1); /* copy str1 into str3 */
   printf("strcpy(␣str3,␣str1)␣:␣␣%s\n", str3 );

   strcat( str1, str2); /* concatenate str1 & str2 */
   printf("strcat(␣str1,␣str2):␣␣␣%s\n", str1 );

   len = strlen(str1); /* length of str1 */
   printf("strlen(str1)␣:␣␣%d\n", len );
   return 0;
}
```

- Using a combination of strlen() and iteration, we can print out each char in a string on a separate line

```
void print_string(char a_string[])
{
    int i = 0;
    while (i < strlen(a_string)){
        printf("%c\n", a_string[i]);
        i++;
    }
}
```

- strlen will return how many elements are in the string
- But remember to access arrays we start counting from 0
- The WRONG way to find the last letter of a string:

```
char my_string[] = "Hello␣World";
int i = strlen(my_string);
printf("%c\n",my_string[i]);    /* ERROR!! */
```

- Remember that the first character in a string has index 0 so the correct code would be:

```
char my_string[] = "Hello␣World";
int i = strlen(my_string);
printf("%c\n",my_string[i-1]);
```

- A common computation to perform on a string (or any collection of data) is start at the beginning, select each character (element) in turn, do something to it (e.g. print it), and continue until the end.
- This pattern of processing is called a traversal.
- We saw how to do this with a while loop on a previous slide

```c
void print_string(char a_string[])
{
    int i = 0;
    while (i < strlen(a_string)){
        printf("%c\n", a_string[i]);
        i++;
    }
}
```

- We can also achieve that traversal by using a for loop:

```
void print_string(char a_string[])
{
   char my_string[] = "Hello␣World";
   int i;
   for(i=0; i<strlen(my_string);i++)
   {
        printf("%c\n", my_string[i]);
   }
}
```

- The structure of a for loop is as follows:

```
for (initialization; termination; increment)
{
        statement 1
        statement 2
        statement n
}
```

- Note, however, that those examples have been carefully chosen
- We have not tried to put Strings into Arrays that are not long enough to hold them
- i.e. we have been very careful to limit the number of characters that we copy into a target string or concatenate onto the end of it to respect the number of characters that the target string can hold
- (We have on occasion left space unused at the end of an Array [e.g. created an Array that can hold 12 elements and used only six to store "Hello"])

- To create exactly the right amount of memory to hold Strings at runtime, we have to understand the way that C handles the memory associated with those arrays
- we will return to that subject next week

- It is, however, worth noting an special use of Strings in C:
- More specifically, it is worth noting that we can send one or more Strings to our main method when running a program.

- In previous exercises, you have been using gcc(with or without flags and specified output files) to compile your C code
- and used only the name of the output file to run that compiled code e.g.

```
$ gcc HelloWorld.c
$ a.out
```

- We can, however pass a string to the HelloWorld program at the point of running it and use that String within the main method
- In order to do so, however, we would need to add instructions to the main method to do exactly that

- More specifically, we need to specify two parameters (”arguments”) in the main method
  - int argc which describes the number of Strings passed as input plus one (since the name of the program is also counted)
  - int *argv which holds the Strings themselves (the program name is held in argv[1]
- an argument introduced at the command line is (unsurprisingly) referred to as a command line argument

```c
/* Original Hello World program */
#include<stdio.h>
main()
{
 printf("Hello␣World");
}
```

```
$ gcc HelloWorld.c
$ a.out
Hello World
```

```
/*
* HelloWorld
* extended to take a command line single argument
* and print it to the screen
* At this point, dont worry about the * preceding argv
*/
#include <stdio.h>
int main( int argc, char *argv[] )
{
     printf("The␣number␣of␣arguments␣passed␣is␣%d\n",
        argc-1);
     printf("The␣name␣of␣your␣program␣is␣%s\n", argv
        [0]);
     printf("The␣argument␣supplied␣is␣%s\n", argv[1]);
}
```

```
$ gcc HelloWorld.c
$ a.out testing
The number of arguments supplied is 1
The argument supplied is testing
```

- NOTE: we can pass multiple command line aguments
- if we had passed a second argument (a Second String) it would now be held in argv[2]
- and the (int) value held in argc would be 3

```
/*
* HelloWorld with guards
*/
#include <stdio.h>
int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
     printf("The␣argument␣supplied␣is␣%s\n", argv[1]);
     printf("The␣argument's␣first␣letter␣is␣%c\n",
        argv[1][0]);
   }
   else if( argc > 2 )
   {
     printf("Too␣many␣arguments␣supplied.\n");
   }
   else
   {
     printf("One␣argument␣expected.\n");
   }
}
```

```
$ gcc HelloWorld.c
$ a.out testing
The argument supplied is testing
The arguments first letter t
```

- NOTE: Do not worry if you do not fully understand the "HelloWorld with guards" code on the previous slides
- We will return to it next week
- For now, just be aware that it gives you example code that you can use to take a command line argument and print it to screen
- remember the incremental programming approach - start with something that works and add to (or subtract from) it