

# Review of the C Programming Language

David W. Walker

[WalkerDW@cardiff.ac.uk](mailto:WalkerDW@cardiff.ac.uk)

# Library Functions

- C provides a set of library functions for carrying out common tasks.
- For example, I/O is not built into the C language. Reading and writing is done using function calls, e.g.,

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

which prints the string “Hello World!” followed by a new line to the standard output (usually the terminal screen).

# Header Files

- Information about a function's interface (the datatypes of its return value, and its inputs) is usually stored in a *header file*.
- Header files are listed at the start of the code using an include statement, e.g.,

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

will include the header file `stdio.h` (needed to use I/O functions such as *printf*).

# Different Types of Header File

- System header files, such as `stdio.h`, are enclosed in pointy brackets. The compiler will look for these in a standard place.
- User header files are enclosed in double quotes, e.g.,

`#include "myfile.h"`

The compiler will look for `myfile.h` in the current directory. If `myfile.h` is somewhere else you can give the path name.

# Hello Word Example

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

Store this in file hello.c and compile like this:

```
gcc -o hello hello.c
```

Then just key in hello to run it.

# Datatypes

- The most common datatypes are:

C datatype	Meaning
int	Integer
float	Floating-point number
char	Character

- Usually int and float are 4-byte numbers, and a char is one byte.

# Formatted Printing

- To print a value use %d for an int, %f for a float, and %c for a char.

```
#include <stdio.h>
int main()
{
    int n=7;
    float v=1.234;
    char a='X';
    printf("n=%d, v=%f, a=%c\n",n,v,a);
}
```

n=7, v=1.234000, a=X

# Formatted Printing

- Specifying %4d will print an int in a field of width 4.
- Specifying %6.1f will print a float in a field of width 6 with one digit after the decimal point

```
#include <stdio.h>
int main()
{
    int n=7;
    float v=1.234;
    printf("n=%4d, v=%6.1f\n",n,v);
}
```

n= 7, v= 1.2



# Octal and Hex Printing

- Can use %o and %x to print an int in octal or hexadecimal form.

```
#include <stdio.h>
int main()
{
    int n=77;
    printf("%4d in octal is %o\n",n,n);
    printf("%4d in hex is %x\n",n,n);
}
```

77 in octal is 115  
77 in hex is 4d

# Pointers

- A pointer in C is an address in memory. For example:

```
int *pn;
```

declares pn as a pointer to an integer, i.e., the address

# Pointers

- A pointer in C is an address in memory. For example, if `n` is an integer then

`pn = &n;`

assigns the address of `n` to the variable `pn`.

- The unary operator `*` accesses an address to fetch its contents. Thus

`m = *pn;`

assigns the value stored at address `pn` to the variable `m`. This is called *dereferencing* the pointer.

- `pn` is declared as:

`int *pn;`

# Pointer Example 1

```
#include <stdio.h>
int main()
{
    int n, m;
    int *pn;
    n = 77;
    pn = &n;
    m = *pn;
    printf("n=%4d, m=%4d, pn=%p\n",n,m,pn);
}
```

n= 77, m= 77, pn=0x7fff5bf72440

# Pointer Example 2

```
#include <stdio.h>
int main()
{
    int n, m;
    int *pn;
    n = 77;
    pn = &n;
    m = *pn;
    printf("n=%4d, m=%4d, pn=%p\n",n,m,pn);
    *pn = 101;
    printf("n=%4d, m=%4d, pn=%p\n",n,m,pn);
}
```

```
n= 77, m= 77, pn=0x7fff52aaf440
n= 101, m= 77, pn=0x7fff52aaf440
```

# Arrays

- Arrays are declared by adding the array size after the variable name. For example

```
int b[10];
```

declares an array of 10 integers.

- Indexing of arrays starts at 0, so the first item in the array b is b[0] and the last is b[9].
- An array can be initialized as follows:

```
int b[10] = {0,1,2,3,4,5,6,7,8,9};
```

# Arrays and Pointers

- The address of the element at index  $i$  is

$pb = \&b[i];$

which can also be written as

$pb = b+i;$

- In particular,

$pb = b;$

assigns the address of the start of array  $b$  to the pointer  $pb$ .

- $*(b+i)$  means the same thing as  $b[i]$ , i.e., it is the value stored at index  $i$  of array  $b$ .

# Array Example

```
#include <stdio.h>
int main()
{
    int b[10]={0,1,2,3,4,5,6,7,8,9};
    int *pb;
    pb = b+5;
    printf("b[5]=%4d\n",b[5]);
    printf("*pb=%4d\n",*pb);
    printf("*(b+5)=%4d\n",*(b+5));
    printf("pb=%p\n",pb);
    printf("b+5=%p\n",b+5);
}
```

```
b[5]=  5
*pb=  5
*(b+5)=  5
pb=0x7fff5b5fa444
b+5=0x7fff5b5fa444
```



# Incrementing a Pointer into an Array

```
#include <stdio.h>
int main()
{
    int b[10]={0,1,2,3,4,5,6,7,8,9};
    int *pb;
    pb = b+5;
    printf("*pb=%4d\n",*pb);
    printf("pb=%p\n",pb);
    pb = pb +2;
    printf("*pb=%4d\n",*pb);
    printf("pb=%p\n",pb);
}
```

```
*pb=  5
pb=0x7fff5474c444
*pb=  7
pb=0x7fff5474c44c
```

# Dynamic Memory Allocation

- Use the library function *malloc* to dynamically allocate memory.
- Pass in the number of bytes required.
- Get back a pointer to the requested memory.
- For example to create a pointer to an array of 5 integers:

```
int *pn = (int *)malloc(5*sizeof(int));
```

# Malloc Example

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

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

Need to include stdlib to use malloc

```
int main()
```

```
{
```

```
    int *pn = malloc(5*sizeof(int));
```

```
    pn[0] = 0;
```

```
    pn[1] = 1;
```

```
    pn[2] = 2;
```

```
    pn[3] = 3;
```

```
    pn[4] = 4;
```

```
    printf("pn = %d,%d,%d,%d,%d\n",pn[0],pn[1],pn[2],pn[3],pn[4]);
```

```
    free(pn);
```

```
}
```

pn = 0,1,2,3,4

# Strings

- In C, a *string* is a character array terminated by a '\0' character. For example:

```
#include <stdio.h>
main()
{
    char s1[6]={'H','e','l','l','o', '\0'};
    printf("s1=%s\n",s1);
    printf("s1+1=%s\n",s1+1);
}
```

- Note the use of the %s format descriptor.

```
s1=Hello
s1+1=ello
```

# Another Way To Initialize a String

- The following code does the same thing as the previous code, but uses a different way to initialize the character array:

```
#include <stdio.h>
int main()
{
    char s1[]="Hello";
    printf("s1=%s\n",s1);
    printf("s1+1=%s\n",s1+1);
}
```

```
s1=Hello
s1+1=ello
```

# Passing Arguments to a Program

- When running a program, arguments may be placed after the program name.
- The arguments can then be processed within the program.
- To pass in arguments the main function must be written as follows:

```
int main(int argc, char **arg)
```

- `argc` is the number of arguments, and `arg` points to an array of pointers to chars, i.e., to an array of strings.

# Passing Arguments Example

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

```
int main(int argc, char **arg)
```

```
{
```

```
    printf("The name of the program is %s\n",arg[0]);
```

```
    printf("The argument is %s\n",arg[1]);
```

```
    printf("The address of the argument is %p\n",arg[1]);
```

```
    printf("The first character of the argument is %c\n",*arg[1]);
```

```
}
```

- The string `arg[0]` is always the name of the program, so `argc` is always at least 1.

The name of the program is args1

The argument is Hello

The address of the argument is 0x7fff5f54561e

The first character of the argument is H

# User-Defined Functions

- Pass in value of input and get output returned.

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

```
int square(int n)
{
    return (n*n);
}
```

```
int main()
{
    int x = 7;
    int y = square(x);
    printf("The square of %d = %d\n",x,y);
}
```

The square of 7 = 49



# Returning Values via the Argument List

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

```
void square(int n, int *m)  
{  
    *m = n*n;  
    return;  
}
```

Store the value of  $n*n$  at the address pointed to by  $m$ .

```
int main()  
{  
    int x = 7, y;  
    square(x,&y);  
    printf("The square of %d = %d\n",x,y);  
}
```

Use  $\&y$  to pass in the address of  $y$

The square of 7 = 49

# Conditional Execution

- Conditional execution with a simple if statement:  
    if (condition) statement;  
    where *condition* evaluates to true (1) or false (0).
- If the statement is compound (made up of more than one statement) use { and }:  
    if (condition){  
        statement1;  
        statement2;  
    }

# Conditions

Condition	Name	Meaning
<code>e1 &amp;&amp; e2</code>	Logical AND of e1 and e2	1 if <code>e1=e2=1</code> , otherwise 0
<code>e1    e2</code>	Logical inclusive OR of e1 and e2	0 if <code>e1=e2=0</code> , otherwise 1
<code>e1 == e2</code>	Equality	1 if e1 equals e2, otherwise false
<code>e1 != e2</code>	Not equal	1 if e1 does not equal e2, otherwise 0
<code>e1 &gt; e2</code>	Greater than	1 if e1 is greater than e2, otherwise 0
<code>e1 &lt; e2</code>	Less than	1 if e1 is less than e2, otherwise 0
<code>e1 &gt;= e2</code>	Greater than or equal	1 if e1 is greater than or equal to e2, otherwise 0
<code>e1 &lt;= e2</code>	Less than or equal	1 if e1 is less than or equal to e2, otherwise 0.

# if-else Construct

- Used to make decisions.
- Execute first sub-clause if condition is true; otherwise execute the second sub-clause.

```
if(condition){  
    statement_true;  
}  
else{  
    statement_false;  
}
```

- In general the statements are compound, but if not then we can write:  
 if (condition) statement\_true;  
 else statement\_false;

# else-if Construct

- Most general case of if statement for choosing from a number of options.

```
if(condition1){  
    statement_1;  
}  
else if(condition2){  
    statement_2;  
}  
else if(condition3){  
    statement_3;  
}  
else{  
    statement_last;  
}
```

# Conditional Example

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **arg)
{
    if (argc==1){
        printf("Usage: %s s, where s is a string\n",arg[0]);
        exit(1);
    }
    else if (argc >2){
        printf("You have supplied more than one argument");
        printf(" - extra arguments ignored\n");
    }
    printf("The first argument is %s\n",arg[1]);
}
```

← Terminate program here if argc is 1

\$ if1 Hello World

You have supplied more than one argument - extra arguments ignored

The first argument is Hello

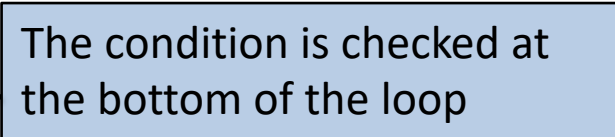
# Loops

- A for loop has the following structure:  

```
for ( variable initialization; condition; variable update ) {  
    Statements to execute while the condition is true  
}
```
- A while loop has the following structure:  

```
while ( condition ) {  
    Statements to execute while the condition is true  
}
```
- A do-while loop ensures at least one pass is made through the loop:  

```
do {  
    Statements  
} while (condition);
```



# For Loop Example

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

```
int main()
```

```
{
```

```
    int i, imax=20;
```


```
    for(i=1;i<=imax;i++){
```

```
        printf("The square of %2d is %3d\n",i,i*i);
```

```
    }
```

```
}
```

i++ means the  
same as  $i = i + 1$



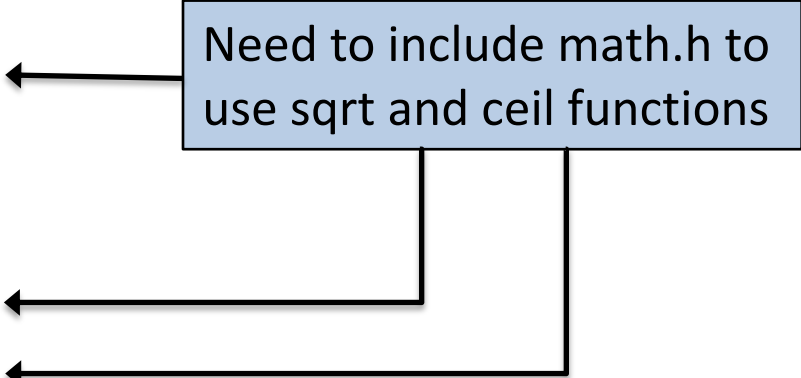
The square of 1 is 1  
The square of 2 is 4  
The square of 3 is 9  
....  
The square of 19 is 361  
The square of 20 is 400




# While Loop Example

```
#include <stdio.h>
#include <math.h>
int main()
{
    int i=1234327;
    float f=sqrt(i);
    int j = ceil(f);
    int n=2, prime = 1;
    while(n<=j){
        if(i%n==0){
            prime = 0;
            break;
        }
        n++;
    }
    if(prime) printf("%d is a prime number\n",i);
    else printf("The smallest divisor of %d is %d\n",i,n);
}
```

Need to include math.h to  
use sqrt and ceil functions



$i \% n$  is the remainder when  $i$  is divided  
by  $n$ , and break causes an immediate  
exit from the loop



The smallest divisor of 1234327 is 29

# Do-While Loop Example

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

```
int main()
```

```
{
```

```
    int x=0;
```

```
    do {
```

```
        printf( "Hello world!\n" );
```

printf will execute once  
even though x=0.

```
    } while ( x != 0 );
```

```
}
```

Hello world!

# #define and #ifdef

```
#include <stdio.h>
#define PI 3.14159265
int main(void)
{
#ifdef PI
    printf("\nPI defined with value %8.6f\n", PI);
#endif
}
```

- Wherever PI appears as a value (not in a string or #ifdef) the compiler substitutes 3.14159265

PI defined with value 3.141593

# Defining Values when Compiling

```
#include <stdio.h>
int main(void)
{
    #ifdef PI
        printf("\nPI defined with value %8.6f\n", PI);
    #endif
}
```

- Compile as:

```
gcc -DPI=3.142 -o define2 define2.c
```

PI defined with value 3.142000

# Defining a Macro with Parameters

```
#include <stdio.h>
#define MAX(a,b) ((a)>(b) ? (a) : (b))
int main()
{
    int i=10, j=22;
    printf("MAX(%d,%d)=%d\n",i,j,MAX(i,j));
}
```

Note:

condition ? expression1 : expression2

evaluates to *expression1* if *condition* is true,  
and to *expression2* if it is false.

MAX(10,22)=22

# Input with scanf

```
#include<stdio.h>
int main()
{
    int k;
    float v;
    char carray[20];
    printf("Enter an integer: ");
    scanf("%d",&k);
    printf("%d\n",k);
    printf("Enter a float: ");
    scanf("%f",&v);
    printf("%f\n",v);
    printf("Enter a string of less than 20 characters: ");
    scanf("%s",carray);
    printf("%s\n",carray);
}
```

Scanf reads input from standard input  
– usually the console.

Second input to scanf is a pointer.

```
$ scanf1
Enter an integer: 8
8
Enter a float: 3.21
3.210000
Enter a string of less than 20 characters: Hello!
Hello!
```

# File I/O with fscanf

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

int main()
{
    char str[10];
    int k;
    float v;
    FILE *fp;
    fp = fopen ("file.txt", "w+");
    fputs("88 CM3103 123.567", fp);
    rewind(fp);
    fscanf(fp, "%d %s %f", &k, str, &v);
    printf("Read integer %d\n",k);
    printf("Read string %s\n",str);
    printf("Read float %f\n",v);
    fclose(fp);
}
```

Contents of file.txt:  
88 CM3103 123.567

Declare a file pointer. Then create and open a file for reading and writing (fopen).

Write a string to the file pointed to by fp (fputs), and then go to the start of the file (rewind).

Read the contents the file pointed to by fp into the variables k, str, and v (fscanf).

Read integer 88  
Read string CM3103  
Read float 123.567001

# Modes for fscanf

Mode	Description
"r"	Open a file for reading. The file must exist.
"w"	Create an empty file for writing. If a file with the same name already exists its content is erased and the file is considered as a new empty file.
"a"	Append to a file. Writing operations append data at the end of the file. The file is created if it does not exist.
"r+"	Open a file for update both reading and writing. The file must exist.
"w+"	Create an empty file for both reading and writing.
"a+"	Open a file for reading and appending.



# Using sscanf

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

int main()
{
    char str[]="88 CM3103 123.567";
    int k;
    float v;
    sscanf(str, "%d %s %f", &k, str, &v);
    printf("Extracted integer %d\n",k);
    printf("Extracted string %s\n",str);
    printf("Extracted float %f\n",v);
}
```

Extracted integer 88  
Extracted string CM3103  
Extracted float 123.567001

# Creating a String with sprintf

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

```
int main()
```

```
{
```

```
    char str[20];
```

```
    int k=123;
```

```
    sprintf(str, "Value of k = %d",k);
```

```
    printf("%s\n",str);
```

```
}
```

Value of k = 123

# Structures

- A structure is a group of variables, possibly with different data types, grouped together under one name.

```
struct date {  
    int day;  
    int month;  
    int year;  
    char month_name[4];  
};
```

*date* is a structure tag  
which serves as a  
shorthand for this sort of  
structure

# Defining Structure Variables

- Given the previous definition of *date* we can declare a variable *d* as a structure of type *date*:

```
struct date d;
```

and we can initialize it as follows:

```
struct date d = {4, 7, 1776, "Jul"};
```

- We can refer to a particular member of a structure using the “dot” operator, e.g.,

```
int y = d.year;
```

# Nested Structures

- Structures can contain structures as members:

```
struct person{  
    char first_name[20];  
    char family_name[20];  
    struct date birthdate;  
};
```

# Pointers to Structures

- We can declare a pointer to a structure as follows:

```
struct date *pd;
```

- Given a pointer to a structure we can refer to one of its members using the “->” notation, e.g.,

```
int y = pd->year;
```

which is the same as:

```
int y = (*pd).year;
```

# Self-referential Structures

- Consider a linked list in which each node contains an integer and a link to the next node in the list:

```
struct listnode{  
    int value;  
    struct listnode *next;  
};
```

- It is illegal for a structure to contain an instance of itself, but it is OK for *listnode* to contain a pointer to a *listnode*.

# Unions

- A union is a variable that may hold (at different times) objects of different types and sizes. For example:

```
union u_tag {  
    int ival;  
    float fval;  
    char *pval;  
};
```

- Can access members of a union using “.” and “->”
- Unions can occur within structures and arrays, and vice versa.



# Slightly Incorrect Example of a union

```
#include <stdio.h>
int main(){
    union u_tag {
        int ival;
        float fval;
        char *pval;
    } uval;
    char str[]="Hello";
    uval.ival = 123;
    uval.fval = 4.567;
    uval.pval = str;
    printf("uval.ival = %d (this is nonsense)\n",uval.ival);
    printf("uval.fval = %f (this is nonsense)\n",uval.fval);
    printf("uval.pval = %s (this is correct)\n",uval.pval);
}
```

```
uval.ival = 1429054480 (this is nonsense)
uval.fval = 11932509667328.000000 (this is nonsense)
uval.pval = Hello (this is correct)
```

# Correct Example of a union

```
#include <stdio.h>
int main(){
    union u_tag {
        int ival;
        float fval;
        char *pval;
    } uval;
    char str[]="Hello";
    uval.ival = 123;
    printf("uval.ival = %d\n",uval.ival);
    uval.fval = 4.567;
    printf("uval.fval = %f\n",uval.fval);
    uval.pval = str;
    printf("uval.pval = %s\n",uval.pval);
}
```

```
uval.ival = 123
uval.fval = 4.567000
uval.pval = Hello
```

# Typedef

- *typedef* is a facility for creating new data type names. For example;

```
typedef int LENGTH;
```

makes LENGTH a synonym for int.

- Similarly, the declaration:

```
typedef char *STRING;
```

makes STRING a synonym for char \* so it can be used in declarations such as:

```
STRING str="Hello";
```

# Syntax of typedef

- The word *typedef* comes first.
- The name of the new type keyword appears in the position of a variable name. i.e., at the end.

# More Complex Example

```
typedef struct listnode {  
    int value;  
    struct listnode *next;  
} LISTNODE, *LISTPTR;
```

- This creates two new type keywords called LISTNODE (a structure) and LISTPTR (a pointer to the structure). We could say:

```
LISTNODE root;  
LISTPTR proot = &root;
```