



PRIFYSGOL
BANGOR
UNIVERSITY

Imperative Programming in C

Laboratory 6: Introduction to Pointers

Pointers

The Reference operator (&)

When a variable is declared, the memory needed to store its value is assigned to a specific memory address. The address of a variable can be obtained using the ampersand sign (&), and in C is referred to as the reference operator. You can think of the & symbol as being the address-of the variable. For example:

```
int myVariable = 100;
```

```
myPointer = &myVariable;
```

would assign the address of myVariable to myPointer. By preceding the name of the variable with the reference operator, we are no longer assigning the content of the variable (100 here), but the address of the variable in memory.

A variable that stores the address of another variable in C/C++ is known as a Pointer. Pointers are said to 'point' to the variable by storing their address in memory.

The Dereference operator (*)

While storing an address, a pointer can also be used to access the variable they point to. This is done by preceding the pointer name with the dereference operator (*). You can think of the * symbol as being the value-of the variable being pointed to.

Leading on from the previous example – the following code is dereferencing the myPointer variable and storing the result in newVariable (100 in this case).

```
int newVariable = *myPointer;
```

The reference and dereference pointers have opposite meanings – one gets the memory address of the variable (&), while the other gets the value stored at an address (*).

Declaring pointers

Once dereferenced, the type of the pointer needs to be known – so as with variables, when creating a pointer we need to define what type of data the pointer will point to, be it an integer, float, char or double.

The declaration of a pointer follows this syntax:

```
type* name;
```

Where type is the data type, and name is the pointer variable's name. Example declaration of pointers:

```
int* myIntPtr;
```

```
float *myFloatPointer;
```

```
char * myCharPointer;
```

```
double * myDoublePointer;
```

(notice it doesn't matter about white spaces).

While all of these are pointers to different data types, they will still usually take the same amount of space in memory (as they store a memory address). However, the variable being pointed to will be using different sizes of memory to store its data.

One thing to note is that the * symbol is used here only to say that we are creating a pointer, and shouldn't be confused with the dereferencing operator *. They are two separate things represented using the same symbol.

Pointer example

```
int var1 = 5, var2 = 10;
int *ptr1, *ptr2;

ptr1 = &var1;    //pointer to address of var1
ptr2 = &var2;    //pointer to address of var2

*ptr1 = 20;      //value pointed to by ptr1 = 20
*ptr2 = *ptr1;   //value pointed to by ptr2 = value pointed to by ptr1

ptr1 = ptr2;     //copy address pointed to by ptr1 to ptr2
*ptr1 = 30;      //value pointed to by ptr1 = 30

printf("\nvar1 is %d", var1);
printf("\nvar2 is %d", var2);
```

Try to follow what the above code is doing before running it. Are the results what you predicted? It is a very good idea to get a pen and paper / paint program and try to map out the changing relationships between the two int variables and the two pointers. If you take things line by line, it should be easy to follow.

Pointers and Arrays

An array isn't that different to a pointer, as an array will also point to its first element, and will contain information about the type of data being stored. In actual fact, we can convert an array to a pointer of the same type.

For example the following is completely valid:

```
int myArray [8];
```

```
int *myPointer = myArray;
```

After this, myPointer and myArray would have very similar properties. Pointers and arrays support the same set of operations. The key difference is that we can always change what myPointer is pointing to, but we can't change myArray once it's created – it's always going to point to the block of 8 integers.

As an example of the interchangeability between arrays and pointers, the following two lines mean the same thing:

```
char myString[] = "Pointer";

myString[6] = 'd';           // a [offset of 6] = 'd'
*(myString+6) = 'd';         // pointed by (a+6) = 'd'

printf("\nmyString : %s \n", myString);
```

Similarly, we can iterate through an array of items using either indices or pointers:

```
int myNumbers[3] = {1,2,3};
int i;

for(i=0; i<3; i++)
    printf("%d ", myNumbers[i]);

printf("\n\n");

for(i=0; i<3; i++)
    printf("%d ", *myNumbers+i);
```

Pointers to Pointers

Of course a pointer doesn't have to point to a variable, it can also point to another pointer. In fact we can have as many indirections as we'd like, although this rapidly becomes too esoteric to have any practical use.

To declare a pointer to a pointer, we use two * symbols like so:

```
int* ptr;
```

```
int **pptr;
```

Where the two asterisks indicate that two levels of pointers are involved.

This is a very simple example of using pointers to pointers:

```
int var;
int *ptr;
int **pptr;    //pointer to ptr

var = 100;

/* take the address of var */
ptr = &var;

/* take the address of ptr using address of operator & */
pptr = &ptr;

/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);

return 0;
```

We will be using pointers to pointers in future labs, especially when returning pointers (and therefore array addresses) to and from functions.

Other Pointer resources

Pointers are notorious for being one of the key concepts in C that beginners can find difficult to grasp, despite them actually being deceptively simple. Learning to understand, use and manage pointers is one of the most important parts of programming in C / C++, and while this will mostly come from practice, you might also find the following resources useful at introducing the basic concepts:

[5 minute guide to C pointers](#)

[Pointers dreaded pointers](#)

[Everything you need to know about pointers](#)

[C Plus Plus guide to pointers](#)

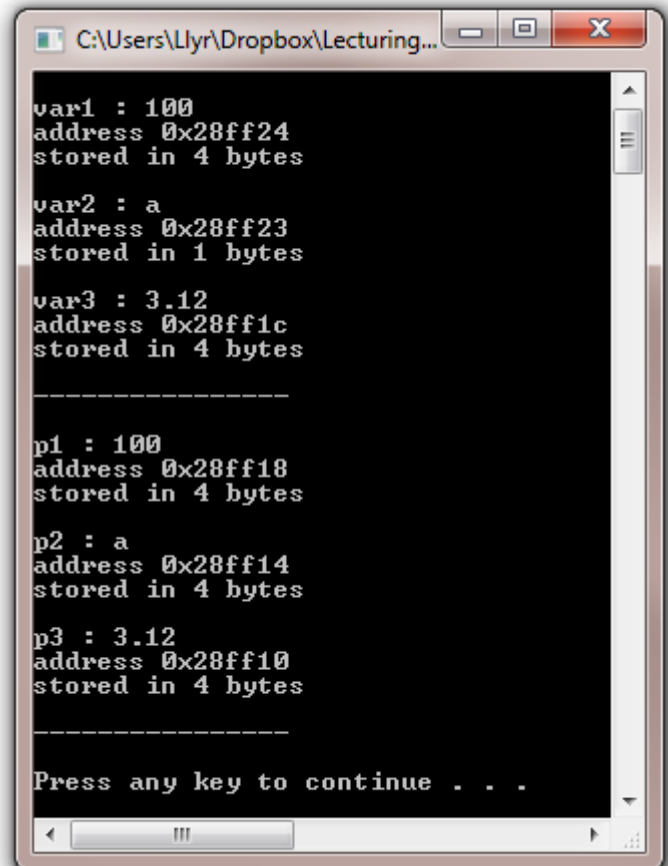
Exercise 1 – Pointer variables

Write a C program that declares and initialises an int, char and float variable. Next declare a pointer to each of these variables.

Your program should then print out the address of, value stored and memory size of each of the six variables.

Use the “0x%x” formatting specifier to print addresses in hexadecimal.

- The “0x” part just tells you that hex notation is being used, the actual address is the rest of the digits
- You’ll need to deference the pointer to get the value of the variable it’s pointing at
- You can use sizeof(variable) to determine the size of a variable in bytes



```
C:\Users\Llyr\Dropbox\Lecturing...  
var1 : 100  
address 0x28ff24  
stored in 4 bytes  
  
var2 : a  
address 0x28ff23  
stored in 1 bytes  
  
var3 : 3.12  
address 0x28ff1c  
stored in 4 bytes  
-----  
p1 : 100  
address 0x28ff18  
stored in 4 bytes  
  
p2 : a  
address 0x28ff14  
stored in 4 bytes  
  
p3 : 3.12  
address 0x28ff10  
stored in 4 bytes  
-----  
Press any key to continue . . .
```

Exercise 2 – Array pointers

The following code snippet prints out all the items in the array myArray.

```
int myArray[] = {10,20,30,40,50,60,70,80};  
  
int i;  
  
for(i=0; i<8; i++)  
    printf("%d ", myArray[i]);
```

Modify this code so that it also prints out the array using pointers (instead of the index [i]).

- You'll need to initialize a pointer variable
- Adding +1 to an array pointer will move to the next item in the array, independent of the size of the data stored (2,4,8 bytes..)

Exercise 3 – Address mystery

Get the following code to run and then modify it to print out the addresses of the variables x and y within the foo1 and foo2 functions. What do you observe? Can you explain this behaviour?

```
#include <stdio.h>

void foo1(int xval)
{
    int x;
    x = xval;

    /* print the address and value of x here */
}

void foo2(int dummy)
{
    int y;

    /* print the address and value of y here */
}

exercise3()
{
    foo1(5);
    foo2(10);
}
```


Exercise 4 – Pointer Reverse

Write a function called **reverse** that accepts a string as input and prints the string reversed. You should use pointers to do this rather than indices.

The function prototype should look like the following:

```
void reverse(char* str);
```

Once your function works you should be able to call it like this:

```
reverse( "lemons" );
```

Hints

- The library <string.h> contains a function called strlen(str) which will return the number of elements stored in the array called str. The size will include the end of string '/0' character

Pointer Reverse Challenge

Write another function which has the following prototype:

```
char* reverse(char* str);
```

Which reverses the string str, and then returns it to the invoker. You should then be able to use your function like this:

```
printf( "\n%s" , reverse( "lemons" ) );
```

- There are many ways of constructing the reversed string
- You can use a temp array to store the reversed string as it is being created
- You can dynamically allocate an array using myArray = malloc(number_of_bytes);

Returning Arrays

We can return an array created within a function using a pointer. For example, in the two functions below we are returning an integer array of size 5.

In the first function we are returning a static array. Being static, we know exactly how big the array will be (it's hard coded as 5). This cannot change at runtime.

In the second function we have created a dynamic array, where we specifically request the machine to allocate $5 * \text{sizeof}(\text{int})$ bytes of storage (using the malloc function). If we didn't do this, the array being returned would be erroneous.

```
int *func_static(void)
{
    static int arr[5] = {0};
    return arr;
}

int *func_dynamic(void)
{
    int *arr = malloc(5 * sizeof(int));
    return arr;
}
```

If we know the size of the array within the function, we can declare it static. However, there are occasions when we don't know how big the array will be until it needs to be created. This requires us to dynamically allocate memory. For the remainder of this lab, we will only ever need to declare static arrays. However in future labs we will look in more detail at dynamic memory allocation.

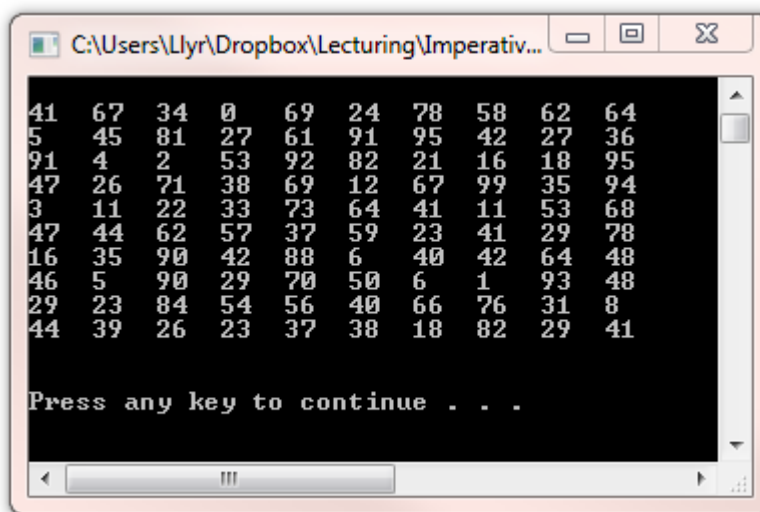
Exercise 5 – Array of randoms

Write a function called `getTenRandomNumbers` that returns an array containing 10 random numbers. The function prototype should look like this:

```
int* getTenRandomNumbers();
```

You should then print out the array of ten numbers on a single line from the invoker.

Extend your program so that the function `getTenRandomNumbers()` is called 10 times, producing an output similar to the following:



Hints

- As we know we are always going to be creating an array of size 10, we can statically declare it inside the `getTenNumbers` function.
- You can use the format specifiers of `printf` to ensure straight, left justified columns of numbers

Assessed Work – Jackpot Dreams

Mad Scientist Roger has invented a cryogenic sleep chamber. He has decided he's going to play the same lottery ticket, every week, until he wins the lottery. He will instruct the computer to put him in deep freeze and only wake him up if and when he wins the lottery jackpot.



Figure 1: Roger's 6 lucky numbers

There is one lottery draw each week. In each lottery draw, 6 numbers are chosen at random from the numbers 1-49. Your task is to write a program to simulate a lottery draw, and compare the results against Roger's lottery ticket each week (52 draws per year) for however many years Roger decides. If 6 numbers are matched, the program should wake him up to tell him the good news! (and what year it is).

If you're unsure how to start, try building your program step by step. Marks will be allocated for completing each stage:

1. Welcome user to the program (0.5 mark)
2. Store Roger's 6 numbers (0.5 mark)
3. Ask how many years to simulate (1 mark)
4. Generate 6 lottery numbers (random between 1-49) (2 marks)
5. Check Roger's numbers against lottery numbers (2 marks)
6. Store number of lottery numbers matched (2 marks)
7. Repeat step 4-6 for X years, or until jackpot is won. Print year and results. (2 marks)

You should keep track of how many times Rogers' ticket matches 0, 1, 2, 3, 4, 5 and 6 lottery numbers each draw.

Your program should feature at least the following function declarations:

```
int* get_otto_draw(); //Returns an array of six random lottery numbers 1-49
```

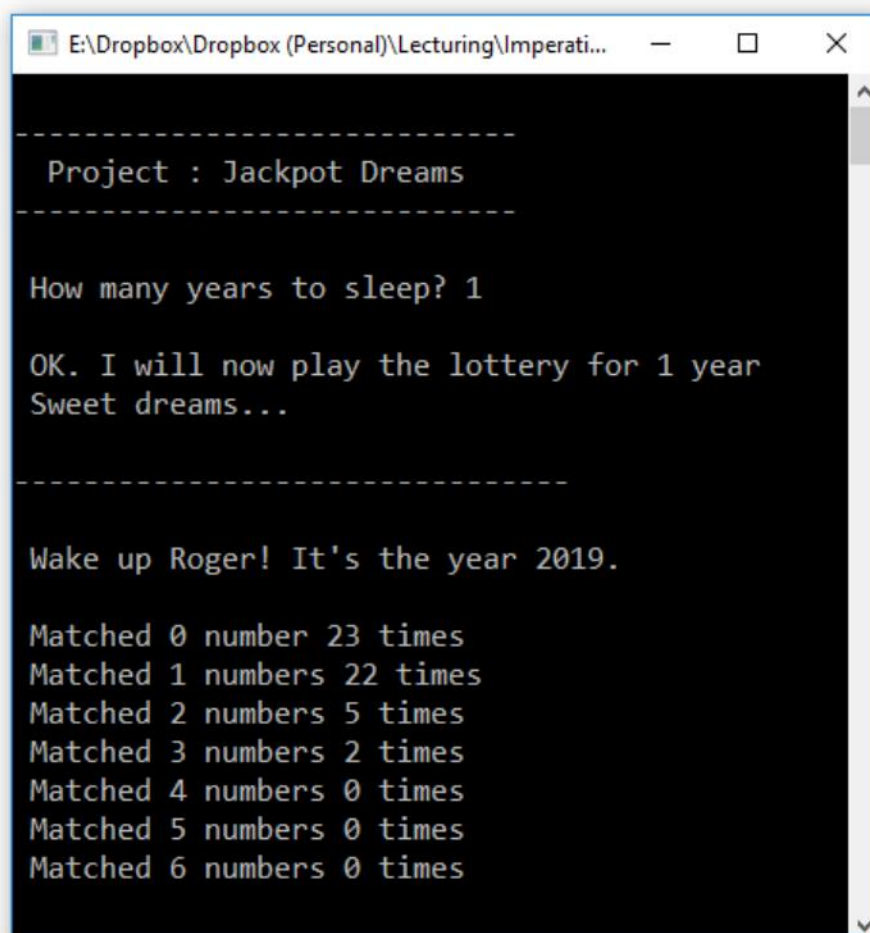
```
void print_array(int* array); //print out the content of an array
```

```
int find_matches(int* array1, int* array2); //returns number of matches between two arrays
```

Hints

- Consider using an array of integers to store the lottery numbers
- Ensure that lottery draws do not contain duplicate numbers
- If a person wins with 6 numbers, they don't win prizes for 3, 4 or 5 numbers so you should only count the highest number matched for every draw
- Consider using dummy lottery values (instead of random ones) to see if your matching algorithm works or not
- You can press CTRL-C to force your program to quit
- You can use the break command to end a loop if 6 numbers are matched
- Make sure your program works by testing it thoroughly
- One year of lottery results should add up to 52 (see screenshot below)
- The odds of matching 6 numbers (1-49) are about 1 in 14 million. So you should win on average every 270,000 years or so.

Example output:



```
E:\Dropbox\Dropbox (Personal)\Lecturing\Imperati...
-----
Project : Jackpot Dreams
-----

How many years to sleep? 1

OK. I will now play the lottery for 1 year
Sweet dreams...

-----

Wake up Roger! It's the year 2019.

Matched 0 number 23 times
Matched 1 numbers 22 times
Matched 2 numbers 5 times
Matched 3 numbers 2 times
Matched 4 numbers 0 times
Matched 5 numbers 0 times
Matched 6 numbers 0 times
```

```
E:\Dropbox\Dropbox (Personal)\Lecturing\Imperative Programming in C\Labs\Virt...
-----
Project : Jackpot Dreams
-----

How many years to sleep? 100000000

OK. I will now play the lottery for 100000000 years
Sweet dreams...

-----

Matched the lottery numbers : 5 42 15 33 11 43
Wake up Roger! It's the year 244064. You've won the lottery!!

Matched 0 number 5381055 times
Matched 1 numbers 5232039 times
Matched 2 numbers 1726195 times
Matched 3 numbers 234049 times
Matched 4 numbers 12869 times
Matched 5 numbers 230 times
Matched 6 numbers 1 times
```

Challenge – A tax on the stupid?

No marks are allocated for this part, but if you'd like to extend your lottery simulator further, give it a try!

Assume Roger pays £2 per lottery ticket.

3 Matches	£25
4 Matches	£100
5 Matches	£1000
6 Matches	£2,000,000

Extend your lottery program so that it also tracks Roger's bank account, assuming it pays by direct debit £2 per lottery draw. He starts with £10,000 in his account. How much debt is he in after 10 years? 100 years? 1000 years? What if he wins a \$10 million jackpot?

Submission

Use **Blackboard** to submit your source code. For this work you only need to submit the .c file that contains your lottery program. Ensure that your code:

- Contains a program header (see Appendix)
- Contains an appropriate level of comments
- Follows a consistent style of indentation
- Follows the usual C programming conventions

The deadline for submission will be published on Blackboard. Late submissions will be penalised in line with School policy.

Marks for this laboratory exercise are awarded for

- Managing input and output
- Meaningful variable names
- Program correctness
- Program testing
- Layout and structure
- Conceptual understanding

When submitting work it is your responsibility to ensure that all work submitted is

- Consistent with stated requirements
- Entirely your own work
- Submitted through Blackboard on time

Please note that there are severe penalties for submitting work which is not your own. If you have used code which you have found on the Internet or from any other source then you must signal that fact with appropriate program comments.

Note also that to obtain a mark you must attend a laboratory session and be prepared to demonstrate your program and answer questions about the coding. Non-attendance at labs will result in your work not being marked.

Appendix

[Online C Programming Resources](#)

[Complete C Reference Library](#)

C Programming IDE's

[Dev-C++](#) (Windows)

[Code::Blocks](#) (Windows, Mac, Linux)

[Visual Studio/C++ Express](#) (Windows)

[Netbeans C/C++](#) (Windows, Mac, Linux)

[Codelite](#) (Windows, Mac, Linux)



PRIFYSGOL
BANGOR
UNIVERSITY