



Coding-bootcamps.com

Introduction to C Programming Language

By Tom Brownlee
from [Coding Bootcamps](https://www.coding-bootcamps.com)



Coding Bootcamps

Session 4

Pointers and Dynamic Allocation

RECAP-Previous Session

3- Fundamental Data Types and Qualifiers

Constants and Strings

Storage Classes

Scope and Block Structure

Scope and Data Hiding

Data Initialization

Examples, as promised last time.

OUTLINE

4- Pointers and Dynamic Allocation

User of Pointers

Pointer and Address Arithmetic

Dynamic Storage Allocation

sizeof Operator

Double Indirection

POINTERS AND DYNAMIC ALLOCATION

Use of Pointers

Pointers are used (in the C language) in a few different ways:

- To create dynamic data structures.
- To pass and handle variable parameters passed to functions.
 - This is called 'passing by reference' as opposed to 'passing by value.'
- To access information stored in arrays. (Especially if you work with links).
- To reduce certain sources of overhead, resulting in faster code.

One way to help understand pointers is to describe them as values and variables that refer to other variables. Suppose you have a variable "int foo;" Memory is allocated for it and that location in memory is referred to by the name "foo." A pointer is a variable whose value refers to some such variable.

On 64 bit architectures pointers take up 8 bytes of memory. On 32 bit architectures, they take up 4 bytes. For large, complex data types it's often much faster to copy one 8-byte value and pass it to a function than it is to pass a whole new copy of that data, and frequently it's more useful too.

POINTERS AND DYNAMIC ALLOCATION

Pointer and Address

To declare a pointer you have to put an `*` in front of its name. A pointer can be **typed** or **untyped**. (A typed pointer points to a particular variable type such as an integer. An untyped pointer points to any data type). See the following example of a declaration of a typed pointer and an untyped pointer:

```
#include<stdio.h>

int main()
{
    int *ptr_A;           /* A typed pointer */
    void *ptr_B;         /* A untyped pointer */
    return 0;
}
```

Put the address of an integer into a “pointer to an integer” by using the `&` operator (address operator) in front of the integer, to get the integer’s address.

Let’s take a look at an example:

```
#include<stdio.h>

int main()
{
    int x;
    int *ptr_p;

    x = 5;
    ptr_p = &x;

    return 0;
}
```

POINTERS AND DYNAMIC ALLOCATION

Pointer and Address

C program to show pointer arithmetic

```
1  #include<stdio.h>
2  #include<conio.h>
3
4  void main() {
5      int int_var = 10, *int_ptr;
6      char char_var = 'A', *char_ptr;
7      float float_val = 4.65, *float_ptr;
8
9      /* Initialize pointers */
10     int_ptr = &int_var;
11     char_ptr = &char_var;
12     float_ptr = &float_val;
13
14     printf("Address of int_var = %u\n", int_ptr);
15     printf("Address of char_var = %u\n", char_ptr);
16     printf("Address of float_var = %u\n\n", float_ptr);
17
18     /* Incrementing pointers */
19     int_ptr++;
20     char_ptr++;
21     float_ptr++;
22     printf("After increment address in int_ptr = %u\n", int_ptr);
23     printf("After increment address in char_ptr = %u\n", char_ptr);
24     printf("After increment address in float_ptr = %u\n\n", float_ptr);
25
26     /* Adding 2 to pointers */
27     int_ptr = int_ptr + 2;
28     char_ptr = char_ptr + 2;
29     float_ptr = float_ptr + 2;
30
31     printf("After addition address in int_ptr = %u\n", int_ptr);
32     printf("After addition address in char_ptr = %u\n", char_ptr);
33     printf("After addition address in float_ptr = %u\n\n", float_ptr);
34
35     getch();
36     return 0;
37 }
```

Output

```
Address of int_var = 2293300
Address of char_var = 2293299
Address of float_var = 2293292
```

```
After increment address in int_ptr = 2293304
After increment address in char_ptr = 2293300
After increment address in float_ptr = 2293296
```

```
After addition address in int_ptr = 2293312
After addition address in char_ptr = 2293302
After addition address in float_ptr = 2293304
```

Pointer Dereferencing

So pointers refer to locations in memory. Fine, how do we get to what's in that spot in memory?

Dereferencing!

Putting a * before the name of a variable when you use it “de-pointer-ifies” or dereferences the value.

Examples back on my scratch sheet.

Stack vs Heap memory

Stack memory

- Allocated by variable declarations
- Memory “frames” are created matching scopes in your program.
- Pushed and popped off a stack structure (last in, first out)
- Managed automatically by C runtime.

Heap memory

- Allocated dynamically at runtime
- Accessed by pointers
- “Borrowed” from the operating system (must be freed/returned manually after you’re done with it)
- Managed by the programmer.

POINTERS AND DYNAMIC ALLOCATION

sizeof Operator

Sizeof is a much used in the C programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type which is usually denoted by size_t. sizeof can be applied to any data-type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes such as Structure, union etc.

POINTERS AND DYNAMIC ALLOCATION

Dynamic Storage Allocation

In C, dynamic memory is allocated from the heap using some standard library functions. The two key dynamic memory functions are `malloc()` and `free()`.

The `malloc()` function takes a single parameter, which is the size of the requested memory area in bytes. It returns a pointer to the allocated memory. If the allocation fails, it returns `NULL`. The prototype for the standard library function is like this:

```
void *malloc(size_t size);
```

The `free()` function takes the pointer returned by `malloc()` and de-allocates the memory. No indication of success or failure is returned. The function prototype is like this:

```
void free(void *pointer);
```

If memory acquired using `malloc()`, `calloc()`, or other such functions is not freed, the memory is lost upon the program's conclusion. This is called a memory leak. Specialized tools like Valgrind exist to detect memory leaks in software.

Example provided on my scratch sheet.

POINTERS AND DYNAMIC ALLOCATION

Double Indirection

Program :

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i =50;
    int **ptr1;
    int *ptr2;
    clrscr();
    ptr2 = &i;
    ptr1 = &ptr2;
    printf("\nThe value of **ptr1 : %d",**ptr1);
    printf("\nThe value of *ptr2  : %d",*ptr2);
    getch();
}
```

Output :

```
The value of **ptr1 : 50
The value of *ptr2  : 50
```

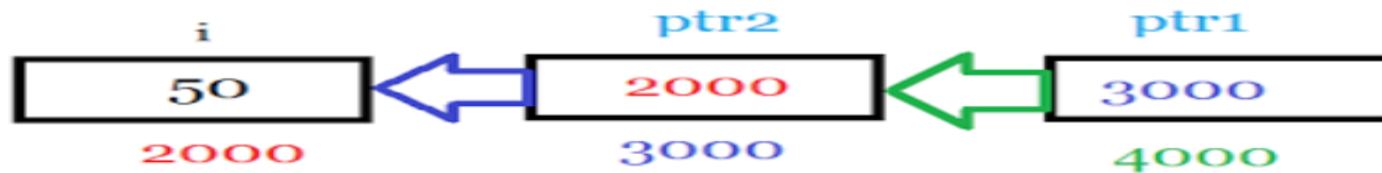
POINTERS AND DYNAMIC ALLOCATION

Double Indirection

Output :

```
The value of **ptr1 : 50  
The value of *ptr2 : 50
```

Explanation :



1. Variable 'i' is initialized to 50.
2. i is stored at Memory Location 2000.
3. Pointer Variable 'ptr2' stores address of variable 'i'.

```
*ptr2 will print [Value Stored at Address 2000 ] i.e 50
```

4. Similarly 'ptr1' is also a pointer variable which stores the address of Pointer variable [i.e ptr2 stores address of integer variable while ptr1 stores address of another pointer variable].
5. So

```
**ptr1 is used to access actual value.
```

SUMMARY

NEXT SESSION

5- Macros

- Functions vs. Inlining
- Purpose of Macros
- Use of Macros
 - Making Code More Readable
 - Auto Adjustment of Compile Time Values
 - Conditional Compilation
 - Making Code Portable
 - Simplifying Complex Access Calculations
- Using Macros to Help Write Portable Programs
- When to Use a Macro instead of a Function
- Using Macros for Debugging

Live private coaching sessions for C

- Private tutoring sessions for software design and engineering- Weekly and monthly plans
- C and C++ programming languages- Private tutoring sessions



Coding-bootcamps.com

Thank You



Coding

Bootcamps