

DPST1091 / CPTG1391

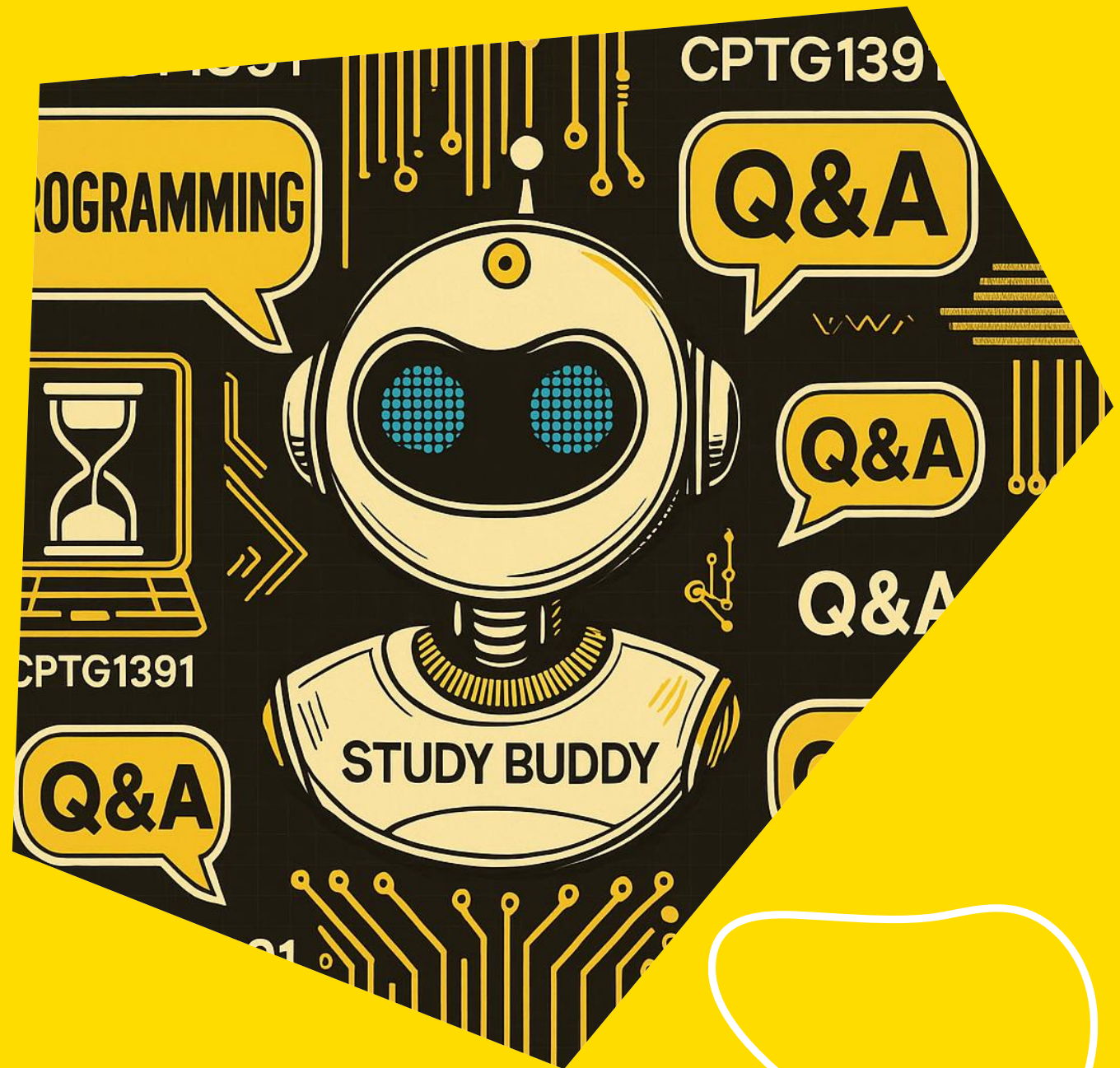
Introduction to Programming

Week 6 – Lecture 1

Lecturer and Course Convener:

Dr Pantea Aria

Pointers



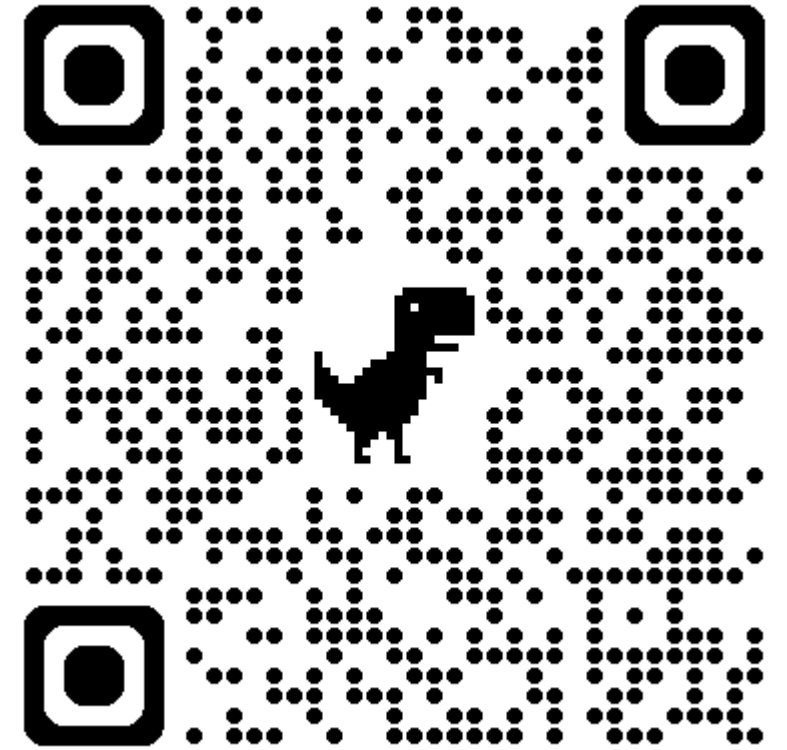
Agenda

- **Today**
 - **Pointers**
 - **Memory, stack**

Link to the Live Lecture Code

<https://cgi.cse.unsw.edu.au/~dp1091/26T2/code/06>

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow
ideal coding style.



Command Line Arguments recap

To use command line arguments, we must change the **main** function header to:

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

argc

- Short for ***argument count***
- Stores **how many** command line arguments were provided
- **Includes the program name itself**

argv

- Short for ***argument vector***
- An array that stores all the command line arguments
- Each argument is stored as a **string** (an array of **char**)

Example

If we run this: `$ ./program 42 apple "Pantea Aria"`

What Does the Program Receive?

- `argc` will be 4
- `argv` will look like this:

Index	Value
<code>argv[0]</code>	<code>"./program"</code>
<code>argv[1]</code>	<code>"42"</code>
<code>argv[2]</code>	<code>"apple"</code>
<code>argv[3]</code>	<code>"Pantea Aria"</code>

`argv`
array

<code>"./program"</code>	<code>"42"</code>	<code>"apple"</code>	<code>"Pantea Aria"</code>
0	1	2	3

Example

```
#include <stdio.h>

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

    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

```
Week5 $ ./command_line_args 42 apple "Pantea Aria" -5
Number of arguments: 5
argv[0] = ./command_line_args
argv[1] = 42
argv[2] = apple
argv[3] = Pantea Aria
argv[4] = -5
Week5 $ █
```

Converting Strings to Integers with *atoi*

Command line arguments are often used in calculations

However, **all command line arguments** are received as **strings**

To work with **numbers**, we must **convert strings to integers**

The `atoi()` Function

- `atoi()` converts a string into an integer
- It is provided in the standard library `<stdlib.h>`

```
int x = atoi("952");
```

- After this line runs, `x` will store the integer value **952**

Using function *atoi* to convert command line args to integers

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

int main(int argc, char *argv[]) {
    int sum = 0;

    for (int i = 1; i < argc; i++) {
        sum = sum + atoi(argv[i]);
    }

    printf("%d is the sum of all command line arguments\n", sum);

    return 0;
}
```

Memory and Addresses

Memory can be thought of as one very **large array made up of bytes**.

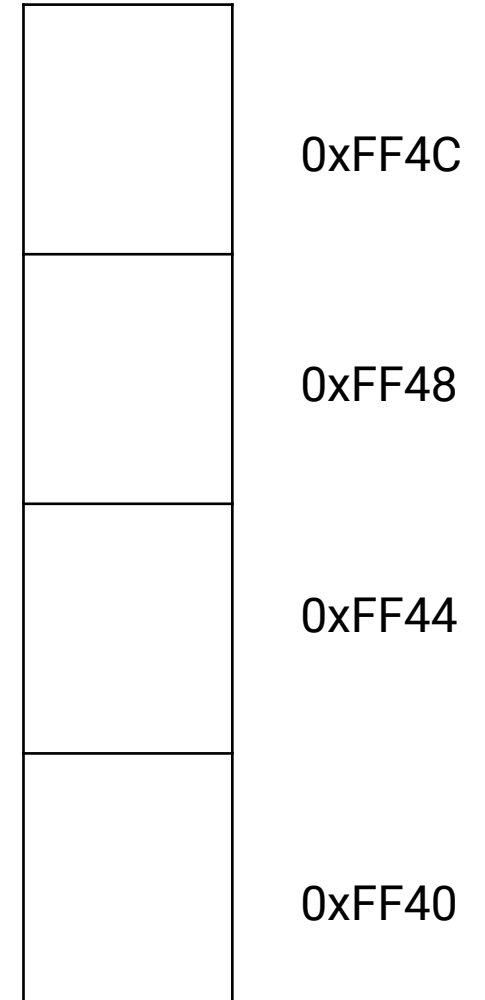
Each **memory address** acts like an **index** that refers to a **specific byte** in that array.

Memory addresses are typically written in **hexadecimal**.

The **prefix 0x** indicates that the value is in hexadecimal.

On modern systems, memory addresses are usually **8 bytes** long and may look like:
0x7ffcaa98655c

Memory Stack



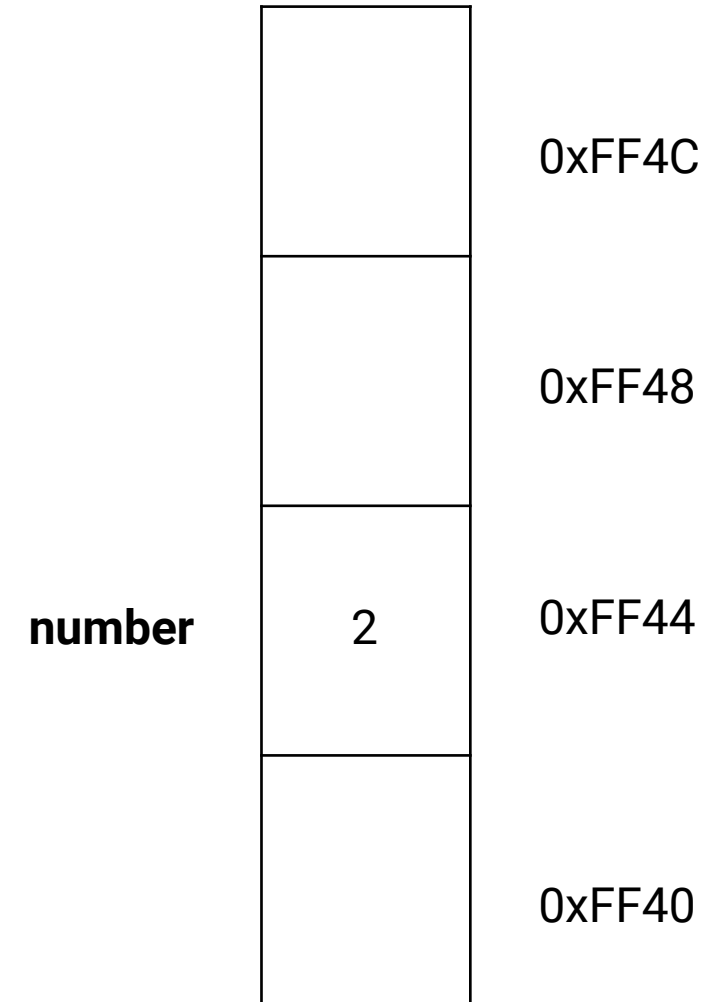
Memory and Variables

```
// Declare a variable of type int  
// Assign the value 2 to it  
int number = 2;
```

During program execution, variables are stored in memory, with each variable located at a specific **memory address**.

number is stored at address 0xFF44

Memory Stack

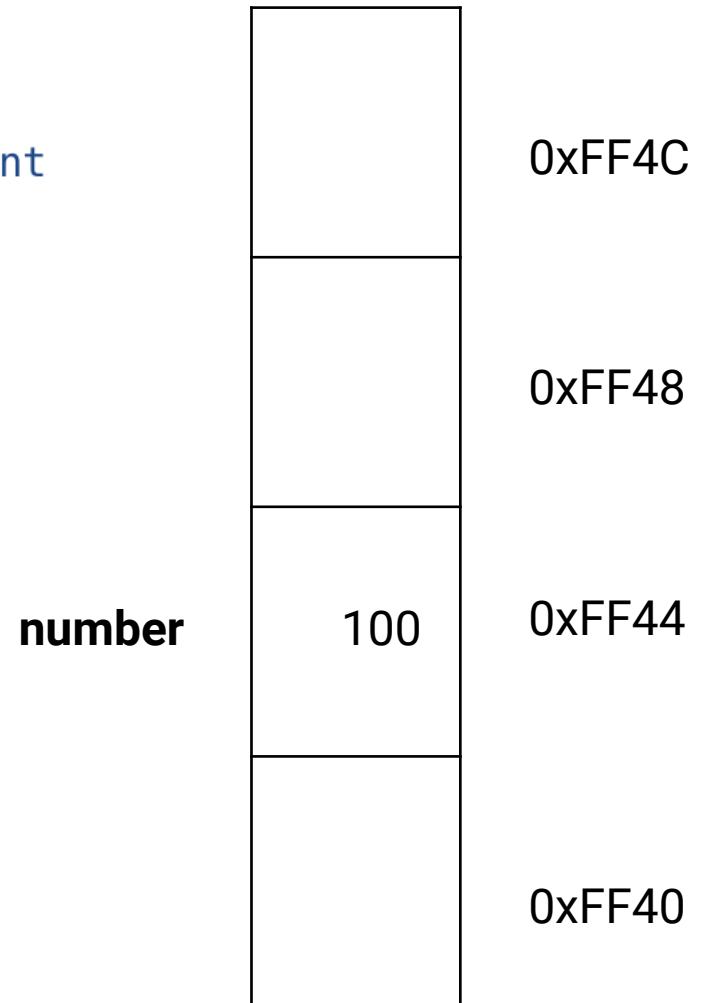


Memory and Variables

```
// Declare a variable of type int
// Assign the value 2 to it
int number = 2;
number = 100;
```

If we change the value of number, the address would be the **same**.

Memory Stack



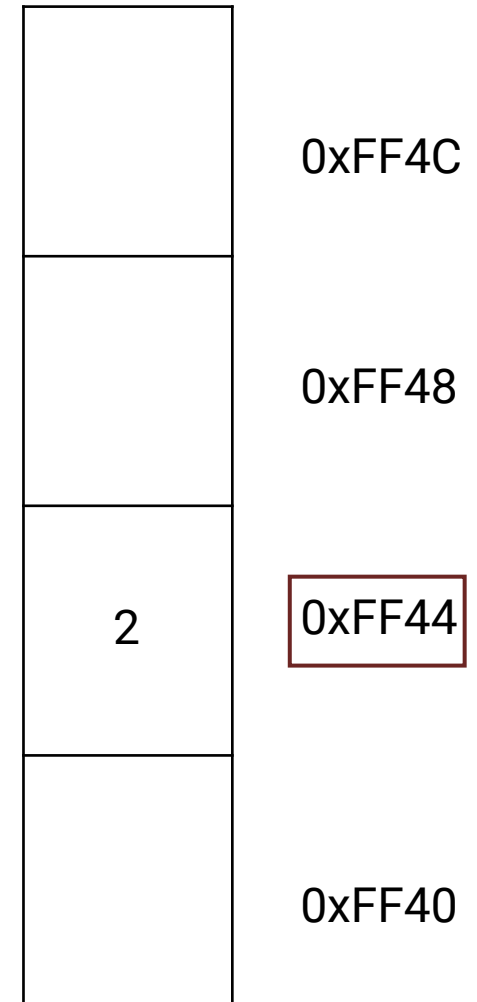
The Address of Operator

&

We can get the address of a variable using the **address of operator &**

```
int number = 2;  
// Print the address of x  
// In this scenario it would print 0xFF44  
printf("%p", &number);
```

Memory Stack



Pointers

Pointers are **variables** that **store the addresses** of other variables in memory.

When **declaring a pointer**, you **specify the data type it points to** and use an asterisk (*) to indicate that it is a pointer.

```
type_pointing_to *name_of_variable;
```

```
int *number_ptr;  
double *real_ptr;  
char *my_ptr;  
struct person *student_ptr;
```

Initialising a Pointer

```
int number = 2;

// number_ptr is declared
// and initialised and
// contains the address
// of int variable number
int *number_ptr = &number;
```

```
double value = 3.14;

// real_ptr declared
double *real_ptr;

// real_ptr is initialised
// and contains the
// address of double
// variable value
real_ptr = &value;
```

Visualise it in the memory

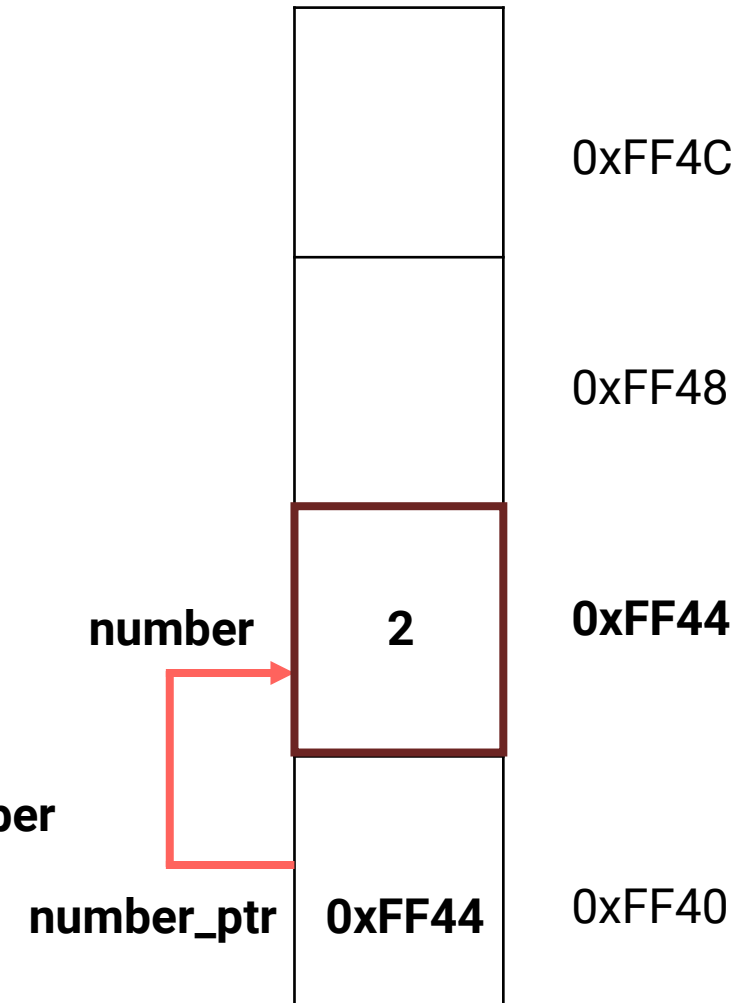
Referencing

```
int number = 2;  
  
// number_ptr is declared  
// and initialised and  
// contains the address  
// of int variable number  
int *number_ptr = &number;
```

So now: **number** is 2
AND
number_ptr is 0xFF44

We can say **number_ptr** **references** **number**
or
number_ptr **points** to **number**

Memory Stack



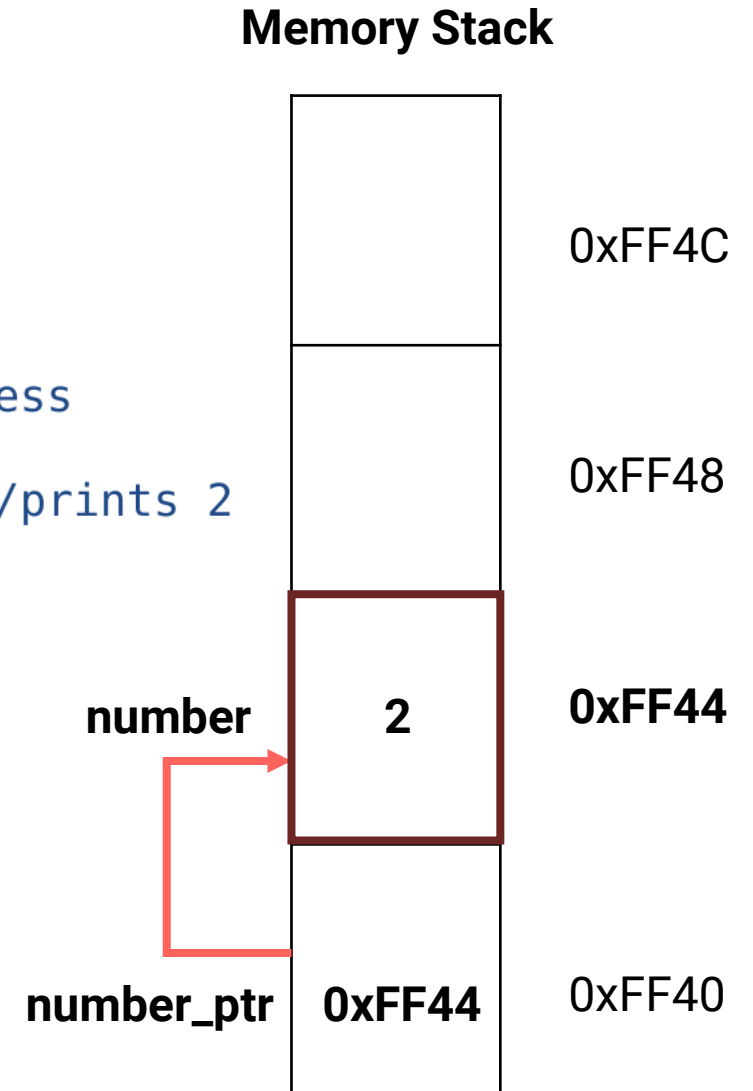
Dereferencing

```
int number = 2;  
int *number_ptr = &number;  
  
// *number_ptr will go to address  
// 0xFF44 and get the value 2  
printf("%d\n", *number_ptr); //prints 2
```

Dereferencing is simply
accessing **the value** at the
address of a pointer

It uses the * symbol again
(which causes confusion)

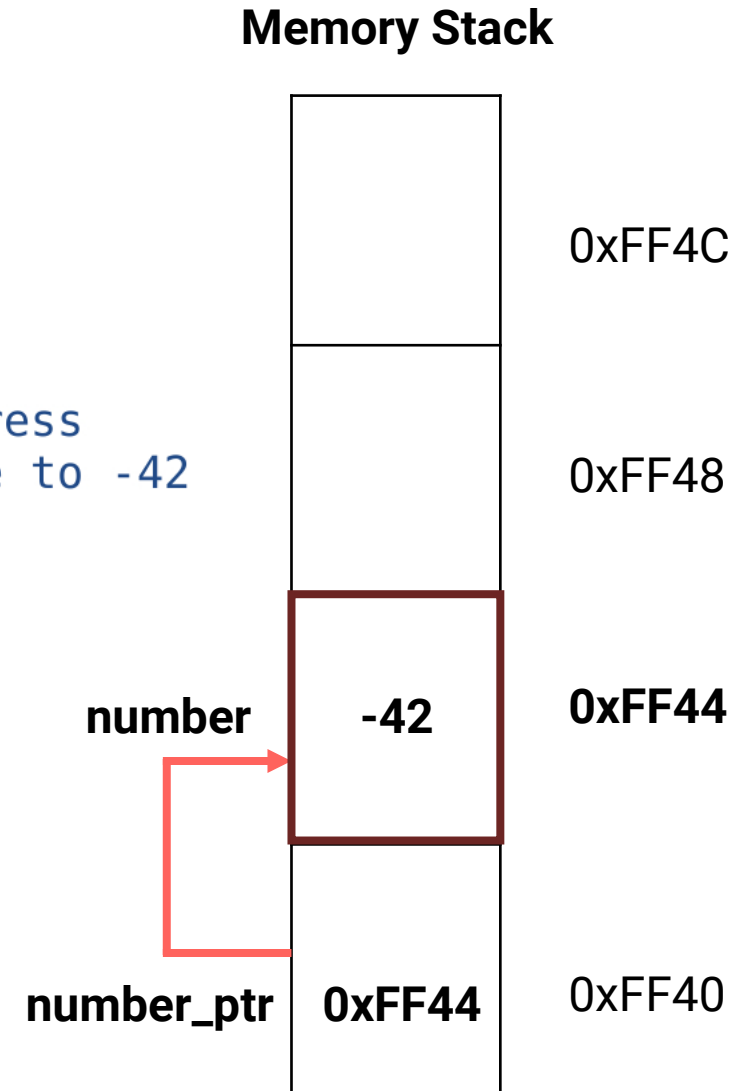
*number_ptr will get the
integer at the address 0xFF44



Indirectly modify a variable using pointers

```
int number = 2;  
int *number_ptr = &number;
```

```
// *number_ptr will go to address  
// 0xFF44 and change the value to -42  
// number is now -42  
*number_ptr = -42;
```



What was pointer again??

Step	Description	Example
Declare a pointer	Use * to declare a pointer and specify the type of value it will point to	int *number_ptr;
Initialise the pointer	Assign the address of a variable to the pointer using the & operator	number_ptr = &x;
Declare and initialise	Declare the pointer and assign it an address in a single statement	int *number_ptr = &x;
Dereference the pointer	Use * to access the value stored at the address the pointer points to	*number_ptr

```
int score = 42;
```

```
// Declare a pointer  
int *score_ptr;
```

```
// Initialise pointer  
score_ptr = &score;
```

```
// Dereference pointer to get  
// 42, then add 1 so result is 43  
int next_score = *score_ptr + 1;
```

Demo

- `pointer_intro.c`
- `play_with_pointers.c`
- `pointer_array.c`

So, what is the point of pointers???

What is the output of this program?

```
#include <stdio.h>

void increment(int x, int y);

int main(void) {
    int x = 3;
    int y = 7;

    printf("Before increment: %d %d\n", x, y);

    increment(x, y);

    printf("After increment:  %d %d\n", x, y);

    return 0;
}

void increment(int x, int y) {
    x = x + 1;
    y = y + 1;

    printf("Inside increment: %d %d\n", x, y);
}
```

And what is the output of this program?

```
#include <stdio.h>

void swap(int x, int y);

int main(void) {
    int x = 3;
    int y = 7;

    printf("Before swap: %d %d\n", x, y);

    swap(x, y);

    printf("After swap:  %d %d\n", x, y);

    return 0;
}

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;

    printf("Inside swap: %d %d\n", x, y);
}
```

Stack Memory



Stack memory stores information needed for your program to run, particularly details about **function calls**.

Each time a function is called, a new block of data is **pushed onto the stack**, including: the **function's local variables** and information about **where the program should return** once the function finishes

When the function (or code block { }) finishes executing, its data, including variables, is **automatically removed from the stack**.

stack

main()

int x =

int y =

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

stack

main()

int x = 3

int y =

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

stack

main()

int x = 3

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);  
    increment(x, y);  
    printf("After increment:  %d %d\n", x, y);  
    return 0;  
}
```

```
void increment(int x, int y) {  
    x = x + 1;  
    y = y + 1;  
    printf("Inside increment: %d %d\n", x, y);  
}
```

stack

main()

int x = 3

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

stack

main()

int x = 3

int y = 7

increment()

int x = 3

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

stack

main()

int x = 3

int y = 7

increment()

int x = 4

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

stack

main()

int x = 3

int y = 7

increment()

int x = 4

int y = 8

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

stack

main()

int x = 3

int y = 7

increment()

int x = 4

int y = 8

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

Output: Inside increment: 4 8

stack

main()

int x = 3

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

Output: Inside increment: 4 8

stack

main()

int x = 3

int y = 7

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

```
void increment(int x, int y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(x, y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y + 1;
```

```
    printf("Inside increment: %d %d\n", x, y);
```

```
}
```

Output: Before increment: 3 7

Output: Inside increment: 4 8

Output: After increment: 3 7

stack

```
#include <stdio.h>

void increment(int x, int y);

int main(void) {
    int x = 3;
    int y = 7;

    printf("Before increment: %d %d\n", x, y);

    increment(x, y);

    printf("After increment:  %d %d\n", x, y);

    return 0;
}

void increment(int x, int y) {
    x = x + 1;
    y = y + 1;

    printf("Inside increment: %d %d\n", x, y);
}
```

Functions And call by value

Variables and data are **passed by value** into functions (note: **arrays** are a special case we will discuss separately)

The function gets **passed copies** of the values

We **can't change the original values** from inside the function

The modified **copies don't even exist** once the **function ends**

**So, can
we do
anything
about it?**

**Yes!
Call by
Reference**

Demo

→ `pointer_square.c`

What did we learn today?

Topic	Summary	Example
Command Line Arguments	Allow values to be passed into a program when it runs. <code>argc</code> stores the number of arguments, <code>argv</code> stores the arguments as strings.	<pre>int main(int argc, char *argv[])</pre>
argc	Argument count (includes program name).	<pre>./program 42 apple → argc = 3</pre>
argv	Array of strings containing command line arguments.	<pre>argv[1] = "42"</pre>
atoi()	Converts a string into an integer. Requires <code><stdlib.h></code> .	<pre>int x = atoi("952"); → x = 952</pre>
Memory Address	The location where a variable is stored. The address does not change when the variable value changes.	<pre>number = 2; then number = 100; same address</pre>
Address-of Operator (&)	Returns the memory address of a variable.	<pre>&number</pre>
Pointer	A variable that stores the address of another variable.	<pre>int *number_ptr;</pre>
Pointer Declaration	Create a pointer using <code>*</code> .	<pre>int *ptr;</pre>
Pointer Initialisation	Store the address of a variable in a pointer.	<pre>int *ptr = &number;</pre>
Referencing	A pointer is said to "point to" a variable when it stores that variable's address.	<pre>ptr = &number;</pre>
Dereferencing (*)	Access the value stored at the address held by a pointer.	<pre>*ptr returns the value of number</pre>
Modify via Pointer	Changing <code>*ptr</code> changes the original variable.	<pre>*ptr = -42;</pre>
Stack Memory	Stores local variables and function information during program execution.	Variables in <code>main()</code> are stored on the stack.
Function Call	Each function call creates a new stack frame.	Calling <code>increment()</code> creates a new frame.
Function End	When a function finishes, its stack frame is removed.	Local variables disappear.
Call by Value	Functions receive copies of variables. Changes inside the function do not affect originals.	<pre>increment(x, y);</pre> leaves original <code>x</code> and <code>y</code> unchanged.
Why Pointers Matter	Pointers allow functions to access and modify original variables (call by reference).	Used later for swapping values and linked lists.



Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me



UNSW
College

Dr Pantea Aria

Pointers and Memory

Voice of the Student



[Voice of the Student – Fill out form](#)

See you soon ...