GMIT
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Advanced Procedural Programming

Lecture 2: Dynamic Memory Allocation

Gemma O'Callaghan

Overview

- Why Pointers?
- What are pointers?
- Pointer Arithmetic
- Pointers and Functions
- Pointers to Pointers
- Pointers and Arrays

Why Pointers?

- A way of managing memory.
- Pointers solve two common software problems.
  - First, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better.
  - Second, pointers enable complex "linked" data structures like linked lists and binary trees.

What are pointers?

- Pointers are basically the same as any other variable.
- However, what is different about them is that instead of containing actual data, they contain a pointer to the memory location where information can be found.

Pointer Creation

- A pointer is a type of variable that is declared in the code.
  - type *name; int *mypointer
- As when you declare any variable, the *type* identifies the pointer as a char, int, float, and so on.
- The *name* is the pointer variable's name, which must be unique, just like any other variable name.
- The asterisk identifies the variable as a pointer, not a regular variable.
- A pointer must always be of the same type as the variable it's pointing at.

Pointer Initialisation

- A pointer must be initialised before it's used.
- To initialise a pointer, you must set its value, just like any other variable.
- The big difference is that a pointer is initialised to the memory location.
- That location isn't a random spot in memory, but rather the address of another variable within the program.
- For example:

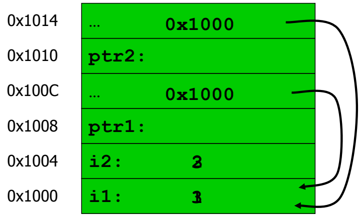
```
int number;
int *mypointer;
mypointer = &number;
```

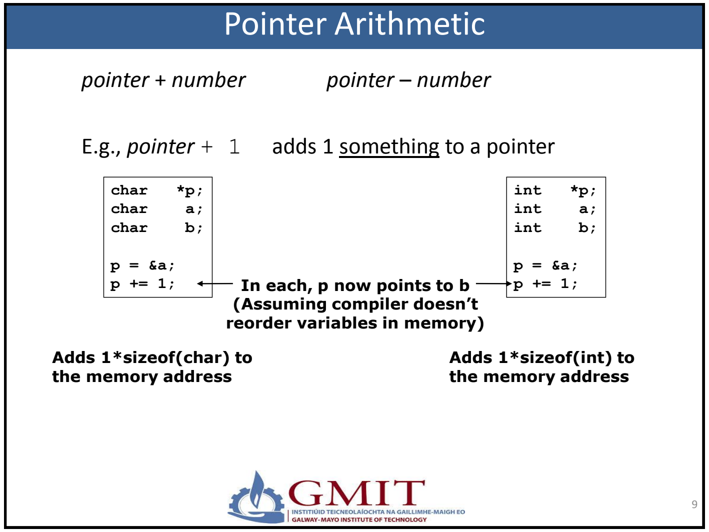
Pointer Dereference

- The indirection operator is used to access the value of a variable, whose address is stored in a pointer.
- For example, **mypointer* means the value of the variable at the address stored in the pointer variable *mypointer* .
- With an asterisk, the pointer references the value at that memory location.

Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```





Pointer Arithmetic

pointer + number *pointer – number*

E.g., *pointer* + 1 adds 1 something to a pointer

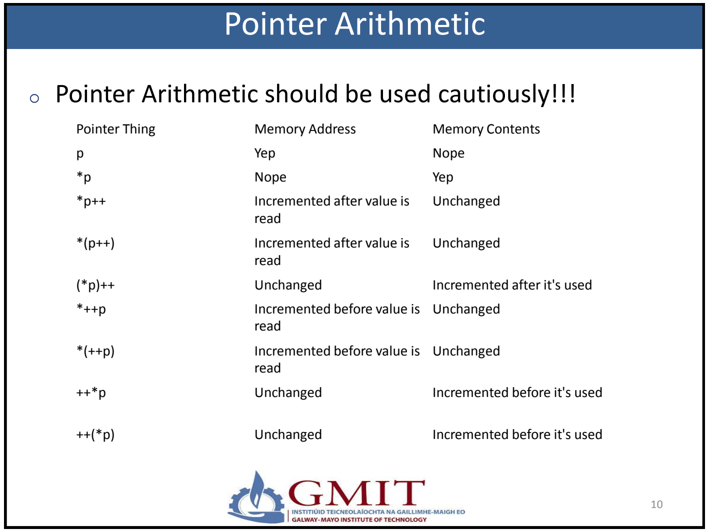
```
char    *p;
char     a;
char     b;

p = &a;
p += 1;
```

```
int    *p;
int     a;
int     b;

p = &a;
p += 1;
```

In each, p now points to b (Assuming compiler doesn't reorder variables in memory)

Adds 1*sizeof(char) to the memory address

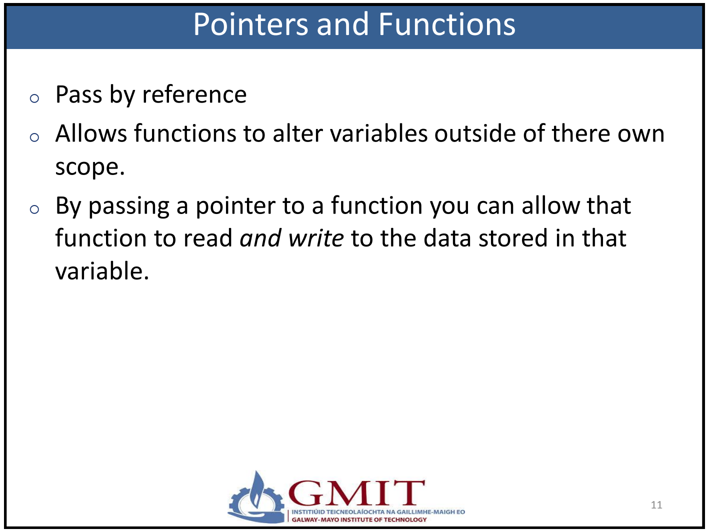
Adds 1*sizeof(int) to the memory address



Pointer Arithmetic

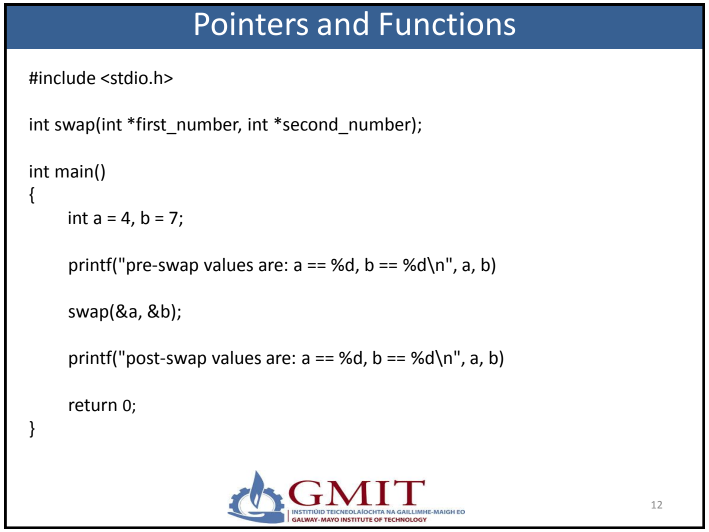
- Pointer Arithmetic should be used cautiously!!!

Pointer Thing	Memory Address	Memory Contents
p	Yep	Nope
*p	Nope	Yep
*p++	Incremented after value is read	Unchanged
*(p++)	Incremented after value is read	Unchanged
(*p)++	Unchanged	Incremented after it's used
*++p	Incremented before value is read	Unchanged
*(++p)	Incremented before value is read	Unchanged
++*p	Unchanged	Incremented before it's used
++(*p)	Unchanged	Incremented before it's used



Pointers and Functions

- Pass by reference
- Allows functions to alter variables outside of there own scope.
- By passing a pointer to a function you can allow that function to read *and write* to the data stored in that variable.



Pointers and Functions

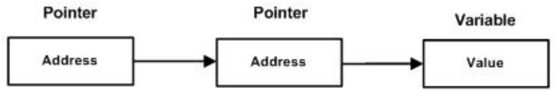
```
#include <stdio.h>

int swap(int *first_number, int *second_number);

int main()
{
      int a = 4, b = 7;

      printf("pre-swap values are: a == %d, b == %d\n", a, b)

      swap(&a, &b);

      printf("post-swap values are: a == %d, b == %d\n", a, b)

      return 0;
}
```

Pointers and Functions

```
int swap(int *first_number, int *second_number)
{
        int temp;

        /* temp = "what is pointed to by" first_number; etc... */
        temp = *first_number;
        *first_number = *second_number;
        *second_number = temp;

        return 0;
}
```

Pointers to Pointers

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.
- Normally, a pointer contains the address of a variable.
- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value.



Pointers to Pointers

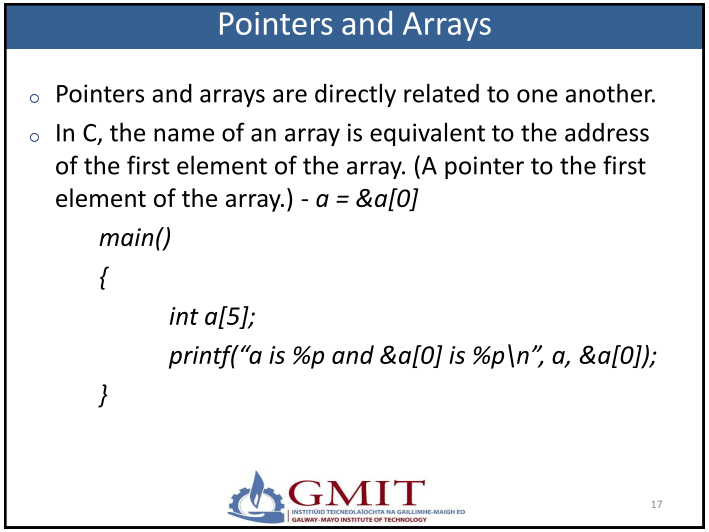
- A variable that is a pointer to a pointer must be declared as that.
- This is done by placing an additional asterisk in front of its name.
- For example, following is the declaration to declare a pointer to a pointer of type int:

  int **mypointer;

- When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice.

Pointers to Pointers

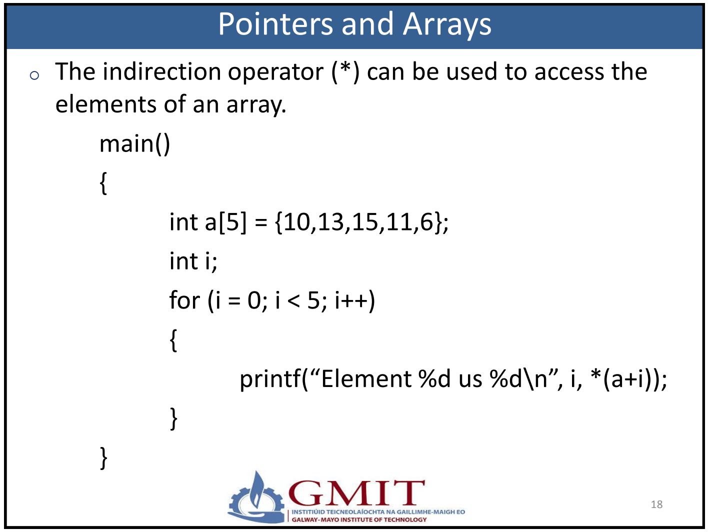
```
#include <stdio.h>

int main ()
{
  int  var;
  int  *ptr;
  int  **pptr;

  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 at *ptr = %d\n", *ptr );
  printf("Value at **pptr = %d\n", **pptr);

  return 0;
}
```



Pointers and Arrays

- Pointers and arrays are directly related to one another.
- In C, the name of an array is equivalent to the address of the first element of the array. (A pointer to the first element of the array.) - *a = &a[0]*

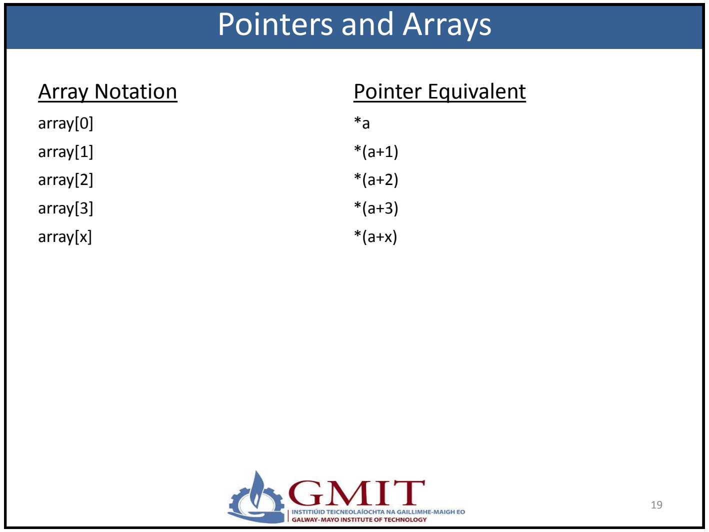
```
main()
{
        int a[5];
        printf(“a is %p and &a[0] is %p\n”, a, &a[0]);
}
```



Pointers and Arrays

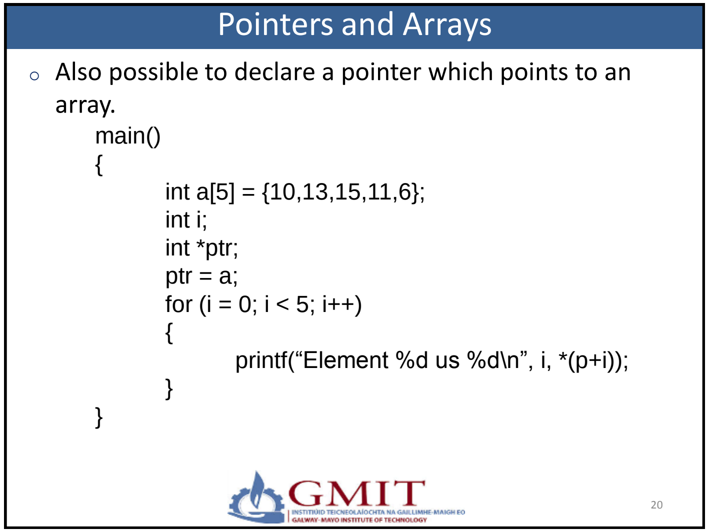
- The indirection operator (*) can be used to access the elements of an array.

```
main()
{
        int a[5] = {10,13,15,11,6};
        int i;
        for (i = 0; i < 5; i++)
        {
                printf(“Element %d us %d\n”, i, *(a+i));
        }
}
```



Pointers and Arrays

Array Notation	Pointer Equivalent
array[0]	*a
array[1]	*(a+1)
array[2]	*(a+2)
array[3]	*(a+3)
array[x]	*(a+x)



Pointers and Arrays

- Also possible to declare a pointer which points to an array.

```
main()
{
        int a[5] = {10,13,15,11,6};
        int i;
        int *ptr;
        ptr = a;
        for (i = 0; i < 5; i++)
        {
                printf(“Element %d us %d\n”, i, *(p+i));
        }
}
```