LECTURE 4:

POINTERS

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)
Beijing Dublin International College

THINKING ABOUT MEMORY

Thinking about memory

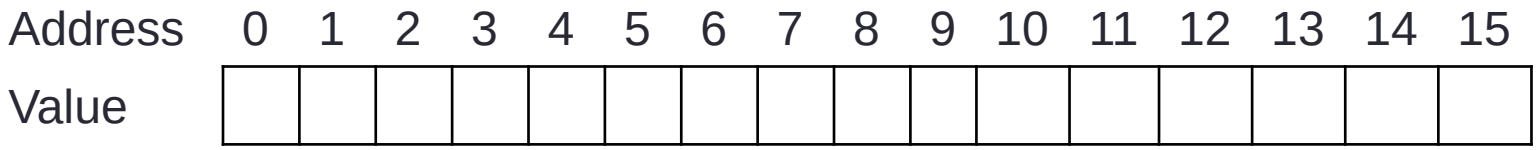
- Whenever we create a variable, we are given some space in memory that we can store some data in.
- As we learned last week, the amount of memory we are given will depend on the data type we use.
- Every variable has a name, which we can use to refer to it:
  - `int x = 7;`
- Whenever we use the variable's name in an expression, we mean "the **value** of that variable":
  - `printf( "%d", x ); // prints 7`

Thinking about memory

- When we imagine memory, it is useful to think of a giant array.
- This array has many *memory cells* that can store some data.
- Every memory cell has an *address* that we can use to refer to it.
  - Very similar to an array index.

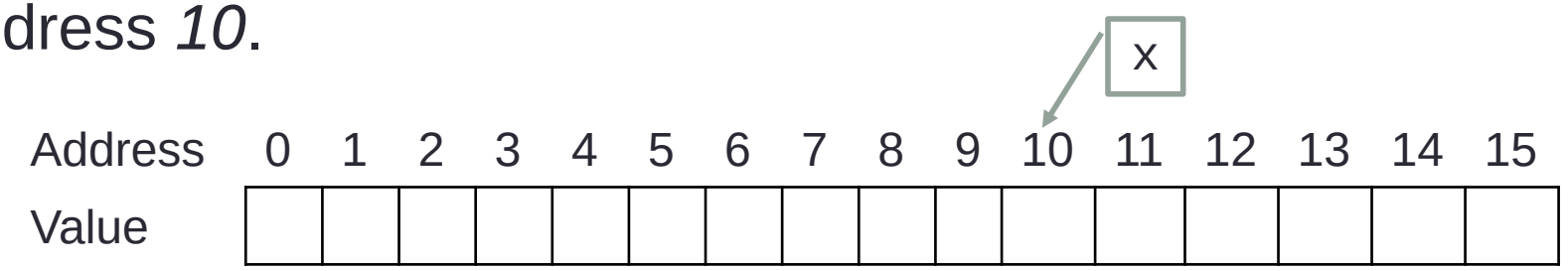
Thinking about memory

- Let us forget about C programming for a moment, and imagine a very simple computer with a very small amount of memory.
- This computer has 16 memory cells (with addresses 0 to 15).
- Each memory cell can store an integer value between 0 and 15 (inclusive).
- Here is a simple diagram:



Thinking about memory

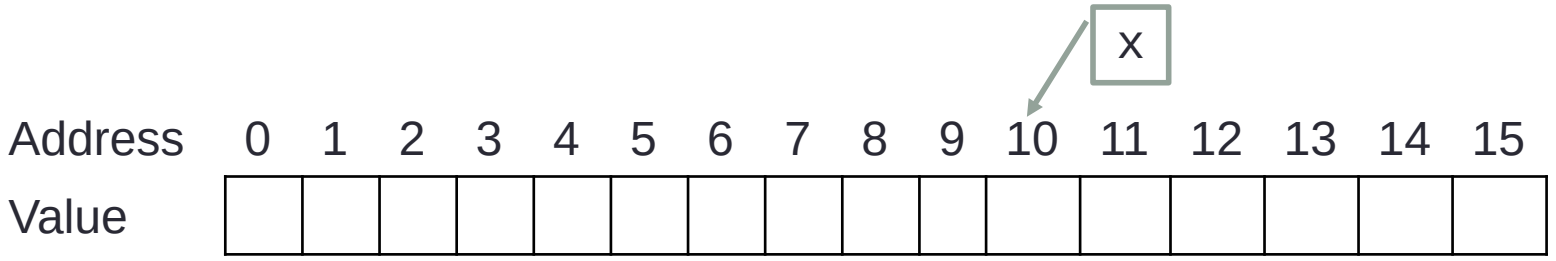
- If we create a variable (e.g. x), the CPU will find a memory cell that is available and associate this variable (x) with the address of the memory cell.
- Now whenever we use x in our program, we are referring to that memory cell.
- In our example, let us assume that x refers to memory address *10*.



Direct vs. Indirect Addressing

- We now have two ways to refer to the same memory cell:
  1. **Using its name (x): direct addressing.**
  2. **Using its address (10): indirect addressing.**

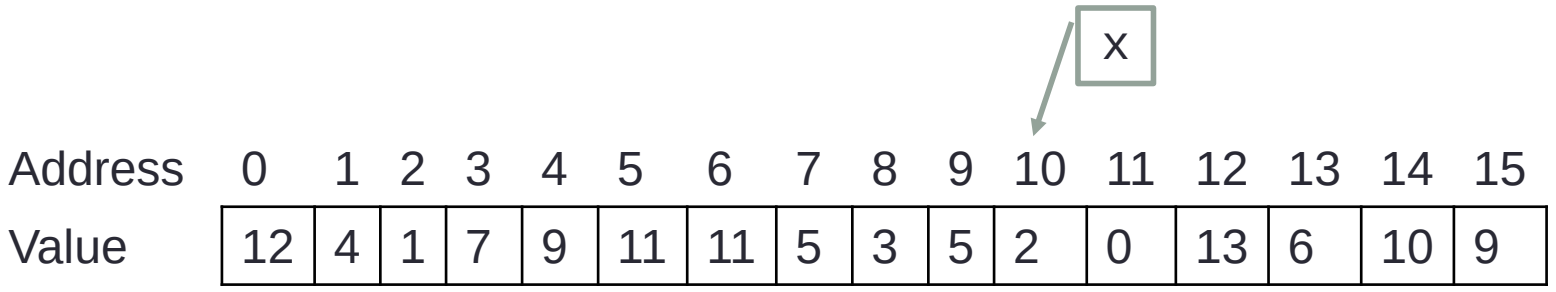
- It's a bit like when I fly on an airplane. You can refer to me in two ways:
  1. Using my name (Brett): direct addressing.
  2. Using my seat number (the person in seat 7C): indirect addressing.



Direct vs. Indirect Addressing

- If we know the name of a variable, we can get its address using the **&** operator (pronounced “ampersand”).

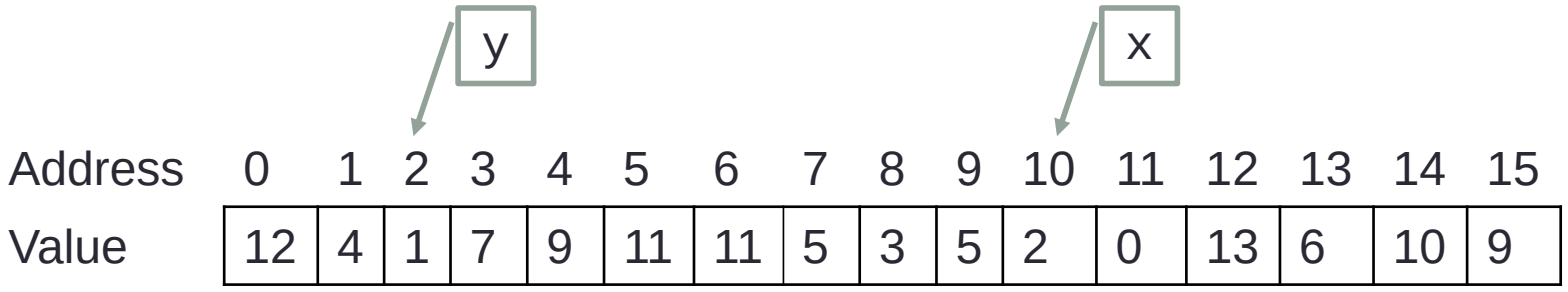
- In the example below (after filling in some values):
  - **x is 2: the value stored in the variable.**
  - **&x is 10: the address that holds the value of the variable.**



Direct vs. Indirect Addressing

- Suppose we have another variable called y that is associated with memory cell 2:
  - What is y?
  - What is &y?

- Answer:
  - y is 1: the value stored in the variable.
  - &y is 2: the address that holds the value of the variable



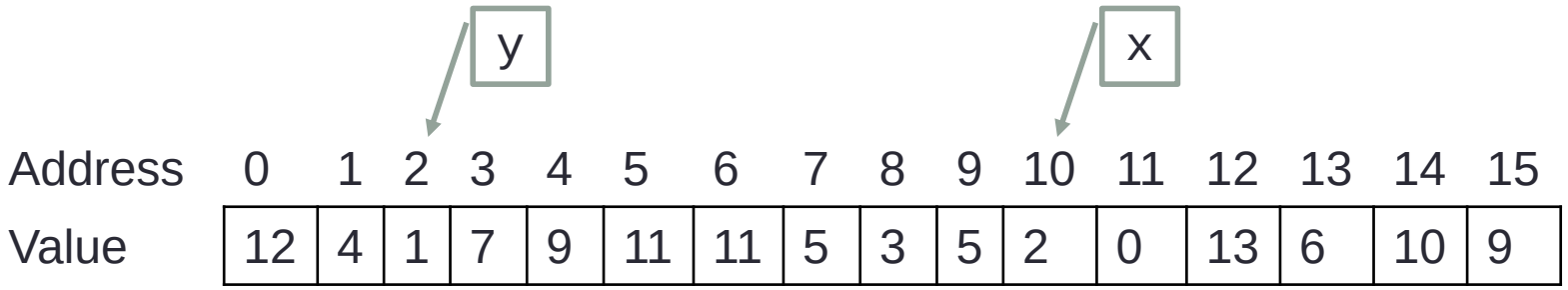
Using addresses

- If we know the address of a memory cell, we can use the * operator (pronounced “asterisk”, or “star”) to find the value stored there.
- *7 means “the value stored at address *7*”. This evaluates to *5*.
- What are the values of *2 and *13?
  - *2 is 1 (the value stored at address 2)
  - *13 is 6 (the value stored at address 13)

Using addresses

- What is the value of *x?
- As we stated earlier, when we use x in an expression, we mean “the value stored in x”.
- Therefore, *x is the same as *2.
  - *x is 1 (the value stored at address 2).

This can get a little confusing!



Pointer Variables

- In that example, we were using x to store the address of another memory cell.

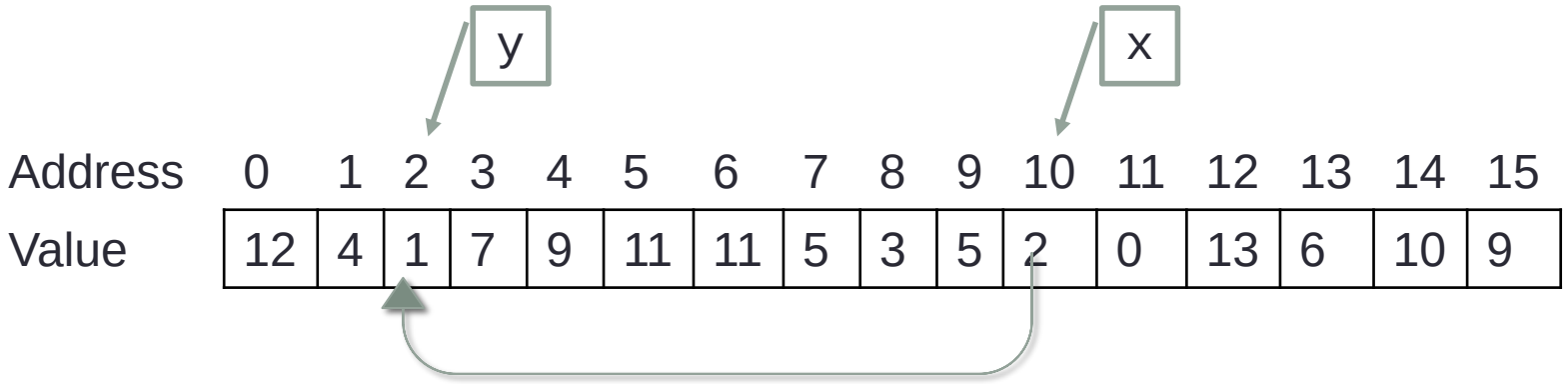
- This is called a *pointer variable*.

- A ***pointer variable*** is a variable that stores a memory address.

Pointer Variables

- When we imagine the memory of a computer using a diagram, it can be useful to draw pointer variables as arrows, so that it is clear what they are.

- The data stored in x is still a number (2), but it is now clearer from the diagram that it is a pointer variable that refers to memory cell 2.



POINTERS IN C

Memory in Real Computers

- In a real computer, the memory addresses will be much bigger numbers than in our simple example.
- Also, not every piece of data will be the same size: it will depend on the data type.
- Generally, memory addresses are written as hexadecimal numbers (e.g. `0x7fff5662ca9c`)
  - However we can use pointers without really needing that detail

Getting the address of a variable

- Remember, we can get the address of a variable using the & operator.

```
int main() {
    int a = 6;

    // print value stored in a
    printf( "%d\n", a );

    // print the address of a
    printf( "%p\n", &a );
}
```

- We can use %p within a `printf` call to print a memory address.

file: address.c

Pointer Variables

- Pointers are variables that contain **memory addresses**.
- A pointer can be assigned to point to **different variables** during the lifetime of the program.
- When we declare a pointer, we must state the type of data it points to.
  - We also use an asterisk (*) before the variable name to show that it is a pointer variable.

Pointer Variable

```
int a = 6;
int *p; //here, the * means we are declaring a pointer
        //variable
p = &a;
```

- In the above example, a is a normal `int` variable.
- `int *p;` shows that p is a pointer variable that will contain the address of an `int`.
  - We can think of this as saying "`*p` is an int type".
- We then store the address of a in that variable.

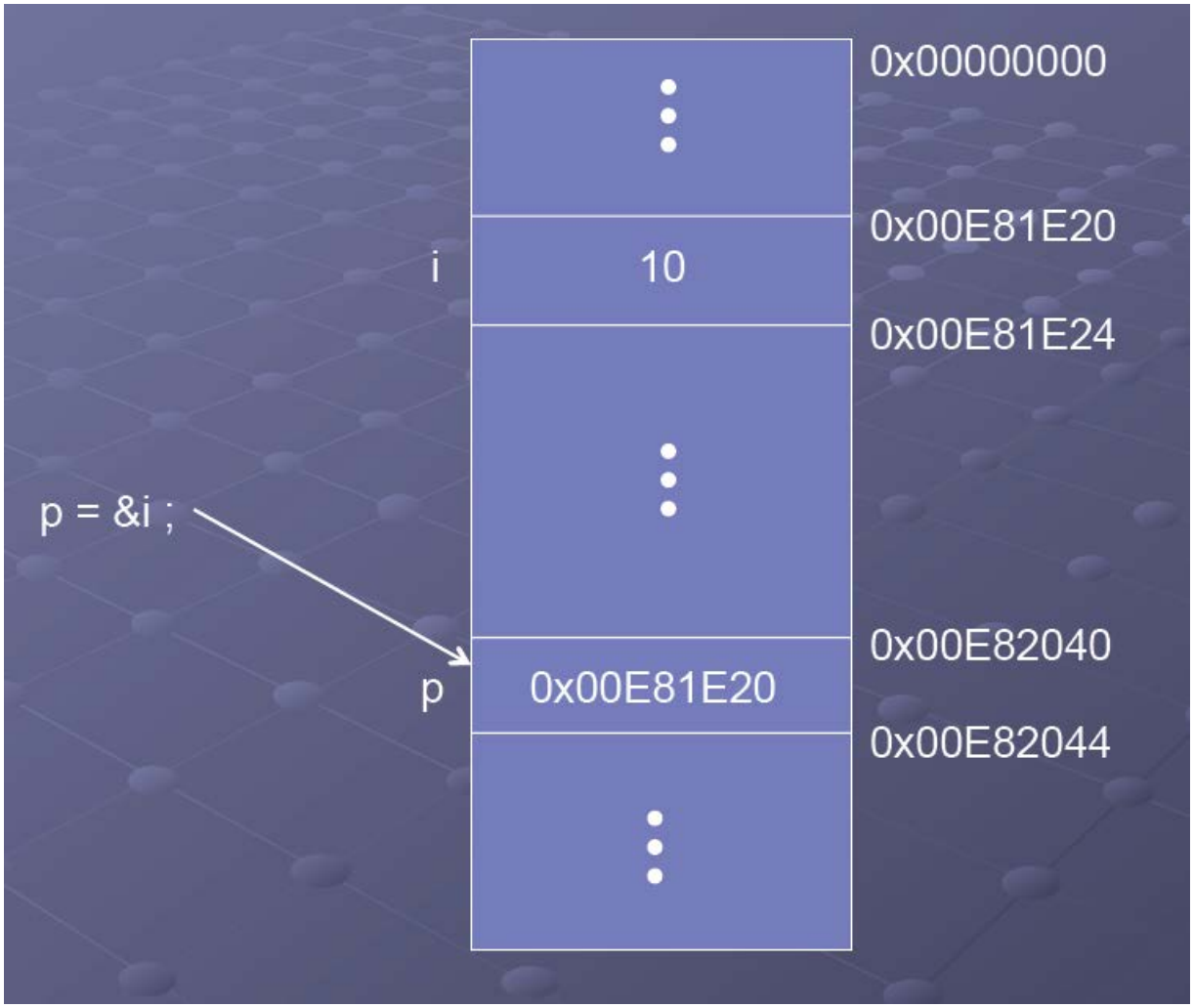
- This can also be written like this:

```
int* p = &a;
```

  - This can be read "p is an int pointer that is assigned the address of a".
  - Different programmers prefer different ways.
  - With this way, you can't declare multiple pointer variables on the same line.

file: pointer.c

Pointers




```
int i = 10;
int* p;
p = &i;
```

What are Pointers?

- If you want to find the value that is stored in the address that a pointer is pointing to, use the * operator.

```
int x = 5;
int *ptr;
ptr = &x;
printf("address of x: %p\n", ptr);
printf("value of x: %d\n", *ptr);
```

- Note that the address of x might change each time we run the program.
- We cannot know the memory addresses in advance (at compile time), so we never write literal memory addresses.

file: value.c

What are Pointers?

- We can also assign a value to the memory address a pointer points to:

```
int x = 5;
int *ptr;
ptr = &x;
printf("address of x: %p\n", ptr);
printf("value of x: %d\n", *ptr);
*ptr = 10;
printf( "value of x: %d\n", x );
printf("value of x: %d\n", *ptr);
```

- The `ptr` variable stores a memory address.
  - This memory address stores the value of x.
- `*ptr` is the value in the memory address `ptr` points to.
- This assignment changes that value to 10.

file: value2.c

Why Pointers?

- C was developed when the hardware was very expensive and computers were not nearly as powerful as they are today
- With C, the user has the ability to work with specific memory locations
- Pointers can be used to optimise a program to run faster, or use less memory, or both!
- Pointers give us a very efficient way of manipulating arrays, strings, and function parameters

Pointers: Summary

- Definition (recall)
  - **A pointer is a variable *containing the address of another variable***
- Remarks
  - C offers two ways of working with the memory: **variables** and **pointers** (many languages only offer the first)
  - A variable is just a *labelled place in memory to store some data*
  - Instead of referring to the data stored in memory by name/label (direct addressing), one can refer to it by its address in memory (indirect addressing).
- Memory Addresses
  - `&var` (& in front of a variable) - gives me the *address* of var
  - `*p` (* in front of a pointer) - gives me the *value* stored at that address
    - unless the variable is being declared. In that case it just means 'I want a pointer variable'

Pointers Summary

- Pointer Operator ‘ * ’
  - ‘ * ’ is really called the *indirection* operator
    - Sometimes called the dereference operator
      - https://en.wikipedia.org/wiki/Dereference_operator
  - To declare a pointer variable, specify the type of data it points to, and use the * before the variable name to show it is a pointer.

```
int *name;
```

  - We also use it to get the value at the address that a pointer points to.

```
int x = *name;
```

Pointers Summary

- Address Operator ‘ & ’
  - Used to obtain the address of a variable
  - The address of a variable is given by adding & in front of the variable
- Example

```
int *pointer, a = 15 ; //pointer is a pointer to an int. a is
                       //an int assigned the value 15.
pointer = &a; // pointer is assigned the address of a.
```

Pointers Summary

- Get the address of a variable
  - `int *pointer, a = 15, b ;`
  - `pointer = &a;` `// pointer gets the address of a`
    `// pointer` *`is equivalent to &a`*
- Get the value of a variable pointer
  - `b = *pointer;` `// the value of b is 15`
  - `*pointer = 45 ;` `// changes the value of a to 45`
    `// *pointer` *`is equivalent to a`*

An example program

```
/* File : usage.c */
#include <stdio.h>

int main () {
  int a = 13, b = -9, i, *p = &a; // p is a pointer to int
  for (i=0; i<10; i++) {
         if ( *p > 0 ) {
                  p = &b ;
         } else if ( *p < 0 ) {
                  p = &a ;
         }
         *p = *p + 1 ;
  }
  printf ( "The value of a and b are : %d and %d \n", a, b);
}
```

Run this program and see what happens. Can you see how it works?

file: usage.c

Note the two different uses of *

- * before a variable name, but after a type (like int) means I want a pointer variable of type int (named p)

```
int *p;
```

- * before a variable name (but not when the variable is being declared) means ‘give me the value [at the address pointed to by the variable]’

```
int x = *p
```

USING POINTERS WITH FUNCTIONS

Using pointers with functions

- A particular benefit of pointers is parameter passing for functions.
  - In C, function calls are **pass-by-value** - this means that the arguments given in the function call are **copied** into the parameters of the function.
  - This passing of the parameters involves copying data from one part of the memory to another.
  - While this is fine for basic data types, it can take a very long time to copy large amounts of data.
  - It also has the issue that if you change the copies, you don't change the originals!

Pointers and Functions

- An alternative approach to functions is known as *pass-by-reference*.
  - In this approach, the **addresses** of the arguments are passed instead of the **values**.
  - This allows the values to be **changed directly** from within the function.
- C supports “simulated” pass-by-reference by using pointers
  - Addresses of variables can be passed to functions instead of the variables themselves.

Pass By Value Example

```
/* File: swap.c */
#include <stdio.h>

void swap (int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
  printf(“x = %d, y = %d\n", x, y);
}

int main () {
  int a =-100, b = 120 ;
  swap (a, b);
  printf ("a = %d, b = %d\n", a, b);
}
```

The **values** of a and b are copied into the function.

The value of a is copied into x, and the value of b is copied into y.

The function swaps the values of x and y, but has no effect on a and b.

file: swap.c

Pass by Reference Example

```
/* File: swap2.c */
#include <stdio.h>

void swap (int *px, int *py) {
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
  printf("value at address pointed to by
  px = %d, value at address pointed to
  by py = %d\n",*px, *py);}

int main () {
  int a =-100, b = 120 ;
  swap (&a, &b);
  printf ("a = %d, b = %d\n", a, b);
}
```

The **addresses** of a and b are copied into the function.

The address of a is copied into px, and the address of b is copied into py.

The function swaps the values that px and py point to (i.e. the values of a and b).

This has the effect of swapping the values of a and b.

file: swap2.c

POINTERS AND ARRAYS

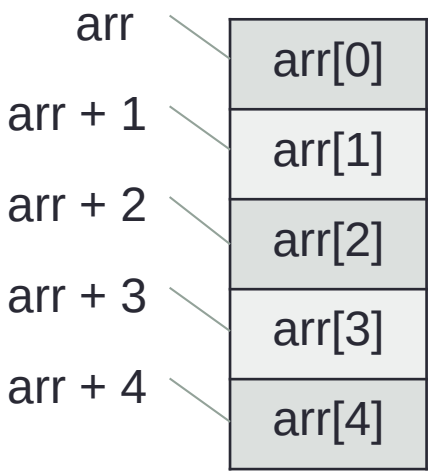
Pointer arithmetic

Arrays and Pointers

- Arrays and pointers have a very strong relationship.
- We can declare an array like this:
  - `int arr[5]:`
- An array name (e.g. `arr`) is actually a pointer to the first element in the array.
  - **Unlike a regular pointer variable, we cannot change where it points to.**
- When we say `arr[2]`, it means that we want to start at the beginning of the array, and then go forward 2 memory cells to get the third element.
  - Remember, arrays are zero-indexed in C

Pointer Arithmetic

- The array name acts as a constant pointer to the *1st element of the array.*
- These two expressions are equivalent:
  - `int *p = arr;`
  - `int *p = &arr[ 0 ];`



- We can use pointer arithmetic to refer to elements in the array.
  - `(arr + 1)` points to the address of `arr[ 1 ]`
  - `(arr + i)` points to the address of `arr[ i ]`
  - `arr[ i ] == *(arr + i);`

- This is why it is important to say the **type** that we point to.
  - What's the difference between (arr + 1) for an int[] array and for a char[] array?

file: array.c

Pointer Arithmetic

```
#include <stdio.h>
int main() {
    int int_arr[5];
    char char_arr[5];

    printf( "1st int addr: %p\n", int_arr );
    printf( "2nd int addr: %p\n", (int_arr + 1));
    printf( "1st char addr: %p\n", char_arr );
    printf( "2nd char addr: %p\n", (char_arr + 1));
}
```

We notice that the difference between (`int_arr`) and (`int_arr+1`) is 4, which shows that an `int` is 4 bytes in size.

The difference for the `char` array is 1.

file: widths.c

PUTTING IT ALL TOGETHER

Passing Arrays to Functions

Passing Arrays To Functions

```
double sum_par (int n, double a[]) {
  double total = 0.0;
  int i;
  for (i=0; i<n; i++) total = total + a[i];
  return total;
}

int main () {
  double A[ 5 ] = {1, 2, 3, 4, 5};
  double B[ 7 ] = {7, 6, 5, 4, 3, 2, 1};
  double sum;
  sum = sum_par( 4, A );
  printf (" sum = %f\n", sum);
  sum = sum_par( 5, B );
  printf (" sum = %f\n", sum);
}
```

file: array2.c

Passing Arrays To Functions

```
double sum_par (int n, double *a) {
  double total = 0.0;
  int i;
  for (i=0; i<=n; i++) total = total + *(a+i);
  return total;
}

int main () {
  double A[ 5 ] = {1, 2, 3, 4, 5};
  double B[ 7 ] = {7, 6, 5, 4, 3, 2, 1};
  double sum;
  sum = sum_par( 4, A );
  printf (" sum = %f\n", sum);
  sum = sum_par( 5, &B[0] );
  printf (" sum = %f\n", sum);
}
```

When we pass an array into a function, it is actually a pointer to the first element that is copied into the function.

For large arrays, this is much faster than copying the entire array.

file: array3.c

Passing Arrays To Functions

- Because we pass in a pointer, it is always possible to change an array's contents within a function.
- **This changes the array itself, not a local copy.**
- For example, a function to double all the values in an array:

```
void twice( int n, int *arr ) {
    int i;
    for (i=0;i<n;i++){
        *(arr+i) *= 2;
    }
}
```

file: double.c