Topic 2

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. Dynamic memory allocation
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Structures
8. Pointers and structures

Arrays and Pointers

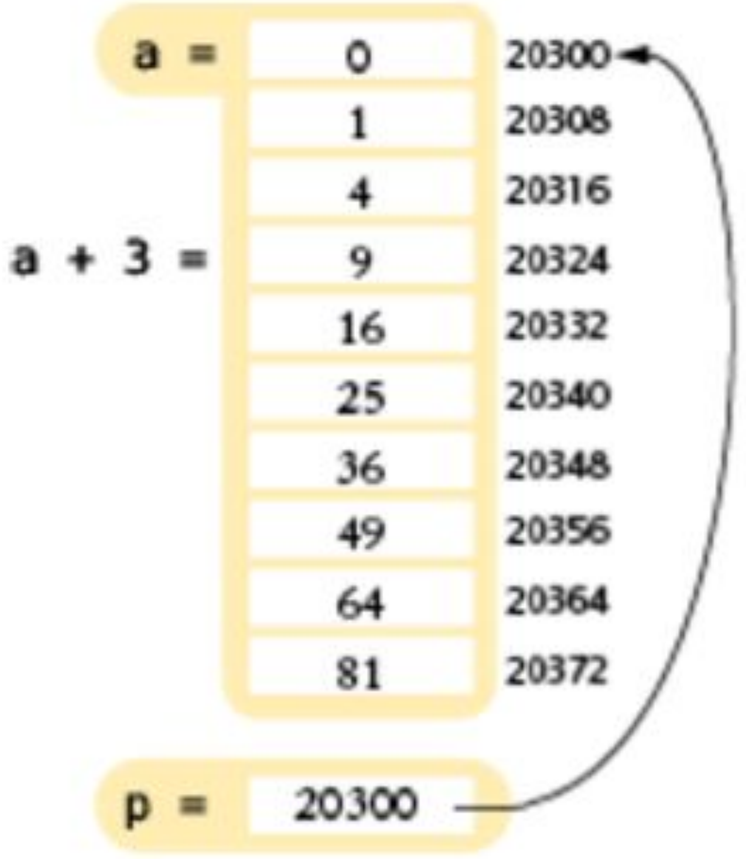
Pointers can help explain the peculiarities of arrays.

The *name* of the array is a pointer to the starting element.

```
int a[10];
```

You can capture the address of the first element in the array in a pointer variable:

```
int* p = a;
// Now p points to a[0]
```



Pointer Arithmetic, and Array/Pointer Duality

You can use the array name `a` as you would a pointer:
These output statements are equivalent:

```
cout << *a;
cout << a[0];
```

Pointer arithmetic means adding integers to pointers (or to array names):

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

`p + 3` is a pointer to the array element with index `3`

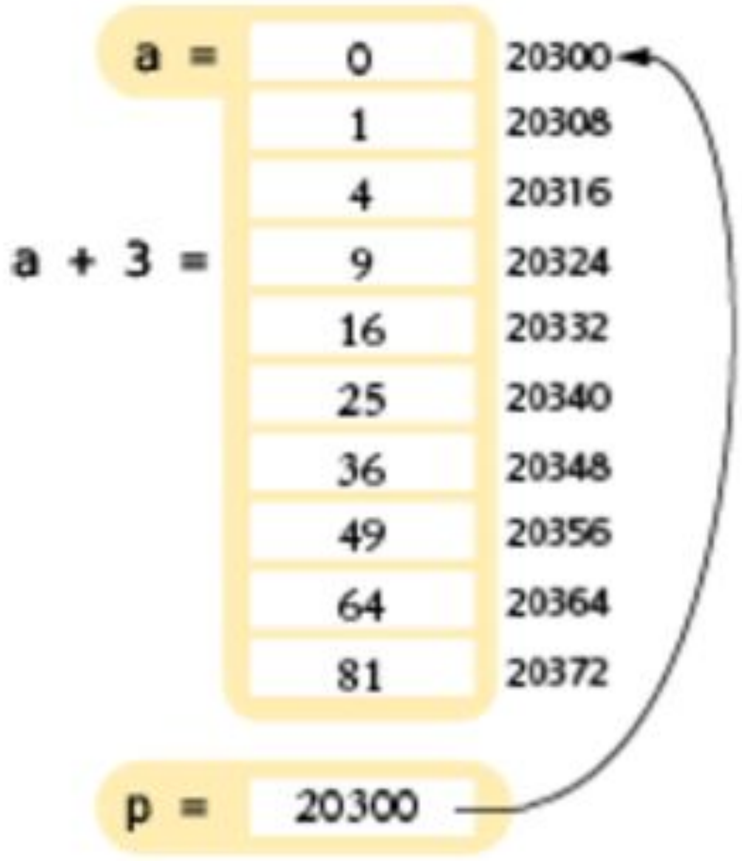
The expression: `*(p + 3)` means the same as `p[3]` and `a[3]`. This is "***array/pointer duality***".

The Array/Pointer Duality Law

This law explains why all C++ arrays start with an index of zero.

The pointer **`a`** (or **`a + 0`**) points to the starting element of the array.

That element must therefore be **`a[0]`**.



Array / Pointer Examples: Table 2

Expression	Value	Comment
`a`	`20300`	Starting address of the array, here assumed 20300.
`*a`	`0`	The value stored at that address. (The array contains values 0, 1, 4, 9, ...)
`a + 1`	`20308`	The address of the next `double` value in the array. A `double` occupies 8 bytes.
`a + 3`	`20324`	The address of the element with index 3, obtained by skipping past 3 × 8 bytes.
`*(a+3)`	`9`	The value stored at address 20324.
`a[3]`	`9`	The same as `*(a + 3)` by array/pointer duality.
`*a + 3`	`3`	The sum of `*a` and 3. Because there are no parentheses, the * refers only to a.
`&a[3]`	`20324`	The address of the element with index 3, the same as `a + 3`.

Array Parameters are Pointer Variables

Consider this function that computes the sum of all values in an array:

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

Syntactic Sugar

In the function header:

```
double sum(double a[], int size)
```

The C++ compiler considers **`a`** to be a pointer, not an array.

The expression **`a[i]`** is *syntactic sugar* for **`*(a + i)`**.

syntactic sugar = a notation that is easy to read for humans and that masks a complex implementation detail.

Equivalent Function Headers

That masked complex implementation detail:

```
double sum(double* a, int size)
```

is how we *should* define the first parameter

but

```
double sum(double a[], int size)
```

looks a lot more like we are passing an array.

Using a Pointer to Step Through an Array

With pointer arithmetic, we can step through an array without using the braces `[ ]`:

```
double sum(double* a, int size) {
   double total = 0;
   double* p = a;
   // p starts at the beginning of the array
   for (int i = 0; i < size; i++){
      total = total + *p;
      // Add the value to which p points
      p++;
      // Advance p to the next array element
   }
   return total;
}
```

Common Error: Returning a Pointer to a Local Variable

Consider this bogus function that tries to return a pointer to an array containing two elements, the first and the last values of an array:

```
double* firstlast(const double values[], int size)
{
   double result[2];
   result[0] = values[0];
   result[1] = values[size - 1];
   return result; // Error! Points to a local array
// that will evaporate as this function returns
}
```

Fixing the Pointer Return Error

The local variable

```
   double result[2];
```

no longer exists when the function exits. Its contents will soon be overwritten by other function calls.

You can solve this problem by passing an array to hold the answer:

```
void firstlast(const double values[], int size,
double result[])
{
   result[0] = values[0];
   result[1] = values[size - 1];
}
```

An alternative fix is to dynamically allocate the `result[]` array in the function using the `new` keyword, but we'll save that for later in the chapter.

Program Clearly, Not Cleverly

Some programmers take pride in minimizing the number of instructions, even if the resulting code is hard to understand.

```
while (size > 0)
 {
    total = total + *p;
    p++;
    size--;
 }
```

could be written as:

```
while (size-- > 0)
    total = total + *p++;
```

Ah, so much better?

Program Clearly

Please do not use such terse programming style.

Your job as a programmer is not to dazzle other programmers with your cleverness,
but to write code that is easy to understand and maintain.

For example, does

```
*p++;
```

mean increment the data that `p` points to, or increment the pointer address? Does it the increment happen before or after the data is accessed? You and the compiler may remember the rules for that syntax, but readers of your code might not.

Constant Pointers for Read-Only Variables and Arrays

A constant pointer:

```
const double* p = &balance;
```

cannot modify the value to which p points.

```
*p = 0; // Error
```

Of course, you can read the value:

```
cout << *p; // OK
```

A constant array parameter is equivalent to a constant pointer.

```
double sum(const double values[], int size)
double sum(const double* values, int size)
```

The function can use the pointer values to read the array elements, but it cannot modify them.