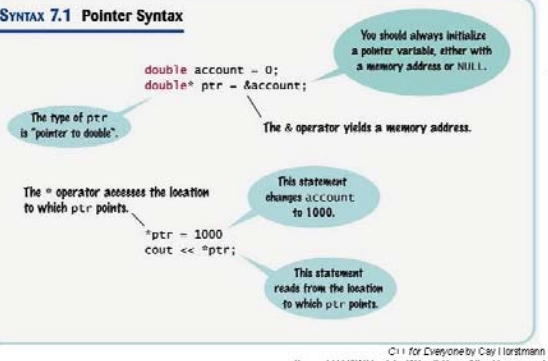
## Syntax of Pointers

**Syntax 7.1 Pointer Syntax**







## Pointer Syntax Examples

Table 1 Pointer Syntax Examples

Assume the following declarations:
```
int n = 10; // Assumed to be at address 20300
int m = 20; // Assumed to be at address 20304
int* p = &n;
```

| Expression | Value | Comment |
|---|---|---|
| p | 20300 | The address of n. |
| *p | 10 | The value stored at that address. |
| &n | 20304 | The address of n. |
| p = &n; | | Set p to the address of n. |
| *p | 20 | The value stored at the changed address. |
| n = *p; | | Stores 20 into n. |
| n = p; | Error | n is an int value; p is an int* pointer. The types are not compatible. |
| &10 | Error | You can only take the address of a variable. |
| &p | The address of p, perhaps 20308 | This is the location of a pointer variable, not the location of an integer. |
| double x = 0; p = &x; | Error | p has type int*, &x has type double*. These types are incompatible. |




## Harry's Banking Program

Here is the complete banking program that Harry wrote. It demonstrates the use of a pointer variable to allow *uniform access* to variables.

ch07/accounts.cpp

```
#include <iostream>
using namespace std;

int main()
{
   double harrys_account = 0;
   double joint_account = 2000;
   double* account_pointer = &harrys_account;
   *account_pointer = 1000; // Initial deposit
```




## Harry's Banking Program

ch07/accounts.cpp

```
   // Withdraw $100
   *account_pointer = *account_pointer - 100;

   // Print balance
   cout << "Balance: " << *account_pointer
      << endl;

   // Change the pointer value so that the same
   // statements now affect a different account
   account_pointer = &joint_account;

   // Withdraw $100
   *account_pointer = *account_pointer - 100;

   // Print balance
   cout << "Balance: " << *account_pointer
      << endl;

   return 0;
}
```

1000 900 2000 1900




## Common Error: Confusing Data And Pointers

A pointer is a memory address
– a number that tells where a value is located in memory.

It is a common error to confuse the pointer with the variable to which it points.




## Common Error: Where's the *?

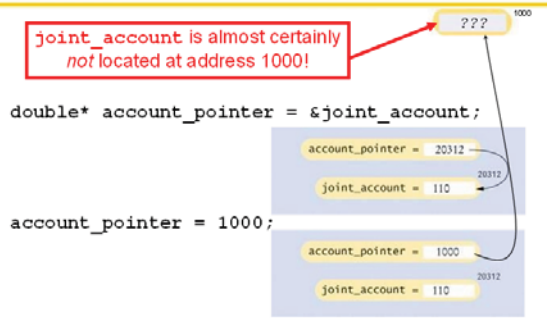
```
double* account_pointer = &joint_account;
account_pointer = 1000;
```

The assignment statement does *not* set the joint account balance to 1000.

It sets the pointer variable, `account_pointer`, to point to memory address 1000.

ERROR

## Common Error: Where's the *?






## Common Error: Where's the *?

Most compilers will report an error for this kind of error.



## Confusing Definitions

It is legal in C++ to define multiple variables together, like this:

```
int i = 0, j = 1;
```

This style is confusing when used with pointers:

```
double* p, q;
```

The * associates only with the first variable.
That is, p is a double* pointer, and q is a double value.

To avoid any confusion, it is best to define each pointer variable separately:

```
double* p;
double* q;
```



## Pointers and References

& == *

?

What are you asking?



## Pointers and References

Recall that the & symbol is used for reference parameters:

```
void withdraw(double& balance, double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}
```

and then sent back

a call would be:

```
withdraw(harrys_checking, 1000);
```

the value is sent



## Pointers and References

We can accomplish the same thing using pointers:

```
void withdraw(double* balance, double amount)
{
   if (*balance >= amount)
   {
      *balance = *balance - amount;
   }
}
```

but the call will have to be:

```
withdraw(&harrys_checking, 1000);
```

the address is sent to the function

## Arrays and Pointers (7.2)

In C++, there is a deep relationship between pointers and arrays.

This relationship explains a number of special properties and limitations of arrays.



## Arrays and Pointers

Pointers are particularly useful for understanding the peculiarities of arrays.

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

```
double a[10];
```

we end up w/a "pointer to double" which points to a[0]



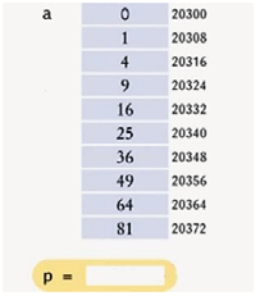
## Arrays and Pointers

Consider this declaration:

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

(Assume we have filled it as shown.)

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





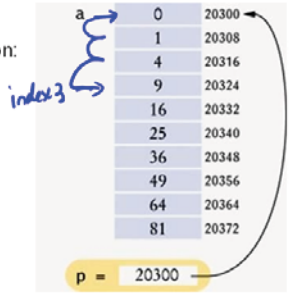

## Arrays and Pointers

Consider this declaration:

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

(Assume we have filled it as shown.)

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




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

the whole array



## Arrays and Pointers – Same Use

You can use the array name `a` as you would a pointer:

These output statements are equivalent:

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

same thing



## Pointer Arithmetic

*Pointer arithmetic* allows you to add an integer to an array name.

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

p points to array a

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

The expression: `*(p + 3)`

a[3]

## The Array/Pointer Duality Law

The *array/pointer duality law* states:

index n

`a[n]` is identical to `*(a + n)`.

where `a` is a pointer into an array and `n` is an integer offset.



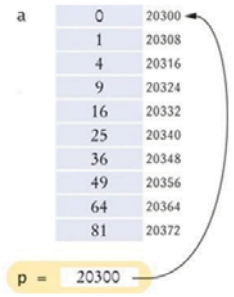
## The Array/Pointer Duality Law

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]`.

You are adding 0 to the start of the array, thus *correctly going nowhere!*






## The Array/Pointer Duality Law

Now it should be clear why array parameters are different from other parameter types.

(if not, we'll show you)



## The Array/Pointer Duality Law

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

*Look at this*

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

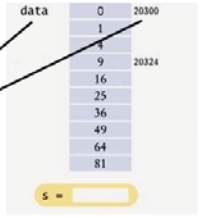


## The Array/Pointer Duality Law

Here is a call to the function.

```
double data[10];
... // Initialise data

double s = sum(data, 10);
```

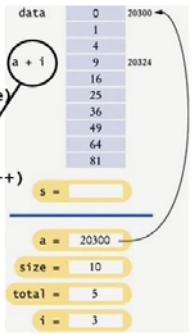





## The Array/Pointer Duality Law

After the loop has run to the point when `i` is 3:

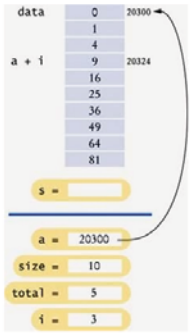
```
double sum(double a[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + a[i];
   }
   return total;
}
```



## The Array/Pointer Duality Law

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

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






## Syntactic Sugar

Computer scientists use the term

*"syntactic sugar"*

to describe a notation that is easy to read for humans and that masks a complex implementation detail.



## Syntactic Sugar

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.

(yummy!)



## Arrays and Pointers

**Table 2 Arrays and Pointers**

| Expression | Value | Comment |
|---|---|---|
| `a` | 20300 | The starting address of the array, here assumed to be 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. Since 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`. |



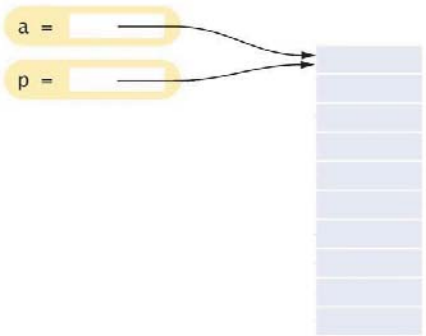
## Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
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;
}
```



## Using a Pointer to Step Through an Array



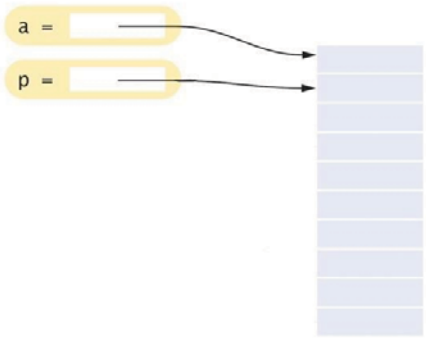
## Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
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;
}
```



## Using a Pointer to Step Through an Array





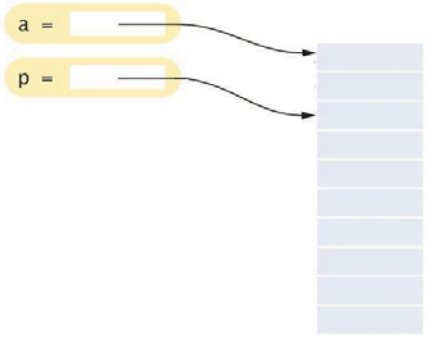

## Using a Pointer to Step Through an Array

Watch variable p as this code is executed.

```
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;
}
```



## Using a Pointer to Step Through an Array





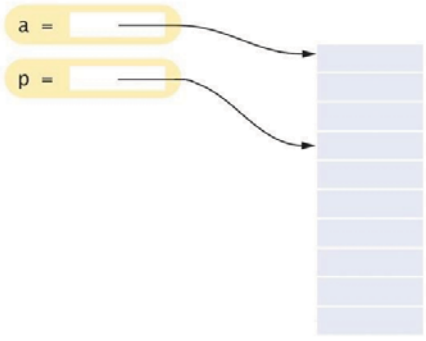

## Using a Pointer to Step Through an Array

Add, then move p to the next position by incrementing.

```
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;
}
```



## Using a Pointer to Step Through an Array



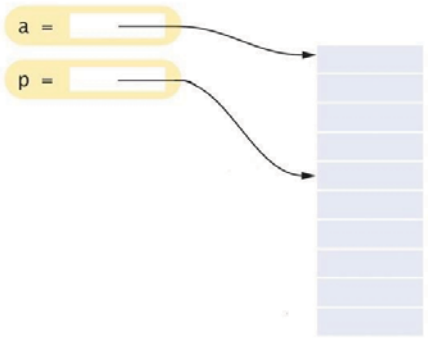
## Using a Pointer to Step Through an Array

Add, then again move p to the next position by incrementing.

```
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;
}
```



## Using a Pointer to Step Through an Array





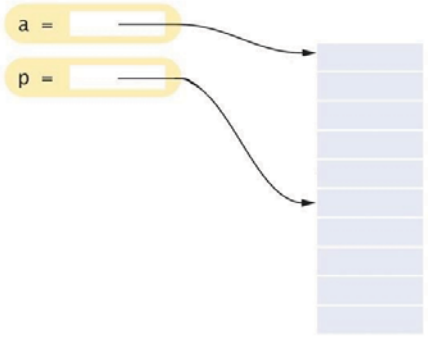

## Using a Pointer to Step Through an Array

Add, then move p.

```
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;
}
```



## Using a Pointer to Step Through an Array





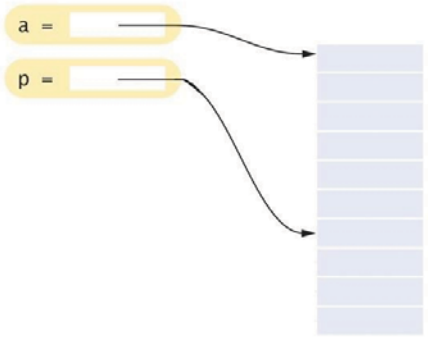

## Using a Pointer to Step Through an Array

Again...

```
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;
}
```



## Using a Pointer to Step Through an Array



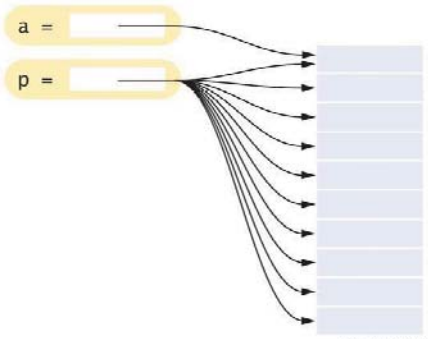
## Using a Pointer to Step Through an Array

And so on until every single position in the array has been added.

```
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;
}
```



## Using a Pointer to Step Through an Array






## Using a Pointer to Step Through an Array

It is a tiny bit more efficient to use and increment a pointer than to access an array element.



## Program Clearly, Not Cleverly

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

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

could be written as:

```
total = total + *p++;
```

Ah, so much better?



## Program Clearly, Not Cleverly

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

could be written as:

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

Ah, so much better?



## Program Clearly, Not Cleverly

Please do not use this 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.

## Common Error: Returning a Pointer to a Local Variable

What would it mean to
"return an array"
?



## Common Error: Returning a Pointer to a Local Variable

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

```
double* firstlast(double a[], int size)
{
   double result[2];
   result[0] = a[0];
   result[1] = a[size - 1];
   return result;
}
```

*Local memory is invalid after the function call has ended!*

*What would the value the caller gets be pointing to?*



## Common Error: Returning a Pointer to a Local Variable

A solution would be to pass
in an array to hold the answer:

```
double* firstlast(double a[], int size,
                  double result[])
{
   result[0] = a[0];
   result[1] = a[size - 1];
}
```



## C and C++ Strings, POP QUIZ (7.3)

"Q: What?"

Really we mean:

"Q: What is this?"

A *C string*, of course!
(notice the double quotes: "Like this")



## C and C++ Strings

C++ has two mechanisms for manipulating strings.

The `string` class
- Supports character sequences of arbitrary length.
- Provides convenient operations such as concatenation and string comparison.

we did this in CIS2

C strings
- Provide a more primitive level of string handling.
- Are from the C language (C++ was built from C).
- Are represented as arrays of `char` values.

char data type could only hold 1 single character



## `char` Type and Some Famous Characters

The type `char` is used to store an individual character.

## char Type and Some Famous Characters

Some of these characters are plain old letters and such:

```
char yes = 'y';
char no = 'n';
char maybe = '?';
```



## char Type and Some Famous Characters

Some are numbers masquerading as digits:

```
char theThreeChar = '3';
```

That is not the number three – it's the *character* 3.
`'3'` is what is actually stored in a disk file
when you write the `int` 3.
Writing the variable `theThreeChar` to a file
would put the same `'3'` in a file.



## char Type and Some Famous Characters

Recall that a stream is a
sequence of characters – `chars`.



## char Type and Some Famous Characters

So some characters are literally what they are:

```
'A'
```

Some represent digits:

```
'3'
```

Some are other things that can be typed:

```
'C'
'+'
'+'
```

but...



## Some Famous Characters

START HERE

Some of these characters are true individuals.
"Characters" you might say (if they were human).

They are quite "special":

```
'\n'
'\t'
```

These are still single (individual) characters:
the ***escape sequence*** characters.

the backslash turns off usual meaning and gives it its special meaning



## Some Famous Characters

And one you can output to the screen
in order to annoy those around you
(if you were naughty and didn't mute your computer when you entered the classroom)

```
'\a'
```

– the *alert* character.

Don't try this at home
– no we mean
*ONLY* try this at home!!!