CS112

Pointers

Pointers

- C and C++ provide a pointer data type to allow programmers to manipulate memory space.
- Pointers are **variables that hold addresses** in C and C++.
- Provide much power and utility for the programmer to access and manipulate data.
- Useful for passing parameters into functions, allowing a function to modify and return values to a calling routine.
- Used in dynamic memory allocation

Pointers

Careful:

- Writing in C or C++ is like running a chain saw with all the safety guards removed. *Bob Gray*
- In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg. *Bjarne Stroustrup*.

Pointers and Memory

- How values for variables are stored in computer memory?
  - Whenever a variable is declared, a memory location is associated with the variable.
  - Whenever the variable is accessed the data value is taken from that particular memory location.
  - A **variable** contains a value, but a **pointer** specifies where a value is located.
  - A **pointer** denotes the memory

    location of a variable.

Memory and Addresses

- Here's a picture of RAM.
  - Every byte in RAM has an address
  - (shown in groups of four bytes)
  - Addresses are always represented in hexadecimal format.

Address	Data value
0x241FF50	
0x241FF54	
0x241FF58	
0x241FF5C	

Hexadecimal numbers

- Everyday numbers use base 10 (decimal)
  - 0 1 2 3 4 5 6 7 8 9
- Computers use base 2 to store information (binary)
  - 0 1
- For memory addresses **base 16** is used (hexadecimal)
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
  - A=decimal 10, B=decimal 11, …, F=decimal 15

Converting hexadecimal to decimal

- For a decimal number 345 we have that

  $3*10^2 + 4*10^1 + 5*10^0 =$

  $300 \quad + 40 \qquad + 5 \qquad\quad = 345$

  using decimals!

- Convert 256 hexadecimal to decimal

  $2*16^2 + 5*16^1 + 6*16^0 =$

  $512 \quad + 80 \qquad + 6 \qquad\quad = 598$

  using decimals!

- Convert FA8 hexadecimal to decimal

  $\text{'F'}*16^2 + \text{'A'}*16^1 + \text{'8'}*16^0 =$

  $15*16^2 + 10*16^1 + 8*16^0 =$

  $3840 \quad + 160 \qquad + 8 \qquad\quad = 4008$

  using decimals!

Hexadecimal numbers

- Why hexadecimal?
  1. Data is binary
  2. Convenience
- One digit represents 4 bits
- FA are 8 bits binary 1111 1010
- FA is decimal 250
- Decimal to binary not as easy
- Prefix 0x is used for hexadecimals
- For example 0x241FF50

4 bits binary	hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Memory and Addresses

- Here's a picture of RAM.
- All variables are stored in memory, each at its own unique address or location.
- For example

```
int count = 5;
```

  - The variable count is associated with memory adress 0x241FF54
  - The value "5" is stored in memory.
  - It can be accessed by using the variable name "count".
  - **Or it be accessed by reference to its address.**

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Every time the program is executed the memory location may change.

Addresses and References

- Accessing memory addresses
  - To access the memory address of a variable, we use the unary operator **&**
  - **&** is called an address operator (or address-of) operator.
- Example

```
cout<< &count << endl;
```

  - This prints address: `0x241FF54`

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Addresses and Pointers

- A pointer is a special type of variable
  - **Normal variables contain a data value.**
  - **Pointer variables contains a memory address as its value.**
  - Data is modified when a normal variable is used.
  - The value of the address stored in a pointer is modified when a pointer is used.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Addresses and Pointers

- How to define a pointer
  - ***datatype* pointername;***
- Example

```
int* ptr;
```

- This declares a pointer variable named `ptr` of type `int*`

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

- Note: Some people prefer to write

```
int *ptr
```

- It's the same. Use one, consistently.

Addresses and Pointers

- Usually, the address stored in the pointer is the address of some other variable.
- Example

```
int  count = 5;
int* ptr   = &count;
```

- This means:
  - `count` is a variable with value 5
  - `ptr` is a pointer with value 0x241FF54

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Dereferencing Pointers

- To get the value that is stored at the memory location pointed by the pointer, one would need to **dereference** the pointer.
- Dereferencing means to read the data value at the address the pointer points to.
- Dereferencing allows manipulation of the **data contained at the memory address stored in the pointer.**

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Dereferencing Pointers

- Dereferencing is done with the unary operator * called a **dereference operator.**
- Syntax
  - ***variablename**
- Example

```
int  count = 5;
int* ptr   = &count;
cout << *prt <<endl;
```

  - This prints the value at address 0x241FF54. Which is 5.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Assigning to Pointers

- Assign an address to a pointer

```
int count = 5;
int* ptr1 = &count;
int* ptr2 = ptr1;
```

  - Both pointers, `ptr1` and `ptr2`, point to the same address.

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	6
	0x241FF58	
	0x241FF5C	

- Set value pointed to directly

```
*ptr1 = 6;
cout << *ptr1;  prints 6
cout << *ptr2;  prints 6
```

  - Because both pointers, `ptr1` and `ptr2`, point to the same address.

Pointers and Addresses

- Normal variables:
  - A normal variable contains a data value

    `int count = 5;`
  - Use `&` to get its address:

    `cout <<  count;` prints 5

    `cout << &count;` prints 0x241FF54
- Pointers:
  - A point variable contains an address

    `int* ptr = &count;`
  - Use `*` to get the value at that address

    `cout <<  ptr;` prints 0x241FF54

    `cout << *ptr;` prints 5

Variable	Address	Data value
	0x241FF50	
count	0x241FF54	5
	0x241FF58	
	0x241FF5C	

Example

- Consider the following piece of code:

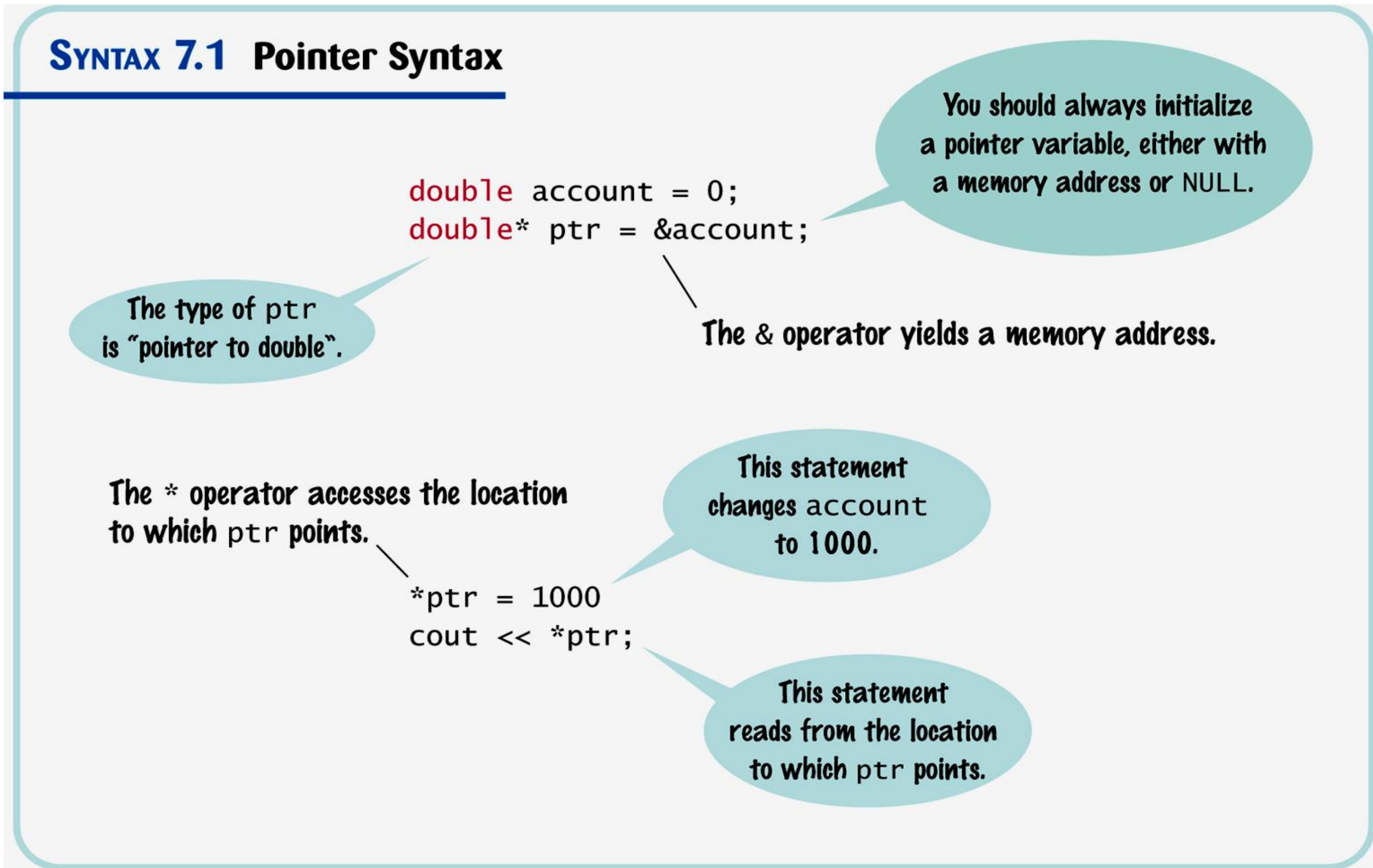
```
int j = 2;
int * pt1 = &j; //pt1 points to j
cout << *pt1;   //prints out value in int j
*pt1 =*pt1 + 2; //adds two to the value
                //pointed to by pt1 (i.e. int j)
```

- The effect of the above statement is equal to

```
int j = 2;
j = j + 2;
```

Of course, usually you use pointers for more useful things. Not for making 2 + 2 complicated.

Syntax of Pointers



Pointer Syntax Examples

Table 1 Pointer Syntax Examples

Assume the following declarations:

```
int m = 10; // Assumed to be at address 20300
int n = 20; // Assumed to be at address 20304
int* p = &m;
```

Expression	Value	Comment
`p`	20300	The address of `m`.
`*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.
`m = *p;`		Stores 20 into `m`.
🚫 `m = p;`	**Error**	`m` 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.

Errors Using Pointers

Uninitialized Pointer Variables

When a pointer variable is first defined,

it contains a random address.

Using that random address is an error.

Errors Using Pointers

Uninitialized Pointer Variables

In practice, your program will likely crash or mysteriously misbehave if you use an uninitialized pointer:

```
double* account_pointer; // No initialization
*account_pointer = 1000;
```

NO!

account_pointer contains an unpredictable value!

Where is the 1000 going?

Errors Using Pointers

Uninitialized Pointer Variables

There is a special value

that you can use

to indicate a pointer

that doesn't point anywhere:

`NULL`

NULL

- If you define a pointer variable and are not ready to initialize it quite yet, it is a good idea to set it to **NULL**
- You can later test whether the pointer is **NULL.**
- If it is, don't use it.
- Example:

```
double* account_pointer = NULL; // Will set later
…                               // Lots of other stuff
if (account_pointer != NULL)    // OK to use?
{
   cout << *account_pointer;
}
```

NULL

Warning:

Trying to access data through a NULL pointer is still illegal,
and
it will cause your program to crash.

```
double* account_pointer = NULL;

cout << *account_pointer;
```

CRASH!!!

NULL

Warning:

Trying to access data through a NULL pointer is still illegal,
and
it will cause your program to crash.

Accidentally dereferencing a NULL Pointer is a serious problem you want to avoid at all costs.

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: Confusing Data And Pointers

A pointer tells where a rabbit is located in the field.

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

Common Error: Where's the *?

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

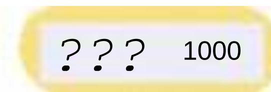
ERROR

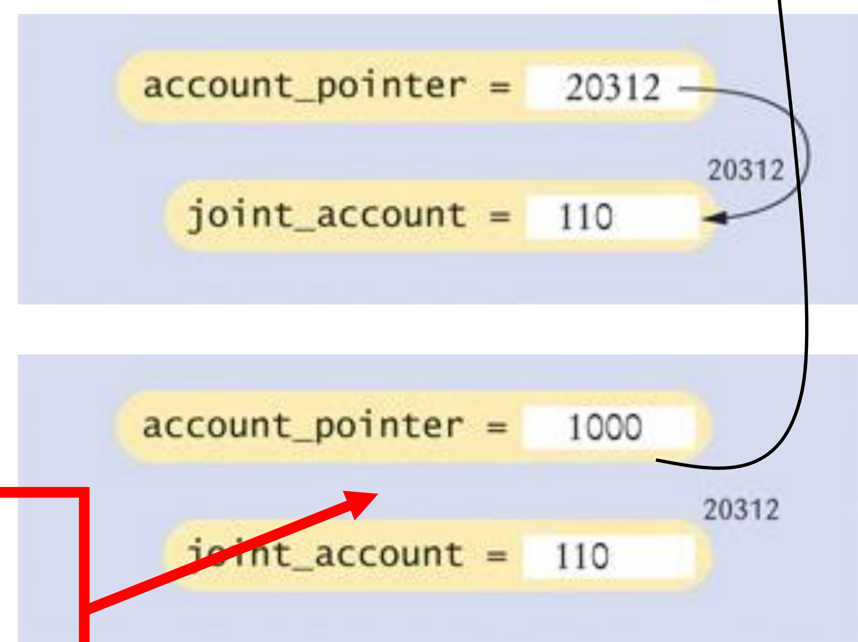
The assignment statement does not set the joint account balance to 1000.

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

Common Error: Where's the *?

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






```
account_pointer = 1000;
```

joint_account is almost certainly not located at address 1000!

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 $\boxed{\&}$ symbol is used for reference parameters:

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

a call would be:

```
withdraw(account, 1000);
```

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(&account, 1000);
```

Summary

- Pointer are variables that refer to addresses in memory
  - Use *datatype** to define pointers.
  - You obtain the value stored at the location a pointer points to by dereferencing it.
  - The dereferencing is done using the * operator.
  - Careful about NULL and undefined pointers. Very careful.
  - You can get the address of a **normal variable** by using the address-of operator &
  - You can assign addresses to pointers. With appropriate type.

Arrays and Pointers

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.

Arrays and Pointers

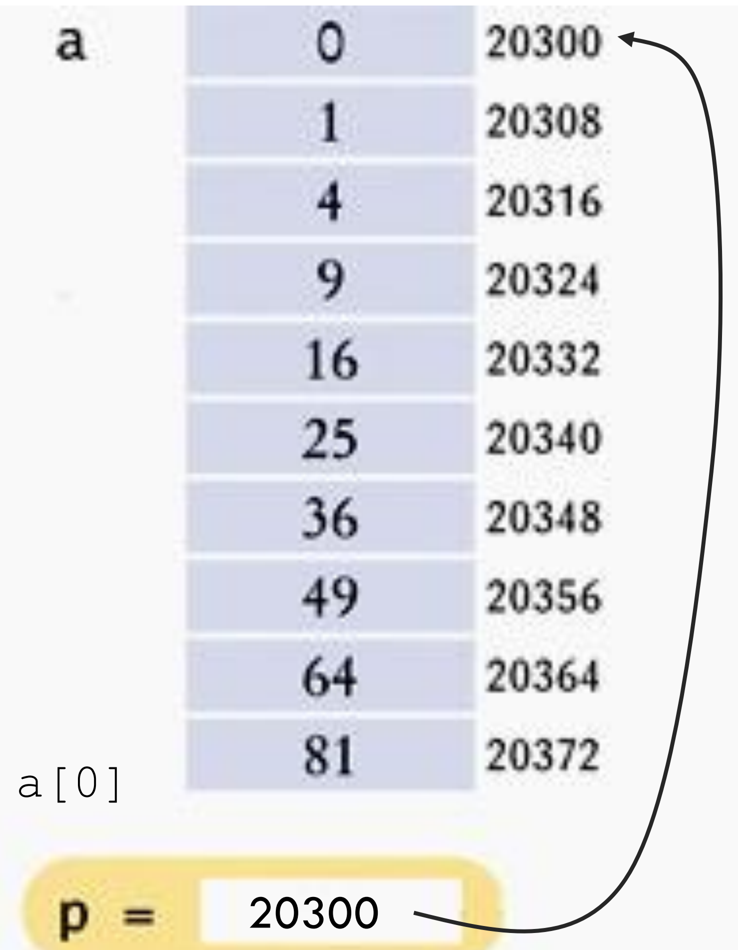
Consider this declaration:

```
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]
```



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];
```

Prints the value

These output statements as well:

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

Prints the address

Pointer Arithmetic

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

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

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

The expression: `*(p + 3)`

is the same as: `a[3]`

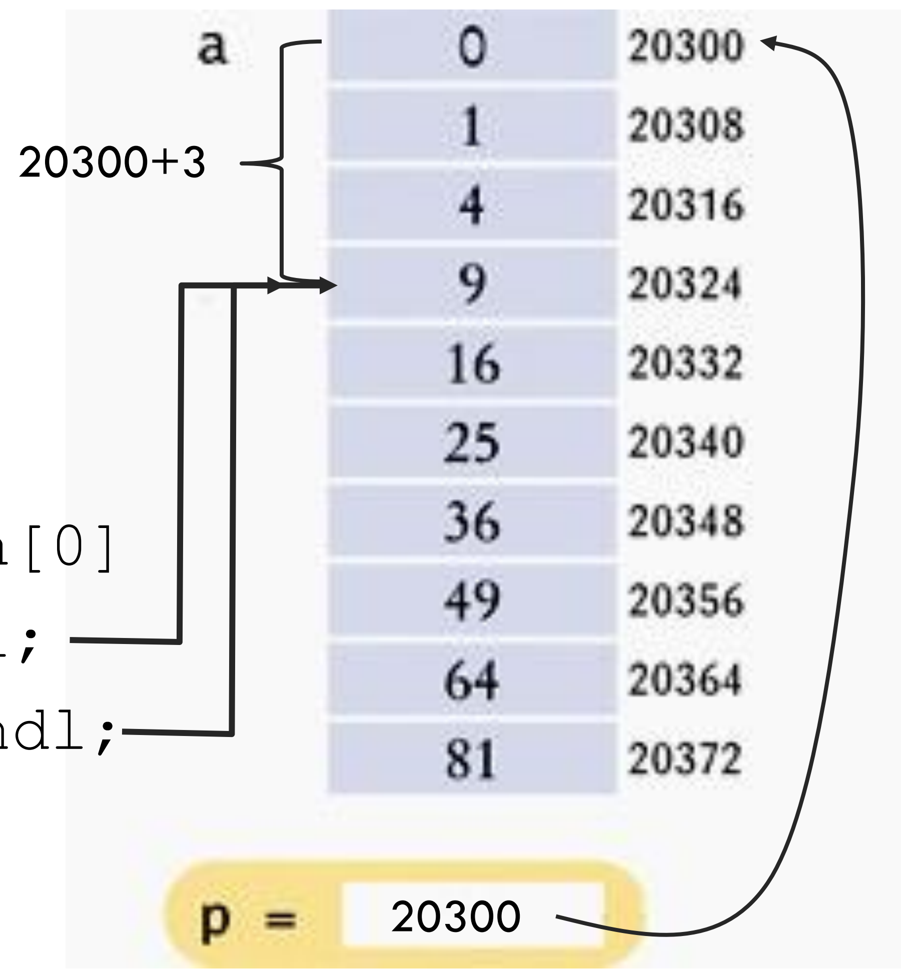
Pointer Arithmetic

The expression: `*(p + 3)`

is the same as: `a[3]`

Really.

```
int a[10];
int* p = a; // or &a[0]
cout << a[3] << endl;
cout << *(p+3) << endl;
```



The Array/Pointer Duality Law

The array/pointer duality law states:

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

where **`a`** is a pointer into an array

and **`n`** is an integer offset.

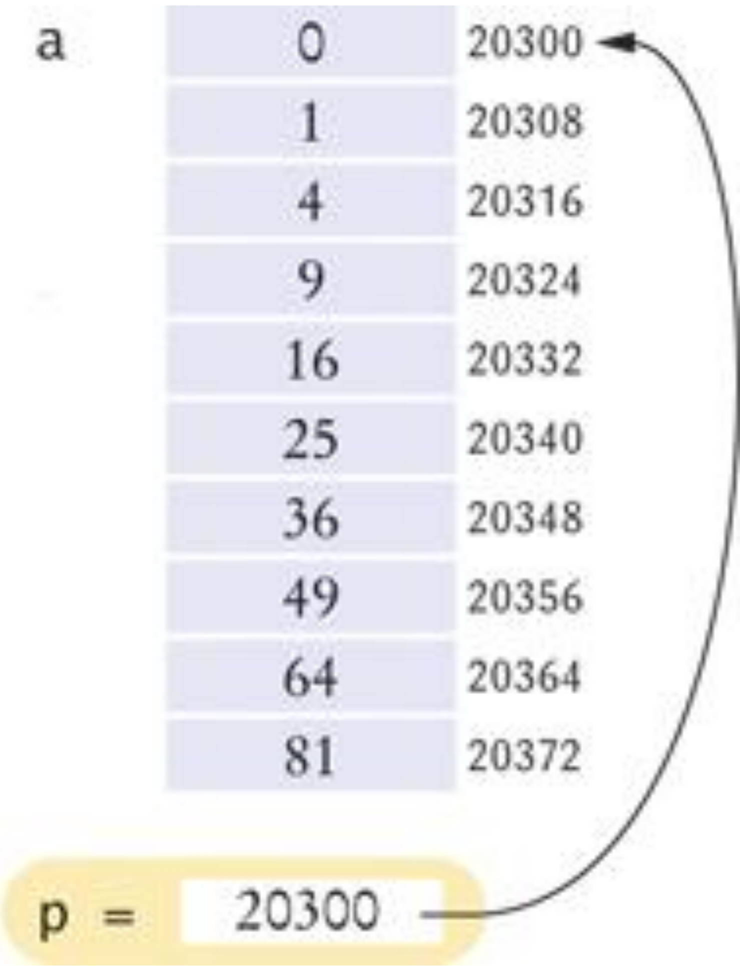
The Array/Pointer Duality Law

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:

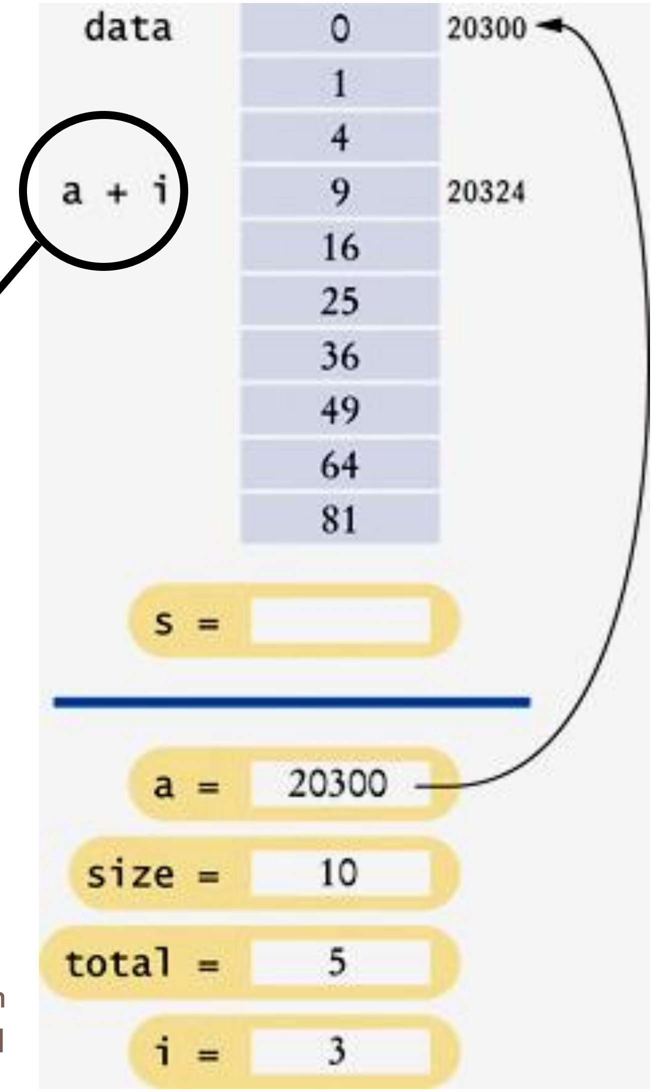
Look at this

```
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

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

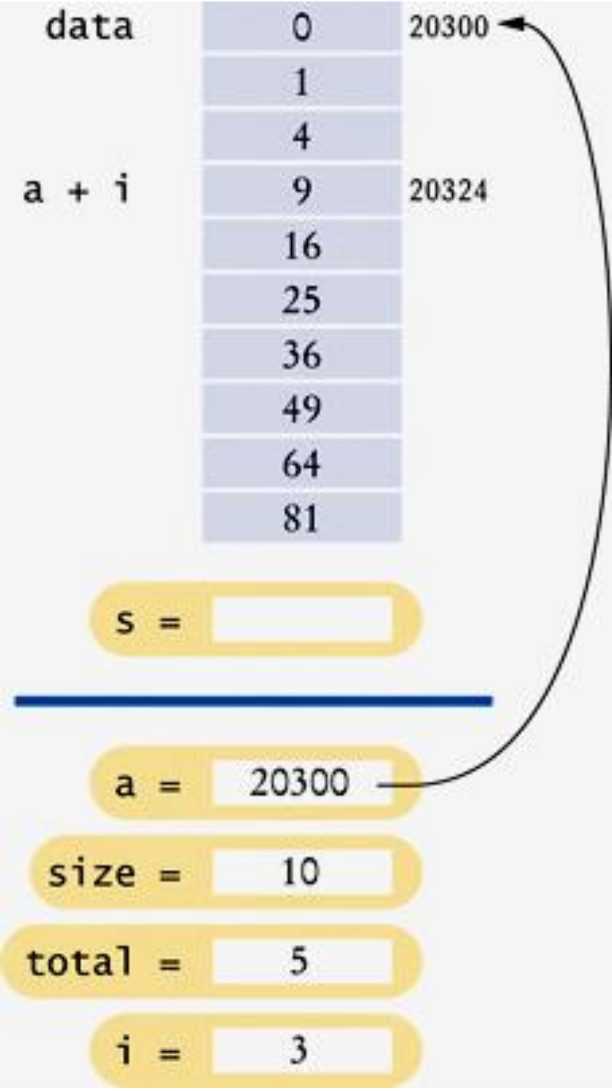
```
double sum(double a[], int size)
{
    double total = 0;
    for (int i = 0; i < size; i++)
    {
        total = total + a[i];
    }
    return total;
}
```



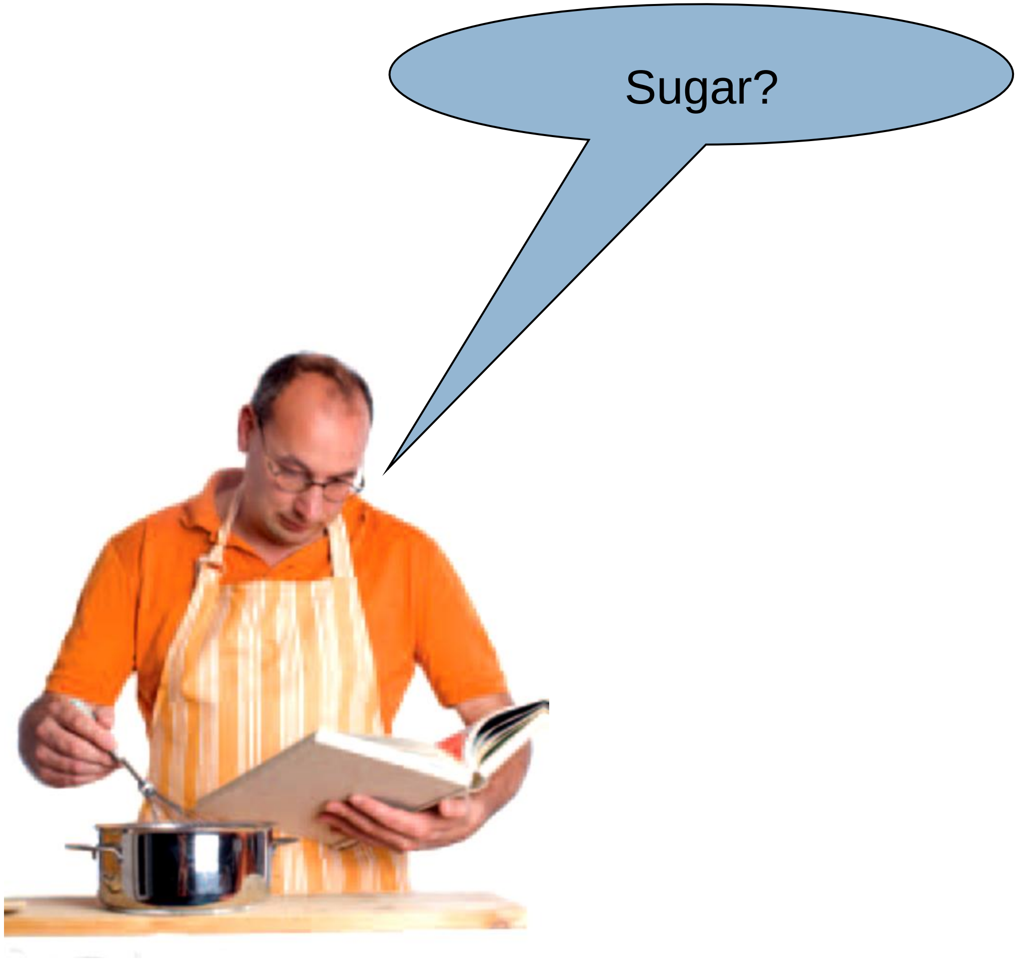
The Array/Pointer Duality Law

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

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



Syntactic Sugar



Syntactic Sugar

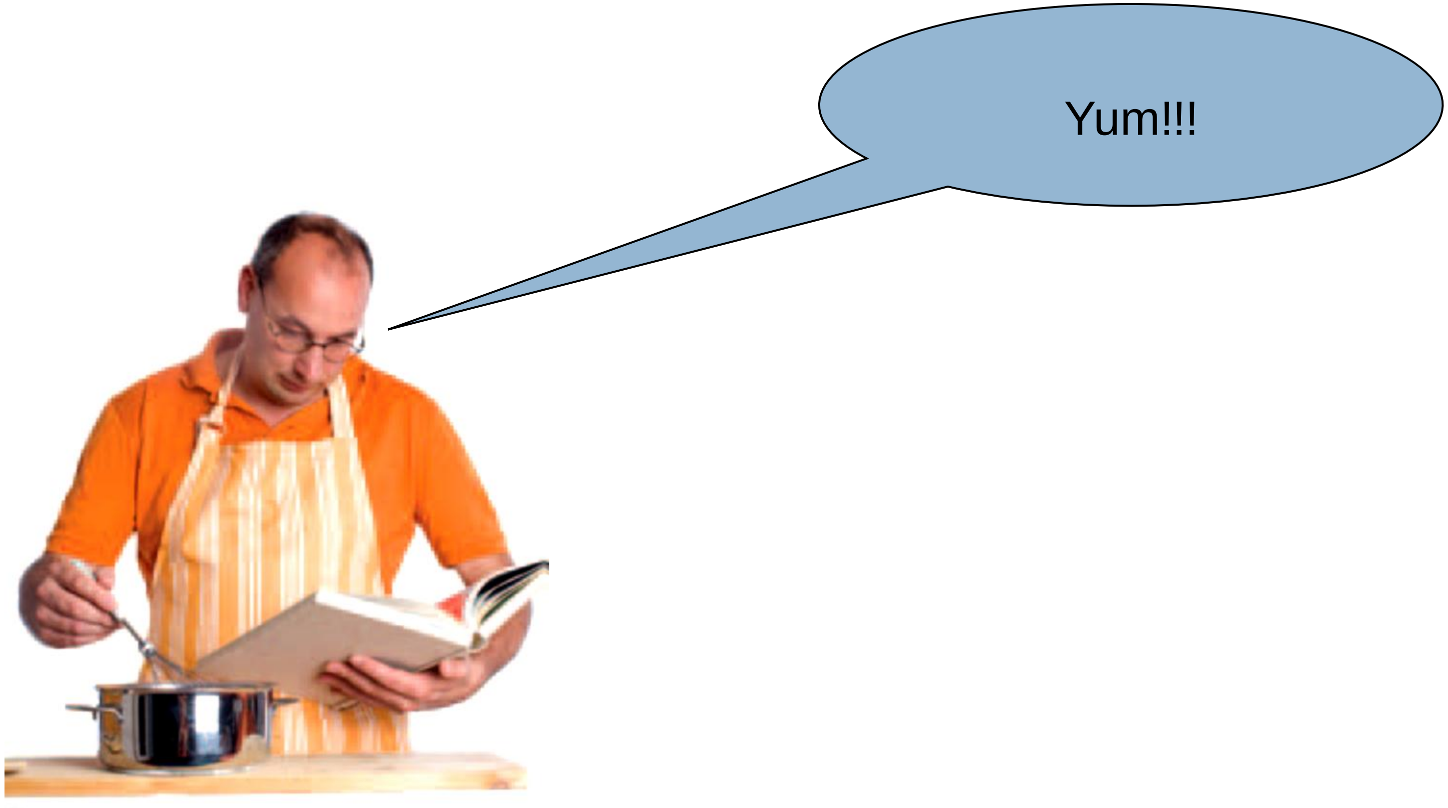
Computer scientists use the term

“syntactic sugar”

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

Yum!

Syntactic Sugar



Syntactic Sugar

That masked complex implementation detail:

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

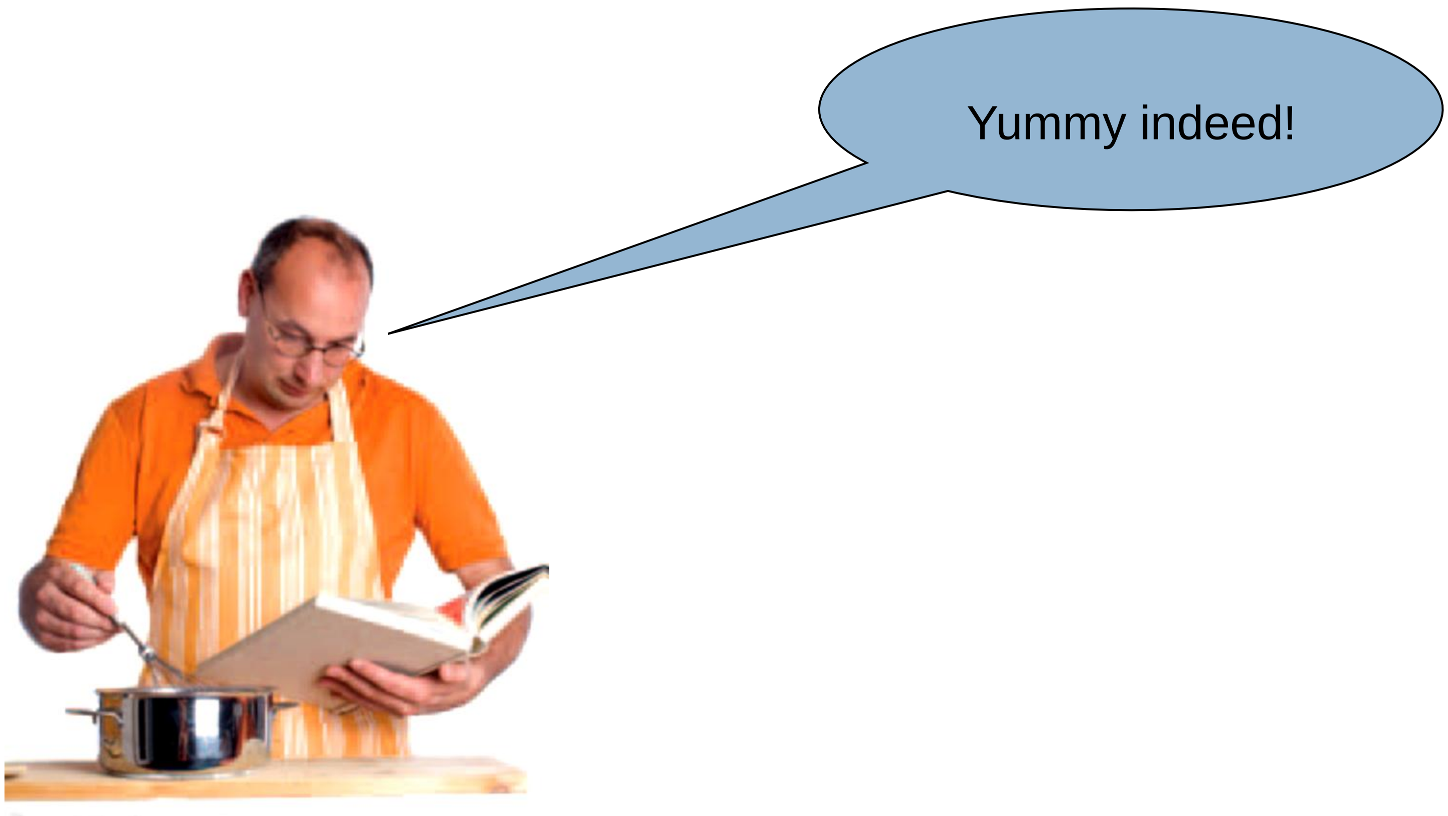
is how we should define the first parameter

but

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

looks a lot more like we are passing an array.

Syntactic Sugar



Syntactic Sugar

The is what the function would look like using pointer notation:

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

Arrays and Pointers

Table 2 Arrays and Pointers

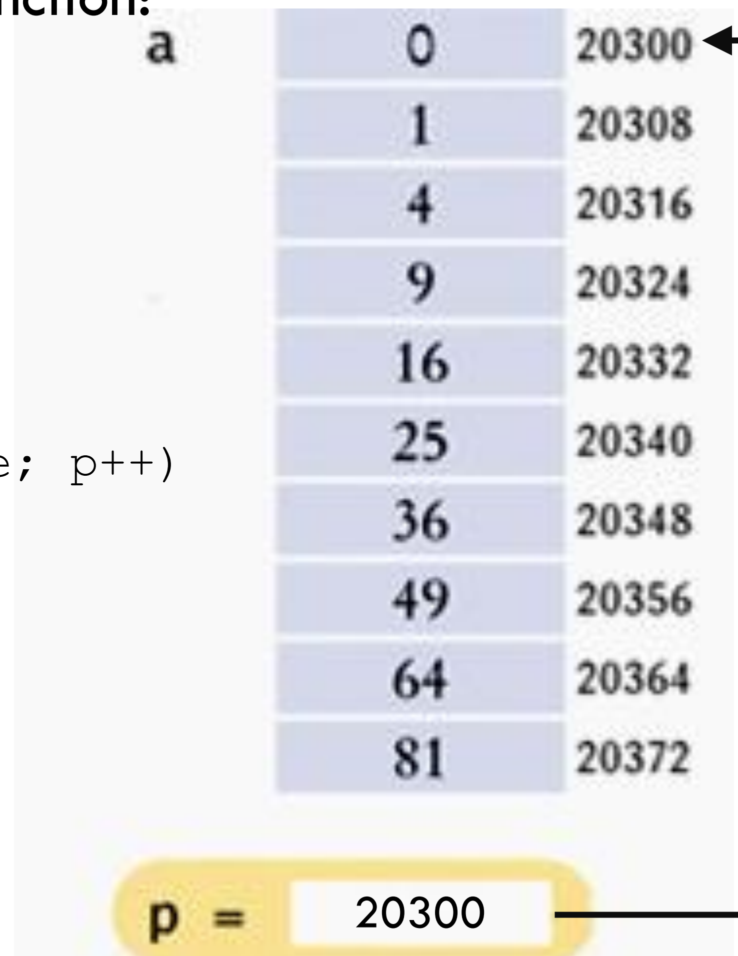
Expression	Value	Comment
`a`	20300	The starting address of the array, here assumed to be 20300.
`*a`	0	The value stored at that address. (The array contains values 0, 1, 4, 9, ...)
`a + 1`	20308	The address of the next `double` value in the array. A `double` occupies 8 bytes.
`a + 3`	20324	The address 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

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
     }
    return total;
}
```

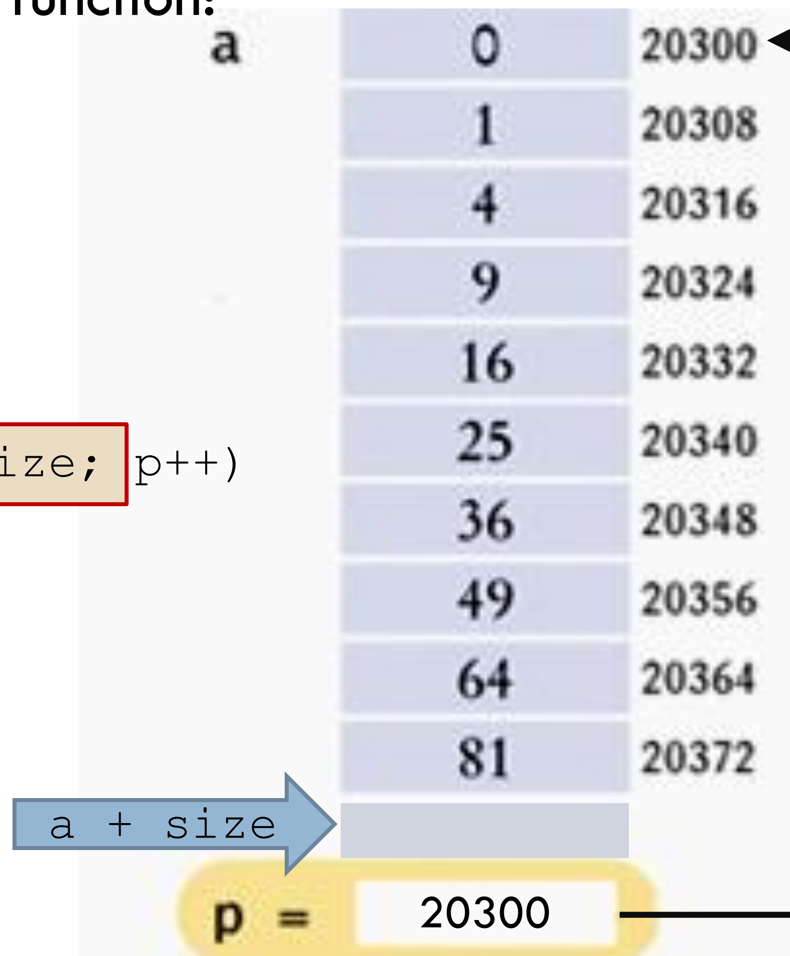


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
   double total = 0;

   // p starts at a[0]
   for (double* p = a; p < a + size; p++)
   {
      total = total + *p;
    }
   return total;
}
```

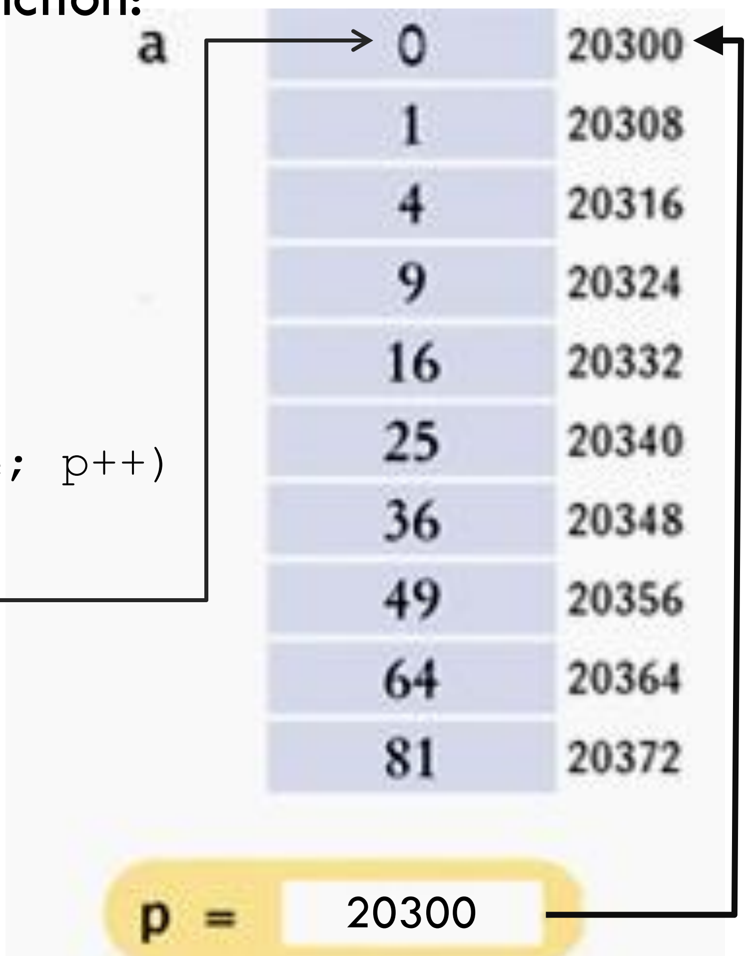


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
   double total = 0;

   // p starts at a[0]
   for (double* p = a; p < a + size; p++)
   {
      total = total + *p;
    }
   return total;
}
```

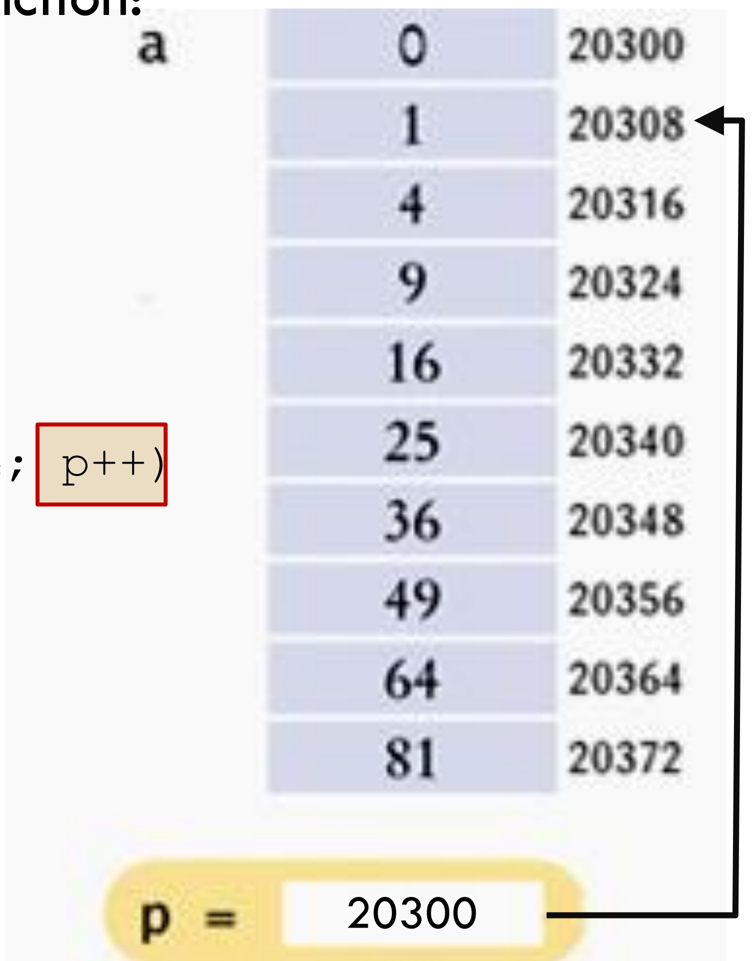


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
   double total = 0;

   // p starts at a[0]
   for (double* p = a; p < a + size; p++)
   {
      total = total + *p;
    }
   return total;
}
```

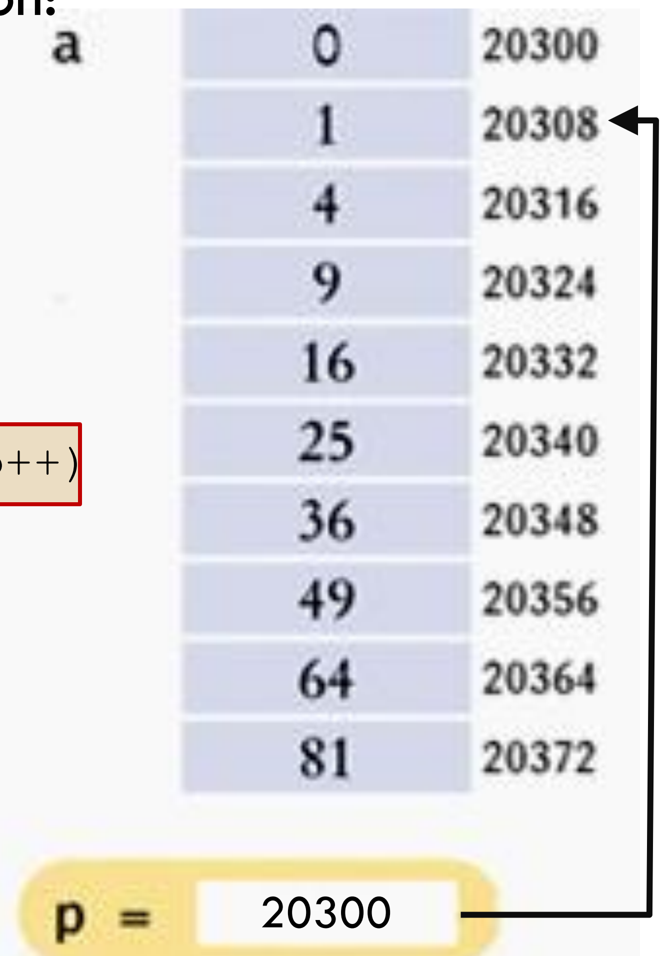


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
     }
    return total;
}
```

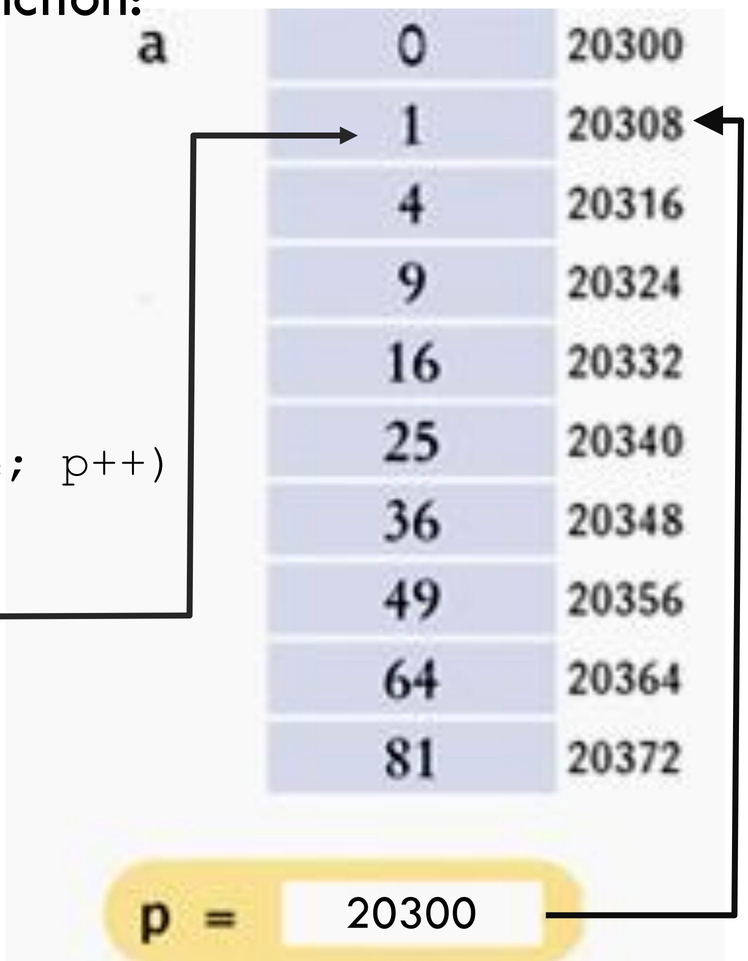


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
    }
    return total;
}
```

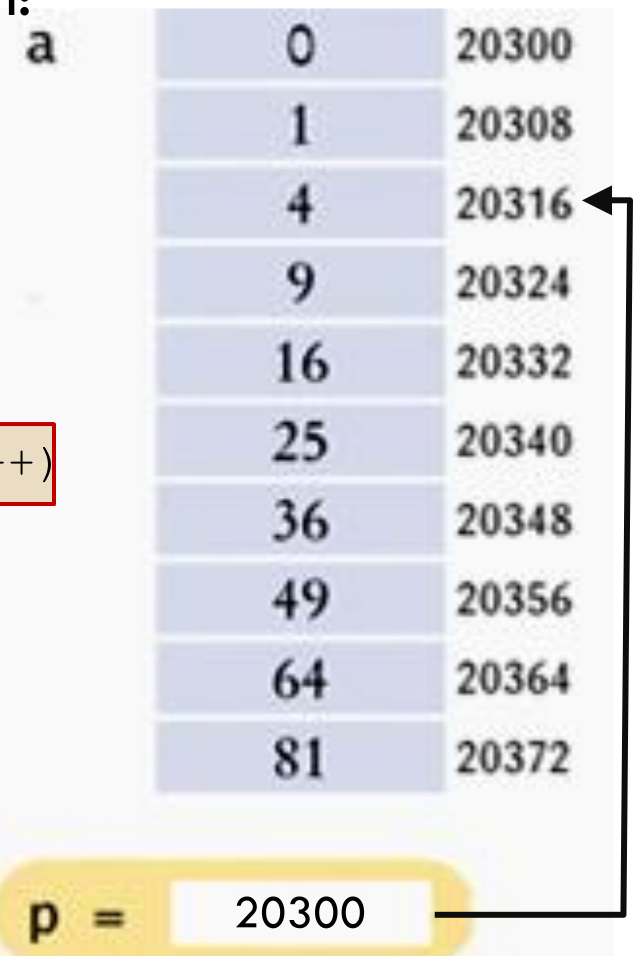


Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
       total = total + *p;
     }
    return total;
}
```



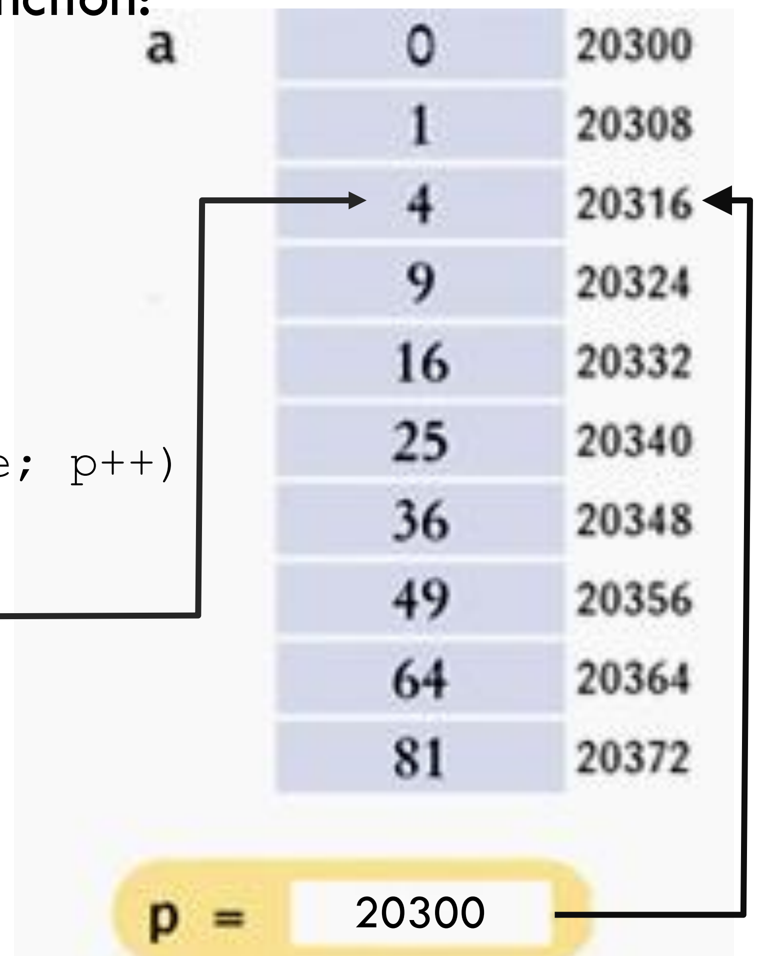
Using a Pointer to Step Through an Array

This is another way to implement the function:

```
double sum(double* a, int size)
{
    double total = 0;

    // p starts at a[0]
    for (double* p = a; p < a + size; p++)
    {
        total = total + *p;
     }
    return total;
}
```

Etcetera



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)
    {
        total = total + *p;
        p++;
        size--;
    }
```

could be written as:

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

This will confuse everybody you work with. Including yourself.

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 Local Pointer

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 Local Pointer

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

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

Common Error: Returning a Local Pointer

Or a pointer.
Remember the duality:

```
void firstlast(double* a, int size,
                     double* result)
{
   *result = *a;
   *(result+1) = *(a+ size - 1);
}
```

Not nearly as nice. Use some syntactic sugar.

Summary

- The name of an arrays is a pointer
- It points to the first element of the array.
- a[n] is identical to *(a + n), where a is a pointer into an array and n is an integer offset.
- Don’t try to be too clever.

C and C++ Strings

More things we didn't tell you before:

C++ has two mechanisms for manipulating strings.

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.
- 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.

Recap on characters

- The type char is used to store an individual character.
- Some of these characters are plain old letters and such as

  `'y', 'n', '3', '?'`

- Some of them are escape characters such as:

  `'\n', '\t', '\a'`

- And then there is the special character `'\0'` to denote the end of a string, the **null terminator**.

Characters

Table 3 Character Literals

`'y'`	The character `y`
`'0'`	The character for the digit `0`. In the ASCII code, `'0'` has the value 48.
`' '`	The space character
`'\n'`	The newline character
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a `char` value

C strings

- C *strings* are arrays of characters.
- Include `#include <cstring>`
- The null always the last character in a C string.
- Literal strings are always stored as character arrays
- Example:
  - "CAT" is really this sequence of characters: `'C' 'A' 'T' '\0'`
  - The null terminator character indicates the end of the C string
  - The literal C string "CAT" is actually an array of <u>four</u> chars stored somewhere in the computer.

Pop Quiz #1.

Q:

Is "C string" a string?

Yes

…wait…

No

…wait…

Pop Quiz #1

Answer:

"C string" is NOT an object of **`string`** type.

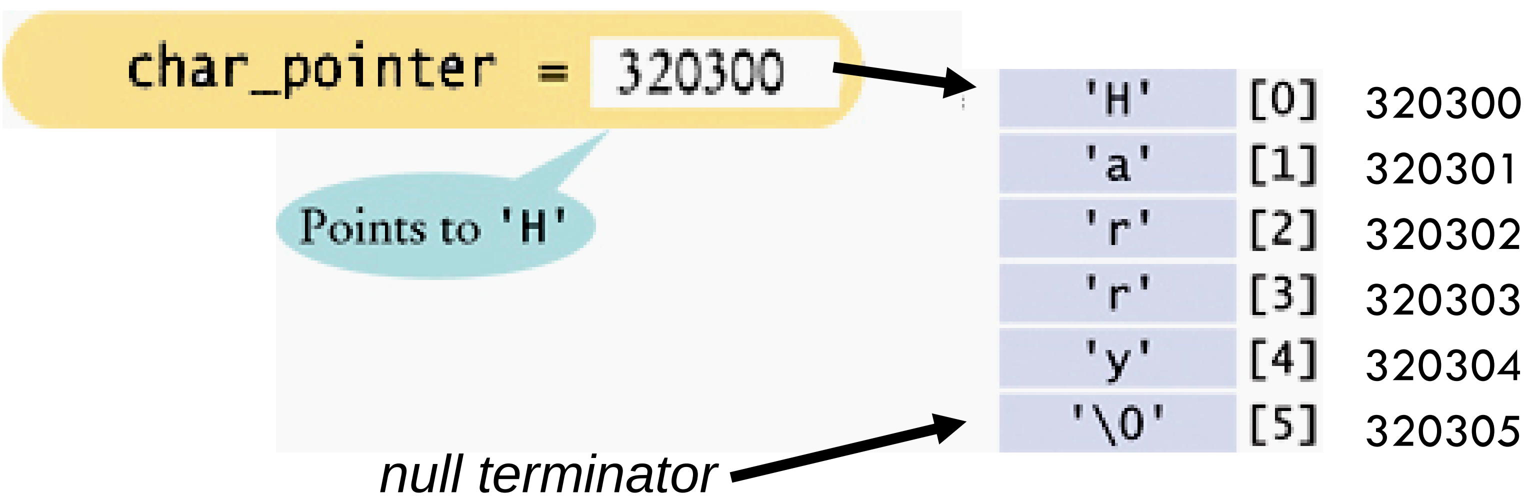
"C string" IS an **array of `chars`** with a null terminator character at the end.

"C string" is a **C string.**

Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry"; // Points to the 'H'
```



Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.

The strlen function returns the length of a C string.

```
#include <cstring>
int strlen(const char s[])
{
   int i = 0;
   // Count characters before the null terminator
   while (s[i] != '\0') {
          i++;
   }
   return i;
}
```

Using the Null Terminator Character

The call `strlen("Harry")` returns 5.

The null terminator character is not counted as part of the “length” of the C string – but it’s there.

Really, it is.

C String Functions

Table 4 C String Functions

In this table, s and t are character arrays; n is an integer.	
Function	**Description**
`strlen(s)`	Returns the length of s.
`strcpy(t, s)`	Copies the characters from s into t.
`strncpy(t, s, n)`	Copies at most n characters from s into t.
`strcat(t, s)`	Appends the characters from s after the end of the characters in t.
`strncat(t, s, n)`	Appends at most n characters from s after the end of the characters in t.
`strcmp(s, t)`	Returns 0 if s and t have the same contents, a negative integer if s comes before t in lexicographic order, a positive integer otherwise.

C String Functions

- Warning:
  - Many C string function have to be used with care.
  - If used incorrectly, they can allow attackers to write to your memory.
  - Many organizations advise to avoid functions such as strcpy altogether.
  - See for example:
    - http://cwe.mitre.org/data/definitions/676.html
    - http://cwe.mitre.org/data/definitions/120.html
    - http://cwe.mitre.org/data/definitions/170.html

C++ strings

- C++ has a <string>library defining the string class
- Include it in your programs when you wish to use strings:

```
#include <string>
```

- This library makes string processing easier than in C
- Notice there is no ".h" in the C++ string header; `<string.h>` is used for C-style strings
- You define a string object as follows:

```
string first_name ="Pete";
```

What's an object?

We'll tell in a few weeks

C++ strings

- The string library provides many useful functions for
  - manipulating string data,
  - comparing strings,
  - searching strings for characters and other strings,
  - tokenizing strings (separating strings into logical pieces),
  - determining the length of strings.

C++ strings

- Example: Concatenation
  - To concatenate two strings, we use the "+" operator

```
string str1= "Hi";
string str2= "5";
string str3 = str1 + str2;
cout<<"str3 = "<<str3<<endl; //displays Hi5
str3 += "!";
cout<<"str3 = "<<str3<<endl; //displays Hi5!
```

C++ strings

- Strings are compared using the following operators: ==, !=, <, <=, >, >=
  - The comparison uses the alphabetical order
  - The result is a Boolean value: *true* or *false*
  - The comparison works as long as at least one of the two arguments is a string object. The other string can be a string object, a C-style string (char array).
  - Example:

```
if(str == "Hi5!"){ … }
```

Comparing <cstring> and <string>

C Library Functions	C++ string operations
strcpy	=
strcat	+=
strcmp	= =, !=, <, >, <=, >=
strlen	.size( )
str[i]	str.substr(i,1);
…	…

Converting Between C and C++ Strings

- To convert a C++ string object to a C string you can use the member function `c_str` (use dot notation)
- Example

```
string cppstr = "Welcome";
char cstr[8]; // 7 for 'Welcome', plus 1 for '\0'
strcpy (cstr, cppstr.c_str());
```

Converting Between C and C++ Strings

- Converting from a C string to a C++ string is very easy:
  - You can just assign a C string (char array) to a string object.
  - Example: `string name = "Harry";`
  - `name` is initialized with the C string `"Harry"`.

Summary

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.
- 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.