



© Suzanne Tucker/iStockphoto.

Chapter Seven: Pointers and Structures

Chapter Goals

- To be able to declare, initialize, and use pointers
- To understand the relationship between arrays and pointers
- To be able to convert between string objects and character pointers
- To become familiar with dynamic memory allocation and deallocation
- To use structures to aggregate data items

Topic 1

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

**A variable *contains* a value,
but a *pointer* specifies *where* a value is located.**

A pointer denotes the
memory location of a variable

Pointer Usages

- In C++, pointers are important for several reasons.
 - Pointers allow sharing of values stored in variables in a uniform way
 - Pointers can refer to values that are allocated on demand (*dynamic memory allocation*)
 - Pointers are necessary for implementing *polymorphism*, an important concept in object-oriented programming (later)

Harry Needs a Banking Program

Harry wants a program to manage bank deposits and withdrawals.

```
... balance += depositAmount ...  
... balance -= withdrawalAmount ...
```

But not all deposits and withdrawals should be from the *same* bank.

By using a **pointer**,
it is possible to *switch* to a different account
without modifying the code for
deposits and withdrawals.

Pointers to the Rescue

Harry starts with a variable for his account balance.
It should be initialized to 0 since there is no money yet.

```
double harrys_account = 0;
```

If Harry anticipates that he may someday use other accounts, he can use a pointer to access any accounts.

So Harry also declares a pointer variable
named **account_pointer**:

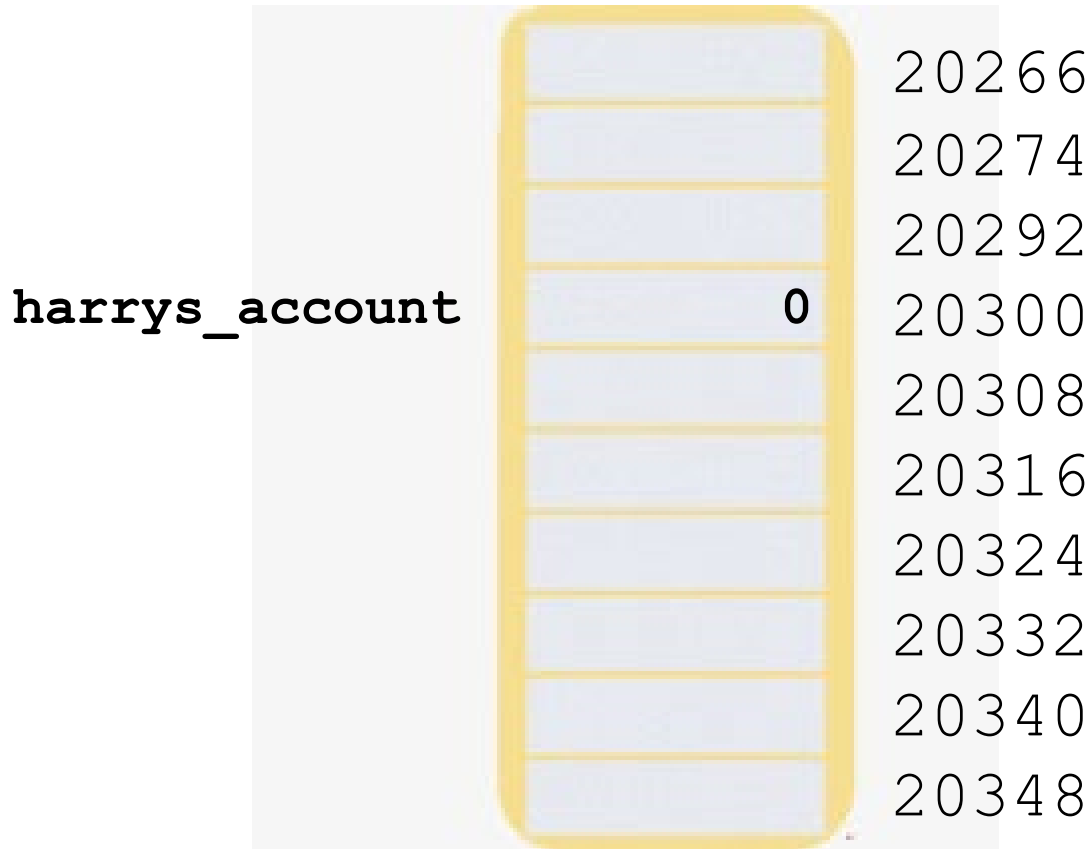
```
double* account_pointer;
```

The type of this variable is “pointer to double”.

Addresses and Pointers

Every byte in RAM has an address as pictured here (this small RAM block is addressed 20266 through 20348, shown in groups of eight bytes)

`harrys_account` as a double, happens to be located at address 20300.



Pointer Initialization

When Harry declares a pointer variable, he initializes it to point to `harrys_account`:

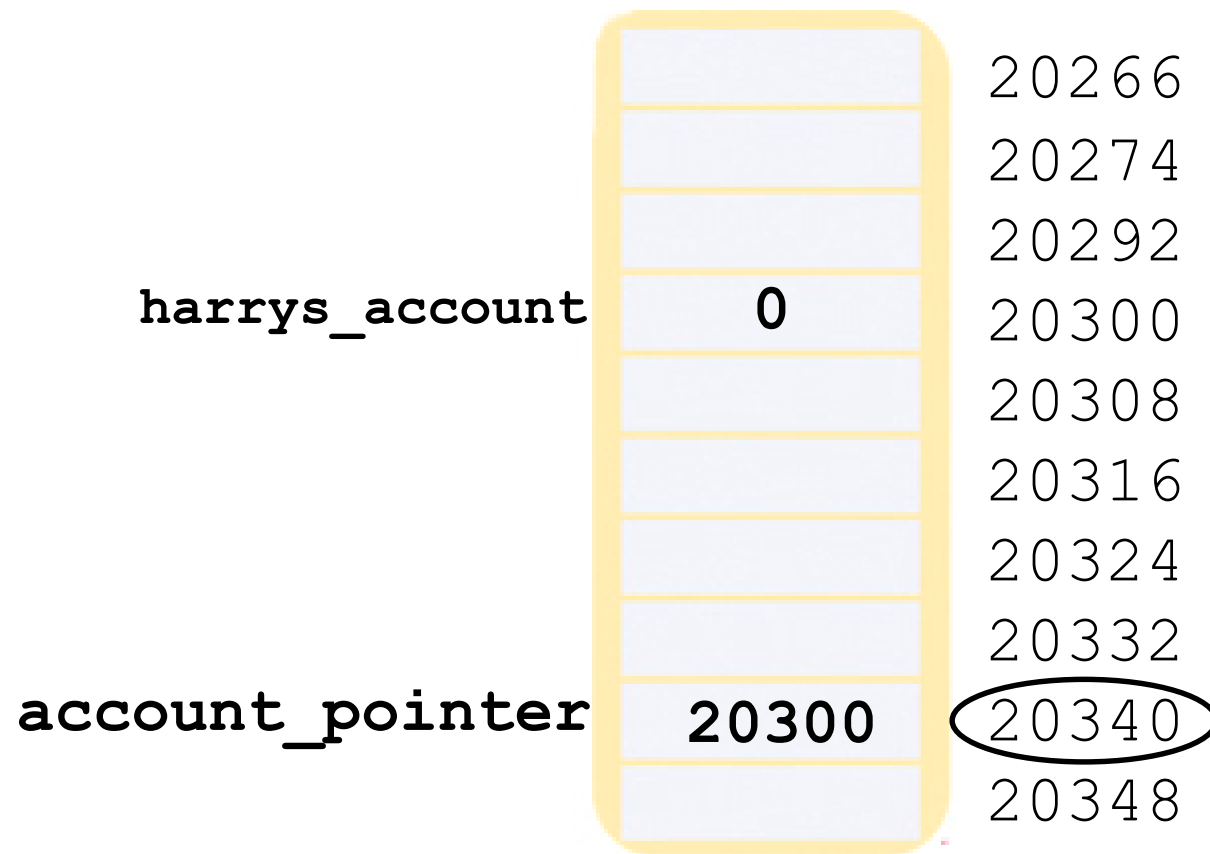
```
double harrys_account = 0;  
double* account_pointer = &harrys_account;
```

- The `&` operator yields the location (address) of a variable.
- Taking the address of a `double` variable yields a value of type `double*` so everything fits together nicely.

`account_pointer` now contains the address of `harrys_account`

Pointers Also Reside in RAM

And, of course, `account_pointer` is *somewhere* in RAM, though we really don't care where it is:



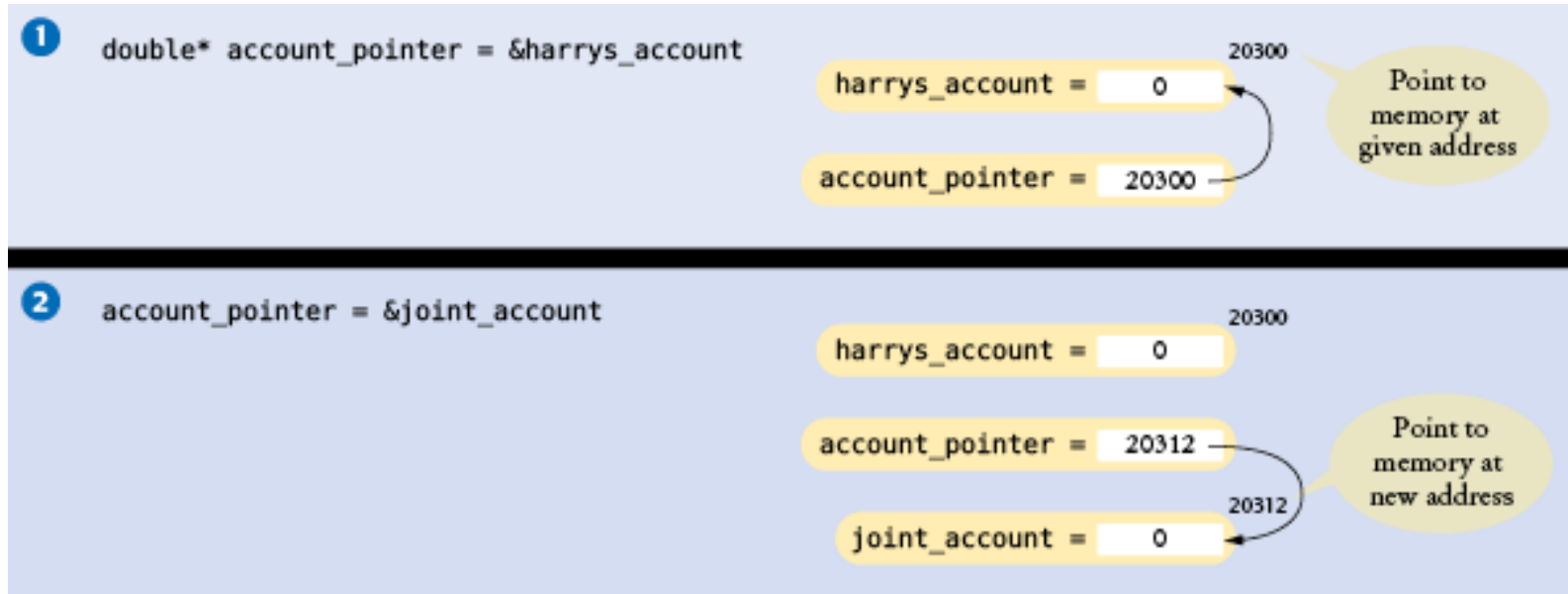
Addresses and Pointers

Harry wanted to use his account, but he found the balance was zero:

```
double harrys_account = 0;  
account_pointer = &harrys_account; //Picture #1  
double joint_account = 1000;
```

To access his joint account hoping it still has a non-zero balance, Harry would change the pointer:

```
account_pointer = &joint_account; //Picture #2
```



Addresses and Pointers – and ARROWS

Do note that the computer stores numbers,
not arrows.

Accessing the Memory Pointed to by A Pointer Variable

The "dereferencing operator" `*` lets you use a pointer to get the data. Use `*account_pointer` as a substitute for the name of the variable the pointer points to:

```
// display the current balance
cout << *account_pointer << endl;
```

It can be used on the left and/or the right of an assignment:

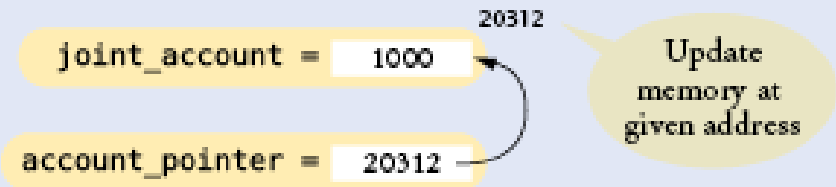
```
// withdraw $100
*account_pointer = *account_pointer - 100;
```

Harry Makes the Deposit

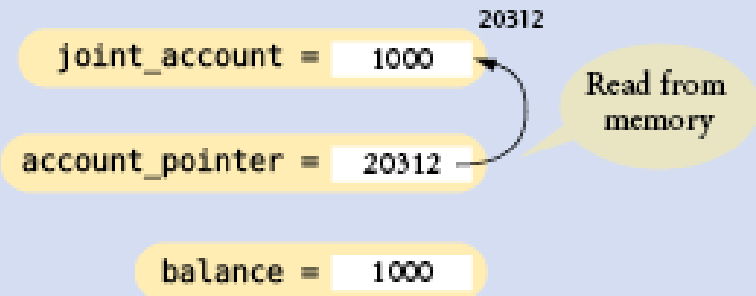
```
// deposit $1000
```

```
*account_pointer = *account_pointer + 1000;
```

1 `*account_pointer = 1000`



2 `balance = *account_pointer`



Pointer Syntax Examples: Table 1, part 1

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;	p gets 20304	Set p to the address of n.
*p	20	The value stored at the changed address.
m = *p;	m gets 20	Stores 20 into m.

Pointer Syntax Examples: Table 1, part 2: Bad Syntax

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
m = p;	Error	m is an <code>int</code> value; p is an <code>int*</code> pointer. The types are not compatible.
&10	Error	You can only take the address of a variable.
&p	The address of p, perhaps 20308	Warning: This is the location of a pointer variable, not the location of an integer. You almost never want to use the address of a pointer variable.
double x = 0; p = &x;	Error	p has type <code>int*</code> , &x has type <code>double*</code> . These types are incompatible.

Errors Using Pointers – Uninitialized Pointer Variables

When a pointer variable is first defined, it is a random address. Using that pointer (and its random address) is an **error**, until the pointer has been initialized.

```
double* account_pointer; // Forgot to initialize  
*account_pointer = 1000; // ERROR! account_pointer  
// contains an unpredictable value, program crashes
```

If you don't already know what the pointer will point to, initialize it with **nullptr**:

```
double* account_pointer = nullptr;
```

Trying to access data through a `nullptr` pointer will cause your program to terminate (but more gracefully than an uninitialized pointer would).

Harry's Banking Program, part 1

```
// Here is the complete banking program
#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

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

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

Harry's Banking Program, part 2

```
// 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 (of joint account)
cout << "Balance: " << *account_pointer << endl;

return 0;

}
```

Practice It

Two groups jointly charter a bus and fill it with travelers. A variable

```
int count = 0;
```

is to be accessed through two pointers `p` and `q`.

1. Declare the pointer variable `p`. Do not initialize:
2. Initialize `p` with the address of `count`:
3. Complete this statement to check whether there is space in the bus for another passenger, using the pointer `p`:
 - `if (_____ < CAPACITY)`
4. Increment the value to which `p` points, using `++`:
5. Declare the pointer variable `q` and initialize it with `p`:

Practice It More

Show the output of each of these code snippets. Answer "?" if the output cannot be determined:

```
int a = 1;
int b = 2;
int* p = &a;
cout << *p << " ";
p = &b;
cout << *p << endl;
```

```
int a = 15;
int* p = &a;
int* q = &a;
cout << *p + *q << endl;
```

```
int a = 15;
int* p = &a;
cout << *p << " " << p << endl;
```

Common Error: Confusing Data And Pointers: 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.

Error: Multiple Pointers Defined in a Single Statement

It is legal 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;
```

Alternatively, you can move the `*` next to the variable name:

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

Function Arguments: Pointers vs. References

Recall that the & symbol is used for reference parameters:

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

A call of this function would be:

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

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 feed the function an address (pointer variable or reference):

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


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

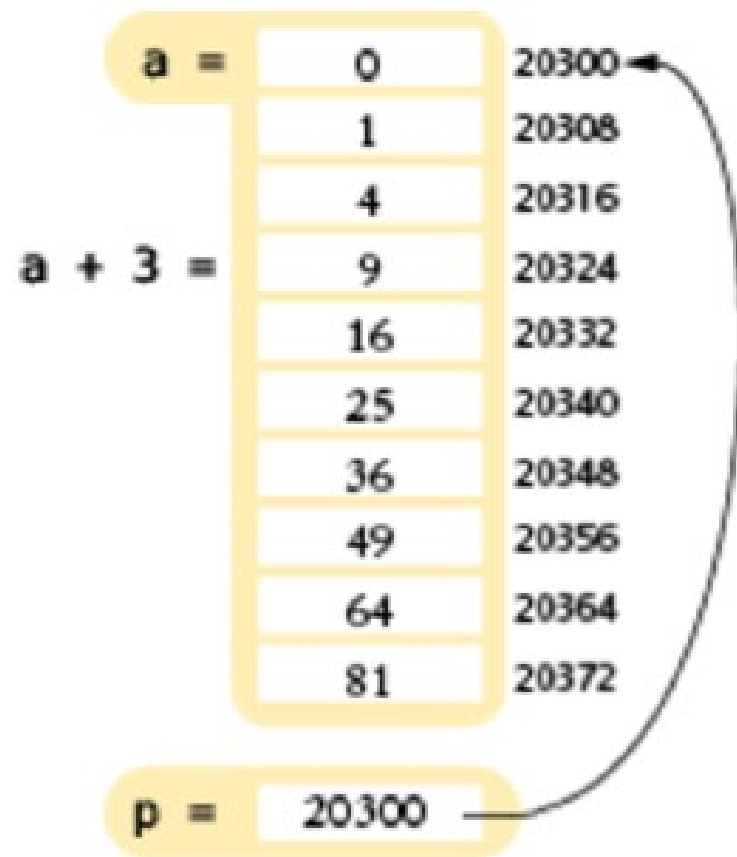
Pointers can help explain the peculiarities of arrays.

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:

```
double* 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;
```

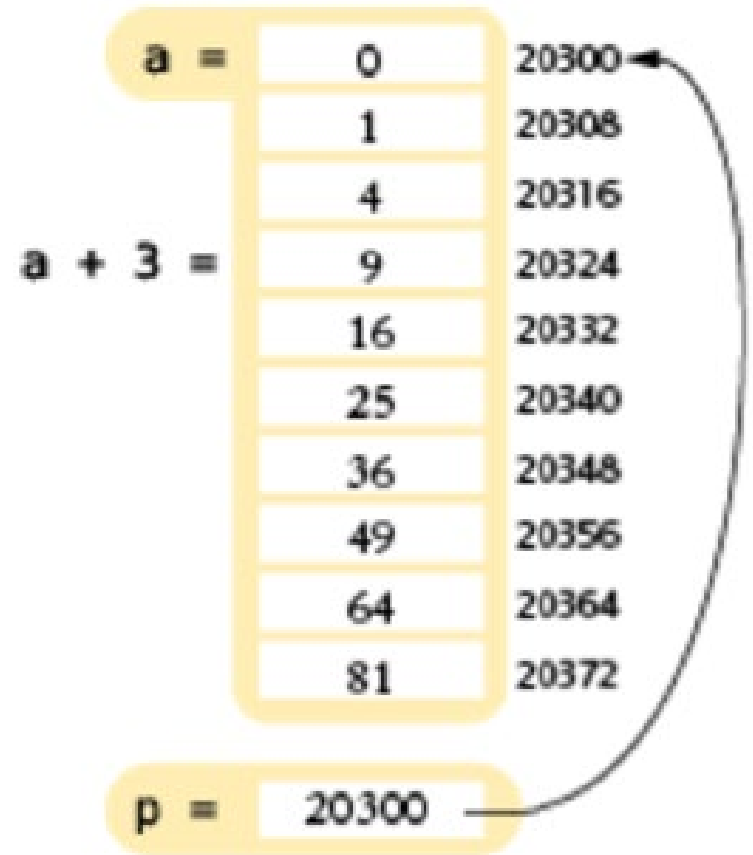
`p + 3` is a pointer to the array element with index 3

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 <code>double</code> value in the array. A <code>double</code> 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 <code>*(a + 3)</code> by array/pointer duality.
*a + 3	3	The sum of <code>*a</code> and 3. Because there are no parentheses, the <code>*</code> refers only to <code>a</code> .
&a[3]	20324	The address of the element with index 3, the same as <code>a + 3</code> .

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 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.

Topic 3

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

C++ has two mechanisms for manipulating strings.

The `string` class

- Supports character sequences of arbitrary length.
- Provides convenient operations: concatenation (+), comparison (`==`, `<`, `>`)

C strings

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

char Type

The type `char` is an individual character.

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

```
char three = '3'; // not binary 3,  
// but the printer-printable ASCII char for  
// the numeral. It happens to be binary  
// 0110 0011 = 51 decimal!
```

Special Characters

'\n': **newline**

'\a': *alert* character – rings the bell

'\t': **horizontal tab**

'\0': **null character (binary zero)**

Etc...

These are still single (individual) characters:
the ***escape sequence*** characters. They control functions
for a display or printer.

Character Literal Examples: Table 3

'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, but a <code>char</code> array of 2 characters (includes a string terminator)

The Null Terminator Character and C Strings

The null character is special to C strings because it is always the last character in them:

"CAT" is really 4 characters, not 3:

'C' 'A' 'T' '\0'

The null terminator character indicates the end of a C string.
Literal strings are always stored as
character arrays.

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'
```

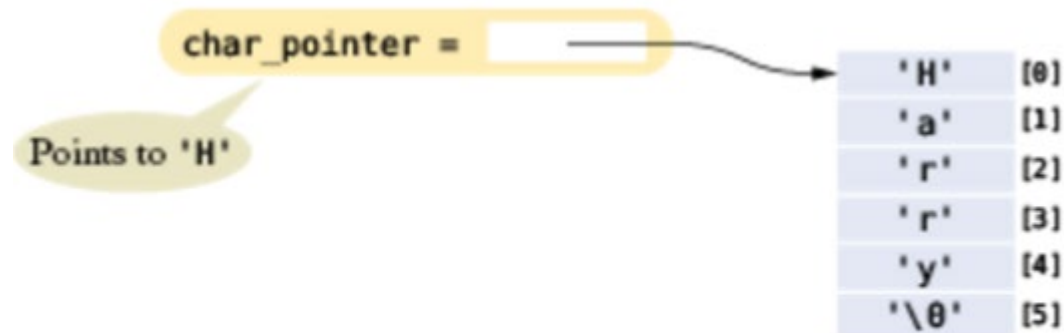


Figure 6 A Character Array

Using the Null Terminator Character

Functions that operate on C strings rely on this terminator. The `strlen` function (from the `<cstring>` library) returns the length of a C string. Here is the source:

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

The call `strlen("Harry")` returns 5. The null terminator character is not counted as part of the “length” of the C string

Character Arrays

If you want to modify the characters in a C string, define a character array to hold the characters.

For example:

```
// An array of 6 characters  
char c[] = "Harry";
```

You can modify the characters in the array:

```
c[0] = 'L'; //now c is Larry
```

Converting Between C and C++ Strings

The `cstdlib` header includes the function:

```
int atoi(const char s[])
```

The `atoi` function returns an `int` equivalent to a character array containing digits:

```
char* year = "2012";  
int y = atoi(year);
```

`y` is the integer 2012

`c_str()` Function converts C++ string to a C string

Older versions of the C++ `<string>` library lack an `atoi()`.

The `c_str` member function offers an “escape hatch”,
converting the C++ `string` to a `char` array:

```
string year = "2012";  
int y = atoi(year.c_str());
```

Again, `y` is the integer 2012

Converting a C string to a C++ string

Converting from a C string to a C++ string is easy – the assignment operator (=) does it:

```
string name = "Harry";
```

```
Char* fred = "Fredrick";
```

```
string name2 = fred;
```

C++ Strings and the `[]` Operator

You can access individual characters in either a C-string or C++ string with the `[]` operator:

```
string name = "Harry";  
name[3] = 'd'; //name now is Hardy
```

```
char c[] = "Mary";  
c[3] = 'k'; // now is Mark
```

Example: Converting Case of a C++ string with toupper()

The `toupper` function is defined in the `<cctype>` header. It converts a lowercase `char` to uppercase. The `tolower` function does the opposite.

You can write a function that will return the uppercase version of a C++ `string`:

```
/**
 * Makes an uppercase version of a string.
 * @param str a string
 * @return a string with the characters in str converted to uppercase
 */
string uppercase(string str)
{
    string result = str; // Make a copy of str
    for (int i = 0; i < result.length(); i++)
    {
        // Convert each character to uppercase
        result[i] = toupper(result[i]);
    }
    return result;
}
```

C String Functions from the `<cstring>` Library: Table 4

In this table, s and t are character arrays; n is an integer.	
Function	Description
<code>strlen(s)</code>	Returns the length of s.
<code>strcpy(t, s)</code>	Copies the characters from s into t.
<code>strncpy(t, s, n)</code>	Copies at most n characters from s into t.
<code>strcat(t, s)</code>	Appends the characters from s after the end of the characters in t.
<code>strncat(t, s, n)</code>	Appends at most n characters from s after the end of the characters in t.
<code>strcmp(s, t)</code>	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++ strings are usually easier than the <cstring> Functions

Consider the task of concatenating 2 names into a string.
The string class makes this easy:

```
string first = "Harry";  
string last = "Smith";  
string name = first + " " + last;
```

With C strings, it is much harder, as we have to worry about sizes of the arrays:

```
const int NAME_SIZE = 40;  
char name[NAME_SIZE];  
strncpy(name, first, NAME_SIZE - 1);  
int length = strlen(name);  
if (length < NAME_SIZE - 1)  
{  
    strcat(name, " ");  
    int n = NAME_SIZE - 2 - length;  
    // Leave room for space, null terminator  
    if (n > 0)  
    {  
        strncat(name, last, n);  
    }  
}
```

Topic 4

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

Dynamic Memory Allocation

You may not know beforehand how many values you need in an array.

To solve this problem, use dynamic memory allocation and ask the C++ run-time system to create new values whenever you need them.

The run-time system keeps a large storage area, called the **free store** or **heap**, that can allocate values and arrays of any type:

```
double *p = new double[n];
```

allocates an array of size *n*, and yields a pointer to the starting element. (Here *n* need not be a constant.)

Dynamic Memory Allocation Examples

You need a pointer variable to hold the pointer you get:

```
//get a single variable  
double* account_pointer = new double;
```

```
//get an array variable  
double* account_array = new double[n];
```

Now you can use `account_array` as an array.

The magic of array/pointer duality
lets you use the array notation
`account_array[i]` to access the `i`th element.

Dynamic Memory Allocation: `delete`

When your program no longer needs the memory that you asked for with the **`new`** operator, you must return it to the heap using the **`delete`** operator for single areas of memory (which you would probably never use anyway).

```
delete account_pointer;  
delete[] account_array;
```


Don't Use a Pointer after delete


After you delete a memory block,
you can no longer use it.

The OS is very efficient – and quick – “your” storage
space may already be used elsewhere.

```
delete[] account_array;  
account_array[0] = 1000;  
    // NO! You no longer own the  
    // memory of account_array
```

Dynamic Memory Allocation – Resizing an Array

account_array = 

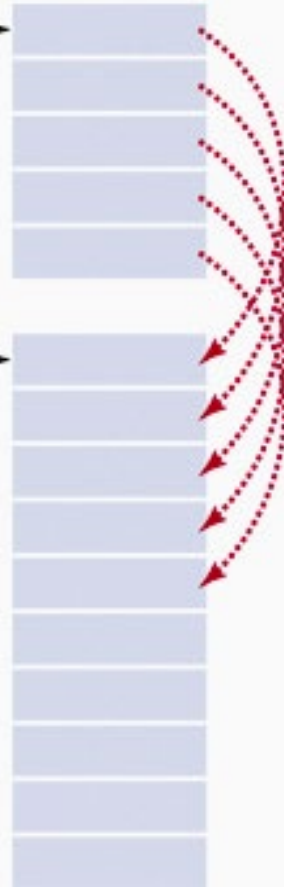
bigger_array = 

Unlike static arrays, you can change the size of a dynamic array.

Make a new, bigger array and copy the old data:

// **n** = size of the original array

```
double* bigger_array = new double[2 * n];  
for (int i = 0; i < n; i++)  
{  
    bigger_array[i] = account_array[i];  
}  
delete[] account_array;  
account_array = bigger_array;  
n = 2 * n;
```



Dynamic Memory Allocation – THE RULES

1. Every call to `new` *must* be matched by exactly one call to `delete`.
2. Use `delete[]` to delete arrays.
And always assign `NULL` to the pointer after that.
3. Don't access a memory block (don't use the pointer) after it has been deleted.

If you don't follow these rules, your program can
crash or run unpredictably

or worse...

Dynamic Memory Allocation – Common Errors: Table 5

Statements	Error
<pre>int* p; *p = 5; delete p;</pre>	There is no call to <code>new int</code> .
<pre>int* p = new int; *p = 5; p = new int;</pre>	The first allocated memory block was never deleted.
<pre>int* p = new int[10]; *p = 5; delete p;</pre>	The <code>delete[]</code> operator should have been used.
<pre>int* p = new int[10]; int* q = p; q[0] = 5; delete p; delete q;</pre>	The same memory block was deleted twice.
<pre>int n = 4; int* p = &n; *p = 5; delete p;</pre>	You can only delete memory blocks that you obtained from calling <code>new</code> .

Common Error: Dangling Pointers

It is a run-time error to use a pointer that points to memory that has already been deleted.

Such a pointer is called a **dangling pointer**.

Because the freed memory will be reused for other purposes, you can do real damage with a dangling pointer. For example:

```
int* values = new int[n];  
// Process values
```

```
delete[] values; //values now dangling
```

```
// Some other work
```

```
values[0] = 42; //ERROR
```

Avoiding Dangling Pointers

To prevent a **dangling pointer**, assign the special value **nullptr**

To any pointer that you delete:

```
int* values = new int[n];  
// Process values
```

```
delete[] values; //values now dangling
```

```
values = nullptr; //makes pointer safe
```


Common Error: Memory Leaks

A memory block that is never deallocated is called a memory leak.

If you allocate a few small blocks of memory and forget to deallocate them, this is not a huge problem.

When the program exits, all allocated memory is returned to the operating system.

Every call to `new` should have a matching call to `delete`.

But if your program runs for a long time, or if it allocates lots of memory (perhaps in a loop) without the `deletes`, then it can run out of memory and crash.

Topic 5

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 Vectors of Pointers

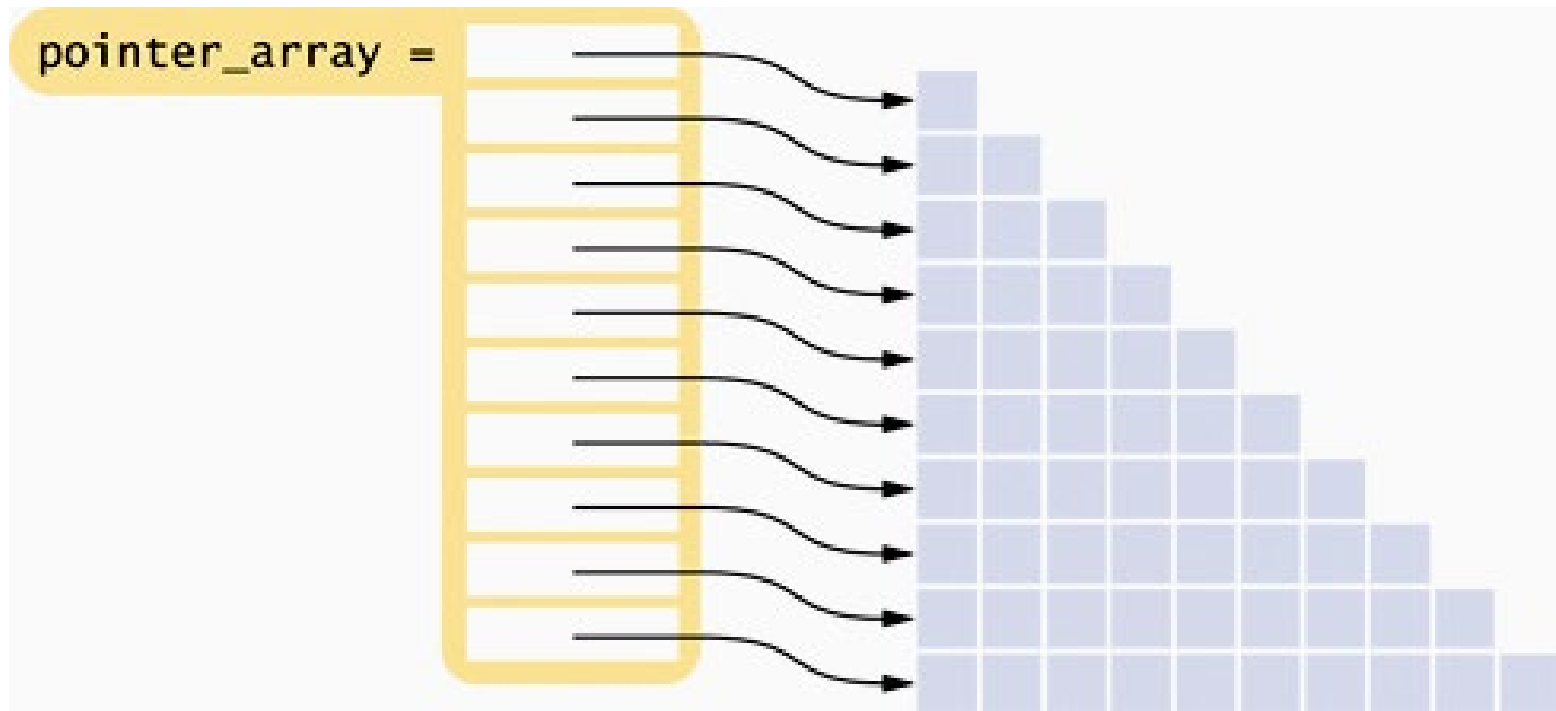
When you have a sequence of pointers,
you can place them into an array or vector.

An array and a vector of ten `int*` pointers are defined as

```
int* pointer_array[10];
```

```
vector<int*> pointer_vector(10);
```

Arrays and Vectors of Pointers – A Triangular Array



In this array, each row is a different length. It would be inefficient to use a two-dimensional array, because almost half of the elements would be wasted

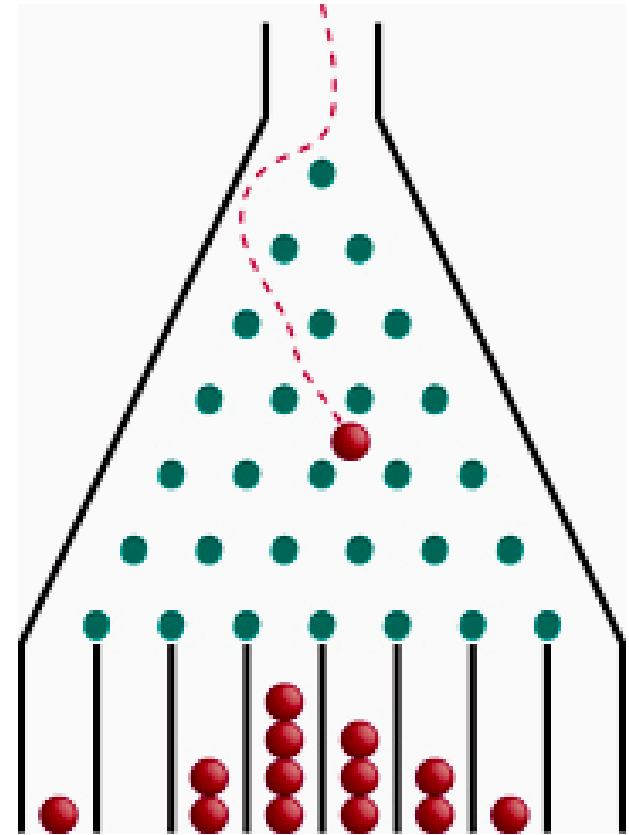
Program Example: A Galton Board

A Galton board consists of a pyramidal arrangement of pegs and a row of bins at the bottom.

Balls are dropped onto the top peg and travel toward the bins.

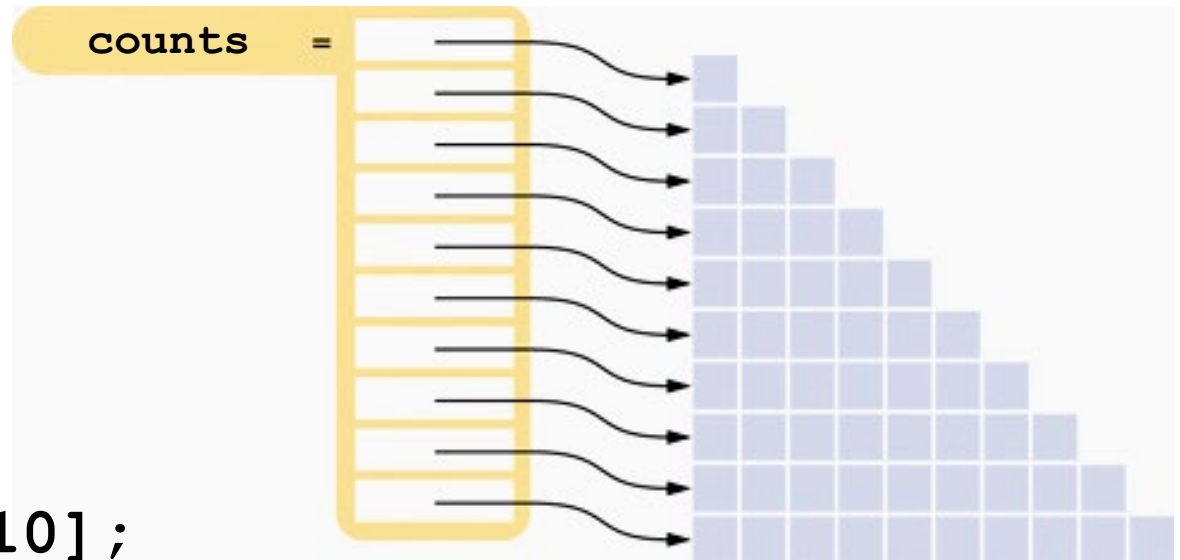
At each peg, there is a 50 percent chance of moving left or right.

The ball counts in the bins approximate a bell-curve distribution.



A Galton Board Simulation

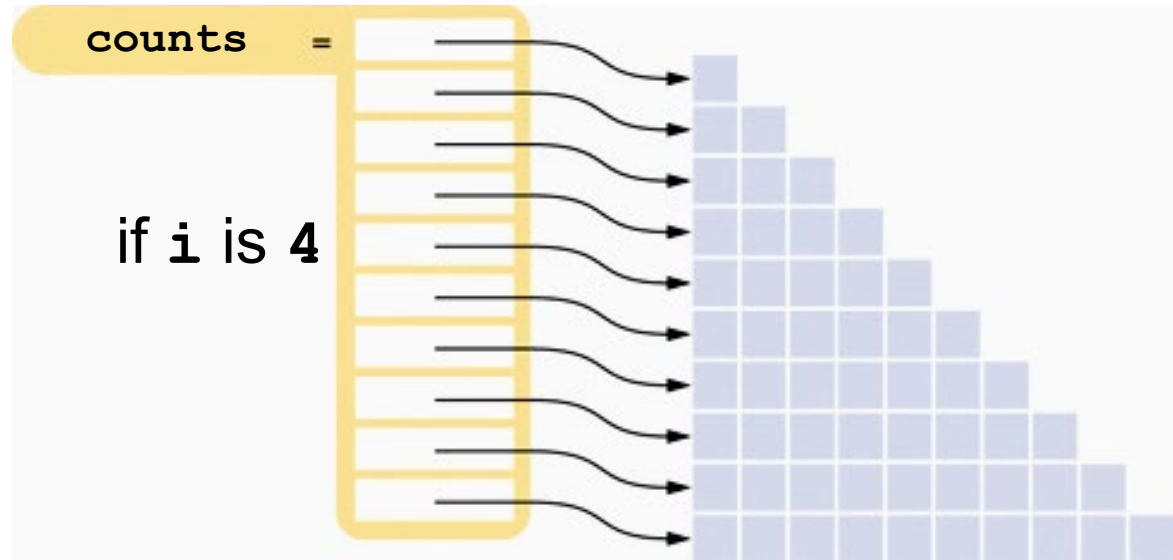
We will simulate a board with ten rows of pegs.
Each row requires an array of counters.
The following statements initialize the triangular array:



```
int* counts[10];  
for (int i = 0; i < 10; i++)  
{  
    counts[i] = new int[i + 1];  
}
```

A Galton Board Simulation: Printing Rows

We will need to print each row:

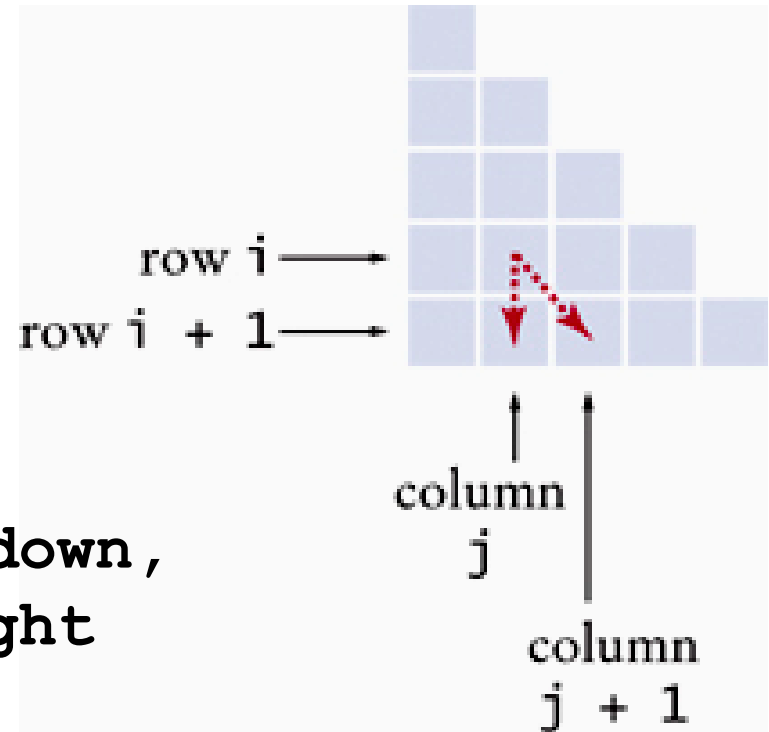


```
// print all elements in the ith row
for (int j = 0; j <= i; j++)
{
    cout << setw(4) << counts[i][j];
}
cout << endl;
```

A Galton Board Simulation: Ball Bouncing on Pegs

We will simulate a ball bouncing through the pegs:

```
int r = rand() % 2;  
// If r is even, move down,  
// otherwise to the right  
if (r == 1)  
{  
    j++;  
}  
counts[i][j]++;
```



A Galton Board Simulation: Complete Code Part 1

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand(time(0));
    int* counts[10];

    // Allocate the rows
    for (int i = 0; i < 10; i++)
    {
        counts[i] = new int[i + 1];
        for (int j = 0; j <= i; j++)
        {
            counts[i][j] = 0;
        }
    }
}
```

A Galton Board Simulation: Complete Code Part 2

```
const int RUNS = 1000;
// Simulate 1,000 balls
for (int run = 0; run < RUNS; run++)
{
    // Add a ball to the top
    counts[0][0]++;
    // Have the ball run to the bottom
    int j = 0;
    for (int i = 1; i < 10; i++)
    {
        int r = rand() % 2;
        // If r is even, move down,
        // otherwise to the right
        if (r == 1)
        {
            j++;
        }
        counts[i][j]++;
    }
}
```

A Galton Board Simulation: Complete Code Part 3

```
// Print all counts
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j <= i; j++)
    {
        cout << setw(4) << counts[i][j];
    }
    cout << endl;
}

// Deallocate the rows
for (int i = 0; i < 10; i++)
{
    delete[] counts[i];
}

return 0;
}
```

A Galton Board Simulation: Results

This is the output from a run of the program, with each number being a count of the balls that hit that peg in the triangle.

Note the bell-curve distribution of balls on the "bottom line":

1000										
480	520									
241	500	259								
124	345	411	120							
68	232	365	271	64						
32	164	283	329	161	31					
16	88	229	303	254	88	22				
9	47	147	277	273	190	44	13			
5	24	103	203	288	228	113	33	3		
1	18	64	149	239	265	186	61	15	2	

Topic 6

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

Problem Solving with Pointer Pictures

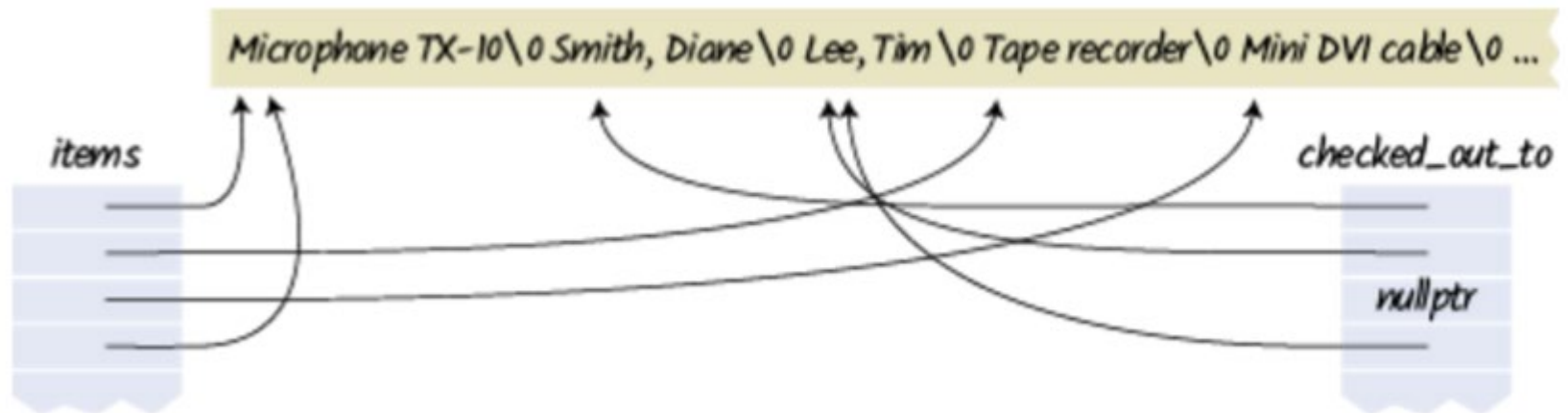
- When designing programs that use pointers, you want to visualize how the pointers connect the data.
 1. Draw the data blocks that will be accessed or modified through the pointers.
 2. Then draw the pointer variables.
 3. Finally, draw the pointers as arrows between those blocks. You may need to draw several diagrams that show how the pointers change.

Problem Solving with Pointer Pictures: Example

The media center loans out equipment (microphones, cables, and so on)

We want to track the name of each item, and the name of the user.

- All equipment and user names are in a long array of characters. New names are added to the end as needed.
- Pointers to the equipment names are stored in an array of pointers called `items`.
- A parallel array `checked_out_to` of pointers to user names. Sometimes, items can be checked out to the same user. Other items aren't checked out at all—the user name pointer is `nullptr`.



Embedded Systems

- An **embedded system** is a computer system that controls a device.
 - a processor and other hardware controlled by a computer program.
 - Unlike a personal computer, which is flexible and runs many different computer programs, the embedded system is tailored to a specific device.
 - increasingly common, in routers, washing machines, medical equipment, cell phones, automobiles, and spacecraft.
 - Probably programmed in C or C++
- Unlike PCs, embedded systems are:
 - cost sensitive: sold in quantities of millions for low prices, having little memory and slow processor
 - mission critical: most must be reliable and bug-free, as changing their program code is non-trivial and can mean life/death in the case of cars and spacecraft
 - optimized to a particular task, which may require significant code streamlining to adapt a cheap CPU to a real-time need

Topic 7

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

Structures: User-defined Mixed Data Types

- To group values of a single type together under a shared name, use an array or `vector`
- To group different types together with one name, use a structured type
 - Like arrays, pointers prove quite useful with structures
- Define a structure type with the `struct` reserved word:

```
struct StreetAddress //has 2 members
{
    int house_number; //first member
    string street_name;
};
```

```
StreetAddress white_house; //defines a variable of the type
```

```
// You use the "dot notation" to access members
white_house.house_number = 1600;
white_house.street_name = "Pennsylvania Avenue";
```

Structures: Assignment, but No Comparisons

Use the = operator to assign one structure value to another. All members are assigned simultaneously.

```
StreetAddress dest;  
dest = white_house;
```

is equivalent to

```
dest.house_number = white_house.house_number;  
dest.street_name = white_house.street_name;
```

However, you cannot compare two structures for equality.

```
if (dest == white_house) // Error
```

You must compare individual members to compare the whole struct:

```
if (dest.house_number == white_house.house_number  
    && dest.street_name == white_house.street_name) // Ok
```

Structure Initialization

- Structure variables can be initialized when defined, similar to array initialization:

```
struct StreetAddress
{
    int house_number;
    string street_name;
};
```

```
StreetAddress white_house = {1600, "Pennsylvania Avenue"}; //
initialized
```

The initializer list must be in the same order as the structure type definition.

Functions and struct

Structures can be function arguments and return values.

For example:

```
void print_address(StreetAddress address)
{
    cout << address.house_number << " " << address.street_name;
}
```

A function can return a structure. For example:

```
StreetAddress make_random_address()
{
    StreetAddress result;
    result.house_number = 100 + rand() % 100;
    result.street_name = "Main Street";
    return result;
}
```

Arrays of Structures

You can put structures into arrays. For example:

```
StreetAddress delivery_route[ROUTE_LENGTH];  
delivery_route[0].house_number = 123;  
delivery_route[0].street_name = "Main Street";
```

You can also access a structure value in its entirety, like this:

```
StreetAddress start = delivery_route[0];
```

Of course, you can also form vectors of structures:

```
vector<StreetAddress> tour_destinations;  
tour_destinations.push_back(white_house);
```



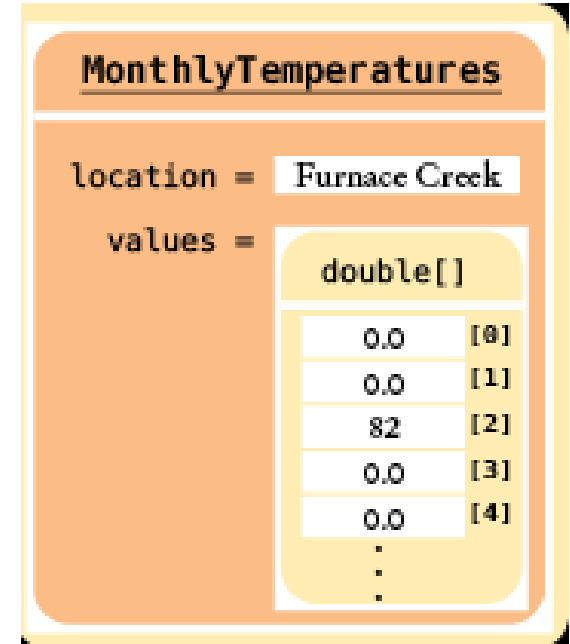
Structures with Array Members

Structure members can contain arrays.
For example:

```
struct MonthlyTemperatures
{
    string location;
    double values[12];
};
```

To access an array element, first select the array member with the dot notation, then use brackets:

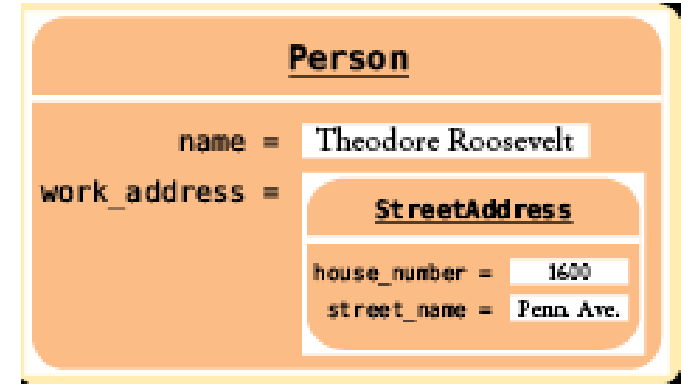
```
MonthlyTemperatures
death_valley_noon;
death_valley_noon.values[2] = 82;
```



Nested Structures

A struct can have a member that is another structure. For example:

```
struct Person
{
    string name;
    StreetAddress work_address;
};
```



You can access the nested member in its entirety, like this:

```
Person theodore;
theodore.work_address = white_house;
```

To select a member of a member, use the dot operator twice:

```
theodore.work_address.street_name =
"Pennsylvania Avenue";
```


Practice It: Structures

Write the code snippets to:

1. Declare a variable "a" of type `StreetAddress`.
2. Set it's house number to 2201.
3. Set the street to "C Street NW".

Topic 8

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

Structure Pointers for Dynamic Allocation

As with all dynamic allocations, you use the `new` operator:

```
StreetAddress* address_pointer = new StreetAddress;
```

The following is incorrect syntax for accessing a member of the structure:

```
*address_pointer.house_number = 1600; // Error
```

...because the dot operator has a higher precedence than the `*` operator. That is, the compiler thinks that you mean `house_number` is itself a pointer:

```
*(address_pointer.house_number) = 1600; // Error
```

Instead, you must first apply the `*` operator, then the dot:

```
(*address_pointer).house_number = 1600; // OK
```

Because this is such a common situation, an arrow operator `->` exists to show `struct` member access via a pointer:

```
address_pointer->house_number = 1600; // OK - use this
```

Structures with Pointer Members

Structures may need to contain pointer members. For example,

```
struct Employee
{
    string name;
    StreetAddress* office;
};

// defining 2 accounting employees:
StreetAddress accounting;
accounting.house_number = 1729;
accounting.street_name = "Park Avenue";

Employee harry;
harry.name = "Smith, Harry";
harry.office = &accounting;
Employee sally;
sally.name = "Lee, Sally";
sally.office = &accounting;
```

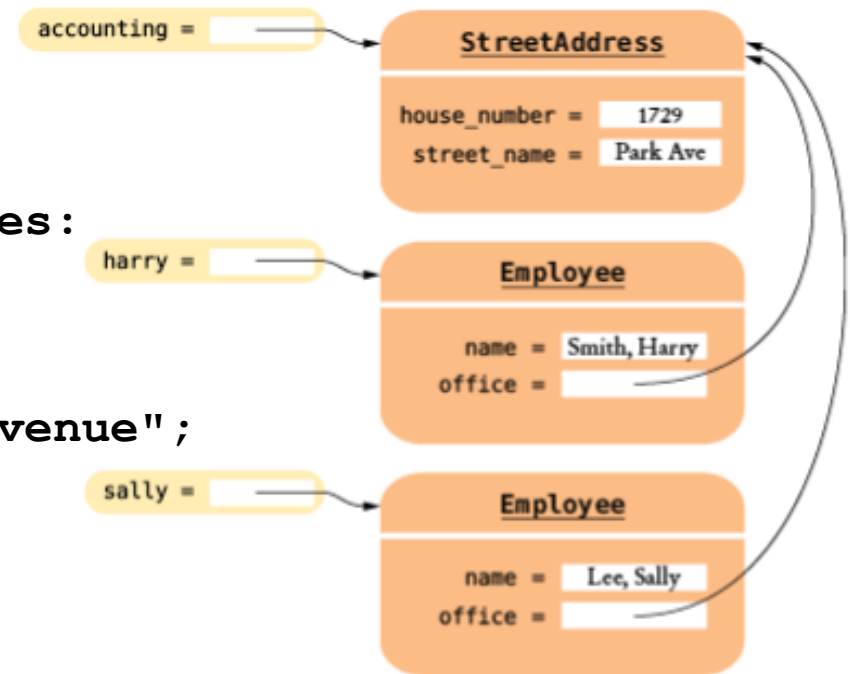


Figure 16 Two Pointers to a Shared Structure

Structures and Pointers: Complete Code Example, Part 1

```
// sec08/streets2.cpp
#include <iostream>
#include <string>
using namespace std;

struct StreetAddress
{
    int house_number;
    string street_name;
};

struct Employee
{
    string name;
    StreetAddress* office;
};

void print_address(StreetAddress address)
{
    cout << address.house_number << " " << address.street_name;
}
```

Structures and Pointers: Complete Code Example, Part 2

```
void print_employee(Employee e)
{
    cout << e.name << " working at ";
    print_address(*e.office);
}

int main()
{
    cout << "A dynamically allocated structure" << endl;
    StreetAddress* address_pointer = new StreetAddress;
    address_pointer->house_number = 1600;
    address_pointer->street_name = "Pennsylvania Avenue";
    print_address(*address_pointer);
    delete address_pointer;

    cout << endl << "Two employees in the same office" << endl;
    StreetAddress accounting;
    accounting.house_number = 1729;
    accounting.street_name = "Park Avenue";
```

Structures and Pointers: Complete Code Example, Part 3

```
Employee harry;  
harry.name = "Smith, Harry";  
harry.office = &accounting;
```

```
Employee sally;  
sally.name = "Lee, Sally";  
sally.office = &accounting;
```

```
cout << "harry: ";  
print_employee(harry);  
cout << endl;
```

```
cout << "sally: ";  
print_employee(sally);  
cout << endl;
```

Structures and Pointers: Complete Code Example, Part 4

```
cout << "After accounting office move" << endl;
accounting.house_number = 1720;

cout << "harry: ";
print_employee(harry);
cout << endl;
cout << "sally: ";
print_employee(sally);
cout << endl;
return 0;
}
```


Smart “shared” Pointers (C++ 11 and Later)

C++ 11 introduced a `shared_ptr<>` type that automatically reclaims memory that is no longer used. For example,

```
shared_ptr<StreetAddress> accounting(new StreetAddress);  
accounting->house_number = 1729;  
accounting->street_name = "Park Avenue";
```

```
Employee sally;  
sally.name = "Lee, Sally";  
sally.office = accounting;
```

Now the `StreetAddress` structure for the accounting office has two shared pointers pointing to it: `accounting` and `sally.office`. When both of these variables go away, then the structure memory is automatically deleted.

We discuss additional strategies for memory management in Chapter 13.

Chapter Summary, Part 1

Define and use pointer variables.

- A pointer denotes the location of a variable in memory.
- The type T^* denotes a pointer to a variable of type T .

```
int* p = nullptr; // can point to an int
```

- The `&` operator yields the location of a variable.

```
int i=0;
```

```
int* p = &i; // p points to i
```

- The `*` operator accesses the variable to which a pointer points.

```
cout << p; // prints value of i, pointed to by p
```

- It is an error to use an uninitialized pointer.
- The `nullptr` pointer does not point to any object.
 - Please initialize unknown pointers to `nullptr`

Chapter Summary, Part 2

Understand the relationship between arrays and pointers in C++.

- The name of an array variable is a pointer to the starting element of the array.
- Pointer arithmetic means adding an integer offset to an array pointer, yielding a pointer that skips past the given number of elements.
- The array/pointer duality law:
 - $a[n]$ is identical to $*(a + n)$, where a is a pointer into an array and n is an integer offset.
- When passing an array to a function, only the starting address is passed.

```
printf(a); //prints array a
```

Chapter Summary, Part 3

Use C++ `string` objects with functions that process character arrays

- A value of type `char` denotes an individual character. Character literals are enclosed in single quotes.
- A literal string (enclosed in double quotes) is an array of `char` values with a zero terminator.
- Many library functions use pointers of type `char*`.
- The `c_str` member function yields a `char*` pointer from a string object.

```
string s = "This is a C++ string object";
```

```
char arr[] = s.c_str(); //copies C++ string to C-string
```

- You can initialize C++ `string` variables with C strings.

```
string t = arr; //copies C-string to C++ string
```

- You can access characters in a C++ `string` object with the `[]` operator.

Chapter Summary, Part 4

Allocate and deallocate memory in programs whose memory requirements aren't known until run time.

- Use dynamic memory allocation if you do not know in advance how many values you need.
- The `new` operator allocates memory from the free store.

```
int* p = new int[50]; // allocate array of 50 ints
```
- You must reclaim dynamically allocated objects with the `delete` or `delete[]` operator.

```
delete[] p; //done using our int array pointed to by p  
p = nullptr; //set p to nullptr to avoid dangling pointer usage
```
- Using a dangling pointer (a pointer that points to memory that has been deleted) is a serious programming error.
- Every call to `new` should have a matching call to `delete`.

Chapter Summary, Part 5

Work with arrays and vectors of pointers.

- Draw diagrams for visualizing pointers and the data to which they point.
 - Draw the data that is being processed, then draw the pointer variables. When drawing the pointer arrows, illustrate a typical situation.

Use structures to aggregate data items.

- A structure combines member values into a single value.
- Use the dot notation to access members of a structure.

```
Streetaddress home;  
home.house_number = 1234;
```
- When you assign one structure value to another, all members are assigned.

Work with pointers to structures.

- Use the -> operator to access a structure member through a pointer

```
Streetaddress* p = new Streetaddress;  
p->house_number = 1234;
```