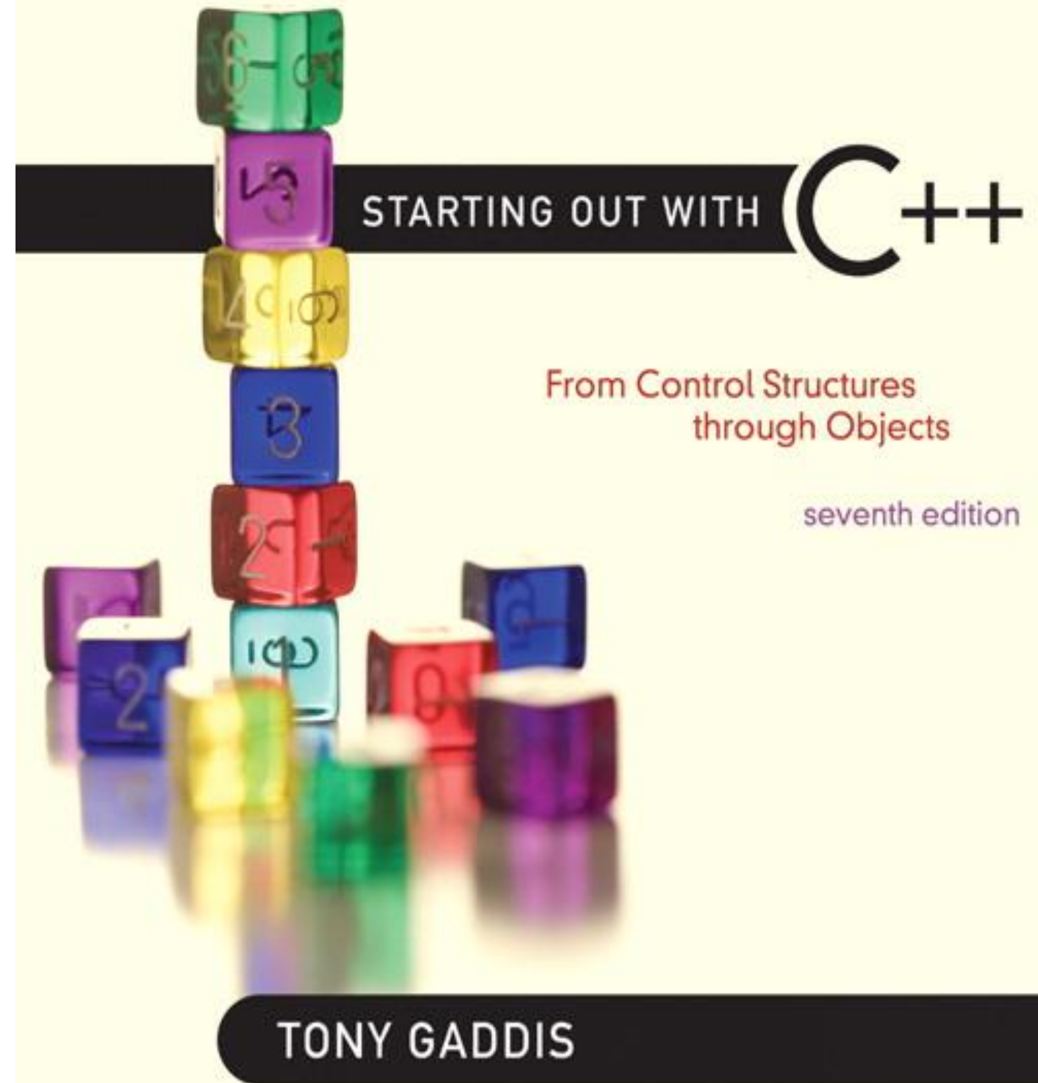
Chapter 9:

Pointers



Addison-Wesley is an imprint of PEARSON

Copyright © 2012 Pearson Education, Inc.

Pointer Variables

- Pointer variable : Often just called a pointer, it's a variable that holds an address

- Because a pointer variable holds the address of another piece of data, it "points" to the data

Copyright © 2012 Pearson Education, Inc.

Something Like Pointers: Arrays

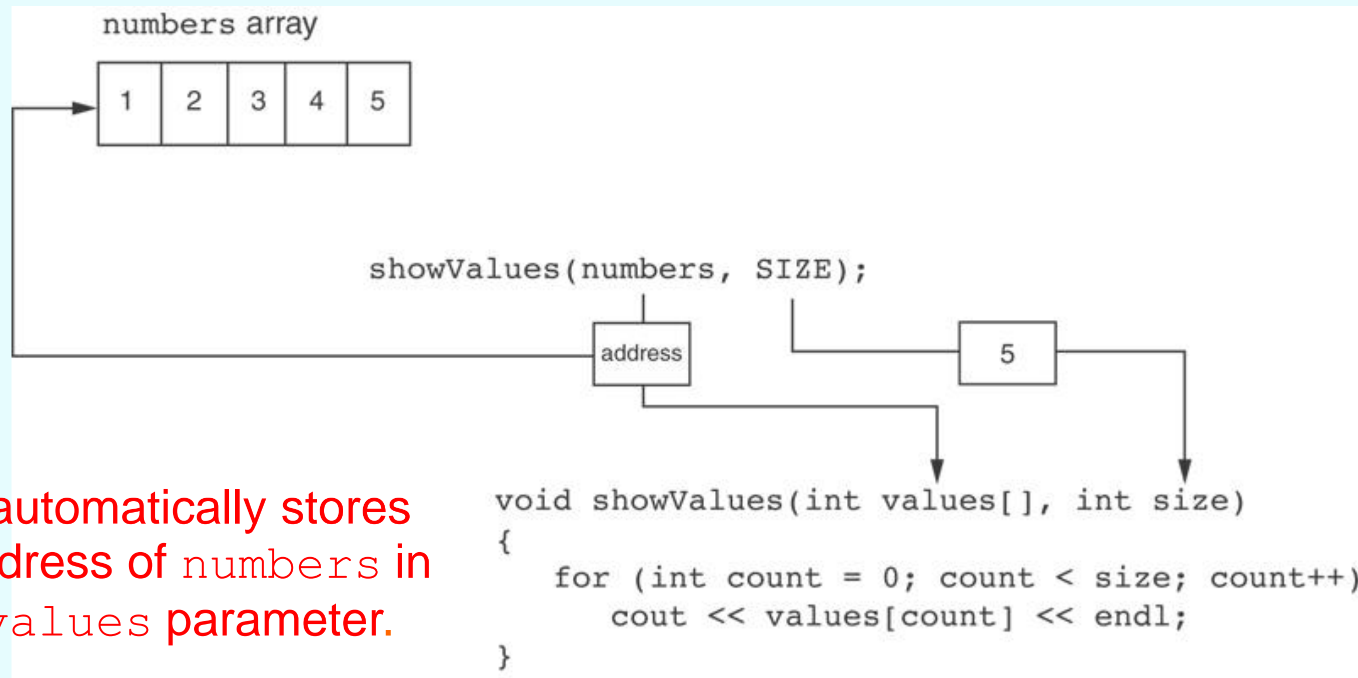
- We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.
- For example, suppose we use this statement to pass the array `numbers` to the `showValues` function:

```
showValues(numbers, SIZE);
```

Copyright © 2012 Pearson Education, Inc.

Something Like Pointers : Arrays

The values parameter, in the showValues function, points to the numbers array.



C++ automatically stores the address of numbers in the values parameter.

Copyright © 2012 Pearson Education, Inc.

Pointer Variables

- Pointer variables are yet another way using a memory address to work with a piece of data.
- Pointers are more "low-level" than arrays and reference variables.
- This means you are responsible for finding the address you want to store in the pointer and correctly using it.

Copyright © 2012 Pearson Education, Inc.

Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:

  "`intptr` can hold the address of an int"

- Spacing in definition does not matter:

```
int * intptr;  // same as above
int*  intptr;  // same as above
```

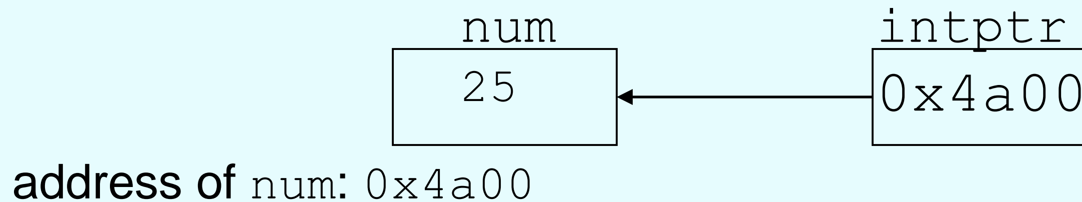
Copyright © 2012 Pearson Education, Inc.

Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr;
intptr = &num;
```

- Memory layout:



Copyright © 2012 Pearson Education, Inc.

The Indirection Operator

- The indirection operator (`*`) dereferences a pointer.
- It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

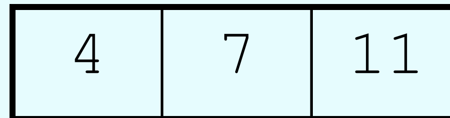
This prints 25.

Copyright © 2012 Pearson Education, Inc.

The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



starting address of `vals`: `0x4a00`

```
cout << vals;            // displays
                         // 0x4a00
cout << vals[0];      // displays 4
```

Copyright © 2012 Pearson Education, Inc.

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;      // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Copyright © 2012 Pearson Education, Inc.

Pointers in Expressions

Given:

```
int vals[]={4,7,11}, *valptr;
valptr = vals;
```

What is `valptr + 1`?

It means (address in `valptr`) + (1 * size of an int)

```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use `( )` as shown in the expressions

Copyright © 2012 Pearson Education, Inc.

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

Copyright © 2012 Pearson Education, Inc.

Array Access

- Conversion:

  `vals[i]` is equivalent to `*(vals + i)`

- No bounds checking performed on array access, whether using array name or a pointer

Copyright © 2012 Pearson Education, Inc.

Pointer Arithmetic

- Operations on pointer variables:

Operation	Example
	`int vals[]={4,7,11};`
	`int *valptr = vals;`

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` (pointer and `int`)	`cout << *(valptr + 2); // 11`

`+=, -=` (pointer and `int`)	`valptr = vals; // points at 4`
	`valptr += 2;   // points at 11`

`-` (pointer from pointer)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

Copyright © 2012 Pearson Education, Inc.

Initializing Pointers

- Can initialize at definition time:

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

```
double cost;
int *ptr = &cost; // won't work
```

- Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```

Copyright © 2012 Pearson Education, Inc.

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2) // compares
                    // contents
```


Copyright © 2012 Pearson Education, Inc.

Pointers as Function Parameters

- A pointer can be a parameter
- Works like reference variable to allow change to argument from within function
- Requires:
  1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr); // ptr is pointer to an int
```

  2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

  3) address as argument to the function

```
getNum(&num);      // pass address of num to getNum
```

Copyright © 2012 Pearson Education, Inc.

Example

```
void swap(int *x, int *y)
{     int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

Copyright © 2012 Pearson Education, Inc.

Pointers to Constants

- Example: Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] =
      { 18.55, 17.45, 12.85,
        14.97, 10.35, 18.89 };
```

- In this code, `payRates` is an array of constant doubles.

Copyright © 2012 Pearson Education, Inc.

Pointers to Constants

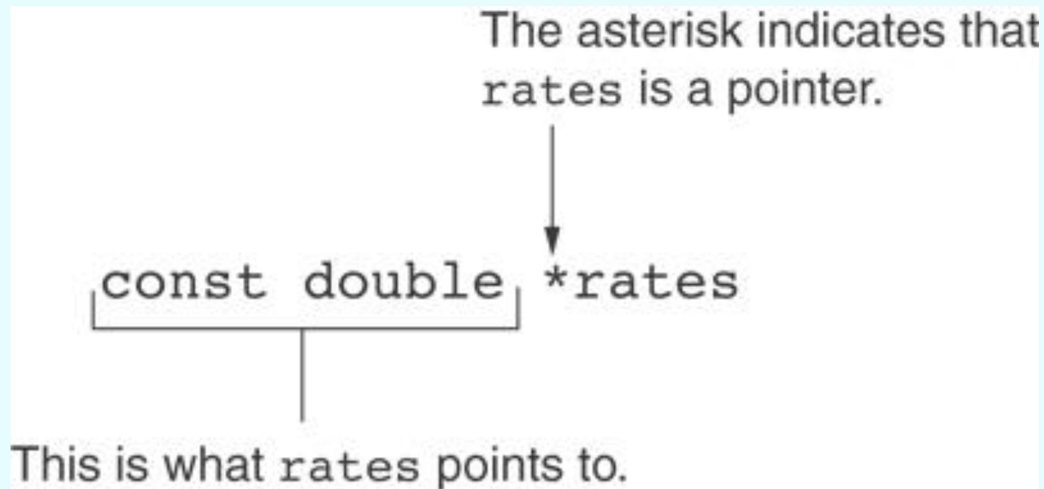
- Suppose we wish to pass the `payRates` array to a function? Here's an example of how we can do it.

```
void displayPayRates(const double *rates, int size)
{
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

The parameter, rates, is a pointer to `const double`.

Copyright © 2012 Pearson Education, Inc.

Declaration of a Pointer to Constant



Copyright © 2012 Pearson Education, Inc.

Constant Pointers

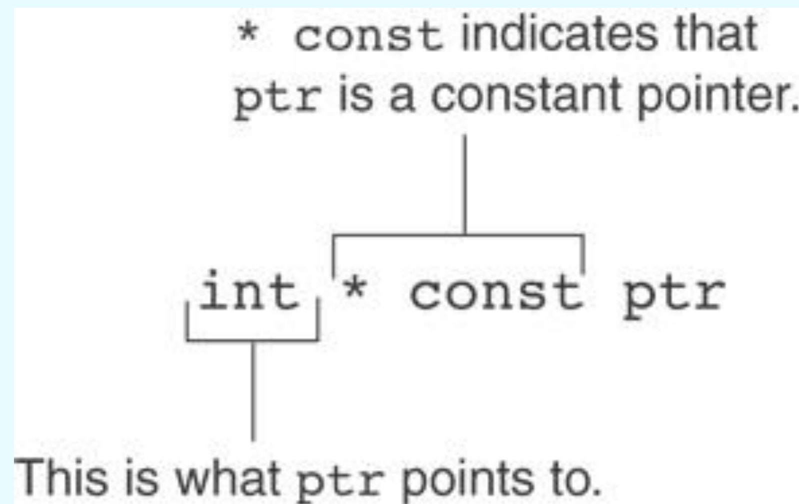
- A constant pointer is a pointer that is initialized with an address, and cannot point to anything else.

- Example

```
int value = 22;
int * const ptr = &value;
```

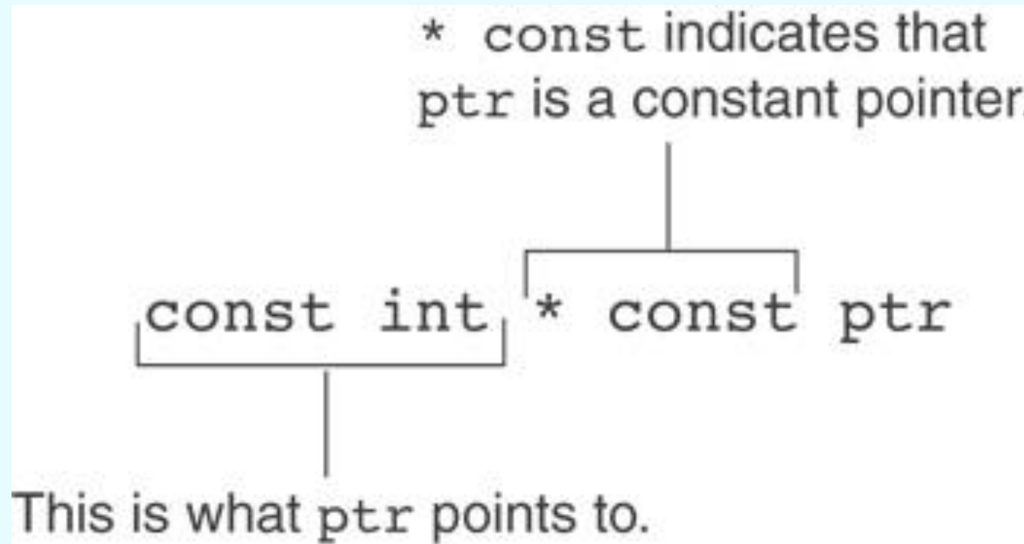
Copyright © 2012 Pearson Education, Inc.

Constant Pointers



Copyright © 2012 Pearson Education, Inc.

Constant Pointers to Constants



Copyright © 2012 Pearson Education, Inc.

Dynamic Memory Allocation: new

- Can allocate storage for a variable while program is running
- Computer returns address of newly allocated variable
- Uses `new` operator to allocate memory:

```
double *dptr;
dptr = new double;
```

- `new` returns address of memory location

Copyright © 2012 Pearson Education, Inc.

Dynamic Memory Allocation

- Can also use `new` to allocate array:

```
const int SIZE = 25;
arrayPtr = new double[SIZE];
```

- Can then use `[]` or pointer arithmetic to access array:

```
for(i = 0; i < SIZE; i++)
    *arrayptr[i] = i * i;
```

or

```
for(i = 0; i < SIZE; i++)
    *(arrayptr + i) = i * i;
```

- Program will terminate if not enough memory available to allocate

Copyright © 2012 Pearson Education, Inc.

Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

Copyright © 2012 Pearson Education, Inc.

Program 9-14

```
1  // This program totals and averages the sales figures for any
2  // number of days. The figures are stored in a dynamically
3  // allocated array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    double *sales,       // To dynamically allocate an array
11           total = 0.0, // Accumulator
12           average;     // To hold average sales
```

Copyright © 2012 Pearson Education, Inc.

Program 9-14 *(continued)*

```
13     int numDays,         // To hold the number of days of sales
14         count;           // Counter variable
15
16     // Get the number of days of sales.
17     cout << "How many days of sales figures do you wish ";
18     cout << "to process? ";
19     cin >> numDays;
20
21     // Dynamically allocate an array large enough to hold
22     // that many days of sales amounts.
23     sales = new double[numDays];
24
25     // Get the sales figures for each day.
26     cout << "Enter the sales figures below.\n";
27     for (count = 0; count < numDays; count++)
28     {
29        cout << "Day " << (count + 1) << ": ";
30        cin >> sales[count];
31     }
32
```

Copyright © 2012 Pearson Education, Inc.

Returning Pointers from Functions

- Pointer can be the return type of a function:

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function.
- A function should only return a pointer:
  - to data that was passed to the function as an argument, or
  - to dynamically allocated memory

Copyright © 2012 Pearson Education, Inc.

From Program 9-15

```
34  int *getRandomNumbers(int num)
35  {
36     int *array;    // Array to hold the numbers
37
38     // Return null if num is zero or negative.
39     if (num <= 0)
40        return NULL;
41
42     // Dynamically allocate the array.
43     array = new int[num];
44
45     // Seed the random number generator by passing
46     // the return value of time(0) to srand.
47     srand( time(0) );
48
49     // Populate the array with random numbers.
50     for (int count = 0; count < num; count++)
51        array[count] = rand();
52
53     // Return a pointer to the array.
54     return array;
55  }
```

Copyright © 2012 Pearson Education, Inc.