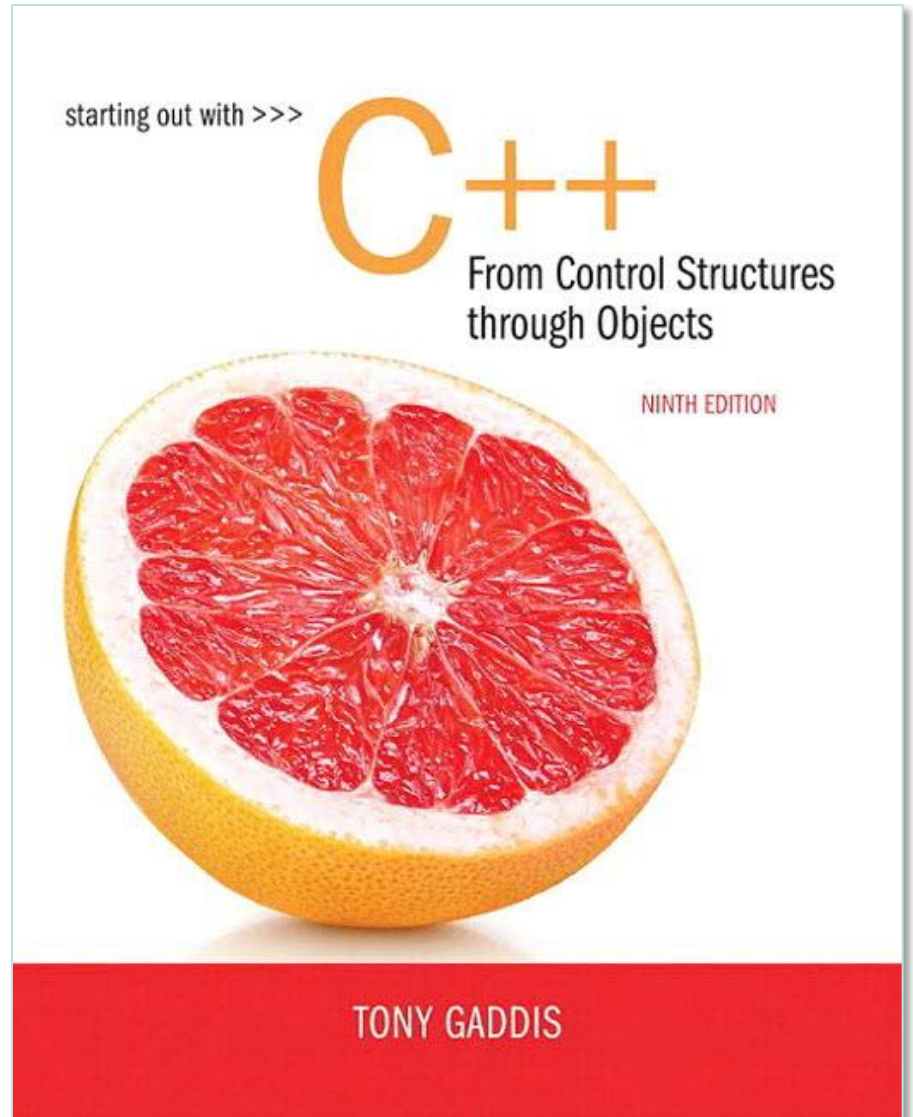


# Chapter 9:

## Pointers





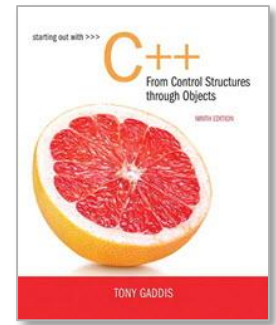
# 9.1

## Getting the Address of a Variable

# Getting the Address of a Variable

- Each variable in program is stored at a unique address
- Use address operator & to get address of a variable:

```
int num = -99;  
cout << &num; // prints address  
               // in hexadecimal
```



# 9.2

## Pointer Variables

# 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

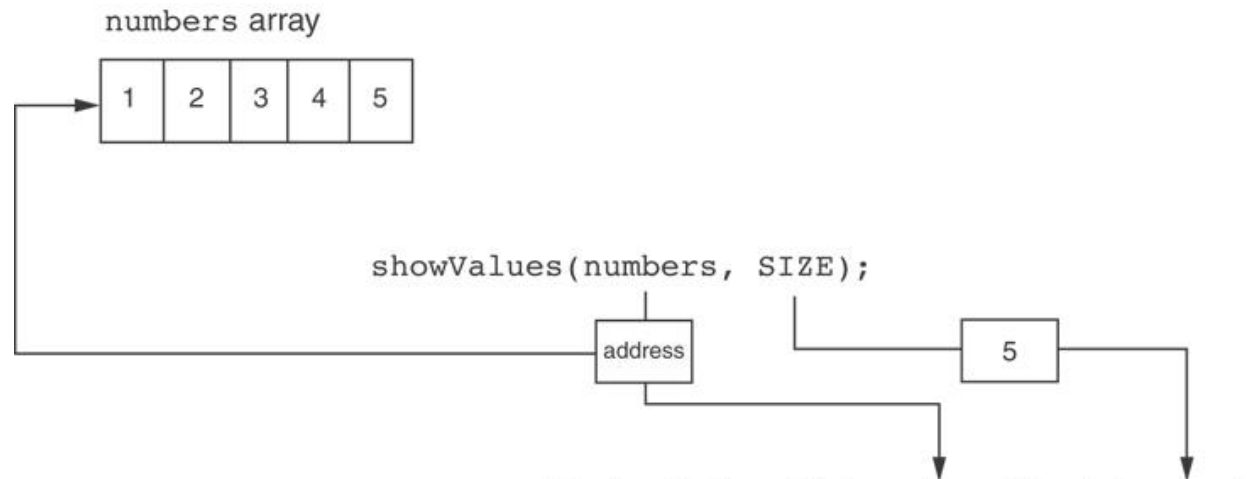
# Something Like Pointers: Arrays

- 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) ;
```

# 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.

```
void showValues(int values[], int size)
{
    for (int count = 0; count < size; count++)
        cout << values[count] << endl;
}
```

# Something Like Pointers: Reference Variables

- We have also worked with something like pointers when we learned to use reference variables. Suppose we have this function:

```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

- And we call it with this code:

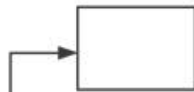
```
int jellyDonuts;
getOrder(jellyDonuts);
```



# Something Like Pointers: Reference Variables

The `donuts` parameter, in the `getOrder` function, points to the `jellyDonuts` variable.

`jellyDonuts` variable



`getOrder(jellyDonuts);`

address

```
void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
```

C++ automatically stores the address of `jellyDonuts` in the `donuts` parameter.

# 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.

# 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
```

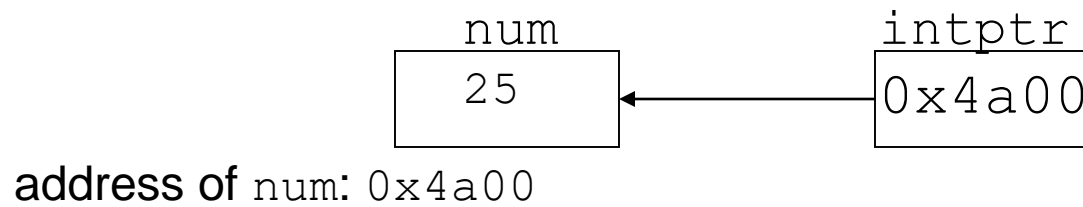
```
int*  intptr;    // same as above
```

# Pointer Variables

- Assigning an address to a pointer variable:

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

- Memory layout:



# Pointer Variables

- Initialize pointer variables with the special value `nullptr`.
- In C++ 11, the `nullptr` key word was introduced to represent the address 0.
- Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```
int *ptr = nullptr;
```

# A Pointer Variable in Program 9-2

## Program 9-2

```
1  // This program stores the address of a variable in a pointer.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      int x = 25;           // int variable
8      int *ptr = nullptr;   // Pointer variable, can point to an int
9
10     ptr = &x;             // Store the address of x in ptr
11     cout << "The value in x is " << x << endl;
12     cout << "The address of x is " << ptr << endl;
13     return 0;
14 }
```

## Program Output

```
The value in x is 25
The address of x is 0x7e00
```

# 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.

# The Indirection Operator in Program 9-3

## Program 9-3

```
1  // This program demonstrates the use of the indirection operator.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      int x = 25;           // int variable
8      int *ptr = nullptr;   // Pointer variable, can point to an int
9
10     ptr = &x;             // Store the address of x in ptr
11
12     // Use both x and ptr to display the value in x.
13     cout << "Here is the value in x, printed twice:\n";
14     cout << x << endl;    // Displays the contents of x
15     cout << *ptr << endl; // Displays the contents of x
16
17     // Assign 100 to the location pointed to by ptr. This
18     // will actually assign 100 to x.
19     *ptr = 100;
```

*(program continues)*



# The Indirection Operator in Program 9-3

## Program 9-3

*(continued)*

```
20
21     // Use both x and ptr to display the value in x.
22     cout << "Once again, here is the value in x:\n";
23     cout << x << endl;    // Displays the contents of x
24     cout << *ptr << endl; // Displays the contents of x
25     return 0;
26 }
```

## Program Output

Here is the value in x, printed twice:

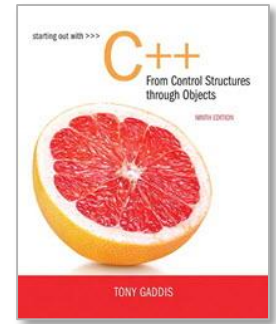
25

25

Once again, here is the value in x:

100

100



# 9.3

## The Relationship Between Arrays and Pointers

# The Relationship Between Arrays and Pointers

- Array name is starting address of array

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

4	7	11
---	---	----

starting address of `vals`: 0x4a00

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

# 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
```

# The Array Name Being Dereferenced in Program 9-5

## Program 9-5

```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      short numbers[] = {10, 20, 30, 40, 50};
9
10     cout << "The first element of the array is ";
11     cout << *numbers << endl;
12     return 0;
13 }
```

## Program Output

The first element of the array is 10

# 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

# Array Access

- Array elements can be accessed in many ways:

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

# Array Access

- Conversion: `vals[i]` is equivalent to `*(vals + i)`
- No bounds checking performed on array access, whether using array name or a pointer



# From Program 9-7

```
9      const int NUM_COINS = 5;
10     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
11     double *doublePtr;    // Pointer to a double
12     int count;            // Array index
13
14     // Assign the address of the coins array to doublePtr.
15     doublePtr = coins;
16
17     // Display the contents of the coins array. Use subscripts
18     // with the pointer!
19     cout << "Here are the values in the coins array:\n";
20     for (count = 0; count < NUM_COINS; count++)
21         cout << doublePtr[count] << " ";
22
23     // Display the contents of the array again, but this time
24     // use pointer notation with the array name!
25     cout << "\nAnd here they are again:\n";
26     for (count = 0; count < NUM_COINS; count++)
27         cout << *(coins + count) << " ";
28     cout << endl;
```

## Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
```



# 9.4

## Pointer Arithmetic

# Pointer Arithmetic

## Operations on pointer variables:

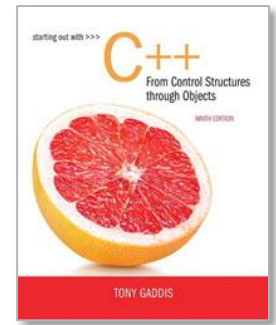
Operation	Example
	<pre>int vals[]={4,7,11}; int *valptr = vals;</pre>
<code>++, --</code>	<pre>valptr++; // points at 7 valptr--; // now points at 4</pre>
<code>+, - (pointer and int)</code>	<pre>cout &lt;&lt; *(valptr + 2); // 11</pre>
<code>+=, -= (pointer and int)</code>	<pre>valptr = vals; // points at 4 valptr += 2;    // points at 11</pre>
<code>- (pointer from pointer)</code>	<pre>cout &lt;&lt; valptr-val; // difference // (number of ints) between valptr // and val</pre>

# From Program 9-9

```
7      const int SIZE = 8;
8      int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9      int *numPtr = nullptr; // Pointer
10     int count;             // Counter variable for loops
11
12     // Make numPtr point to the set array.
13     numPtr = set;
14
15     // Use the pointer to display the array contents.
16     cout << "The numbers in set are:\n";
17     for (count = 0; count < SIZE; count++)
18     {
19         cout << *numPtr << " ";
20         numPtr++;
21     }
22
23     // Display the array contents in reverse order.
24     cout << "\nThe numbers in set backward are:\n";
25     for (count = 0; count < SIZE; count++)
26     {
27         numPtr--;
28         cout << *numPtr << " ";
29     }
30     return 0;
31 }
```

## Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```



# 9.5

## Initializing Pointers

# Initializing Pointers

- Can initialize at definition time:

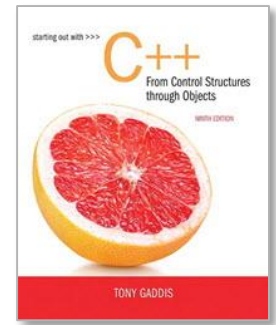
```
int num, *numptr = &num;  
int val[3], *valptr = val;
```

- Cannot mix data types:

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

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

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



# 9.6

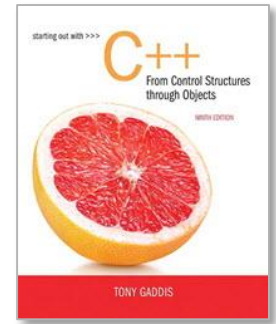
## Comparing Pointers

# 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
```





# 9.7

## Pointers as Function Parameters

# 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`

# Example

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

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

# Pointers as Function Parameters in Program 9-11

## Program 9-11

```
1  // This program uses two functions that accept addresses of
2  // variables as arguments.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNumber(int *);
8  void doubleValue(int *);
9
10 int main()
11 {
12     int number;
13
14     // Call getNumber and pass the address of number.
15     getNumber(&number);
16
17     // Call doubleValue and pass the address of number.
18     doubleValue(&number);
19
20     // Display the value in number.
21     cout << "That value doubled is " << number << endl;
22     return 0;
23 }
24
```

*(Program Continues)*

# Pointers as Function Parameters in Program 9-11

## Program 9-11 (continued)

```
25  //*****
26  // Definition of getNumber. The parameter, input, is a pointer. *
27  // This function asks the user for a number. The value entered *
28  // is stored in the variable pointed to by input.                *
29  //*****
30
31  void getNumber(int *input)
32  {
33      cout << "Enter an integer number: ";
34      cin >> *input;
35  }
36
37  //*****
38  // Definition of doubleValue. The parameter, val, is a pointer. *
39  // This function multiplies the variable pointed to by val by    *
40  // two.                                                            *
41  //*****
42
43  void doubleValue(int *val)
44  {
45      *val *= 2;
46  }
```

### Program Output with Example Input Shown in Bold

```
Enter an integer number: 10 [Enter]
That value doubled is 20
```

# Pointers to Constants

- If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const.

# 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.

# 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`.

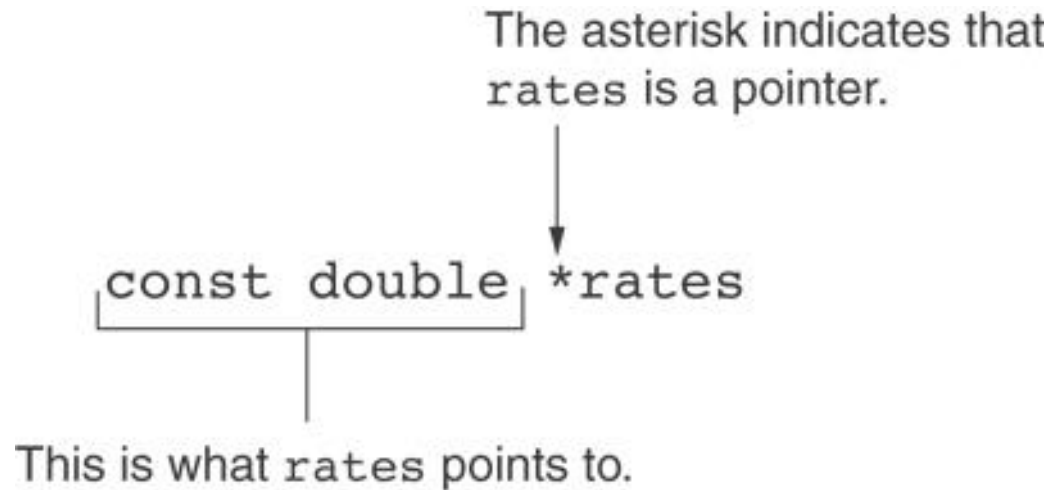


# Declaration of a Pointer to Constant

The asterisk indicates that rates is a pointer.

`const double *rates`

This is what rates points to.



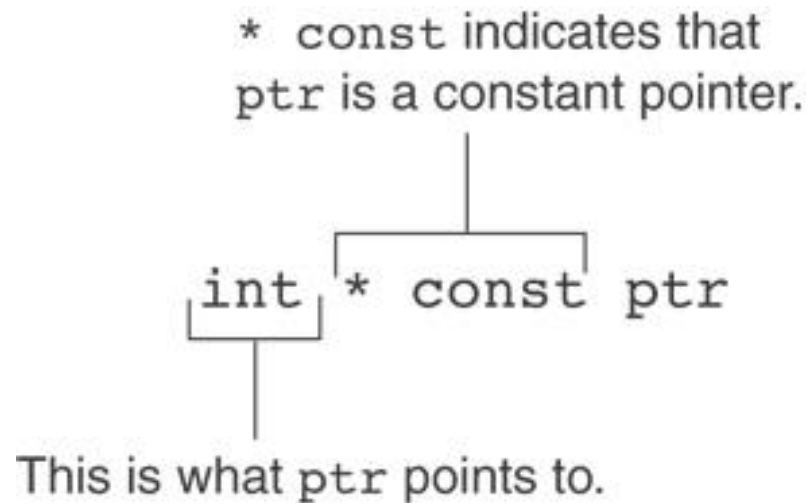
# Constant Pointers

- 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;
```

# Constant Pointers



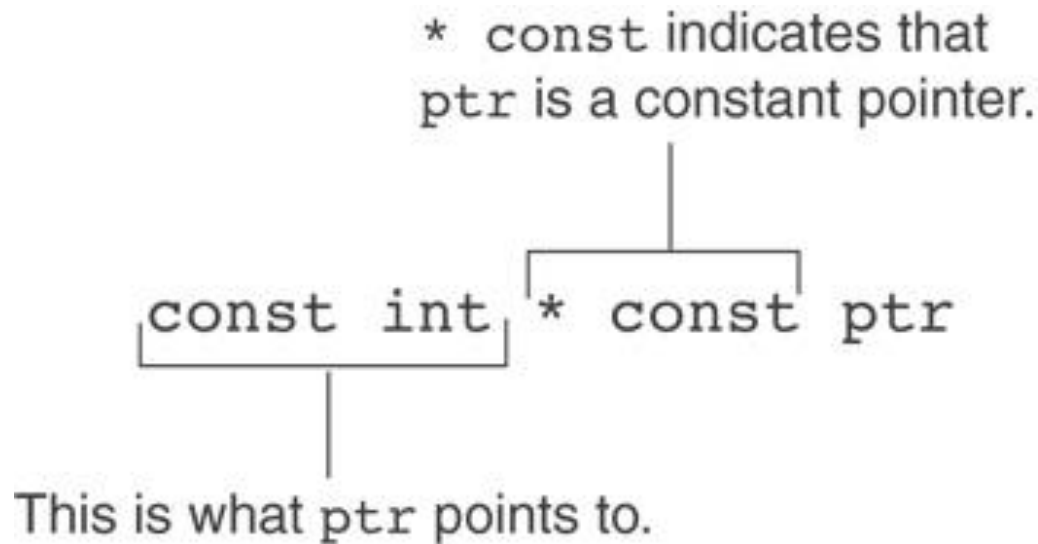
# Constant Pointers to Constants

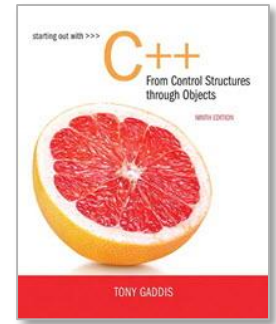
- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

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

# Constant Pointers to Constants





# 9.8

## Dynamic Memory Allocation

# Dynamic Memory Allocation

- 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 = nullptr;  
dptr = new double;
```
- `new` returns address of memory location

# 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



# Releasing Dynamic Memory

- Use `delete` to free dynamic memory:

```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

# Dynamic Memory Allocation in Program 9-14

## 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 = nullptr, // To dynamically allocate an array
11           total = 0.0,       // Accumulator
12           average;           // To hold average sales
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;
```

# Dynamic Memory Allocation in Program 9-14

```
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
33     // Calculate the total sales
34     for (count = 0; count < numDays; count++)
35     {
36         total += sales[count];
37     }
38
39     // Calculate the average sales per day
40     average = total / numDays;
41
42     // Display the results
43     cout << fixed << showpoint << setprecision(2);
44     cout << "\n\nTotal Sales: $" << total << endl;
45     cout << "Average Sales: $" << average << endl;
```

Program 9-14 (Continued)

# Dynamic Memory Allocation in Program 9-14

## Program 9-14 (Continued)

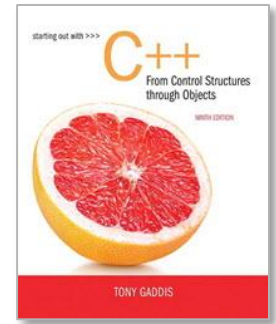
```
46
47     // Free dynamically allocated memory
48     delete [] sales;
49     sales = nullptr;    // Make sales a null pointer.
50
51     return 0;
52 }
```

### Program Output with Example Input Shown in Bold

```
How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

Total Sales: $4067.13
Average Sales: $813.43
```

*Notice that in line 49 `nullptr` is assigned to the `sales` pointer. The `delete` operator is designed to have no effect when used on a null pointer.*



# 9.9

## Returning Pointers from Functions

# Returning Pointers from Functions

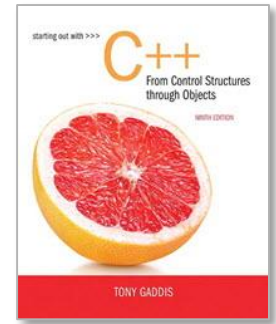
- Pointer can be the return type of a function:

```
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

# From Program 9-15

```
34 int *getRandomNumbers(int num)
35 {
36     int *arr = nullptr; // Array to hold the numbers
37
38     // Return a null pointer if num is zero or negative.
39     if (num <= 0)
40         return nullptr;
41
42     // Dynamically allocate the array.
43     arr = 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         arr[count] = rand();
52
53     // Return a pointer to the array.
54     return arr;
55 }
```



# 9.10

## Using Smart Pointers to Avoid Memory Leaks



# Using Smart Pointers to Avoid Memory Leaks

- In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it.
- Three types of smart pointer:

```
unique_ptr  
shared_ptr  
weak_ptr
```

- Must `#include` the memory header file:

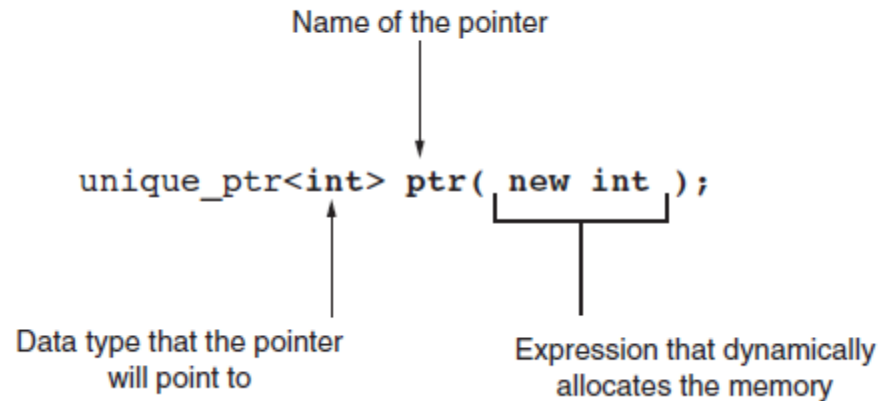
```
#include <memory>
```

- In this book, we introduce `unique_ptr`:

```
unique_ptr<int> ptr( new int );
```

# Using Smart Pointers to Avoid Memory Leaks

Figure 9-12



- The notation `<int>` indicates that the pointer can point to an `int`.
- The name of the pointer is `ptr`.
- The expression `new int` allocates a chunk of memory to hold an `int`.
- The address of the chunk of memory will be assigned to `ptr`.

# Using Smart Pointers in Program 9-17

## Program 9-17

```
1  // This program demonstrates a unique_ptr.
2  #include <iostream>
3  #include <memory>
4  using namespace std;
5
6  int main()
7  {
8      // Define a unique_ptr smart pointer, pointing
9      // to a dynamically allocated int.
10     unique_ptr<int> ptr( new int );
11
12     // Assign 99 to the dynamically allocated int.
13     *ptr = 99;
14
15     // Display the value of the dynamically allocated int.
16     cout << *ptr << endl;
17     return 0;
18 }
```

## Program Output

99