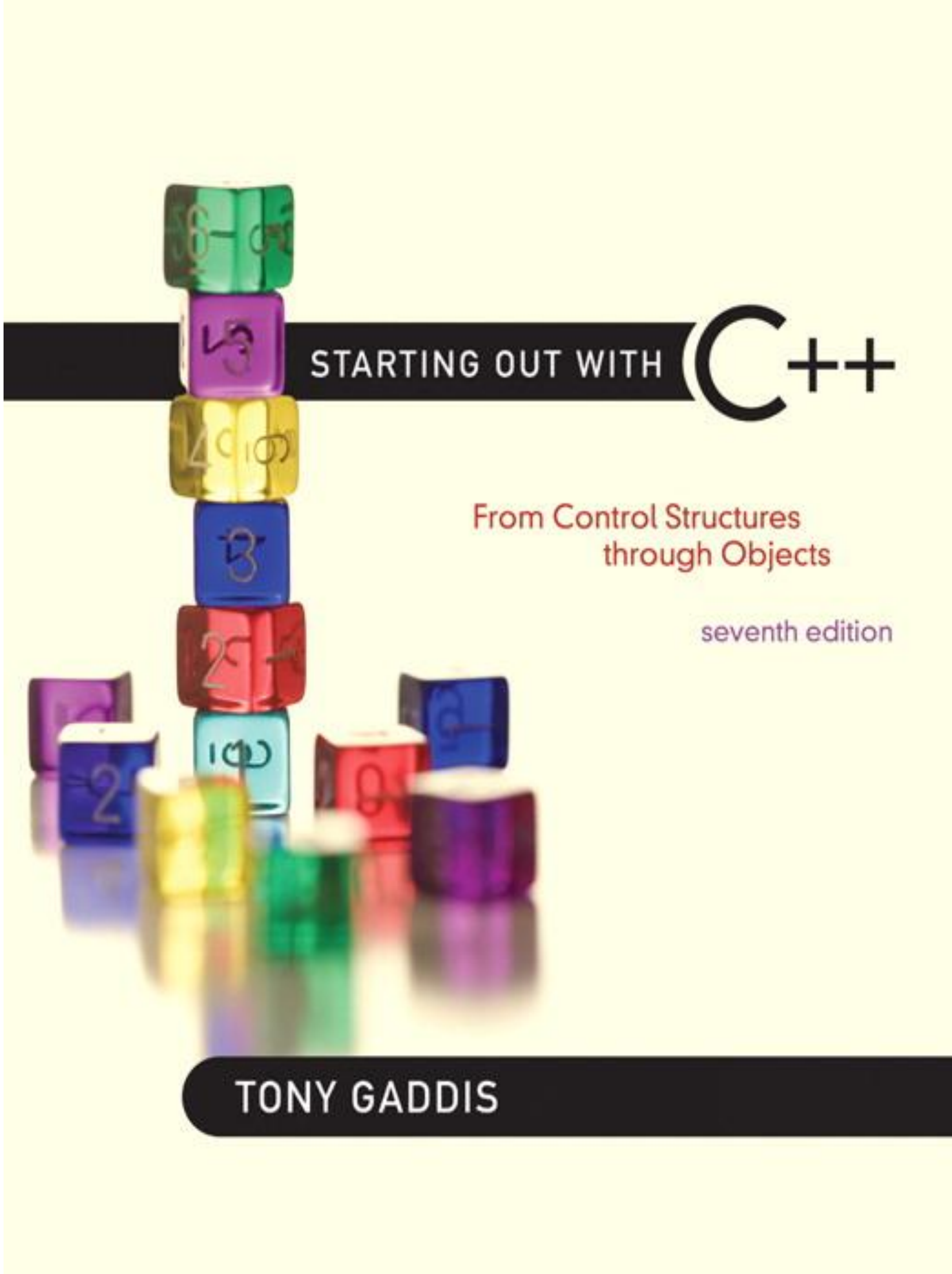
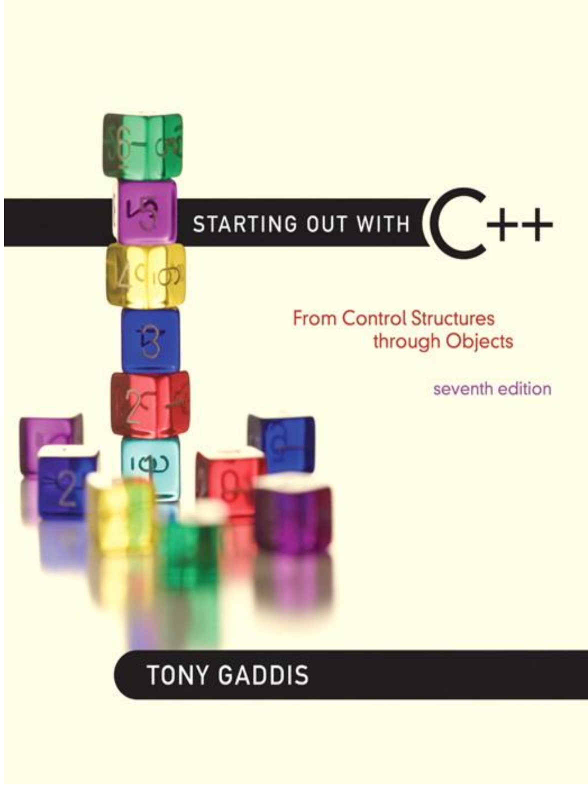
Chapter 9:

Pointers



Addison-Wesley
is an imprint of
PEARSON



9.1

Getting the Address of a Variable

C++ Variables [not in book]

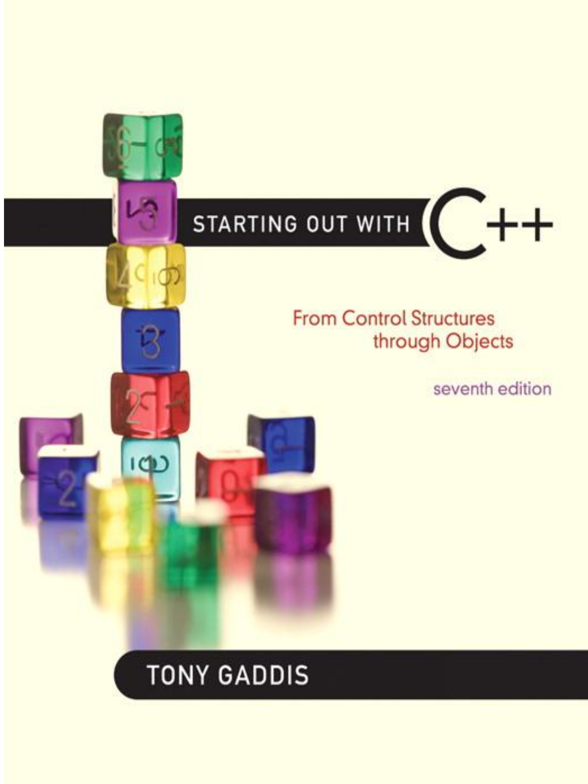
A Variable has all of the following attributes:

1.	**name**	
2.	**type**	
3.	**size**	
4.	**value**	
5.	**storage class**	**static or automatic**
6.	**scope**	**where it is known in the program**
7.	**linkage**	**use of extern and static qualifiers**
8.	**address**	**the address in memory of the 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
```



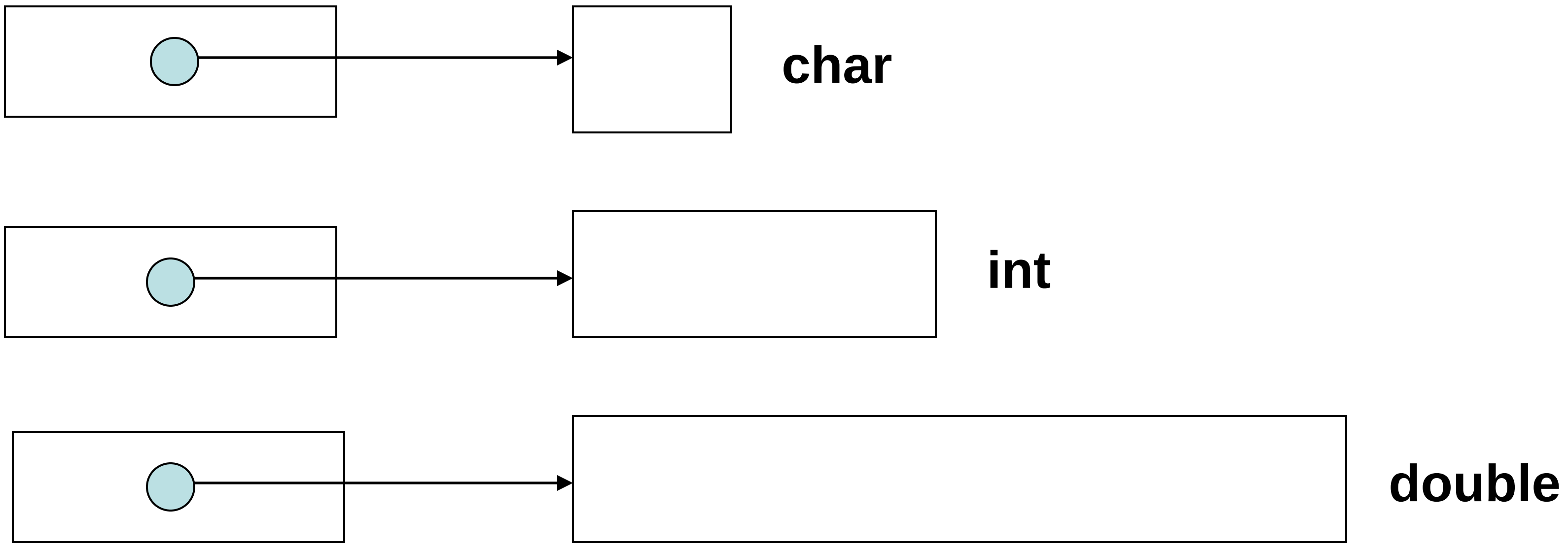
9.2

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

Chapter 9 – Pointers



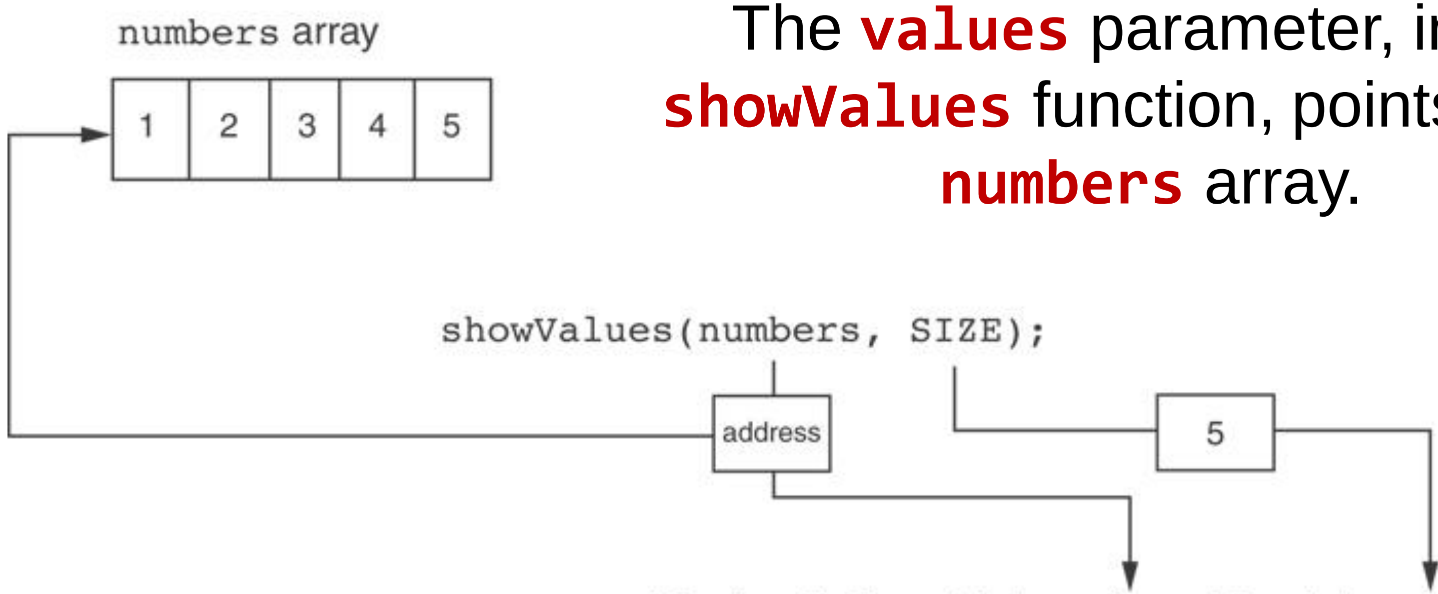
Something Like Pointers: Arrays

- 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.

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

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

Something Like Pointers: Reference Variables

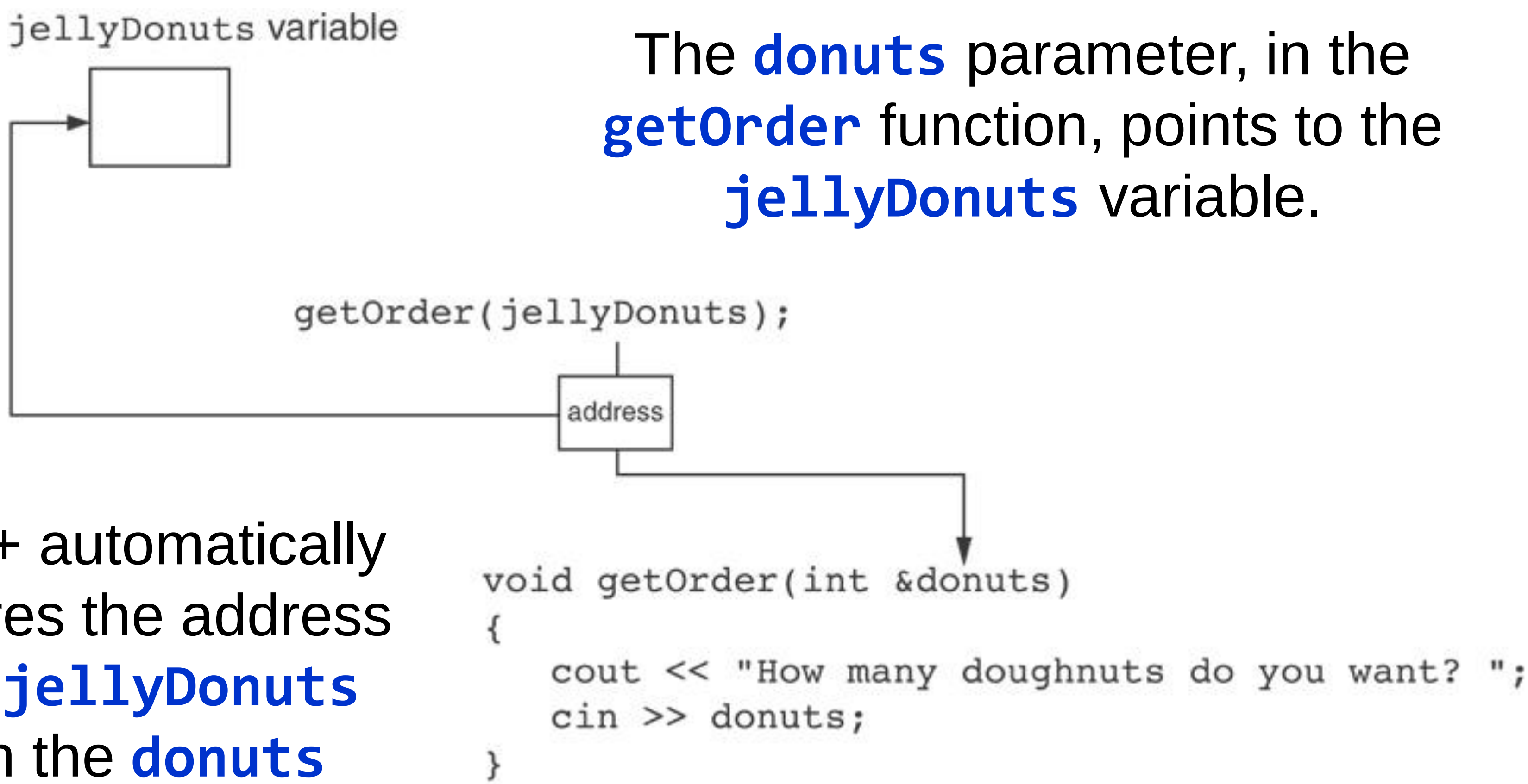
- We have also worked with something like pointers when we learned to use reference variables. Suppose we have this function:

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

- And we call it with this code:

```
int jellyDonuts;
getOrder(jellyDonuts); //no ampersand in call…
```

Something Like Pointers: Reference Variables



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

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

Pointer Variables

- Pointer variables are yet another way of 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 num = 25;
int *intptr;
intptr = &num;
```

- Memory layout:

address of **num**: 0x4a00

Program 9-2

```
// This program stores the address of a variable in a pointer.
#include <iostream>
using namespace std;

int main()
{
   int x = 25;      // int variable
   int *ptr;        // Pointer variable, can point to an int

   ptr = &x;        // Store the address of x in ptr
   cout << "The value in x is " << x << endl;
   cout << "The address of x is " << ptr << endl;
   return 0;
}
```

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.

Program 9-3

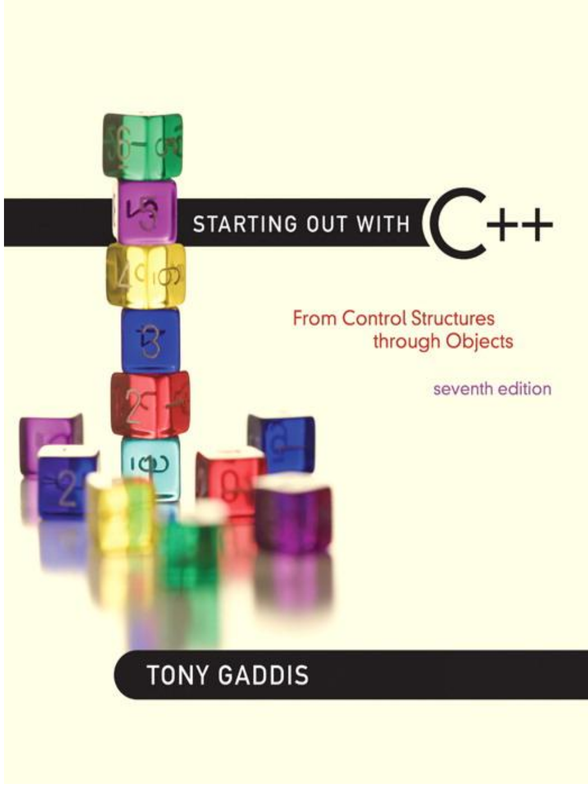
```
// This program demonstrates the use of the indirection operator.
#include <iostream>
using namespace std;

int main()
{
   int x = 25;    // int variable
   int *ptr;      // Pointer variable, can point to an int

   ptr = &x;      // Store the address of x in ptr

   // Use both x and ptr to display the value in x.
   cout << "Here is the value in x, printed twice:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x

   // Assign 100 to the location pointed to by ptr. This
   // will actually assign 100 to x.
   *ptr = 100;

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

Program 9-3 *(continued)*

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};
```

starting address of `vals`: **0x4a00**

4	7	11

```
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
[ ] dereferences the array item
```

The Relationship Between Arrays and Pointers

- Array name can be used as a constant pointer:

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

`*vals` is equivalent to `vals[0]`

- Pointer can be used as an array name:

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

Program 9-5

```cpp
// This program shows an array name being dereferenced with the *
// operator.
#include <iostream>
using namespace std;

int main()
{
   short numbers[] = {10, 20, 30, 40, 50};

   cout << "The first element of the array is ";
   cout << *numbers << endl;
   return 0;
}
```

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 `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and pointer arithmetic	`*(vals + 2) = 17;`
pointer to array and pointer arithmetic	`*(valptr + 2) = 17;`

Use of **[ ] subscript** and *** offset** notation

Notice that the array name and pointer are interchangeable !

Array Access

- Conversion:

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

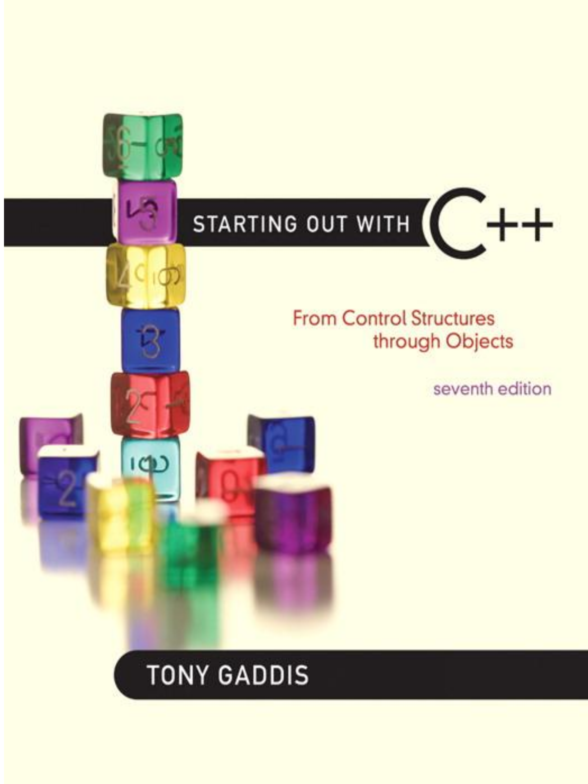
- No bounds checking performed on array access, whether using array name or a pointer

From Program 9-7

```
    const int NUM_COINS = 5;
    double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
    double *doublePtr;    // Pointer to a double
    int count;            // Array index

    // Assign the address of the coins array to doublePtr.
    doublePtr = coins;

    // Display the contents of the coins array. Use subscripts
    // with the pointer!
    cout << "Here are the values in the coins array:\n";
    for (count = 0; count < NUM_COINS; count++)
       cout << doublePtr[count] << " ";

    // Display the contents of the array again, but this time
    // use pointer notation with the array name!
    cout << "\nAnd here they are again:\n";
    for (count = 0; count < NUM_COINS; count++)
       cout << *(coins + count) << " ";
    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
	`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`

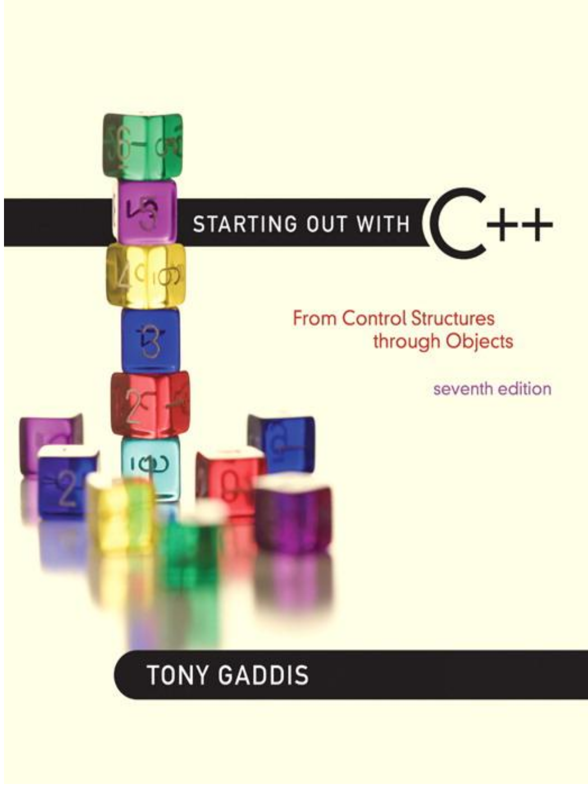
`-` (**pointer** from **pointer**)	`cout << valptr-val; // difference`
	`//(number of ints) between valptr`
	`// and val`

From Program 9-9

```
    const int SIZE = 8;
    int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *numPtr;    // Pointer
    int count;      // Counter variable for loops

    // Make numPtr point to the set array.
    numPtr = set;

    // Use the pointer to display the array contents.
    cout << "The numbers in set are:\n";
    for (count = 0; count < SIZE; count++)
    {
       cout << *numPtr << " ";
       numPtr++;
    }

    // Display the array contents in reverse order.
    cout << "\nThe numbers in set backward are:\n";
    for (count = 0; count < SIZE; count++)
    {
       numPtr--;
       cout << *numPtr << " ";
    }
```

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:

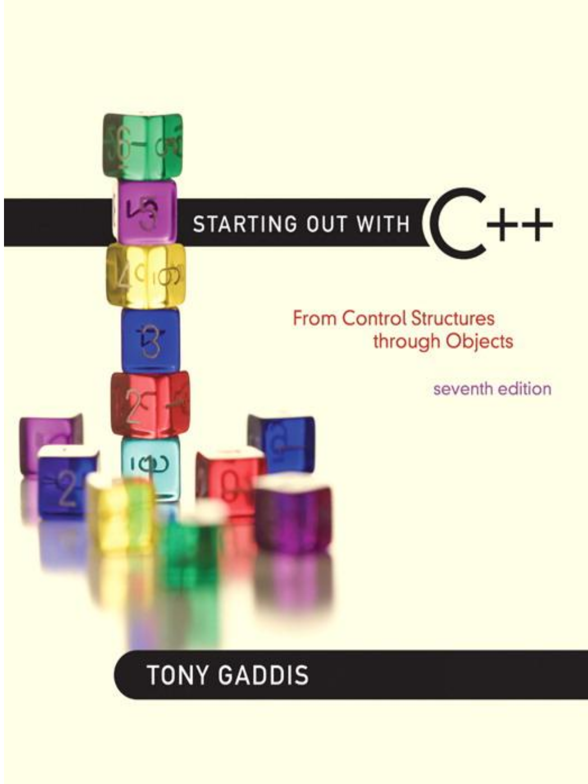
```
int num,    *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

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

- Can test for a null pointer for `ptr` with:

```
if ( !ptr ) ...   //null pointer == 0
```



9.6

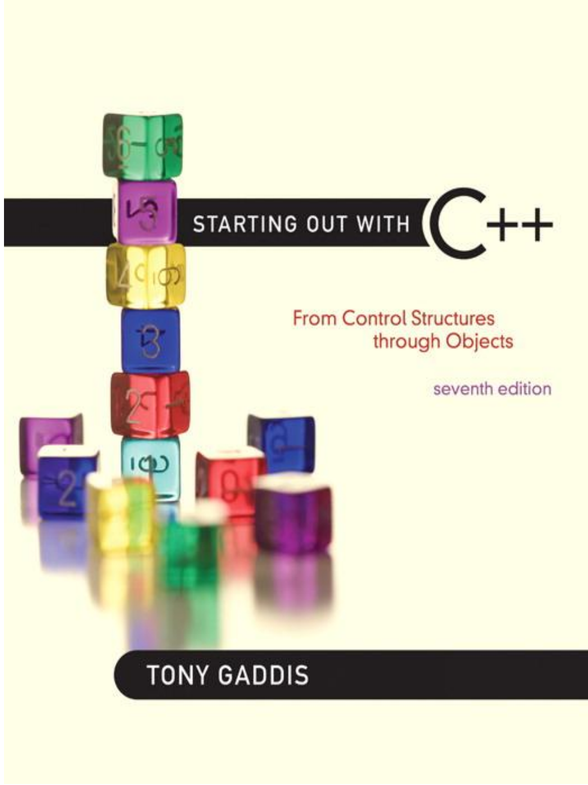
Comparing Pointers

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 a reference variable to allow changes to an argument from within a 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 (using pointers)

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

      temp = *x;
      *x   = *y;
      *y   = temp;
}

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

Example (using references)

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

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

Program 9-11

```
// This program uses two functions that accept addresses of
// variables as arguments.
#include <iostream>
using namespace std;

// Function prototypes
void getNumber(int *);
void doubleValue(int *);

int main()
{
   int number;

   // Call getNumber and pass the address of number.
   getNumber(&number);

   // Call doubleValue and pass the address of number.
   doubleValue(&number);

   // Display the value in number.
   cout << "That value doubled is " << number << endl;
   return 0;
}

```

(Program Continues)

Program 9-11 *(continued)*

```
//*****************************************************************
// Definition of getNumber. The parameter, input, is a pointer. *
// This function asks the user for a number. The value entered  *
// is stored in the variable pointed to by input.               *
//*****************************************************************

void getNumber(int *input)
{
   cout << "Enter an integer number: ";
   cin >> *input;
}

//*****************************************************************
// Definition of doubleValue. The parameter, val, is a pointer. *
// This function multiplies the variable pointed to by val by   *
// two.                                                         *
//*****************************************************************

void doubleValue(int *val)
{
   *val *= 2;
}
```

Program Output with Example Input Shown in Bold

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

Keyword const with pointers/data

constant **data**	constant **pointer**

[const] <type> *** [const] <variable>**

```
int x = 100;

const int * const xPtr = &x;

// xPtr is a constant pointer to an integer constant
```

Creating a constant pointer

- The const qualifier is used by a programmer to inform the compiler that the value of a particular variable should not be modified.
  - The **const** qualifier can be applied to a **pointer** making it a constant pointer.
  - **int x = 100, y = 10;**
  - **int * const intPtr = &x;**
    // defines intPtr as a constant pointer to an integer
  - **intPtr = &y;**
    // error! intPtr is a constant pointer
  - ***intPtr += 100;**
    // x now contains 200 (constant pointer, not data)
  - **A constant pointer cannot be assigned a new address**

Creating a pointer to constant data

– A pointer can be declared to point to a constant type. In this situation the pointer can be assigned a new address but the data in the object it points to cannot be modified.

– **int x = 100, y = 10;**

– **const int * intPtr = &x;**
// intPtr is a <u>pointer to an integer constant</u>

– **intPtr = &y;**
// intPtr now points to y

– ***intPtr += 100;**
// error! Cannot change constant data

– A constant variable cannot be updated through a dereferenced pointer.

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 double constants.

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.

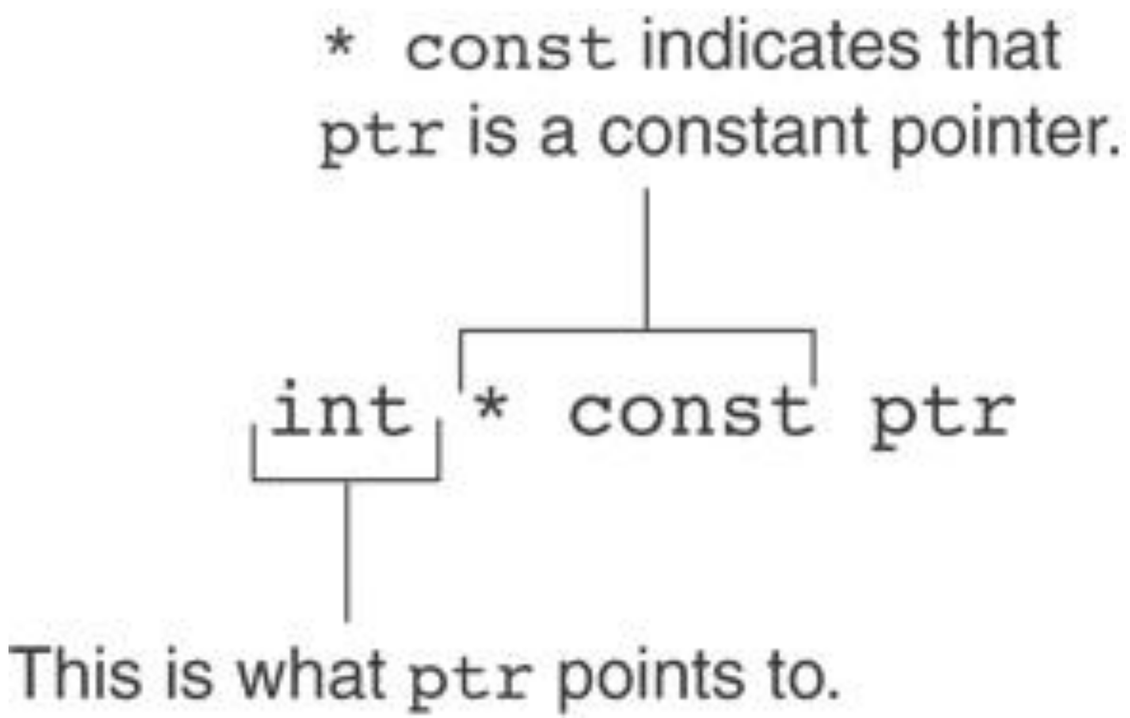
Constant Pointers

- 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 Pointer to Constant Data

- 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:

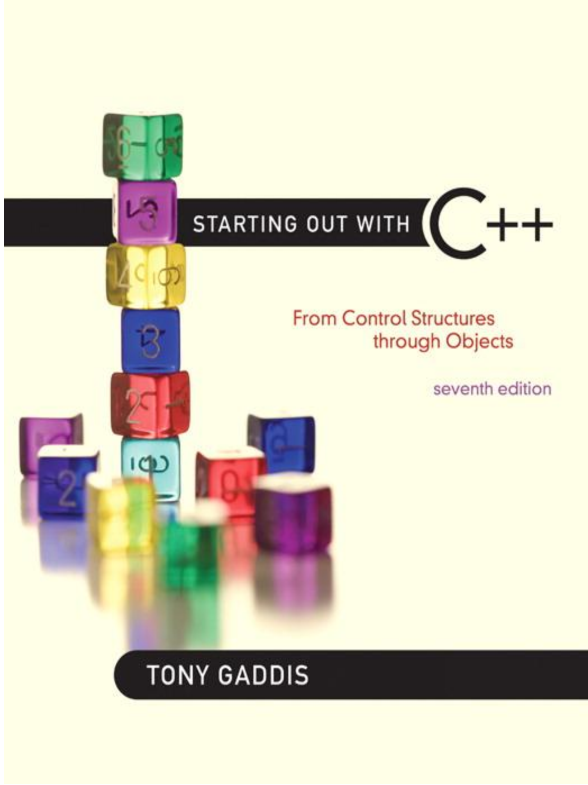
```
int value = 22;
const int * const ptr = &value;
```

Constant Pointer to Constant Data

`* const` indicates that `ptr` is a constant pointer.

```
const int * const ptr
```

This is what `ptr` points to.



9.8

Dynamic Memory Allocation

Dynamic Memory Allocation [C++]

- 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`** operator returns address of memory location

Dynamic Memory Allocation

- Can also use `new` to allocate an array:

```
const int SIZE = 25;
double * 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!

Program 9-14

```
// This program totals and averages the sales figures for any
// number of days. The figures are stored in a dynamically
// allocated array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   double *sales,       // To dynamically allocate an array
          total = 0.0, // Accumulator
          average;     // To hold average sales
```

Program 9-14 *(continued)*

```
   int numDays,           // To hold the number of days of sales
       count;             // Counter variable

   // Get the number of days of sales.
   cout << "How many days of sales figures do you wish ";
   cout << "to process? ";
   cin >> numDays;

   // Dynamically allocate an array large enough to hold
   // that many days of sales amounts.
   sales = new double[numDays];

   // Get the sales figures for each day.
   cout << "Enter the sales figures below.\n";
   for (count = 0; count < numDays; count++)
   {
      cout << "Day " << (count + 1) << ": ";
      cin >> sales[count];
   }

```

Program 9-14 (Continued)

```
    // Calculate the total sales
    for (count = 0; count < numDays; count++)
    {
       total += sales[count];
    }

    // Calculate the average sales per day
    average = total / numDays;

    // Display the results
    cout << fixed << showpoint << setprecision(2);
    cout << "\n\nTotal Sales: $" << total << endl;
    cout << "Average Sales: $" << average << endl;

    // Free dynamically allocated memory
    delete [] sales;
    sales = 0;       // Make sales point to null.

    return 0;
}
```

Program Output with Example Input Shown in Bold

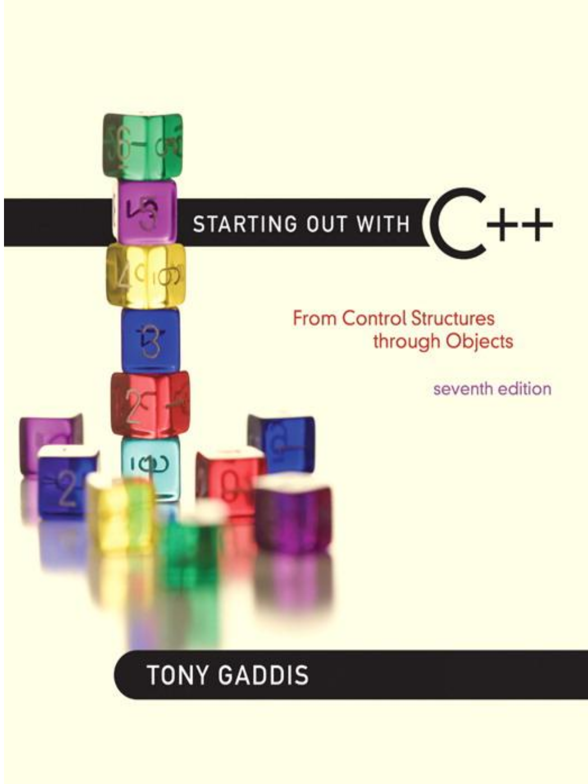
```
How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

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

Notice that in line 49 the value 0 is assigned to the `sales` pointer.

It is a good practice to store 0 in a pointer variable after using delete on it.

> First, it prevents code from inadvertently using the pointer to access the area of memory that was freed.
>
> Second, it prevents errors from occurring if `delete` is accidentally called on the pointer again.

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

```
int *getRandomNumbers(int num)
{
   int *array;    // Array to hold the numbers

   // Return null if num is zero or negative.
   if (num <= 0)
      return NULL;

   // Dynamically allocate the array.
   array = new int[num];

   // Seed the random number generator by passing
   // the return value of time(0) to srand.
   srand( time(0) );

   // Populate the array with random numbers.
   for (int count = 0; count < num; count++)
      array[count] = rand();

   // Return a pointer to the array.
   return array;
}
```

A Final array example

```
#include <iostream>
using namespace std;
int main()
{
    int sales[3][4] = { {100,500,200,250}, {300,250,400,500}, {450,350,400,200} };

    cout << "The sales array is defined as:  int sales[3][4]" << endl << endl;
    cout << "The value of sales is:    " << sales  << endl;

    for (int j =0; j<3; j++)
           cout << "The value of sales[" << j << "] is: " <<  sales[j] << endl;
    cout << endl;

    for (int m=0; m<3; m++)
       for (int n=0; n<4; n++)
           cout << sales[m][n] << "  ";
    cout << endl;

    for (m=0; m<3; m++)
       for (int n=0; n<4; n++)
           cout << *(*(sales+m)+n) << "  "; // array name – offset notation for a
    cout << endl << endl;                     // two-dimensional array
    return 0;
}

    *(*(*(*(name+d1)+d2)+d3)+d4)…        // equivalent to name[d1][d2][d3][d4]…

    name is the name of the array or pointer
    d1, d2, d3 … represent values used for the different dimensions in a
                 multi-dimensional array
```

The sales array is defined as: `int sales[3][4];`

The value of `sales` is: 0x0012FF50

The value of `sales[0]` is: 0x0012FF50

The value of `sales[1]` is: 0x0012FF60

The value of `sales[2]` is: 0x0012FF70

100 500 200 250 300 250 400 500 450 350 400 200

100 500 200 250 300 250 400 500 450 350 400 200