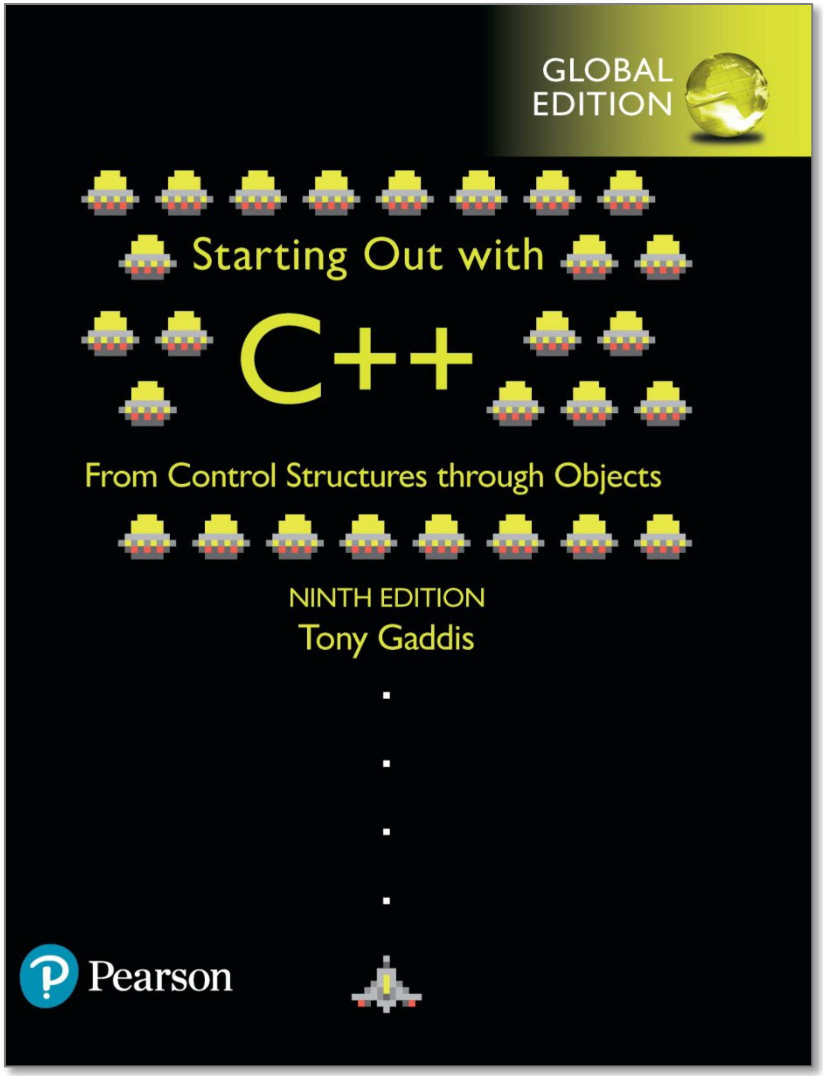
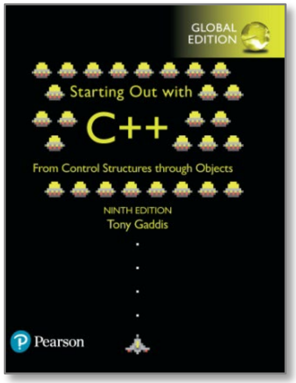
Chapter 9:

Pointers





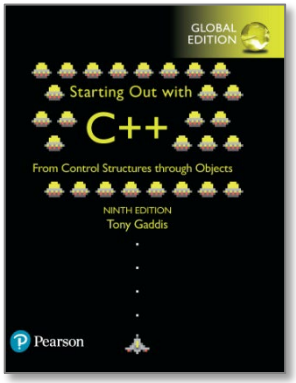
9.1

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

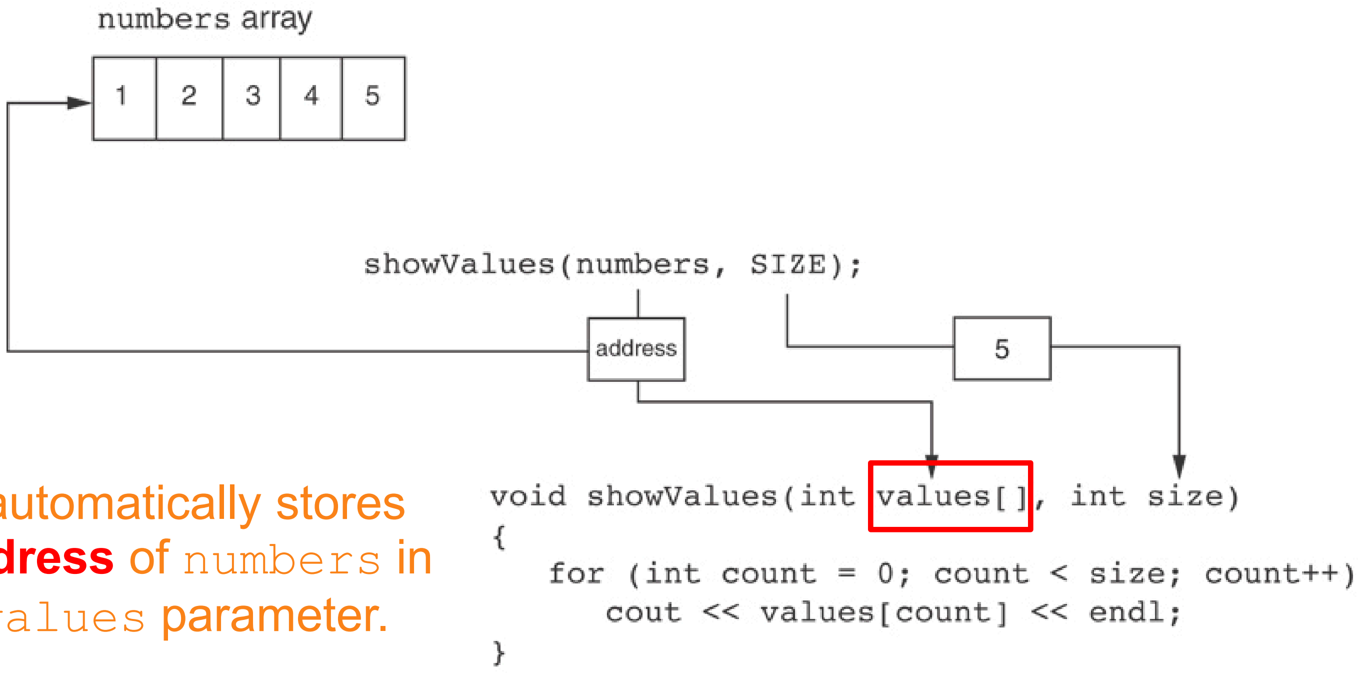
Something Like Pointers: Arrays

- 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.



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

```
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:

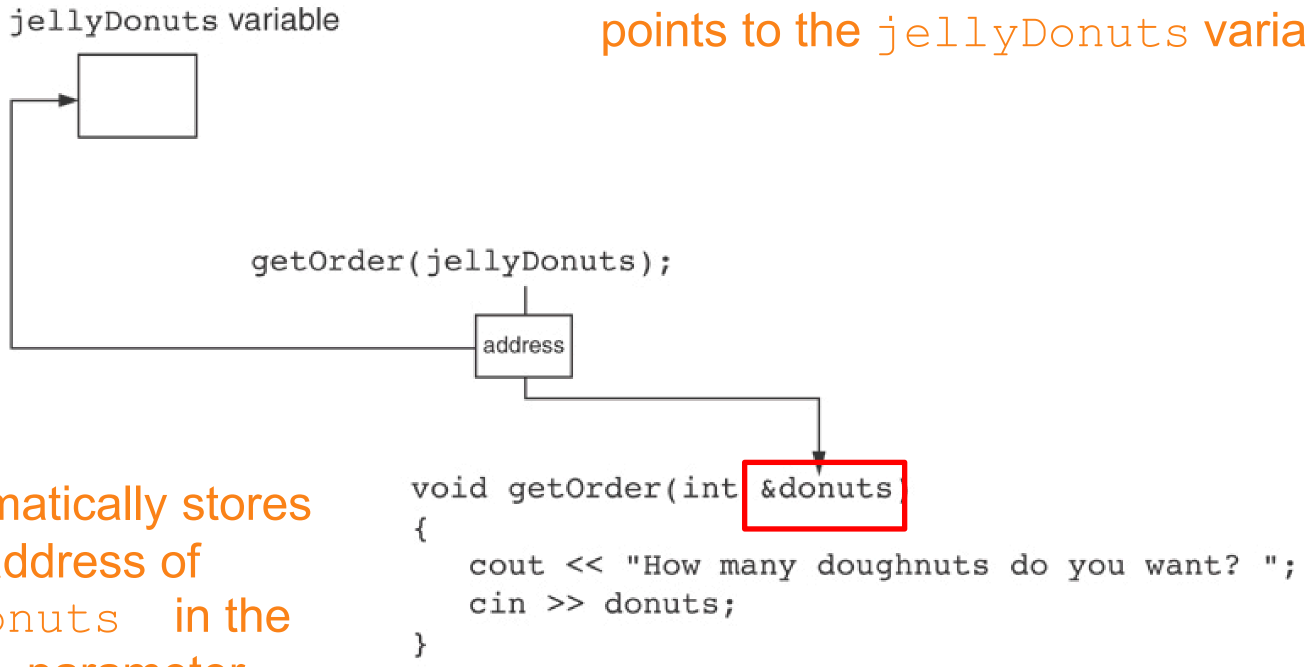
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

- And we call it with this code:

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

Something Like Pointers: Reference Variables

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



```
getOrder(jellyDonuts);
```

```
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
```

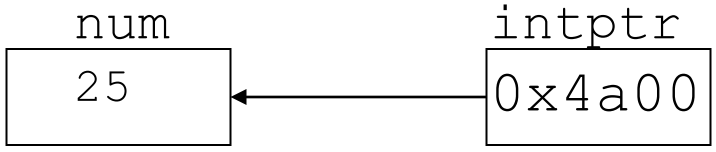
Pointer Variables

- Assigning an address to a pointer variable:

```
int *intptr; //pointer variable
int num = 25; //int variable
intptr = &num; //assign variable address
```

The ampersand operator can be read as "address of".

- Memory layout:



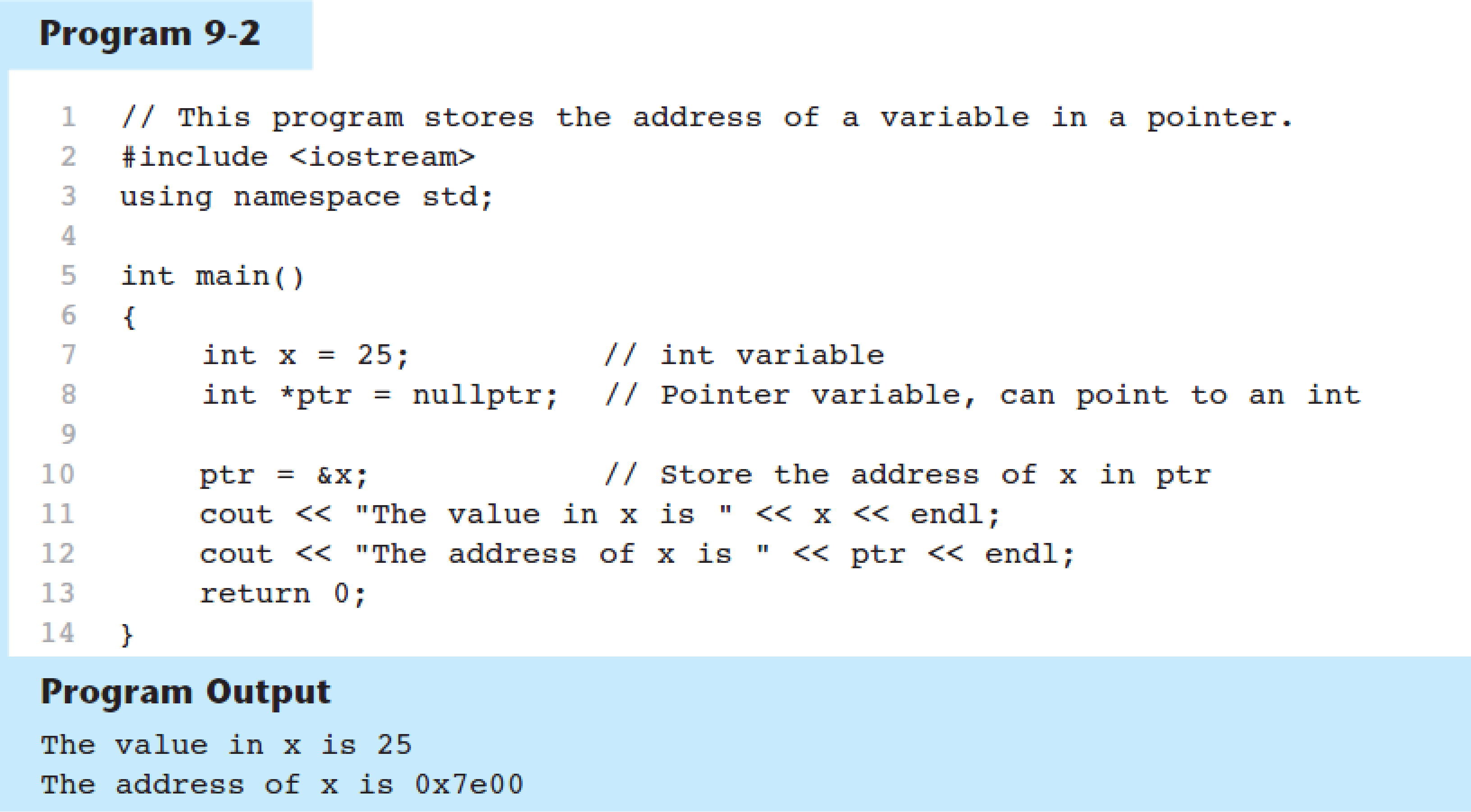
address of `num`: `0x4a00`

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

```
// 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 = nullptr;   // 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
```

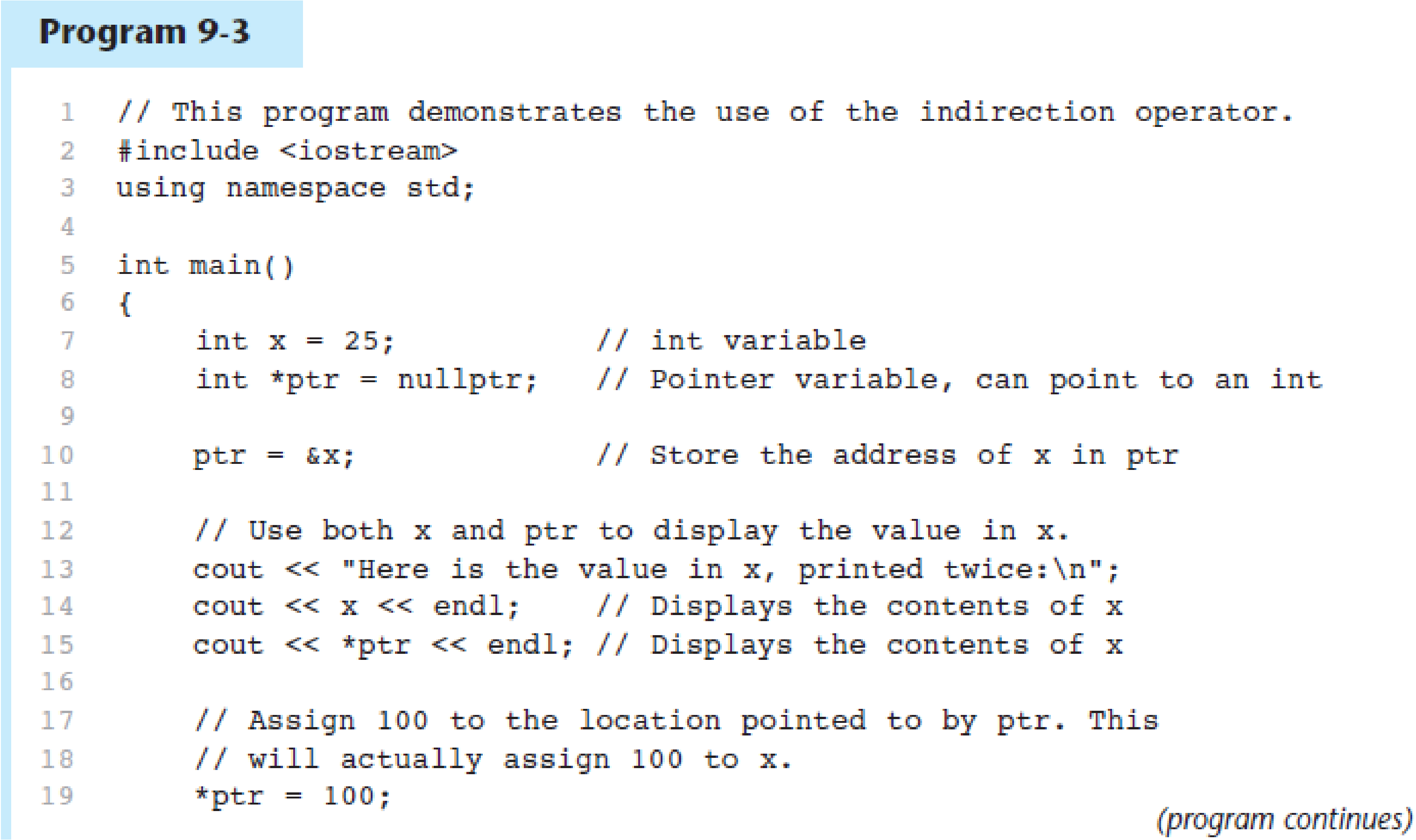
The Indirection Operator

- 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.

The Indirection Operator in Program 9-3

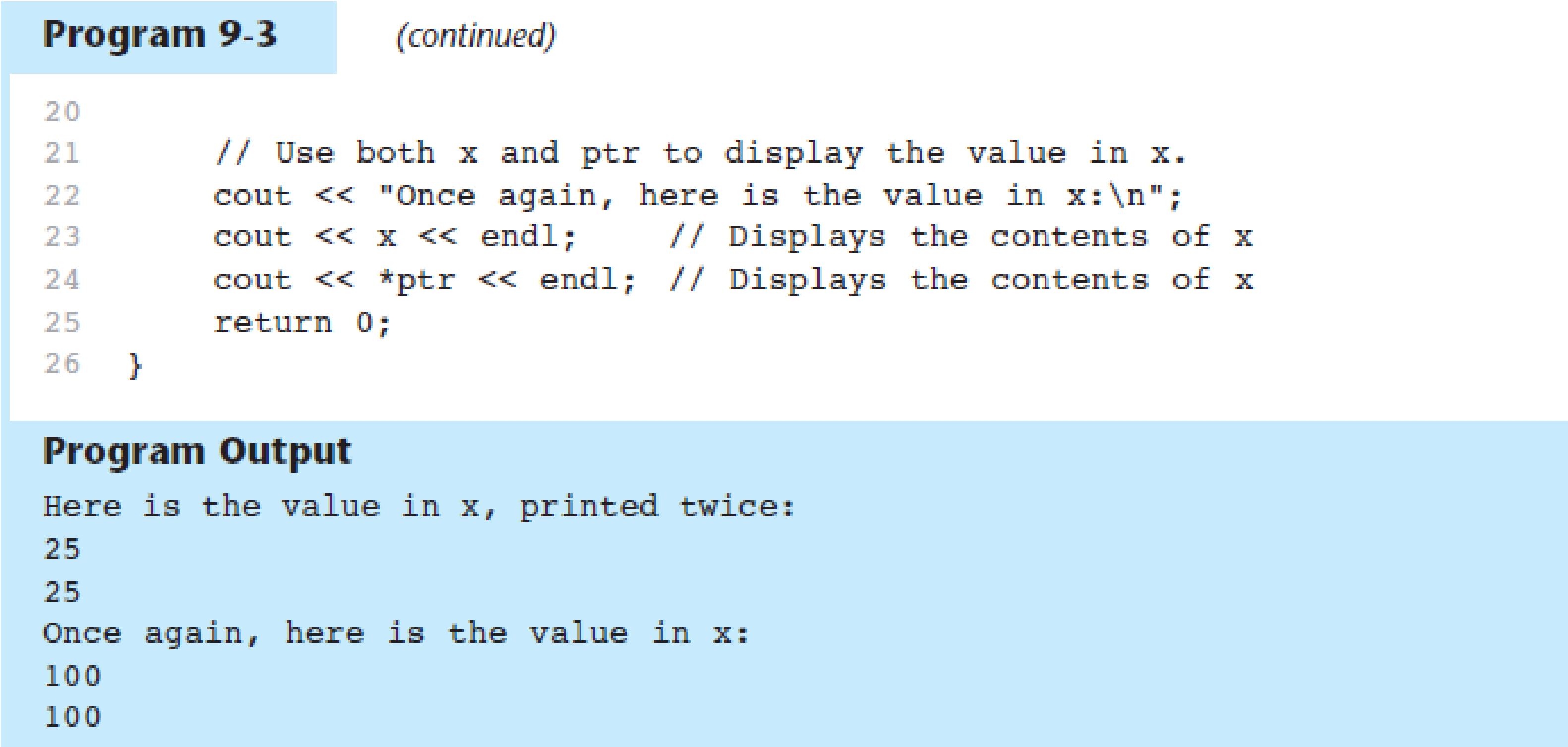


Program 9-3

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

int main()
{
    int x = 25;           // int variable
    int *ptr = nullptr;   // 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;
```

(program continues)

The Indirection Operator in Program 9-3

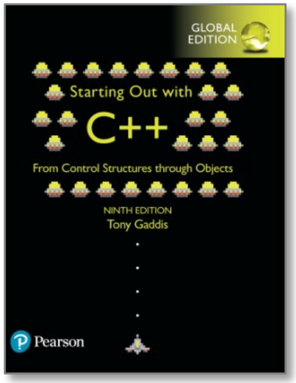


Program 9-3 *(continued)*

```
    // 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 Output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```



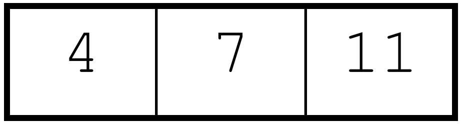
9.3

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

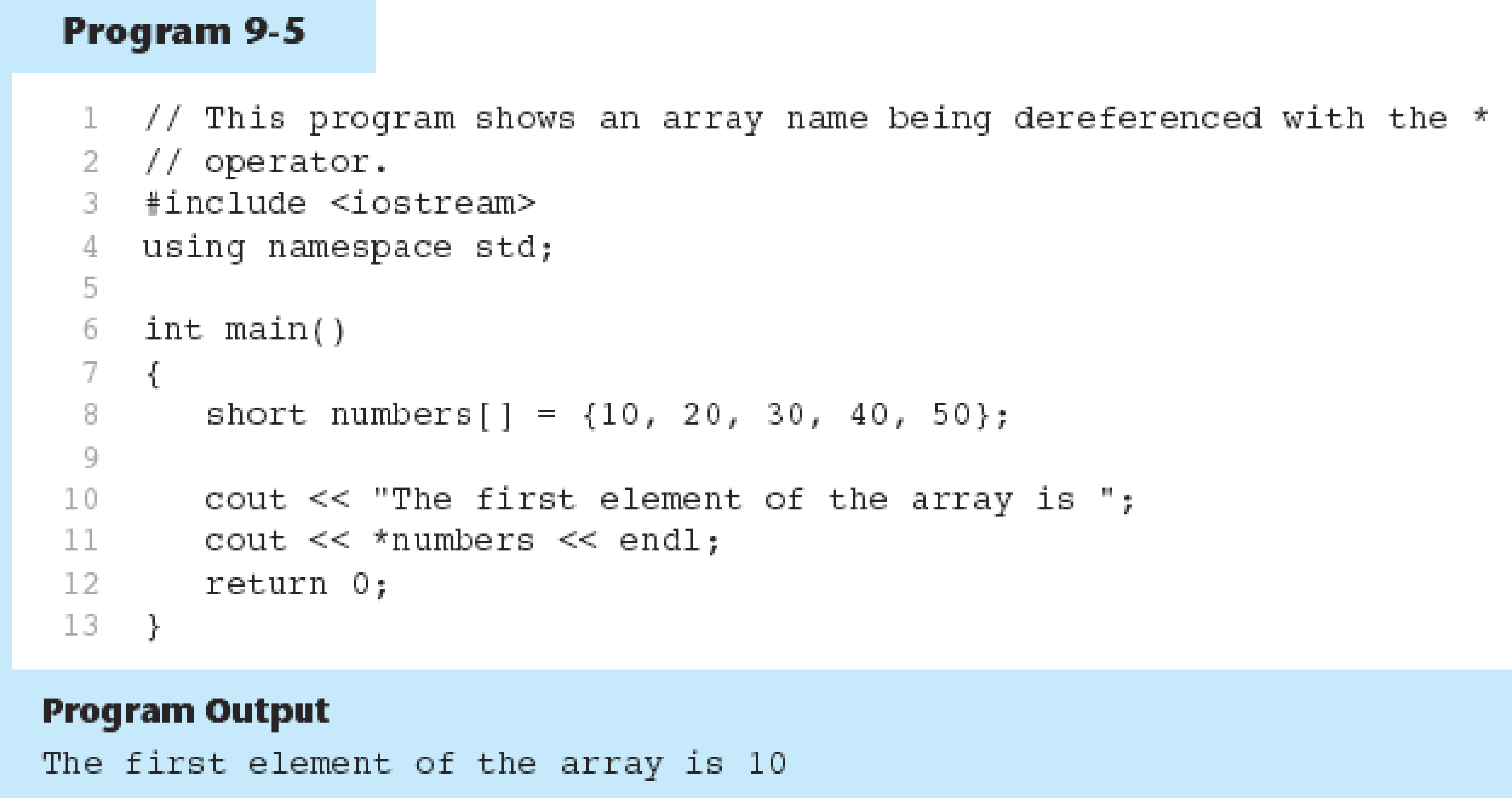
- Array name can be used as a pointer **constant**:

```
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

```
// 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

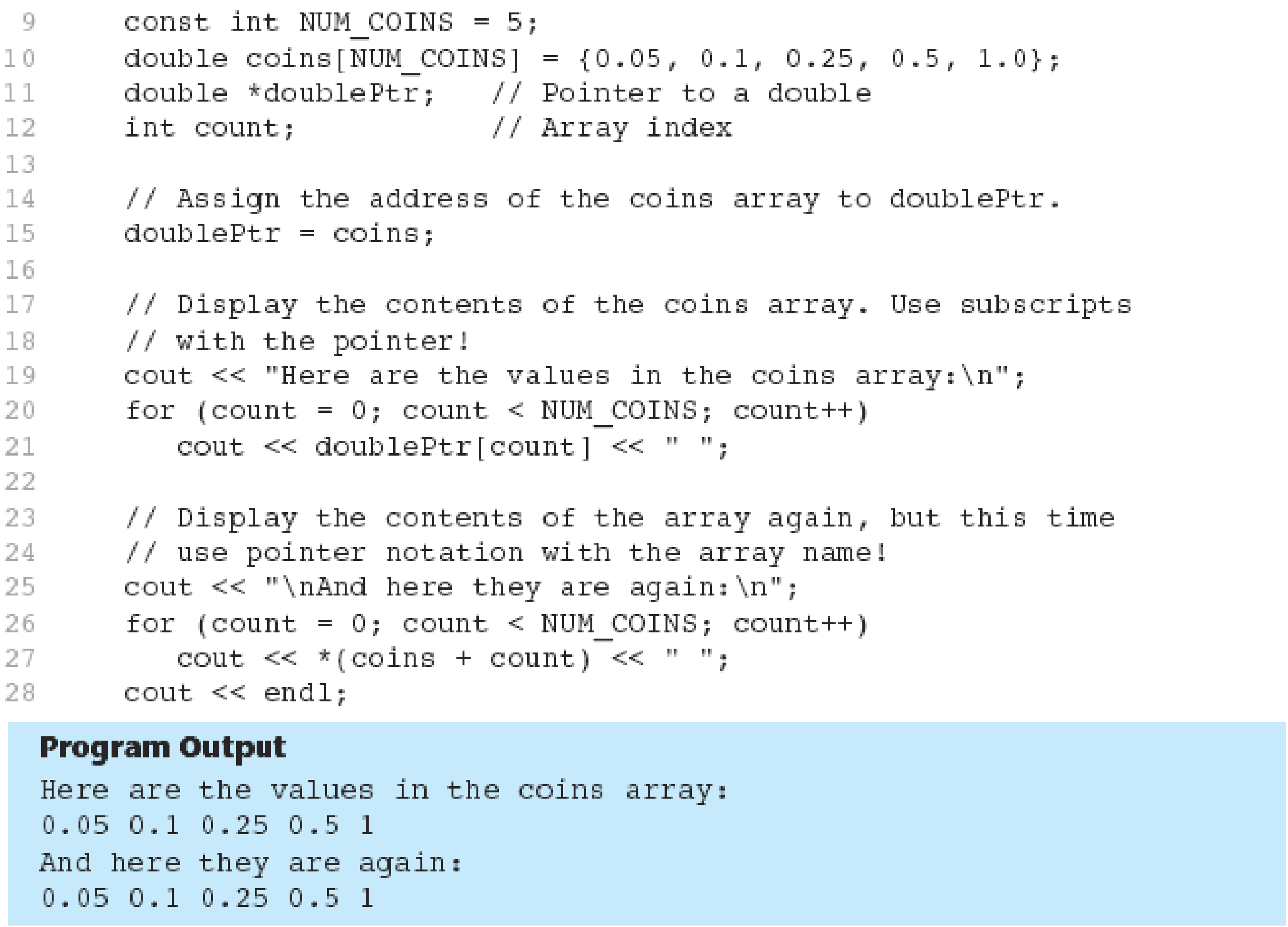
- Array elements can be accessed in many ways:

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

Array Access

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

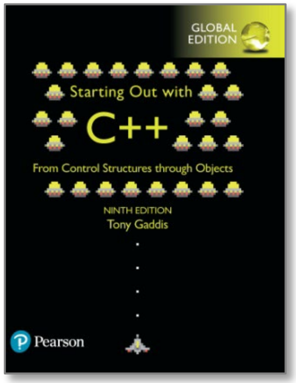
From Program 9-7



```
    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
```



GLOBAL EDITION

Starting Out with C++

From Control Structures through Objects

NINTH EDITION

Tony Gaddis

Pearson

9.4

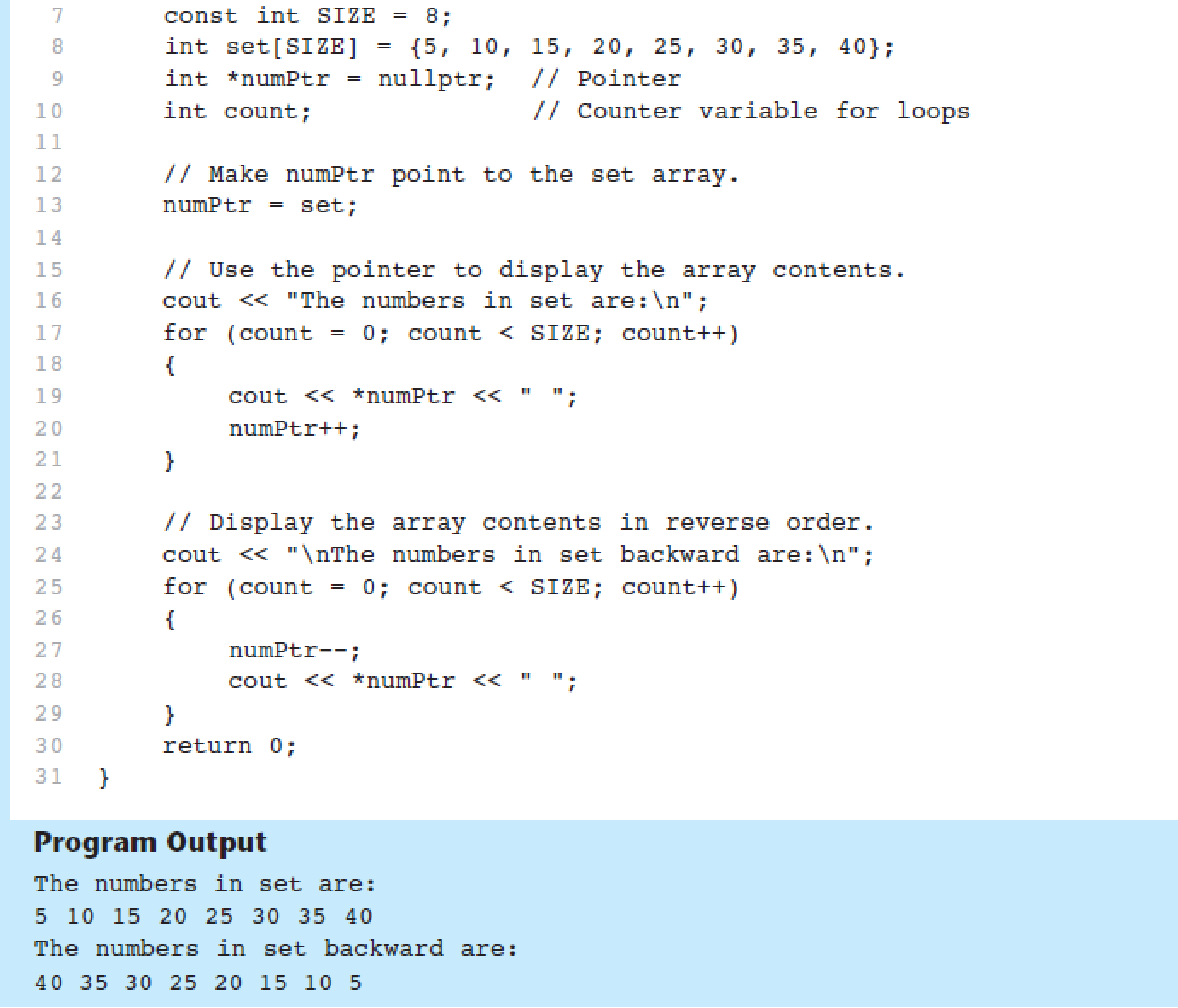
Pointer Arithmetic

Pointer Arithmetic

- Operations on pointer variables:

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

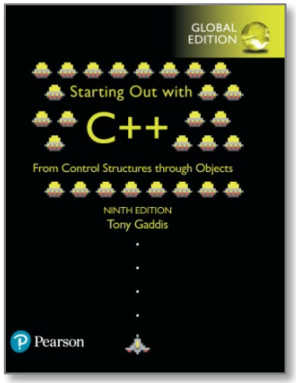
From Program 9-9



```
    const int SIZE = 8;
    int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *numPtr = nullptr;  // 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 << " ";
    }
    return 0;
}
```

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**:

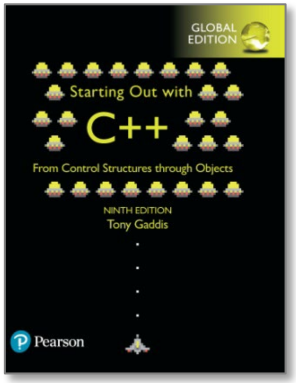
```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

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

- Can test for an invalid address (e.g. null) for `ptr` with:

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



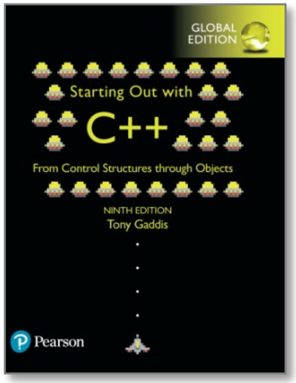
9.6

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)


if (*ptr1 == *ptr2)
```



GLOBAL EDITION

Starting Out with C++

From Control Structures through Objects

NINTH EDITION

Tony Gaddis

Pearson

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
```

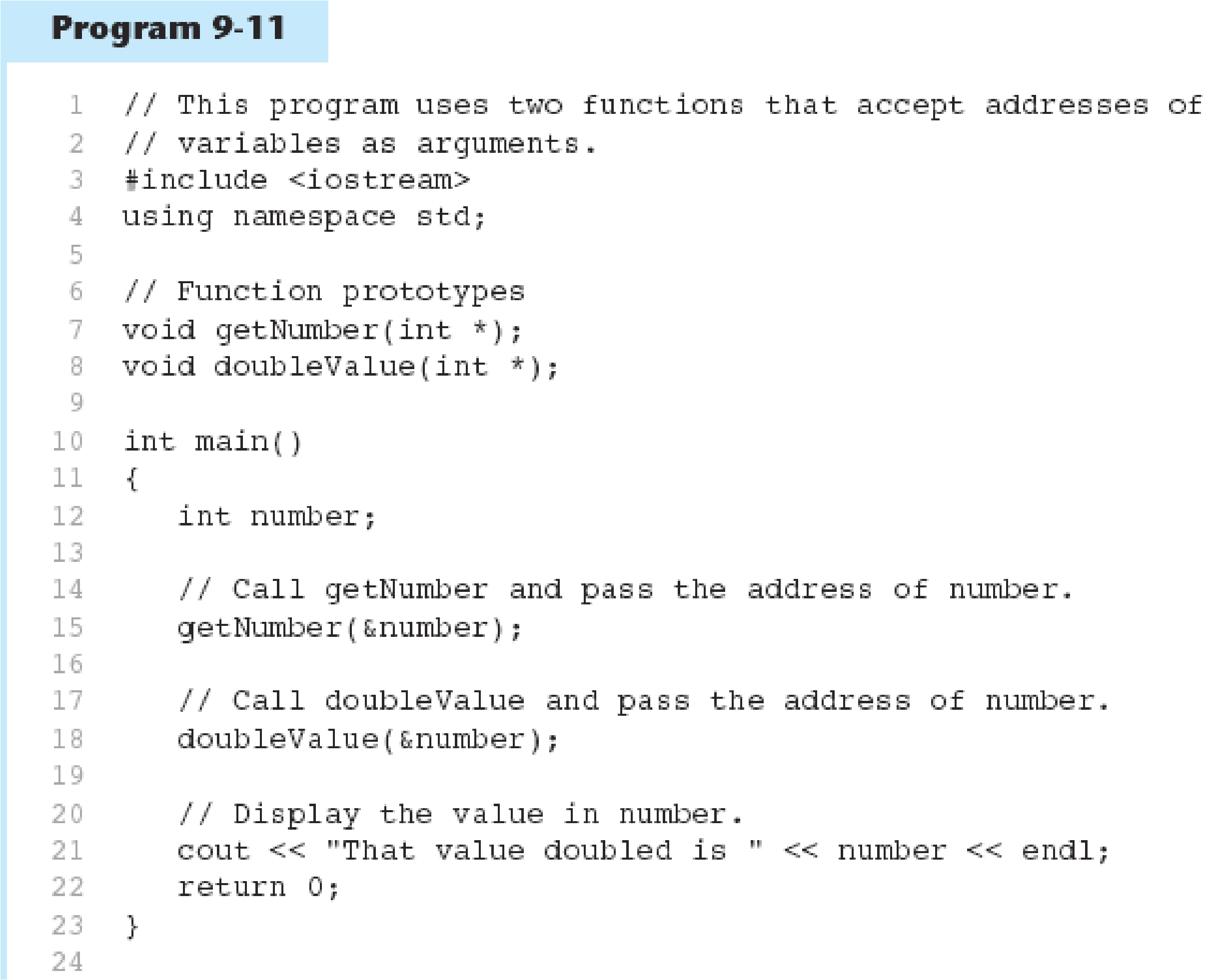
Example

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

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

Comparison to passing by reference

```
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

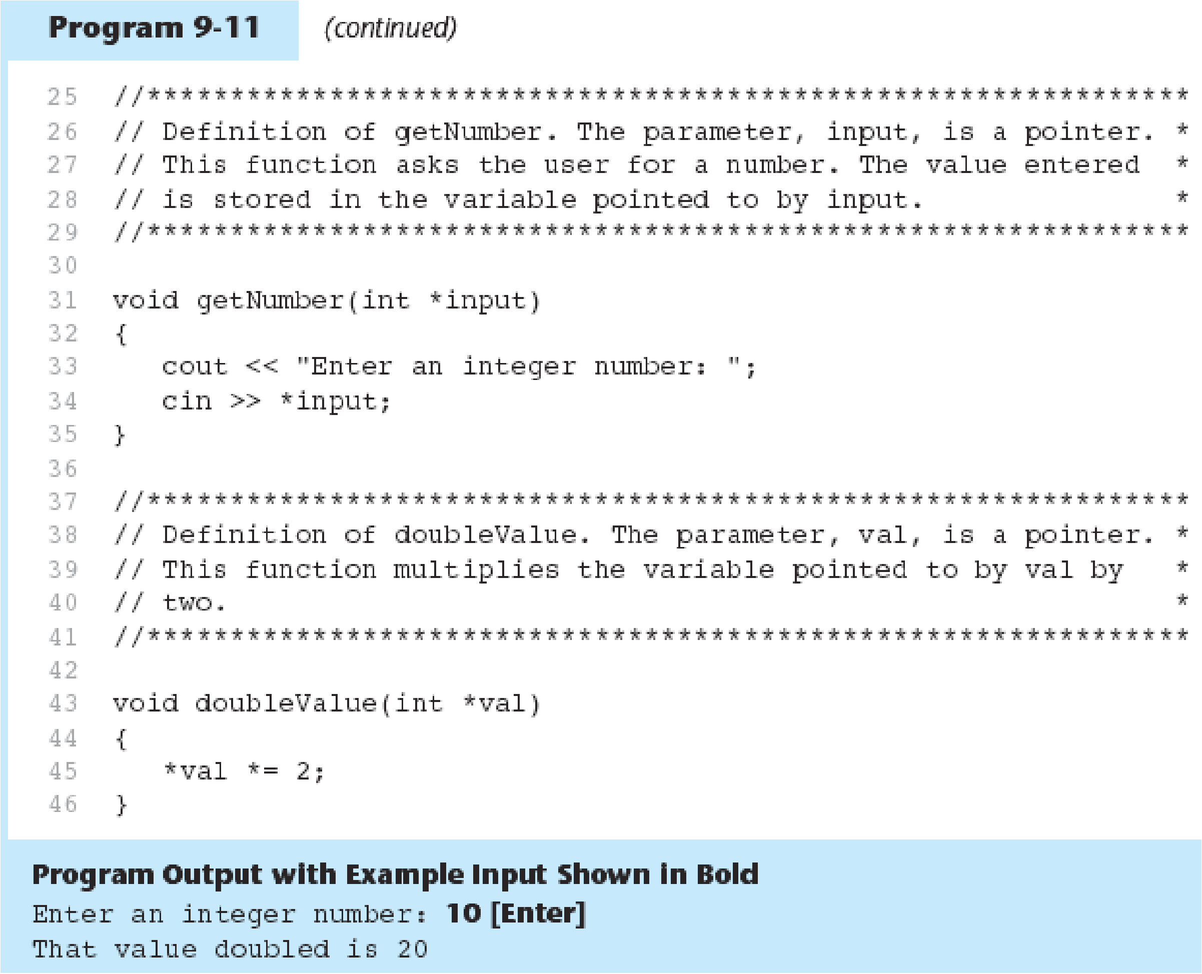


Program 9-11

```cpp
// 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)

Pointers as Function Parameters in Program 9-11

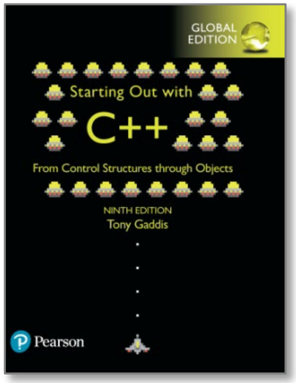


Program 9-11 *(continued)*

```
//***************************************************************
// 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
```



GLOBAL EDITION

Starting Out with C++

From Control Structures through Objects

NINTH EDITION
Tony Gaddis

Pearson

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:

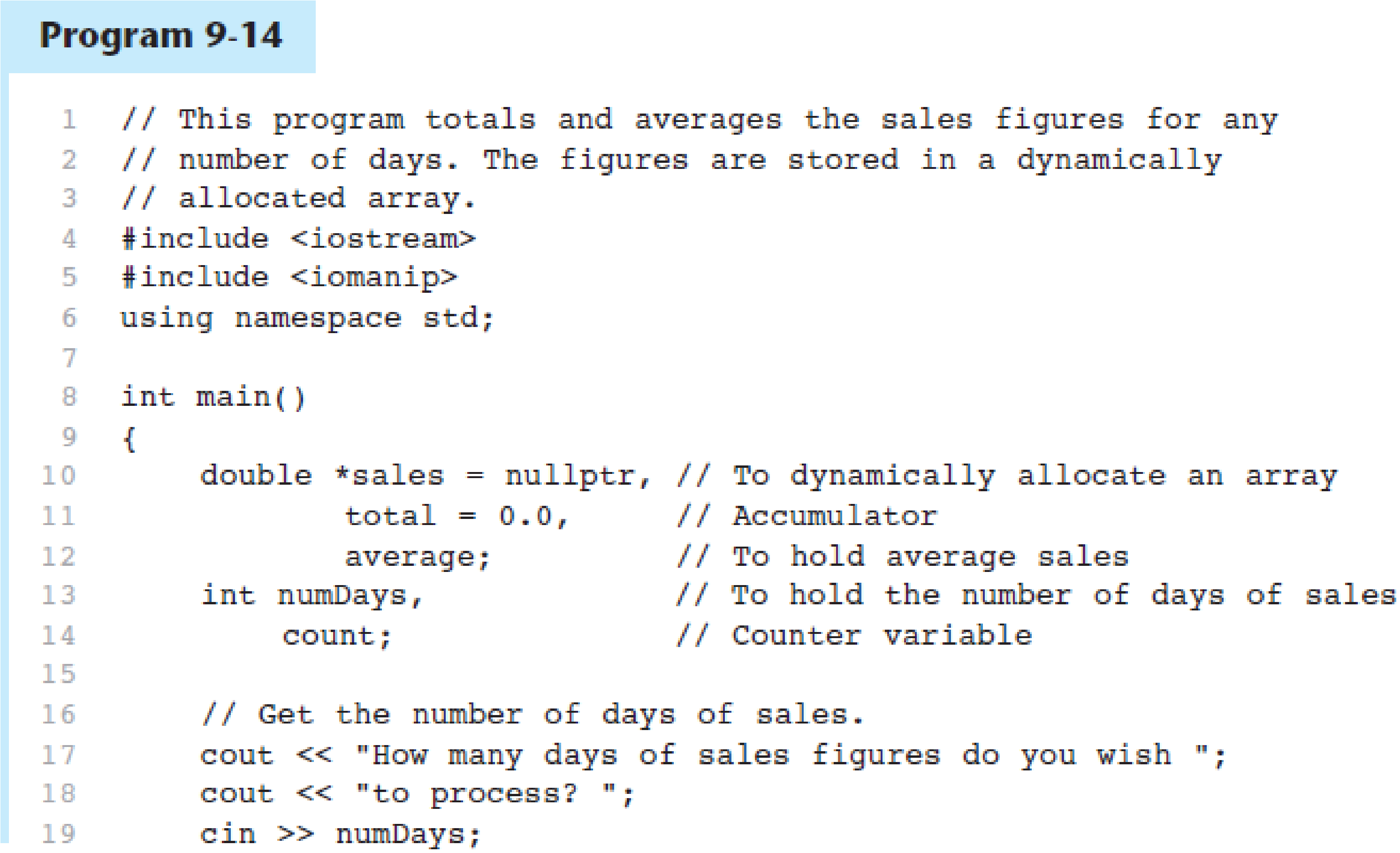
```
delete fptr;
```

- Use `[]` to free dynamic array:

```
delete [] arrayptr;
```

- Only use `delete` with dynamic memory!

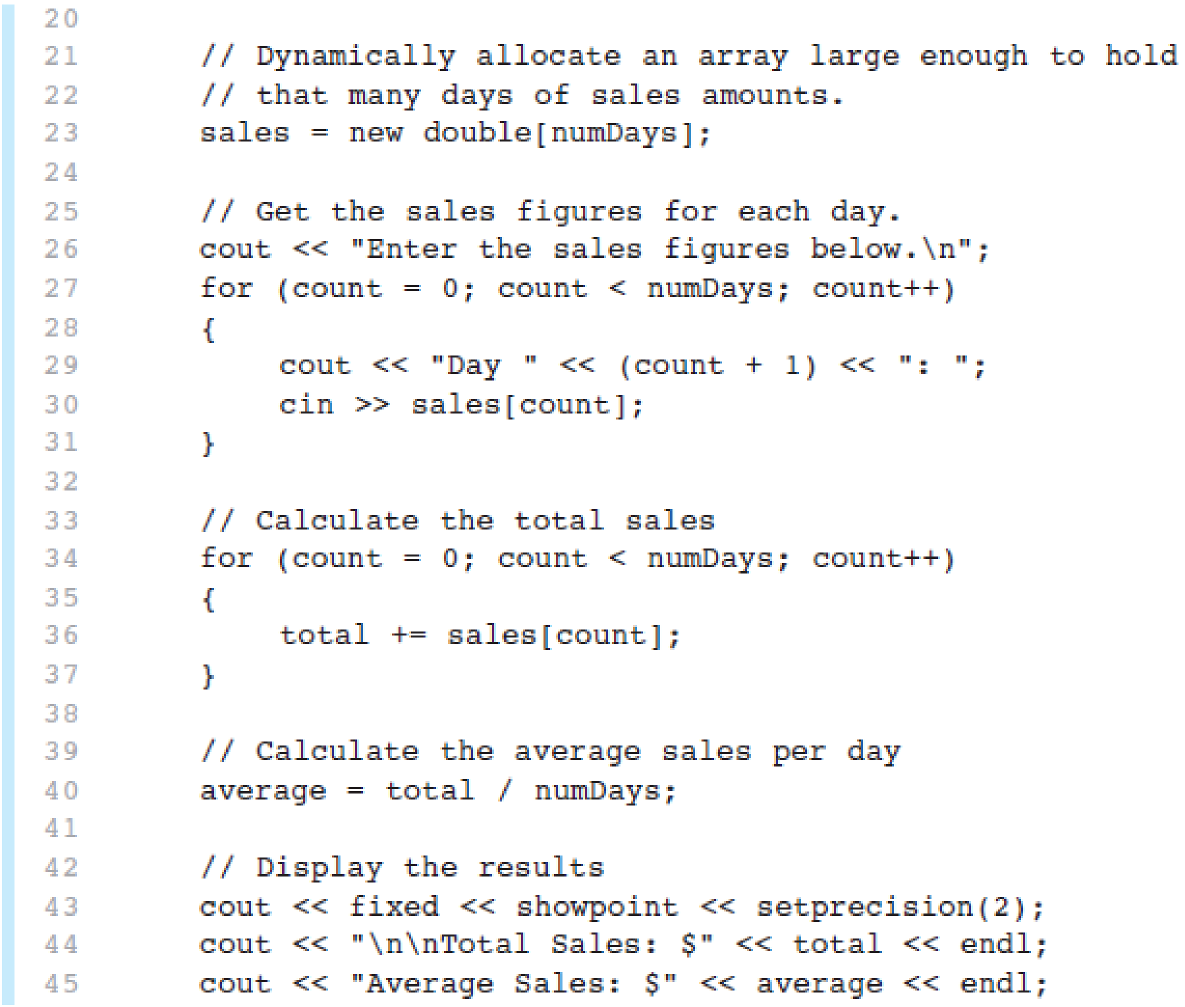
Dynamic Memory Allocation in Program 9-14



Program 9-14

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

Dynamic Memory Allocation in Program 9-14

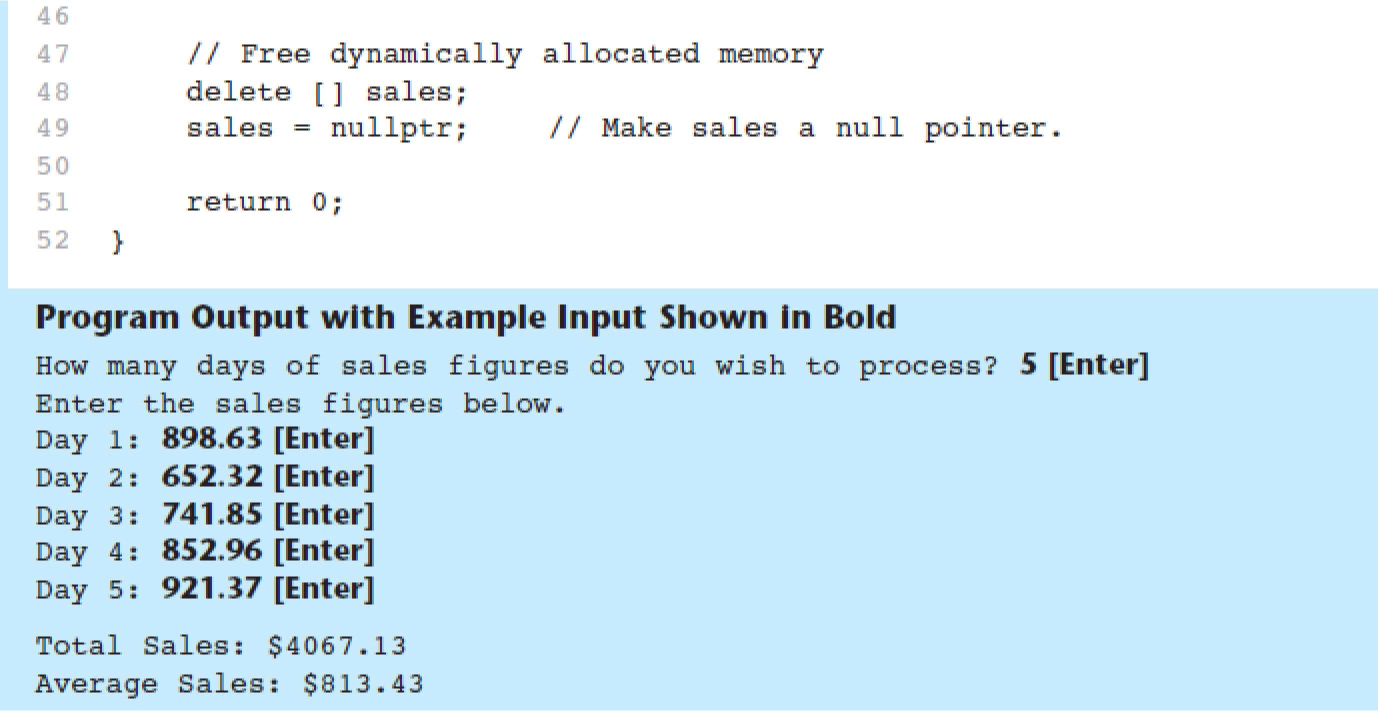


```

        // 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];
        }

        // 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;
```

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.*