**Chapter 9**

**Pointers**

**The & operator**

* 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

**Pointer Variables**

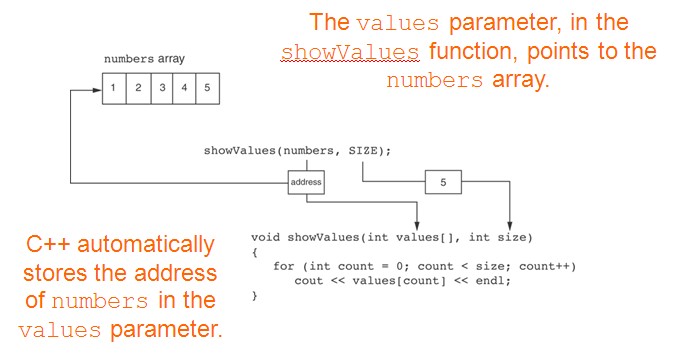
* Pointer variable : Often just called a *pointer*, it's a variable that holds an address
* Because a pointer variable holds the address of another piece of data, it "points" to the data

**Examples of Pointers that we’ve already used**

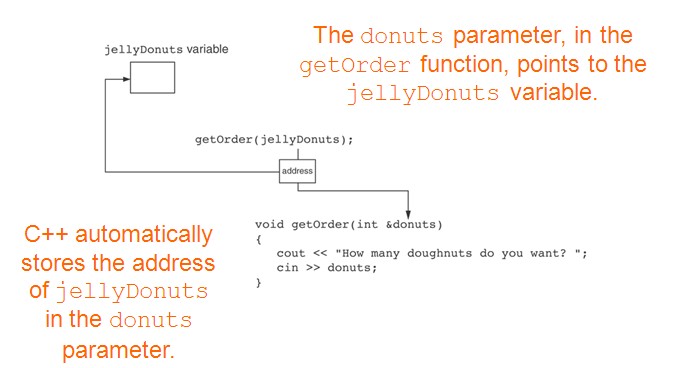
We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.

const int SIZE = 5;

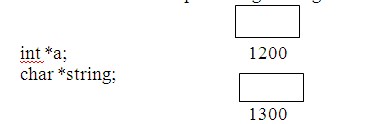
int numbers[SIZE];



* 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);



A **pointer** is a datatype (just like int, char, float is a datatype) used for declaring variables which can contain addresses.

**Syntax:**

Data\_type\* var\_name;

Examples:

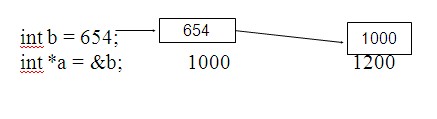
int\* a;

double\* b;

char\* string;

Spacing not important when declaring a pointer.

**Initializing Pointers**

****

**Indirection Operator (\*) - Dereferencing a Pointer**

* 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

In other words, dereferencing a pointer consists of:

1. Go to the address that I (i.e., the pointer variable) contains

2. Access what is at that address

**Example of using Indirection Operator**

int b = 654;

int\* a = &b;

cout << \*a << endl;

cout << b << endl;

cout << &a << endl;

//654 is displayed by the first cout statement

//654 is displayed by the second cout statement

//1200 is displayed by the third cout statement

What happens here?

int\*\* c = &a;

**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

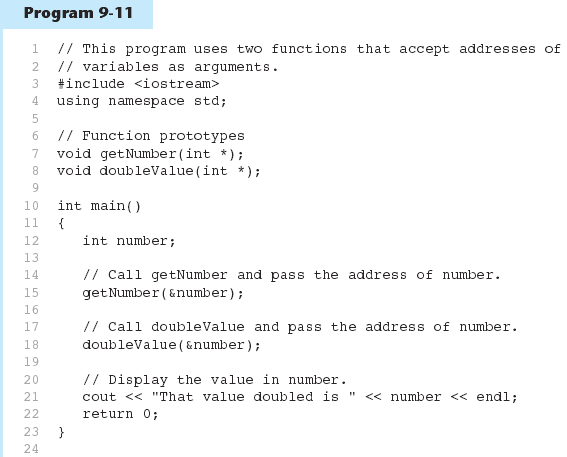
1. asterisk **\*** in body to dereference the pointer

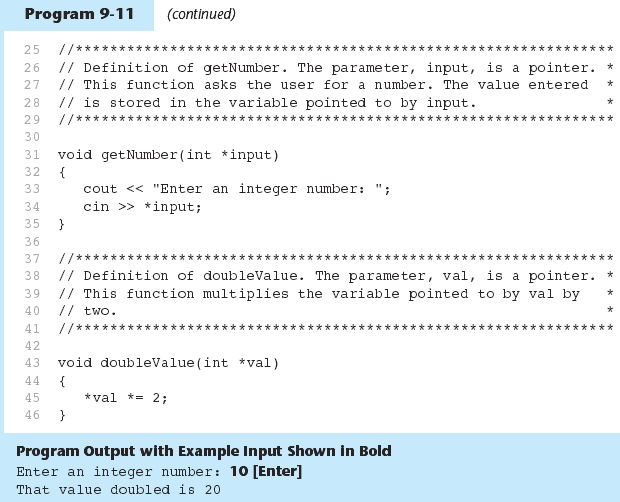
cin >> \*ptr;

3) address as argument to the function

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

**Example**

****

****

**Relationship Between Arrays and Pointers**

|  |  |  |
| --- | --- | --- |
| **4** | **7** | **11** |

Array name is starting address of array

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

cout << vals; // displays address of ele sub 0

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

Arrays are constant pointers - cannot ever be changed.

* 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:

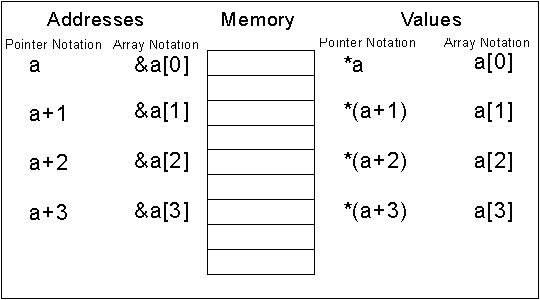
int\* valptr = vals;

cout << valptr[1]; // displays 7

**Pointer Arithmetic**

Consider the following array declaration:

int a[4] = {0,1,2,3};

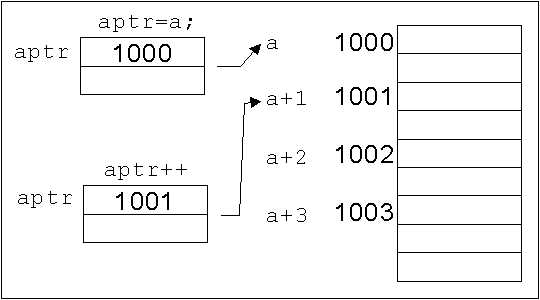


We can refer to the address of an element by either (a+index) or &a[index]. We can refer to the value stored in an array by either \*(a+index) or a[index]. Even though 4 bytes are allocated

for each element, we don’t have to add the 4 bytes .

Consider the following declaration:

int\* aptr = a;



We place the address of the array a into a pointer variable

called aptr. We can refer to aptr in much the same way that

we refer to a. We can use aptr++ or aptr--, but not a++

or a--!