07: POINTERS

Programming Technique I

(SCSJ 1013)

Topic Outline

- 1 - Getting the Address of a Variable
- 2 - Pointer Variables
- 3 - The Relationship Between Arrays and Pointers
- 4 - Pointer Arithmetic
- 5 - Initializing Pointers
- 6 - Comparing Pointers
- 7 - Pointers as Function Parameters
- 8 - Dynamic Memory Allocation
- 9 - Returning Pointers from Functions

1- Getting the Address of a Variable

Addresses and Pointers

Address:

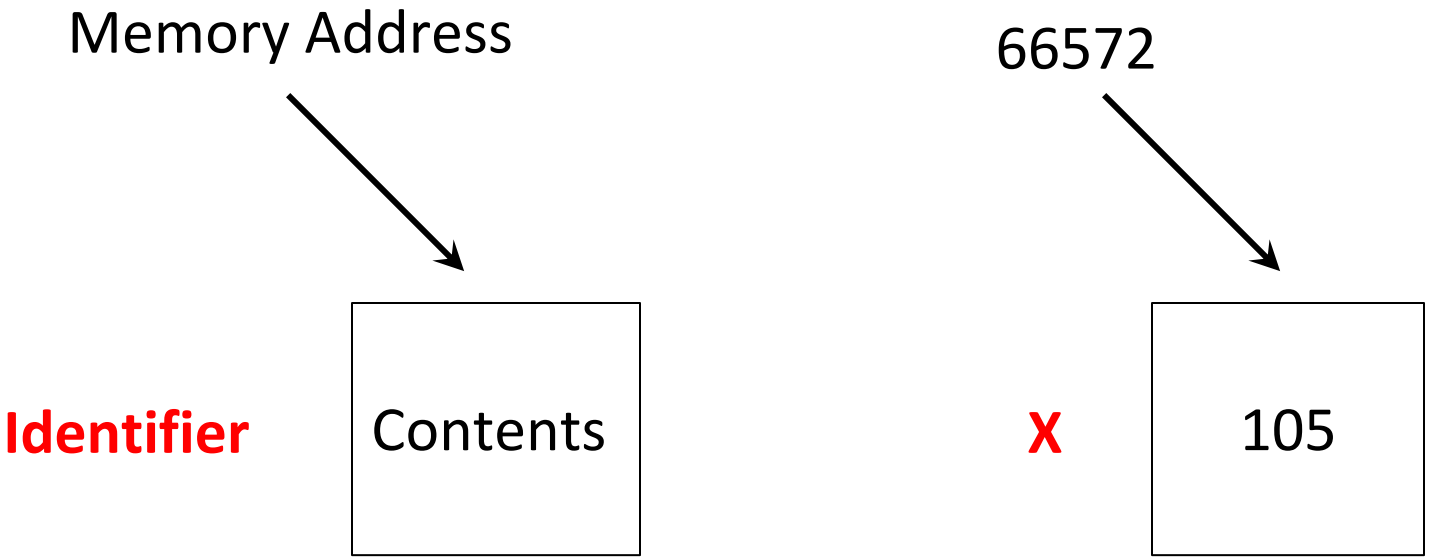
- A uniquely defined memory location which is assigned to a variable.
- Example - a positive integer value

<An analogy with post box>

Post office box number 78	**Individual name John Ruiz**	**Contents Catalog**
Memory Address 66572	Identifier X	Contents 105

Notation for Memory Snapshot

Memory Address	Identifier	Contents
66572	X	105



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:

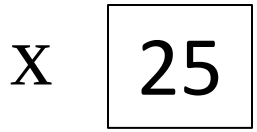
```
int num = -99;
cout << &num; // prints address
              // in hexadecimal
```

Example 1.1

```
#include <iostream>
using namespace std;

int main()
{
  int x=25;
   cout<<"The address of x is= "<<&x<<endl;
   cout<<"The value in x is "<< x<<endl;
}
```

Result of Example 1.1




```
The address of x is  0x8f05
The value in x is 25;
```

Exercise 1

- Type & execute the following program
- Check with your friend the address displayed.

```
#include <iostream>
using namespace std;

int main()
{
  int x=25;
   cout<<"The address of x is= "<<&x<<endl;
   cout<<"The value in x is "<< x<<endl;
}
```

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

Pointer Variables (cont.)

- Definition:

```
int  *intptr;
```

- Read as:

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

- Spacing in definition does not matter:

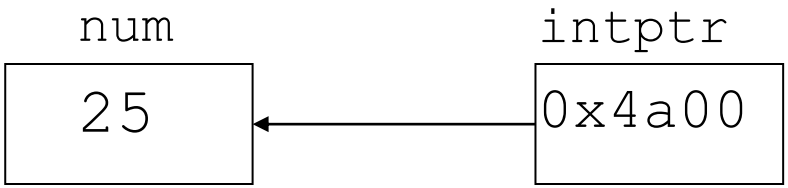
```
int * intptr;  // same as above
int*  intptr;  // same as above
```

Pointer Variables (cont.)

- Assigning an address to a pointer variable:

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

- Memory layout:



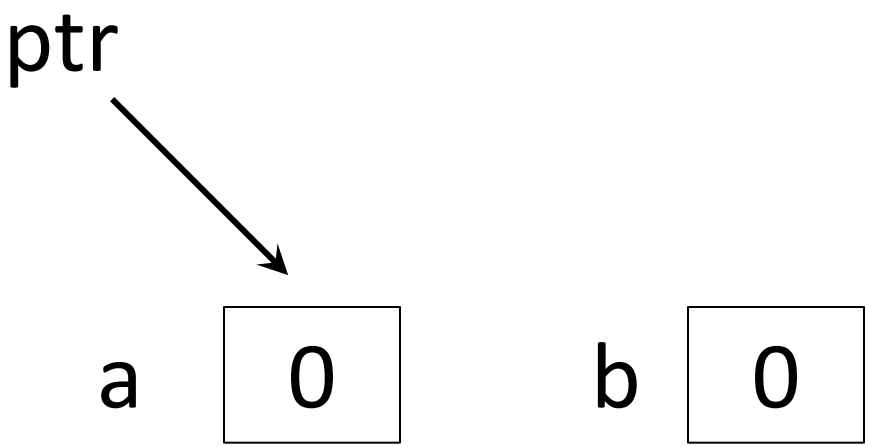
address of `num`: `0x4a00`

Pointer Variables (cont.)

```
int  a,  b,  *ptr;
ptr  =  &a;
```

or

```
int a, b, *ptr =  &a;
```



Example 2.1

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;       // 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.

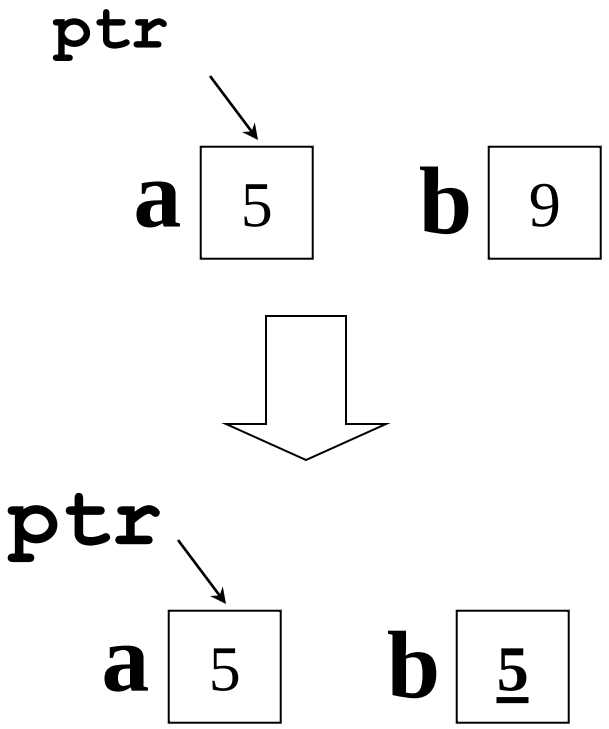
```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25.

Pointer Variables (cont.)

```
int a = 5, b = 9,
*ptr = &a;


b = *ptr;
```



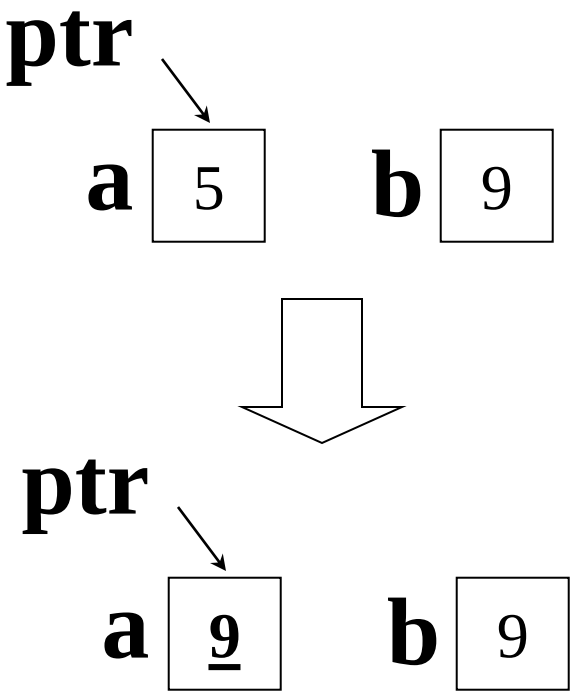
b=*ptr : **b** is assigned the value pointed to by **ptr**

Pointer Variables (cont.)

```
int a = 5, b = 9,
  *ptr = &a;



*ptr = b;
```



`*ptr` = `b`: the value pointed to by `ptr` is assigned the value in `b`.

Example 2.2

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;       // 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;
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 }
```

Exercise 2

- Give memory snapshots after each of these sets of statements are executed.

1.
```
int a=1, b=2, *ptr;
...
ptr = &b;
```

2.
```
int a=1, b=2, *ptr=&b;
...
a = *ptr;
```

3.
```
int a=1, b=2, c=5, *ptr=&c;
...
b = *ptr;
*ptr = a;
```

4.
```
int a=1, b=2, c=5, *ptr;
...
ptr = &c;
c = b;
a = *ptr;
```

Exercise 3

- Refer to Exercise 10.14 no 3 (b) pg. 297.
- Solve the problem.

Something like Pointers: Arrays

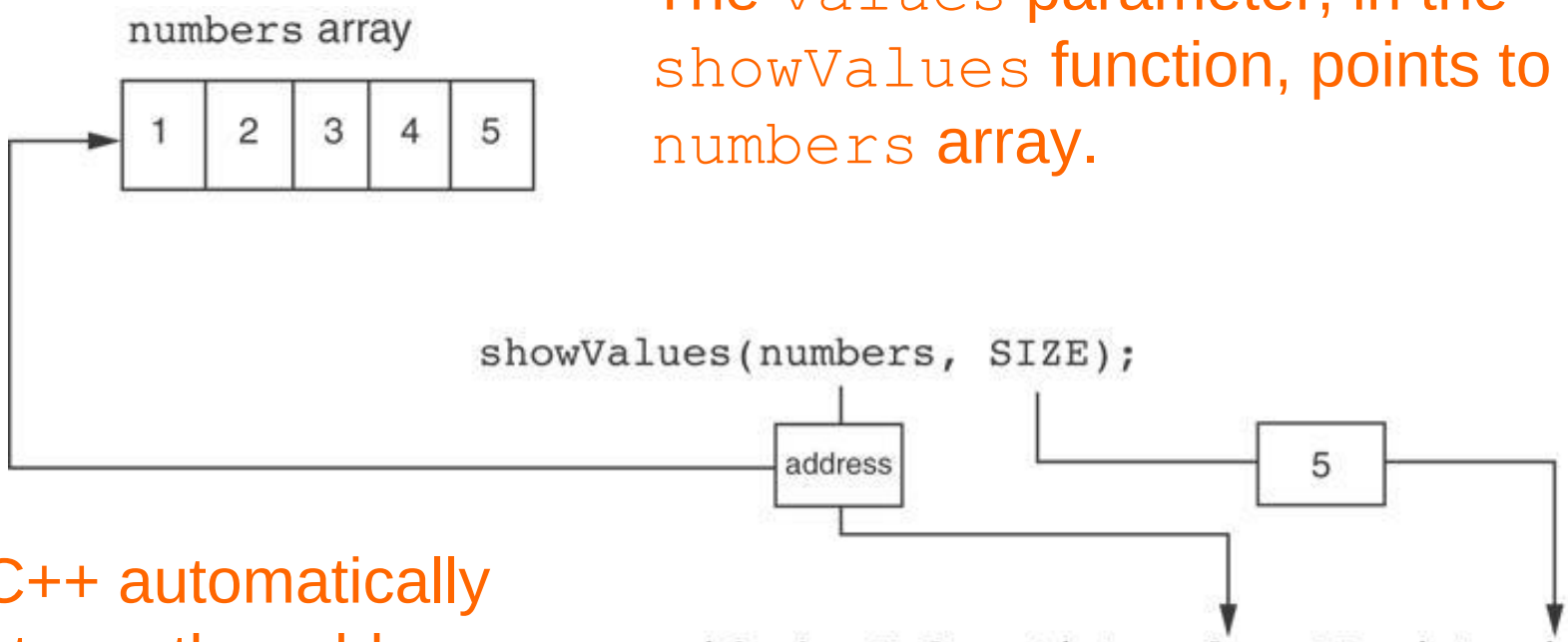
- We have already worked with something similar to pointers, when we learned to pass arrays as arguments to functions.

- 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

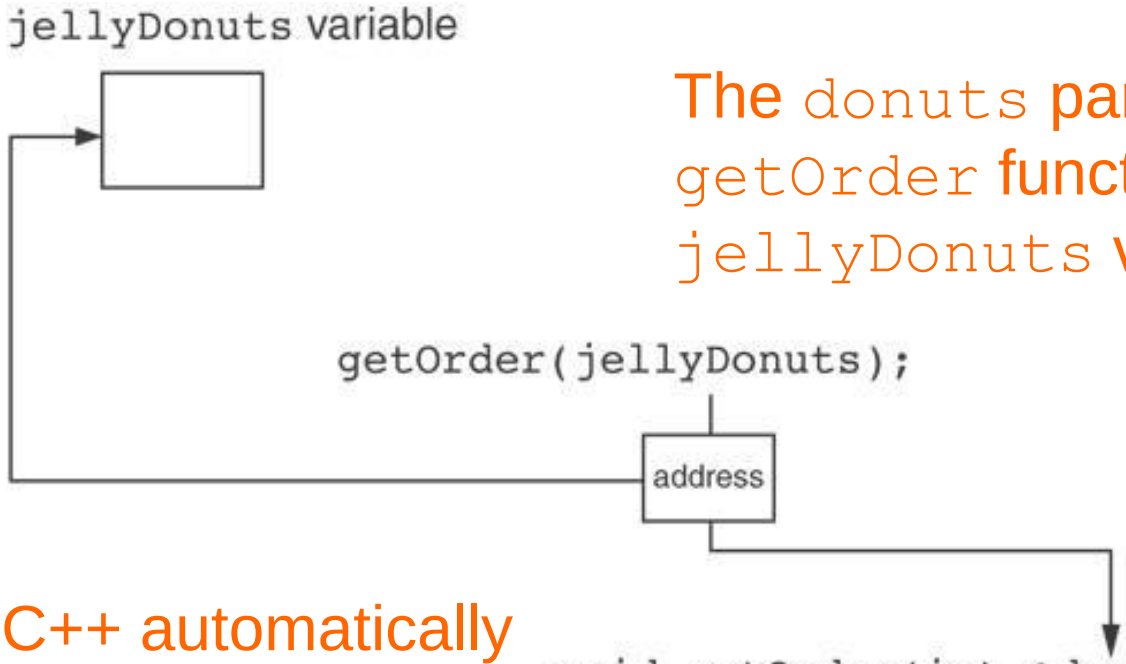
- We have also worked with something like pointers when we learned to use reference variables.
- Suppose we have this function:

```
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.

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

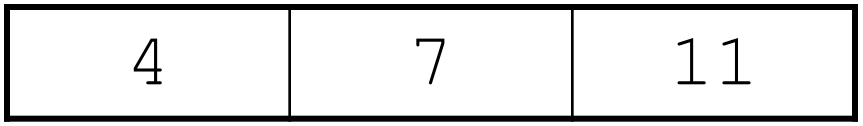
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

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};
```

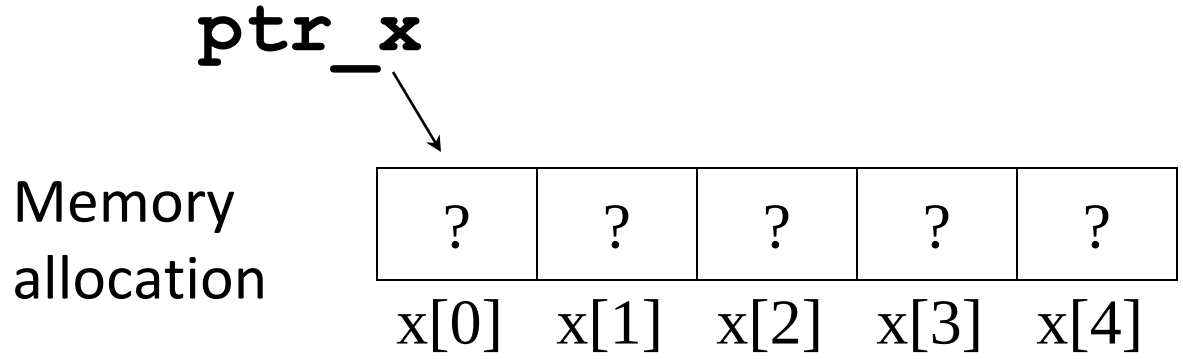


starting address of vals: 0x4a00

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

The Relationship Between Arrays and Pointers

```
int x[5], *ptr_x;
ptr_x = &x[0];
```



The memory location for **x[1]** is immediately follow the memory location of **x[0].**

The Relationship Between Arrays and Pointers (cont.)

- 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
```

Example 3.1

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
```

Exercise 4

- Refer to previous slide in Program 9-5.
- Print the third element in the array using pointer **`number`**.

4 - Pointers Arithmetic

Pointers 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`

Example 4.1

```
 7       const int SIZE = 8;
 8       int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
 9       int *numPtr;    // 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       }
```

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

Pointers in Expressions

- depends on the machine used
- depends on the variable type
- For examples,
  - Short integers (2 byte)
    - Beginning : **ptr = 45530**
    - After **ptr++** : **ptr = 45532**
  - Floating point values (4 byte)
    - Beginning : **ptr = 50200**
    - After **ptr++** : **ptr = 50204**

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

```
 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;
```

Exercise 5

- Refer to Exercise 10.14 No. 4(b) in pg. 298.
- Solve the problem.

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) ...
```

Exercise 6

For each of the following problems, give a memory snapshot that includes both variables and pointer references after the problem statements are executed. Include as much information as possible. Use question marks to indicate memory locations that have not been initialized.

1.
```
double x=15.6, y=10.2, *ptr_1=&y, *ptr_2=&x;
...
*ptr_1 = *ptr_2 + x;
```

2.
```
int w=10, x=2, *ptr_2=&x;
...
*ptr_2 -= w;
```

3.
```
int x[5]={2,4,6,8,3};
int *ptr_1=NULL, *ptr_2=NULL, *ptr_3=NULL;
...
ptr_3 = &x[0];
ptr_1 = ptr_2 = ptr_3 + 2;
```

4.
```
int w[4], *first_ptr=NULL, *last_ptr=NULL;
...
first_ptr = &w[0];
last_ptr = first_ptr + 3;
```

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
```

Exercise 7

```
#include <iostream>
using namespace std;

int main()
{
   int value=7;
   int *ptr1 = &value;
   int *ptr2 = &value;

   if (ptr1==ptr2){
    cout << ”Pointers are Equal”;
    }else{
    cout << "Pointers are Not Equal”;}
return 0;
}
```

Pointers are Equal

7 - Pointers as Function Parameters

Pointers as Function Parameters

- A pointer can be a parameter
- implements call-by-address references
- allows to modify the values by statements within a called function
- Requires:

1) asterisk * on parameter in prototype and heading

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

2) asterisk * in body to dereference the pointer

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

2) address as argument to the function

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

Example 7.1

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

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

Example 7.2

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
```

Example 7.2 (cont.)

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
```

Exercise 8

- Refer to Exercise 10.14 No. 5 in pg. 298.
- Solve the problem.

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.
- 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.

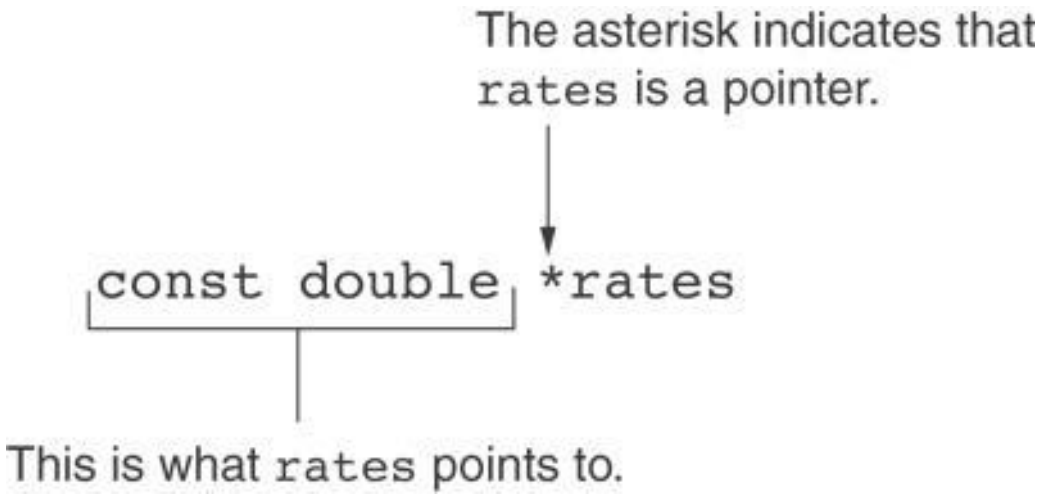
Pointers to Constants

- 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



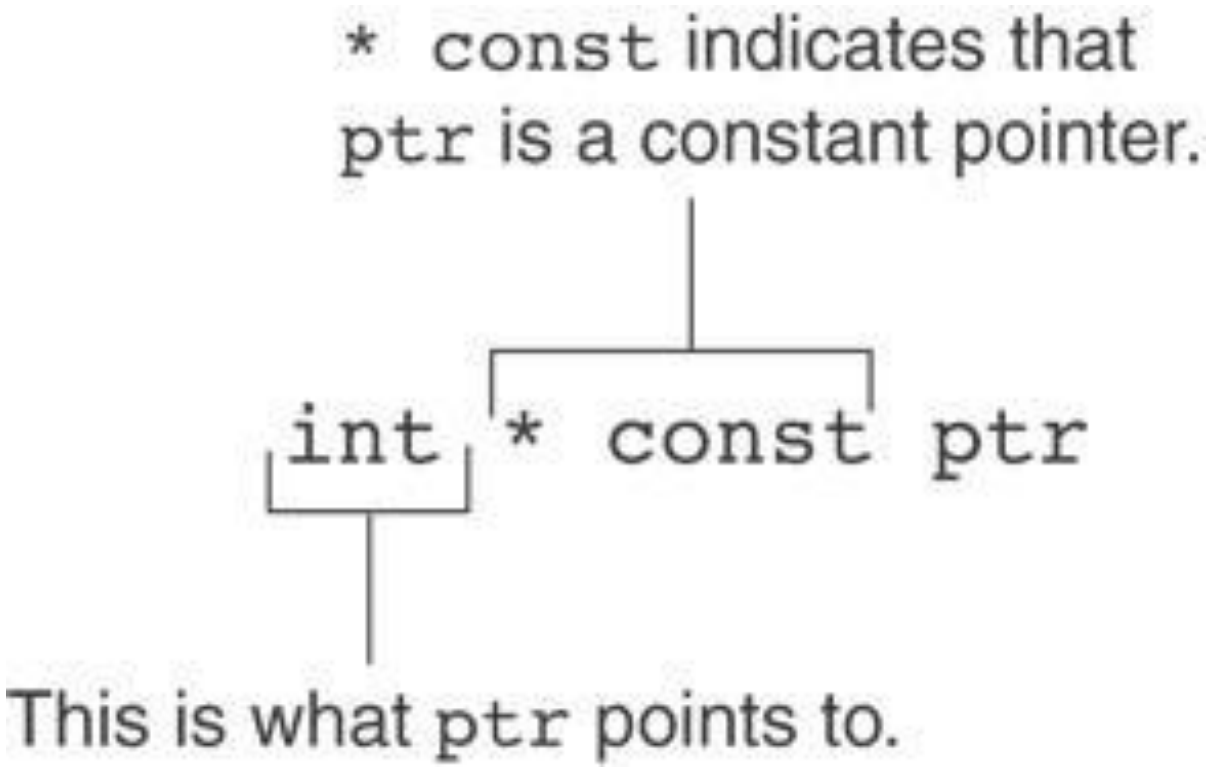
Constant Pointer

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

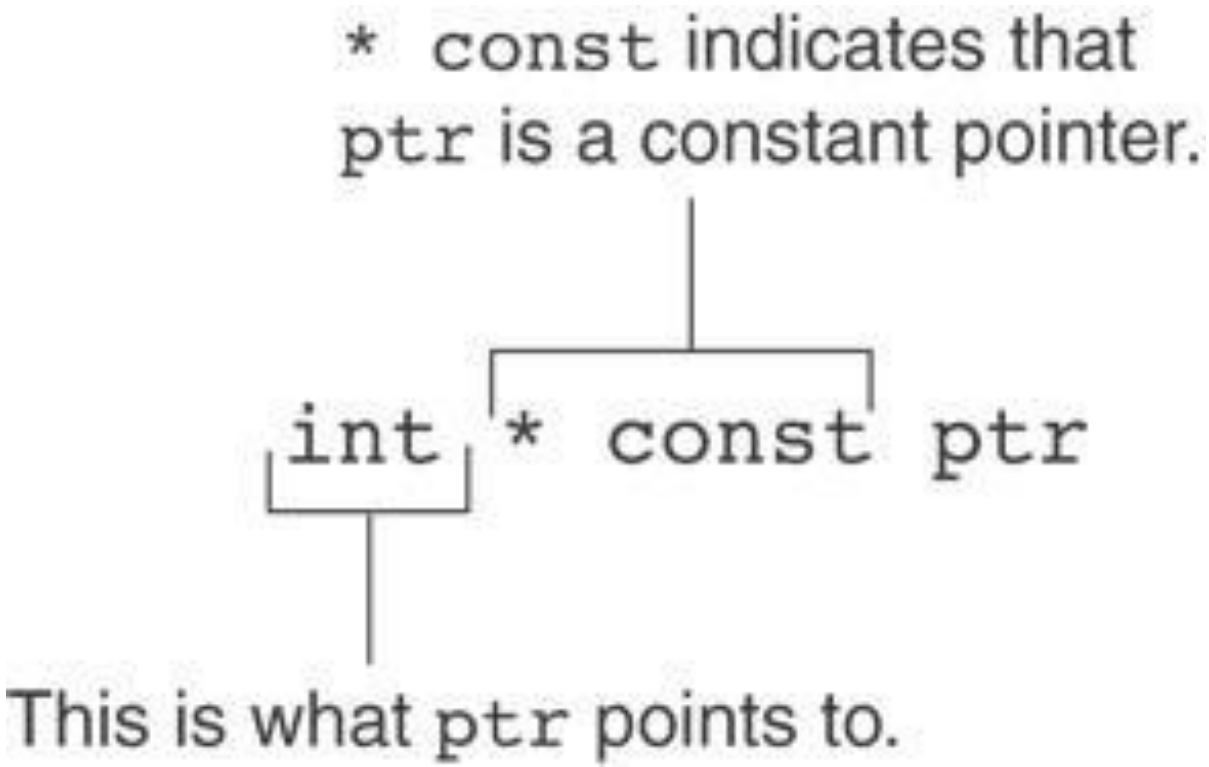
- Example

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

Declaration of a Constant Pointers



Declaration of a Constant Pointers



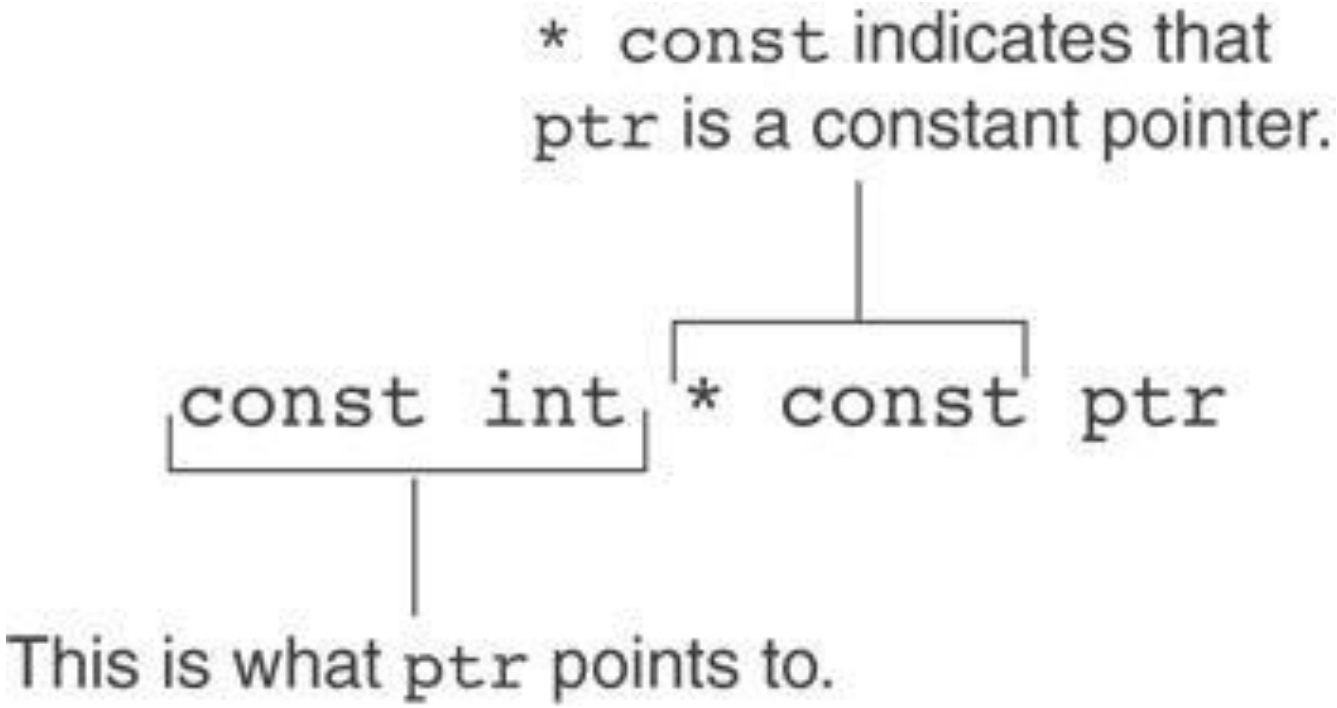
Constant Pointer to Constants

- 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 Pointer to Constants



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;
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!

Example 7.8

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
```

Example 7.8 (cont.)

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
```

Example 7.8 (cont.)

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

Exercise 9

- Given the following program with 3 errors. Rewrite the program to store the power value of the array's index and print the values.

```
int main(){
 const int SIZE = 25;
 int *arrayptr;
 arrayptr = new double[SIZE];
 for(int i = 0; i < SIZE; i++)
        *arrayptr[i] = i * i;
 for(int i = 0; i < SIZE; i++)
        cout <<*arrayptr + i<<endl;
 return 0;
}
```

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

Example 7.9

```
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  }
```


Thank You
Q
&
A