Pointers

PROF. MADYA NORANIAH MOHD. YASSIN
Adopted from: En. Jumail bin Taliba
Faculty of Computing
Universiti Teknologi Malaysia

Outline

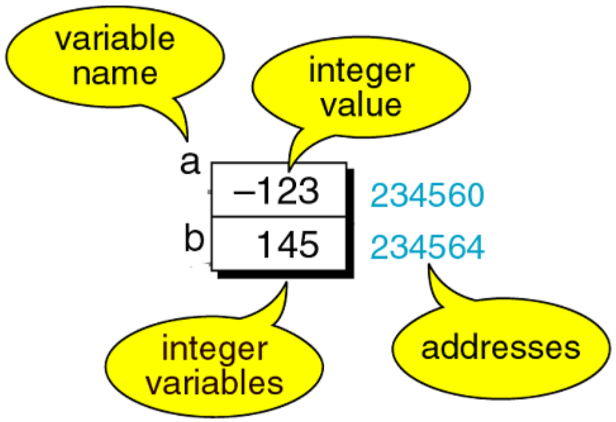
- Concept of Pointers
- Pointer Syntax
- Pointers to Pointers
- Pointers and Arrays
- Pointer Arithmetic
- Pointers and Function Parameters
- Dynamic Variables

Concept of Pointers

Variables:

- Variable is a space in memory that is used to hold data.
- Each variable has a name, content and address
- The variable name is used by the program to refer the space
- The address is the actual location in memory and used by the computer to refer the space

```
int a=-123;
int b=145;
```



Concept of Pointers *(cont)*

Pointers:

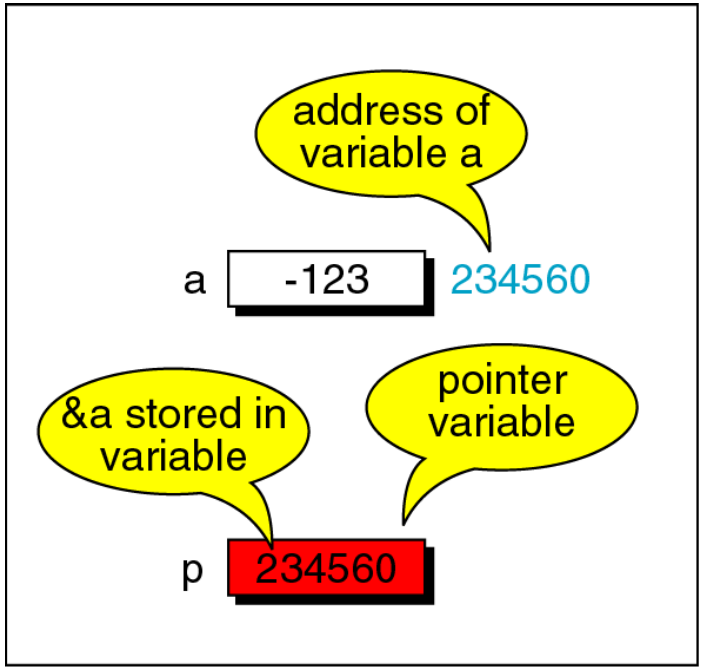
- What we have used so far are ordinary variables or also called data variables.
- A data variable contains a value (e.g. integer number, a real number or a character) .
- A pointer variable is another type of variable that contains the address of other variable. Pointer also known as address variable.

Concept of Pointers *(cont)*

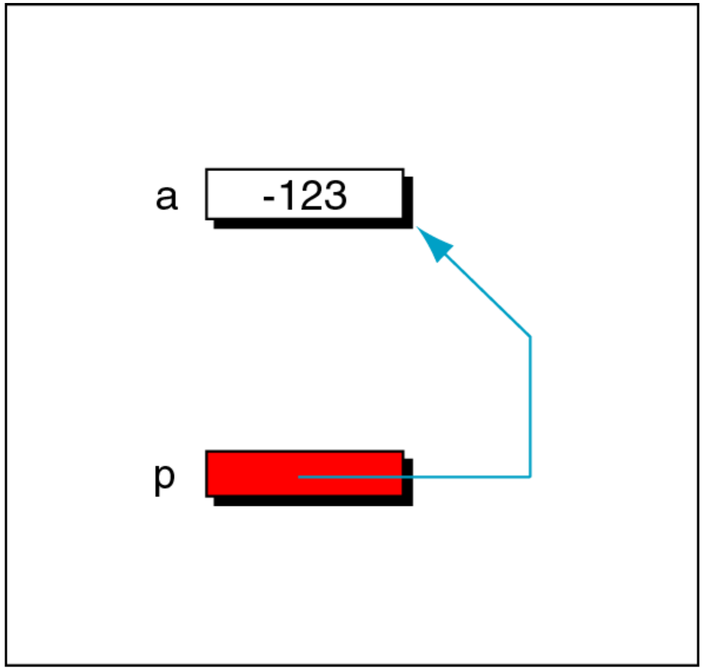
Example:

a is a data variable

p is a pointer variable that stores the address of **a**



Physical representation



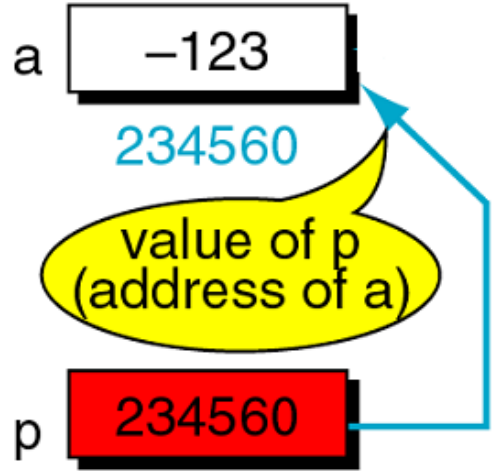
Logical representation

Pointer Syntax: Declaring and Assigning Pointers

- A pointer is declared like the data variable. The difference is, we need to put an asterisk (*) .
- To assign the address of a variable to a pointer, use & operator

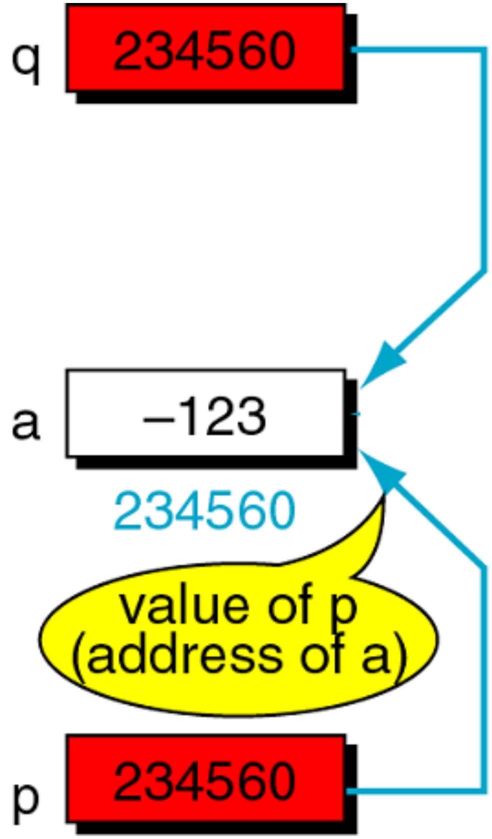
Example:

```
int a=-123; // declaring a as a data variable
int *p;     // declaring p as a pointer variable
p = &a;     // Assigning p with the address of variable a
```



Example: **p** and **q** point to the same variable.

```
int a=-123;
int *p;
int *q;

p=&a;
q=p;
```



The type of a pointer must be matched with the type of the variable that the pointer points to.

```
int n;
int *p;
double *q;

p=&n;   // this is OK
q=&n;   // this is wrong, type mismatch
q=p;    // this is also wrong
```

Pointer Syntax: Multiple Pointer Declarations

- To declare multiple pointers in a statement, use the asterisk for each pointer variable

  Example:

```
int *p1, *p2;
```

  Only `p1` is a pointer variable; `p2` is an ordinary variable

- You may also use **`typedef`** to make the declaration clear

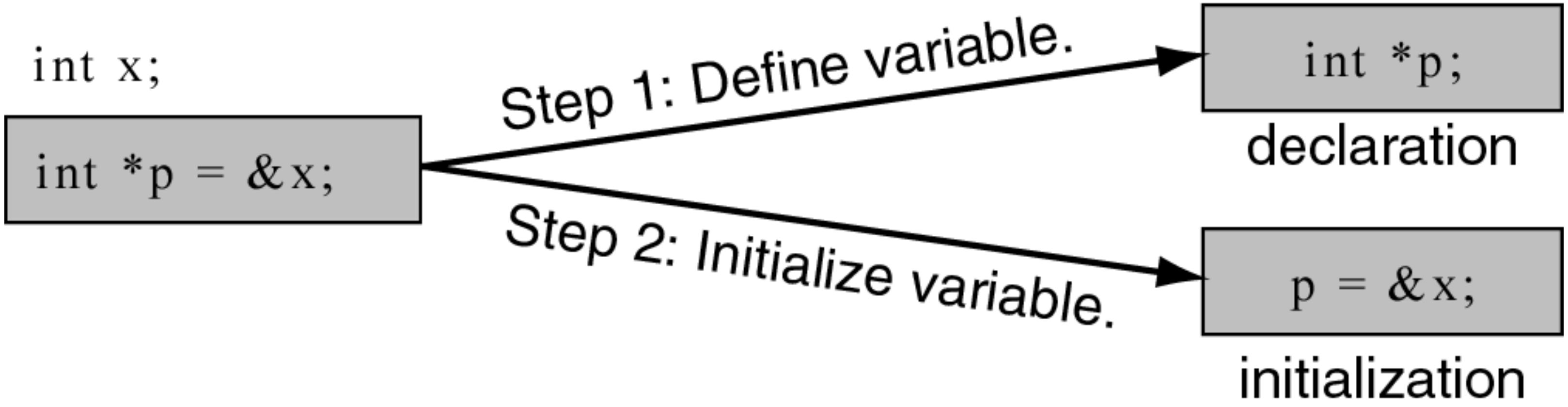
  Example:

```
typedef int *IntPtr;
IntPtr p1, p2;
```

  Now, both `p1` and `p2` are pointer variables

Pointer Syntax: Initializing pointers

Pointers can also be declared and initialized in a single statement



Pointer Syntax: The `*` and `&` operators

- There are two special operators for pointers: address operator (**&**) and indirection operator (*).

- Address operator, **&**
  - is used to get the **address** of a variable
  - Example: **`&n`**

    means : *"give me the address of variable n"*

- Indirection operator, *
  - is used to get the **content** of a variable *whose address is stored in the pointer*.
  - Example: **`*ptr`**

    means: *"give me the content of a variable whose address is in pointer ptr"*

  - ***Indirection operator must only be used with a pointer variable****. If* ***`n`*** *is a data variable, the following would be an error.* ***`*n`***

```
int main()
{ int a = 5;
  int *ptr = &a;

  cout << ptr;      // Prints 2000

  cout << &a;       // Prints 2000

  cout << &ptr;     // Prints 2004

  cout << *ptr;
```

Prints **5.**
This is how it works. **ptr** contains **2000**.
Go to the address 2000 and get its content => **5**. Means that, the value of ***ptr** is 5.

```
  *ptr = *ptr + 5;
```

This means, **a** = **a** + **5**, because ptr holds the address of **a**. The new value of a is **10**

```
  cout << *ptr;     // Prints 10

  cout << a;        // Prints 10
}
```

Memory

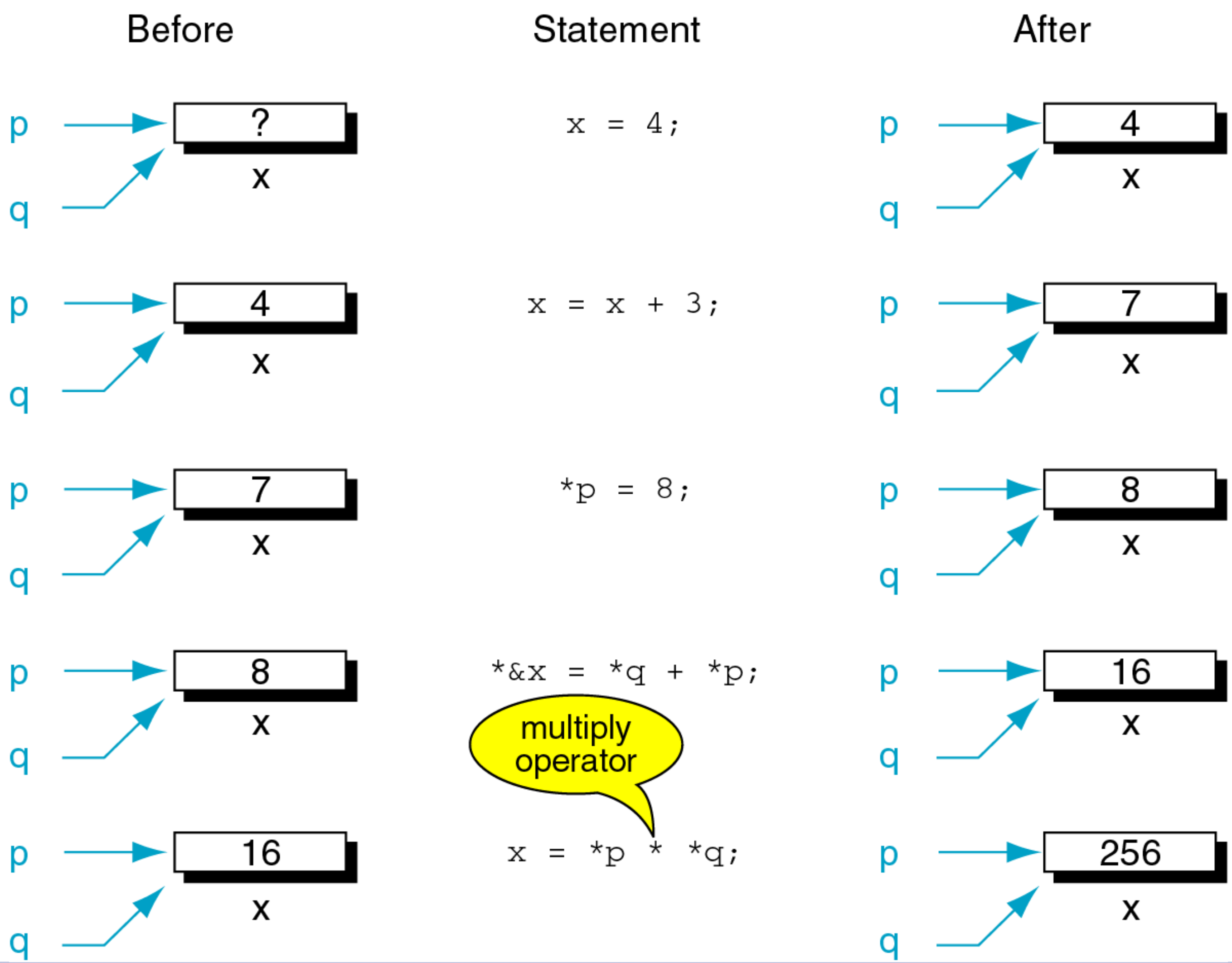
Address	***Content***	
2000	5	**a**
2004	2000	**ptr**
2008		
2009		
2010		
2011		

The addresses are specified by the Operating System. The addresses above are used only for this example.

```
cout << *a;   // This would be an error, because
              //     a is not a pointer variable
```

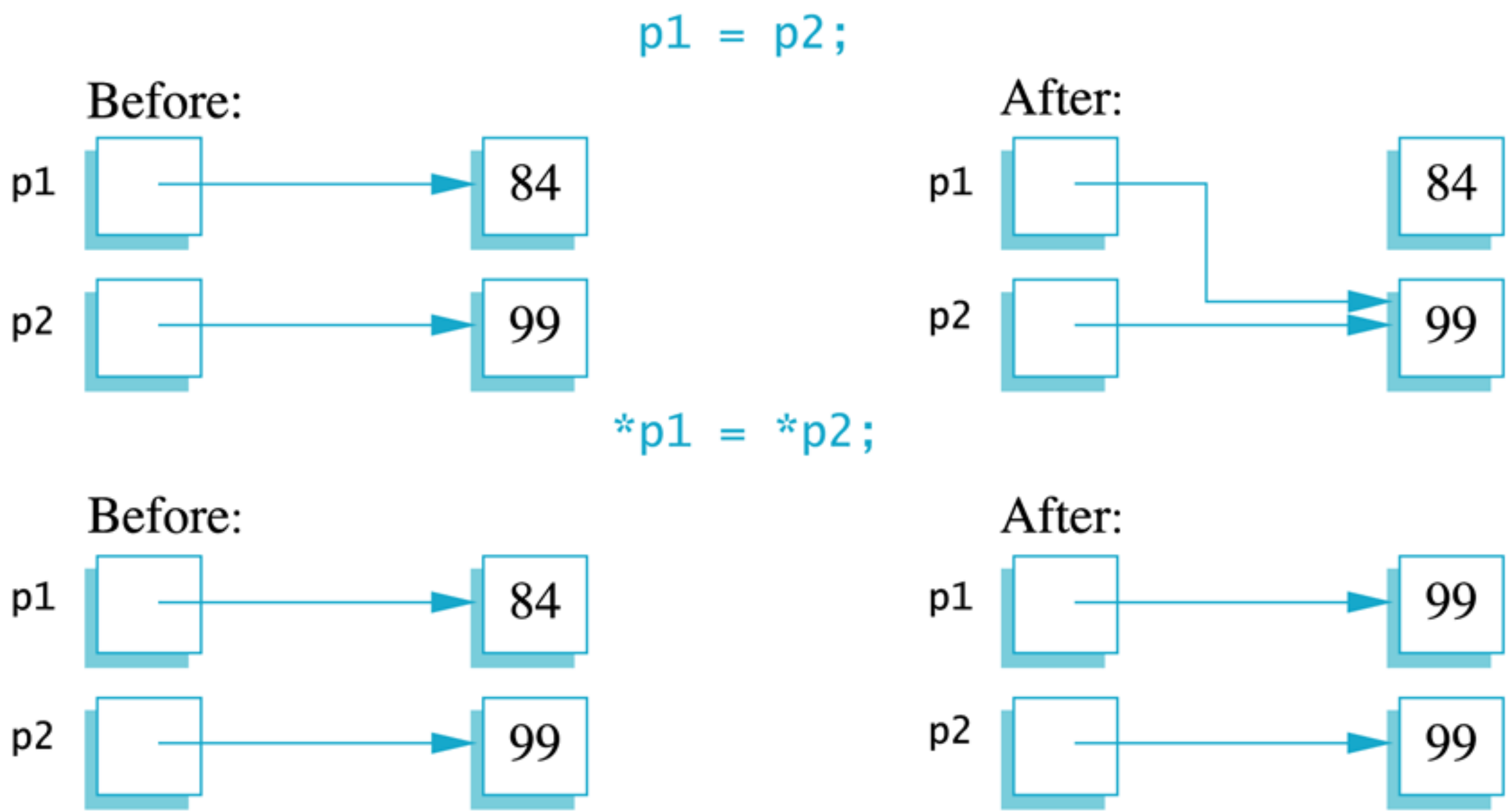
Example

```
int x;
int *p=&x;
int *q=&x;
```



Pointer Syntax: The = operator

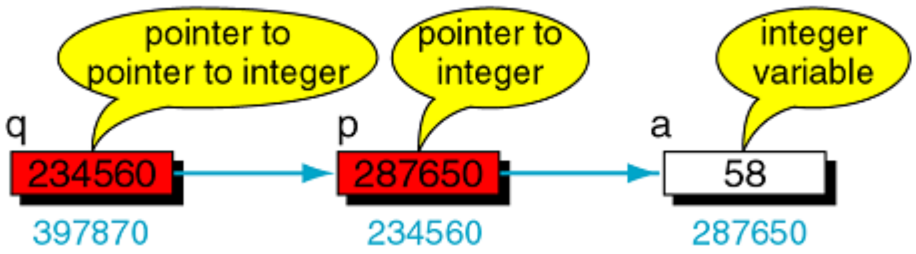
Uses of the Assignment Operator



Pointers to Pointers

Example:

```
int a;
int *p;
int **q;

a = 58;
p = &a;
q = &p;
```



You have three ways to access the content of the integer variable: **a**, ***p** and ****q**

Example: All the three couts print the same thing, i.e. 58

```
cout << a   << endl;
cout << *p  << endl;
cout << **q << endl;
```

Pointers and Arrays

- Array name is starting address of array

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

4	7	11

starting address of `vals`: `0x4a00`

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

Pointers and Arrays *(cont.)*

- Array name can be used as a pointer constant:

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

- Pointer can be used as an array name:

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

Example:

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); // prints 7
cout << *(valptr+2); // prints 11
```

Must use `( )` as shown in the expressions

Array Access

Array elements can be accessed in many ways:

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

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;`

- Accessing the content of an array element

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

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

- Accessing the address of an array element

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

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
```

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); // prints 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
```

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
```

Pointers as Function Parameters

Example: Another way to pass by reference is using pointers

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

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

In function declaration and definition, declare formal parameters as pointers

Must use * operator, when referring to the parameters

At function call, must use & operator when sending arguments

Example

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
```

Pointers to Constants

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

- Example: Suppose you 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.

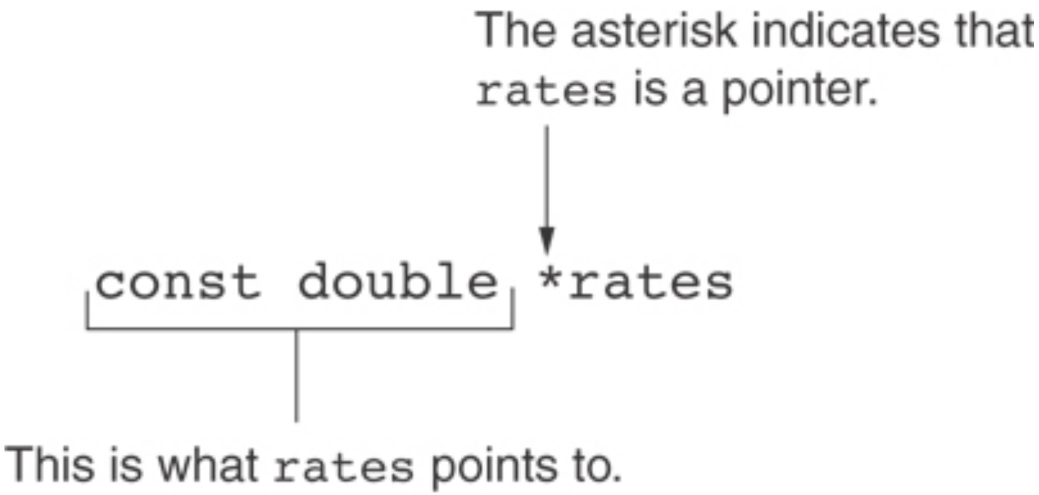
Pointers to Constants *(cont.)*

- To pass the `payRates` to a function

```
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 a Constant



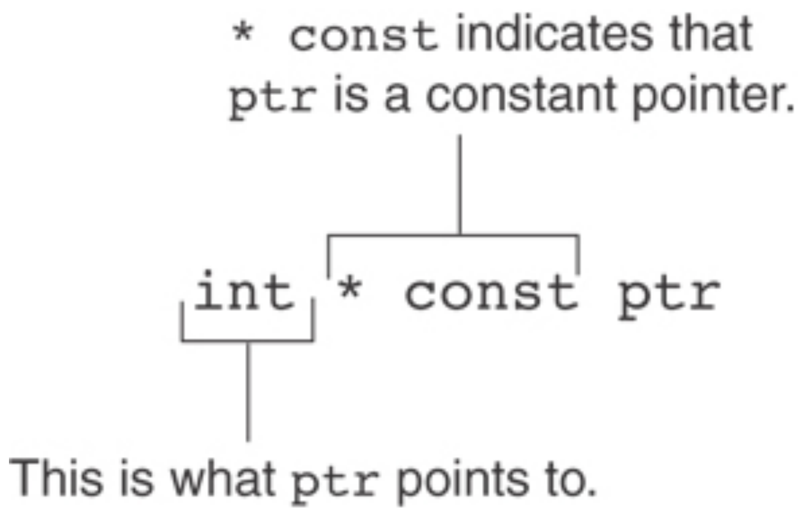
Constant Pointers

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

- Example

```
int value1 = 22;
int value2 = 11;
int * const ptr = &value1; // ptr is a constant pointer
                           //   and pointing to value1

 ptr = &value2; // Error! The pointing of a constant pointer
                //    cannot be changed
```

Constant Pointers *(cont.)*



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

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

```
const int * const ptr
```

This is what `ptr` points to.

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;
}
```

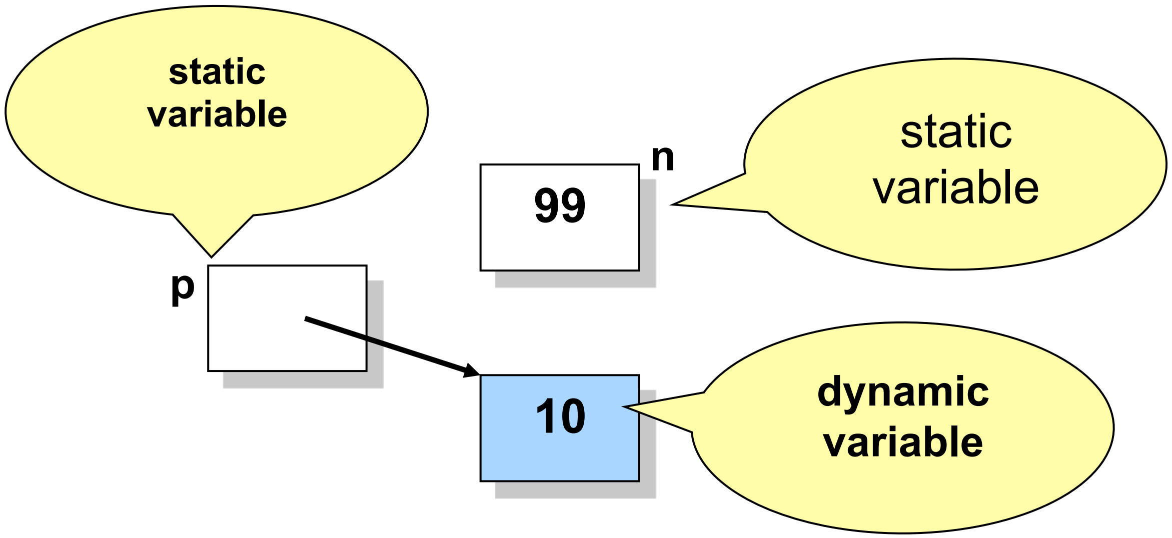
Dynamic Variables

- Dynamic variables are variables that are created and destroyed while the program is running.

- Static variables (sometimes called *automatic variables*) are variables that are automatically created and destroyed by the computer.

Example:

```
int n=99;
int *p;

p = new int;
*p = 10;
```



The `new` Operator

- Using pointers, variables can be manipulated even if there is no identifier for them

  - To create a pointer to a new "nameless" variable of type int:

    ```
    p1 = new int;
    ```

  - The new variable is then referred to as **`*p1`**
  - It can be used anyplace as integer variable can

```
cin >> *p1;
*p1 = *p1 + 7;
```

Example

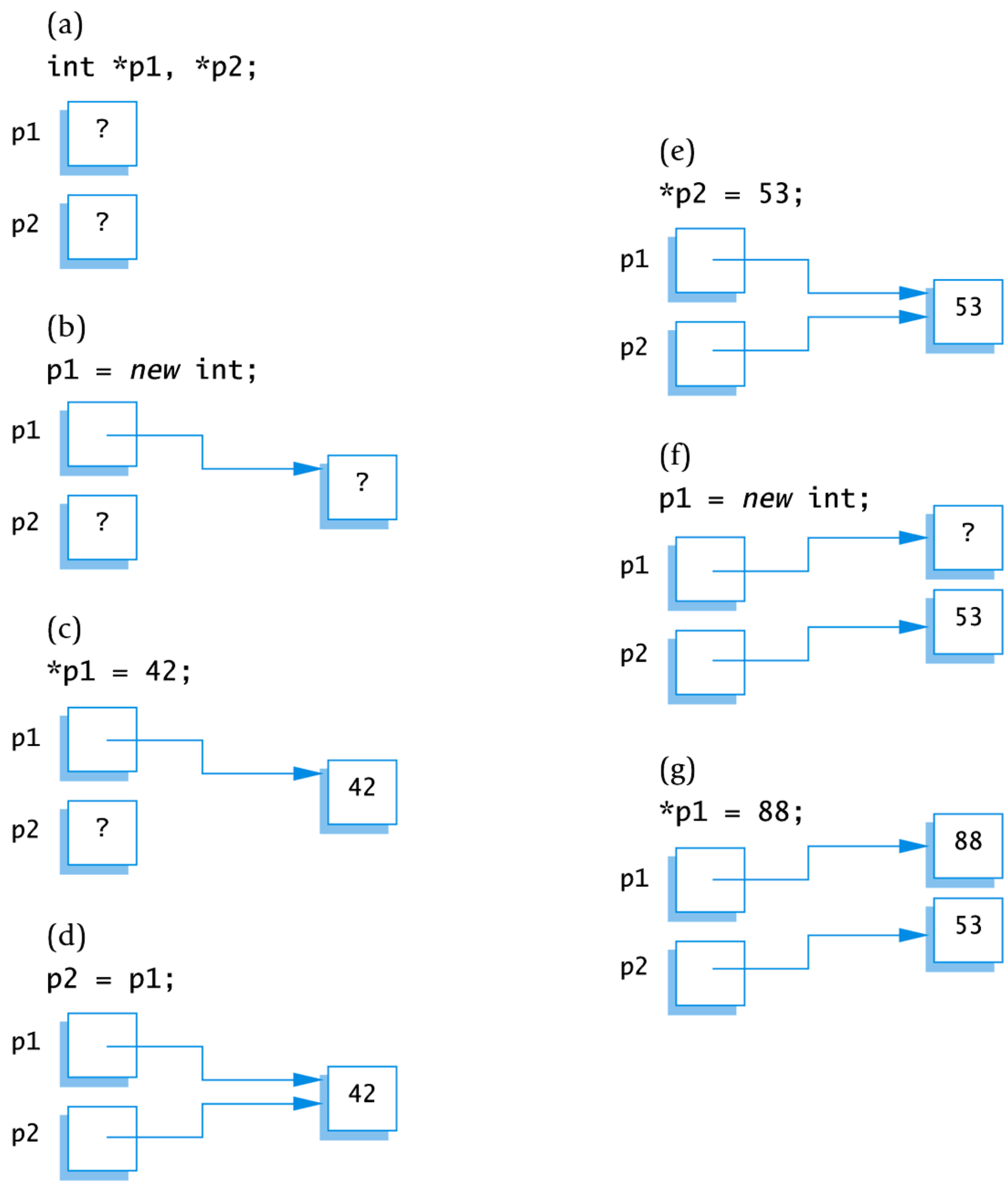
Basic Pointer Manipulations

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

DISPLAY 9.3 Explanation of Display 9.2



Basic Memory Management

- An area of memory called the freestore is reserved for dynamic variables
  - New dynamic variables use memory in the freestore
  - If all of the freestore is used, calls to new will fail

- Unneeded memory can be recycled
  - When variables are no longer needed, they can be deleted and the memory they used is returned to the freestore

The delete Operator

- When dynamic variables are no longer needed, delete them to return memory to the freestore

  Example:

  ```
  delete p;
  ```

  The value of `p` is now undefined and the memory used by the variable that p pointed to is back in the freestore