Lesson: 8 : Pointers and Dynamic

Objectives:

To:

- Learn about Pointers in C++
- Become aware of the basic properties of pointers
- Explore the application of pointers
- Learn about Dynamic Arrays
- Discover how to create and manipulate a dynamic array
- Learn how to use a dynamic array

Structure of the Lesson:

8.1. Introduction to pointers and pointer

Pointers are variables that hold addresses in C and C++. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages. They are also useful for passing parameters into functions in a manner that allows a function to modify and return values to the calling routine. When used incorrectly, they also are a frequent source of both program bugs.

We can define a variable in C++ to store a *memory address*. A **pointer** in **C++** is said to "point to" the memory address that is stored in it. Also, when defining a ***C++*** pointer variable, we must specify the type of variable to which it is pointing. For example, to define a pointer, which will store a memory address at which exists an int, we can do the following:

```
//Sample program for c++ pointer
 main()
{
    int* p;
    // p contains no particular value in this C++
code.
}
```

The asterisk in the above specifies that we have a pointer variable. Let's say we want to define an int variable and then we want to define a pointer variable, which will store the memory address of this int:

```
//c++ pointer using an int variable

main()
{
   int number(7);
   int* p_number;
   // p_number contains no particular value.
   p_number = &number;
   //Now, p_number in this c++ program
contains the

   //memory address of the variable myval
}
```

With **&number**, & is referred to as "the address-of operator". The expression &number is of the c++ type int*. We then store this int* number in our int* variable, which is p_number. Now, we will actually use this pointer:

```
//Sample program for c++ pointer

signed main()
{
   int number = 7;
   int* p_number = &number;
   *p_number = 6;
}
```

With *p_number = 6, the asterisk is referred to as "the dereference operator". It turns the expression from an int* into an int. The statement has the effect of setting the value of number to 6.

Pointers to Pointers: An int* c++ pointer points to an int, so an int** points to an int*. The variable p_p_n below stores a memory address. At that memory address exists a

variable of type int*. This int* variable also stores a memory address, at which exists an int.

//sample for c++ pointer to pointers .

```
int main()
{
   int n(7);
   int* p_n = &n;
   int** p_p_n(&p_n);
   int*** p_p_p_n = &p_p_n;
}
```

8.2. Dynamic Variables

In C++, space in memory for variables may be either statically or dynamically allocated. Statically allocated objects are those that are not created with the memory allocator **new**, that is, they are just the ordinary objects we use.

```
int x;
float money;
Employee emp;
```

These objects are of fixed, known size and the compiler arranges the required space as it turns source code into an object program. Statically allocated objects that are of local scope are put into a memory space known as the stack. Statically allocated objects of global scope live in the global address space. The key point is that for these objects their size is fixed at compile time.

Sometimes we don't know the size of an object until the program execution. Examples of this are a buffer to hold a

block of text of variable size, or an array with an undetermined number of elements, You could try to size the buffer or array to be large enough to hold the worst case, that is, to be big enough to hold anything we should encounter. But, there are two problems with this strategy. First, it consumes memory unnecessarily. Second, we can never be sure that the object is large enough, no matter how much memory is statically set aside for our object. This is a serious problem. The solution to this problem is dynamic memory allocation.

Allocating Single Objects

During program execution dynamically allocated memory comes form a pool of memory known as the heap or free store. It is allocated using the C++ operator "new" and freed using the operator "delete". To see how this works let's dynamically allocate some objects of intrinsic data types.

```
int *IDpt = new int;
float *theMoney = new float;
char *letter = new char;
```

The "new" operator returns the address to the start of the allocated block of memory. This address must be stored in a pointer. New allocates a block of space of the appropriate size in the heap for the object. For instance, "new int" reserves four bytes (on most operating systems), while "new char" reserves a single byte. Also, notice that the reserved block of memory is anonymous; it has no identifier (name). Dynamically allocated memory is accessed indirectly via a pointer. It is possible for new to fail. This will be the case if no memory is available. In this case new throw a "bad_alloc" exception.

Objects dynamically allocated using the above syntax are uninitialized. They contain whatever random bits happen to

be at their memory location. Before use, a value must be assigned.

```
int *IDpt = new int;
*IDpt = 5;
```

Alternatively, C++ provides a syntax, which initializes the allocated object via the "new" operator.

```
int *IDpt = new int(5);     //Allocates an int object and
initializes it to value 5.
char *letter = new char('J');
```

Dynamically allocated objects introduce a new twist; they must be explicitly deleted when no longer needed by a program. This is done using the "delete" operator. Delete releases the memory used by the object. That memory is then available for reuse.

```
delete IDpt;
delete theMoney;
delete letter;
```

Memory for statically allocated variables is reclaimed when they go out of scope. For instance, when execution enters a function, the function's statically allocated local variables are created on the stack. When the function exits, these variables are popped off the stack and the memory they occupied is available for reuse. The memory of dynamically allocated objects is not automatically released. It must be explicitly released using the "delete" operator. Suppose we have a function that dynamically allocated some variables

and that we neglect to call delete before exiting that function.

```
void fun()
{
   int *pt;
   int ordinaryVariable;

   pt = new int(1024);
   ....
   ....
   // dynamic variable not deleted.
}
int main()
{
   while (some condition exists) // Pseudo-code
   {       fun();
   }
   return 0;
}
```

"ordinaryVariable" is created on the stack when entering the function and popped off when exiting the function, so there is no problem. Likewise for "pt". "pt" is a local variable in the function. It holds the address of the dynamically allocated object. When the function exists, pointer is popped off the stack like any local variable, but the dynamically allocated object still exists. We just no longer have a pointer to it. Since we no longer have the pointer, the memory of the dynamically allocated object can no longer be released using delete. This is known as a memory leak. As the program continued to operate, more and more memory will be lost from the heap (free store). If the program runs long enough, eventually no memory will be available, and the program will no longer operate. Additionally, even if we don't run out of memory, the reduced pool of available memory affects system performance. The moral of all this: Be sure to delete.

Every new should be paired with a delete in your code to avoid memory leaks.

8.3. Dynamic Arrays

Dynamically Allocating Arrays
Arrays of built-in and user-defined data types may be dynamically allocated. User-defined data types include classes.

```
int *pt = new int[1024];
 //allocates an array of 1024 ints
double *dbs = new double[1000];    /* Allocates an
array of 1000 doubles to hold the amounts of the bills
*/
```

Observe the difference between:

```
int *pt = new int[1024];
 //allocates an array of 1024 ints
int *pt = new int(1024);
//allocates a single int with value 1024
```

A dynamically allocated array is best initialized using a loop, as follows.

```
int *list = new int[1024];
for (i = 0; i < 1024; i++)
{
   *list = 52; //Assigns 52 to each element;
   list++;
}
```

or equivalently

```
int *list = new int[1024];
for (i = 0; i < 1024; i++)
{
   list[i] = 52; //Assigns 52 to each element;
}
```

The syntax of the delete operator to delete dynamically allocated arrays is slightly different from what we saw for single objects.

```
delete[] pt;
delete[] dbs;
```

The square brackets after the delete tell the compiler to delete a dynamic array rather than a single object.

Dangling Pointers
Take a look at this snippet of code.

```
int *dptr, *dup;

dptr = new int(10);
dup = dptr;
cout << "The value of dptr is " << *dptr << endl;

delete dup;
*dptr = 5;
cout << "The value of dptr is " << *dptr << endl;
```

In the above example dptr is a dangling pointer. We have released the memory of the object whose address dptr holds and then continued to use it. This problem is that although the program may run, this section of memory may be used by another dynamic object allocated after the delete. The values in that object will be corrupted by the continued use of dptr. This is a very subtle programming bug and is very difficult to isolate. To avoid this bug, always set a pointer to 0, after the delete is called. Subsequent attempts to use the pointer will result in a run-time exception. This will immediately allow the bug to be identified and fixed. The corrected code is given below.

```
int *dptr;
dptr = new int(10);
cout << "The value of dptr is " << * dptr << endl;

delete dptr;
dptr = 0;
*dptr = 5;
//This statement will cause an run-time exception, now.

cout << "The value of dptr is " << * dptr << endl;
```

An example program on dynamic arrays is given below.

```
//Sorts a list of numbers entered at the keyboard.
#include <iostream.h>
#include <stdlib.h>
#include <stddef.h>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);
//Precondition: size is the size of the array a.
//Postcondition: a[0] through a[size-1] have been
//filled with values read from the keyboard.

void sort(int a[], int size);
/*Precondition: size is the size of the array a.
The array elements a[0] through a[size - 1] have values.
Postcondition: The values of a[0] through a[size-1] have
been rearranged so that a[0] <= a[1] <= ... <=
a[size-1].*/

//The following prototypes are to use in the definition of
//sort:

void swap_values(int& v1, int& v2);
//Interchanges the values of v1 and v2.

int index_of_smallest(const int a[], int start_index, int
number_used);
//Precondition: 0 <= start_index < number_used.
//Referenced array elements have values.
//Returns the index i such that a[i] is the smallest of the
values
//a[start_index], a[star_index + 1], ..., a[number_used -
1].

int main( )
{
   cout << "This program sorts numbers from lowest to
```

```
highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];
    if (a == NULL)
    {
        cout << "Error: Insufficient memory.\n";
        exit(1);
    }

    fill_array(a, array_size);
    sort(a, array_size);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [] a;

    return 0;
}




//Uses the library iostream.h:
void fill_array(int a[], int size){
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)
{
```

```
    int index_of_next_smallest;
         for (int index = 0; index < size - 1; index++)
    {//Place the correct value in a[index]:
        index_of_next_smallest =
                        index_of_smallest(a, index, size);
        swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of
the original array
        //elements. The rest of the elements are in the
remaining positions.
    }
}

void swap_values(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int index_of_smallest(const int a[], int start_index, int
number_used)
{
    int min = a[start_index],
        index_of_min = start_index;
    for (int index = start_index + 1; index < number_used;
index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through
a[index]
        }

    return index_of_min;
}
```

How to avoid Memory leaks and Dangling pointers:

A memory leak happens when you forget to free a block of memory allocated with the new operator or when you make it impossible to do so. As a consequence your application may eventually run out of memory and may even cause the system to crash. The following are a few practices to avoid memory leaks and dangling pointers

- **Delete a dynamic array before reallocating it**

```
char *string;
string = new char[20];
string = new char[30];
delete[] string;
```

In the above example, we have two consecutive memory allocations to the string pointer. This leads to memory leak. We should have a delete [] statement after the first allocation and then try to reallocate using a different size parameter. If we choose not to, the second allocation will assign a new address to the string pointer while the previous one will be lost. This makes it impossible to free the first dynamic variable further on in the code, resulting in a memory leakage. The corrected code is:

```
char *string;
string = new char[20];
delete[] string;
string = new char[30];
delete[] string;
```

- **Be sure you have a pointer to each dynamic variable**

  What do you think will happen at the end of this code fragment?

```
char *first_string = new char[20];
char *second_string = new char[20];
strcpy(first_string, "leak");
second_string = first_string;
strcpy(second_string, first_string);
delete [] second_string;
```

There is a memory leak in the above example, because you have lost the address of the dynamic variable associated with *second_string* (as a side-effect of the pointer assignment) so you cannot delete it from the heap anymore. Thus the last line of code only frees the dynamic variable associated with *first_string*, which is not what we wanted.

The main idea is to try and not lose the addresses of dynamic variables, so that you will be able to free them, after their purpose is over.

- **look for local pointers**

  Consider the following function :

```
Void leak() {
   int k;
   char *cp = new char('E');
   delete cp;
}
```

Obviously both the k and cp variables are local so they are allocated on the stack segment. Then when it comes the time to exit the function they will be freed from memory as the stack is restored.

The last statement, **delete cp;** is essential as it frees up the memory consumed in the function. If this statement is absent, the memory location pointed to by cp will no longer be accessible, once the control is returned out of the function. C++ does not take any responsibility to free such memory locations. The programmer has to take care of such things.

- **Careful with functions returning dynamic variables**

  Let us take a look at the following program.

```
#include <iostream>

char* tostring(int n) {
   char *S = new char[100];
   char aux;
   int i, j;

   for (i = 0; n; n /= 10, ++i)
      S[i] = n % 10 + '0';
   for (j = 0; j < i / 2; ++j) {
      aux = S[j];
      S[j] = S[i - j - 1];
      S[i - j - 1] = aux;
   }
   S[i] = '\0';

   return S;
}

void main() {
   cout << tostring(23) << tostring(146) << endl;
```

```
    char *temp;
    temp = tostring(23); cout << temp; delete []
temp;
    temp = tostring(146); cout << temp; delete []
temp;
}
```

Obviously the function char* tostring(int n) converts the integer n to a string, but that is not of our interest right now. You may have noticed that the string stored in S is not freed from the heap before exiting the function. We have just been warned about local pointers though. The reason for this is that the string should also be available within the calling function main() as we need to print it out to screen. To solve this "contradiction" we should first assign the return value to a temporary pointer variable inside main(), print it out and be sure to delete [] it right away, as shown above.

You may ask yourself why use a supplementary pointer here, why not stick to the previous variant which is also more compact ? The answer is simple - we may not be able to delete [] the dynamic variable returned by the tostring() function call as its address would eventually be lost if we do not store it somewhere. For example the calls tostring(23), tostring(146) within the cout statement return two dynamic variables whose adresses are only used at printing, they are then lost. This leads to memory leakage.

8.4. Classes and Dynamic Arrays

A dynamic array can have a base type, which is a class. A class can have a member variable, which is a dynamic array. The techniques of dynamic arrays and classes can be combined in many ways. The *string* class is an example

of such combination. The C++ string class contains a dynamic character array, that can be created to a desired size and can be initialized with a chosen value. Each object of the *string* class then represents a string. Other member functions to do basic operations on strings can be applied on these objects. The following C++ code defines and exercises a *string* class.

```
#ifndef STRVAR_H
#define STRVAR_H
#include <iostream.h>

class StringVar
{
public:
    StringVar(int size);
    //Initializes the object so it can accept string values up
to size
    //in length. Sets the value of the object equal to the
empty string.

    StringVar( );
    //Initializes the object so it can accept string values of
length 100
    //or less. Sets the value of the object equal to the
empty string.

    StringVar(const char a[]);
    //Precondition: The array a contains characters
terminated with '\0'.
    //Initializes the object so its value is the string stored in
a and
    //so that it can later be set to string values up to
strlen(a) in length

    StringVar(const StringVar& string_object);
    //Copy constructor.

    ~StringVar( );
```

```
    //Returns all the dynamic memory used by the object to
the heap.

    int length( ) const;
    //Returns the length of the current string value.

    void input_line(istream& ins);
    //Precondition: If ins is a file input stream, then ins has
    //already been connected to a file.
    //Action: The next text in the input stream ins, up to
'\n', is copied
    //to the calling object. If there is not sufficient room,
then only as
    //much as will fit is copied.

    friend ostream& operator <<(ostream& outs, const
StringVar& the_string);
    //Overloads the << operator
    //so it can be used to output values of type StringVar
    //Precondition: If outs is a file output stream, then outs
    //has already been connected to a file.
private:
    char *value; //pointer to the dynamic array that holds
the string value.
    int max_length; //declared max length of any string
value.
};

#endif //STRVAR_H
```

```
#include <iostream.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include "strvar.h"
```

```
//Uses stddef and stdlib.h:
StringVar::StringVar(int size)
{
   max_length = size;
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }

   value[0] = '\0';
}

//Uses stddef and stdlib.h:
StringVar::StringVar( )
{
   max_length = 100;
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }

   value[0] = '\0';
}

//Uses string.h, stddef, and stdlib.h:
StringVar::StringVar(const char a[])
{
   max_length = strlen(a);
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }
```

```
    strcpy(value, a);
}


//Uses string.h, stddef.h, and stdlib.h:
StringVar::StringVar(const StringVar& string_object)
{
    max_length = string_object.length( );
    value = new char[max_length + 1];//+1 is for '\0'.
    if (value == NULL)
    {
        cout << "Error: Insufficient memory.\n";
        exit(1);
    }

    strcpy(value, string_object.value);
}


StringVar::~StringVar( )
{
    delete [] value;
}

//Uses string.h:
int StringVar::length( ) const
{
    return strlen(value);
}

//Uses iostream.h:
void StringVar::input_line(istream& ins)
{
    ins.getline(value, max_length + 1);
}

//Uses iostream.h:
ostream& operator <<(ostream& outs,
                     const StringVar& the_string)
```

```
{
   outs << the_string.value;
   return outs;
}
```

```
#include <iostream.h>
#include "strvar.h"


void conversation(int max_name_size);
//Carries on a conversation with the user.


int main( )
{
   conversation(30);
   cout << "End of demonstration.\n";
   return 0;
}

// This is only a demonstration function:
void conversation(int max_name_size)
{
   StringVar                    your_name(max_name_size),
our_name("Borg");

   cout << "What is your name?\n";
   your_name.input_line(cin);
   cout << "We are " << our_name << endl;
   cout << "We will meet again " << your_name << endl;
}
```

8.5. Copy Constructors and Destructors

A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy. By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created. This is called a member wise or shallow copy. Although it may seem to be the desired behavior, in many cases a shallow copy is not satisfactory. For the *string* class, the default behavior of copy constructor is not is not sufficient for *string* class, as the data of the object is kept in a dynamic array. A member-by- member copy can only copy the address of the dynamic character array of the string object in to that of destination object. But the actual requirement is to copy the character sequence.

Hence the *string* class in the above listing redefines the default behavior of the copy constructor. In general notation the copy constructor can be written as A(A&) where A is the class name. The copy constructor of the string class is given below.

```
StringVar::StringVar(const StringVar& string_object)
{
   max_length = string_object.length( );
   value = new char[max_length + 1];   //+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }
   strcpy(value, string_object.value);
}
```

The copy constructor should be defined such that the object being initialized becomes a complete, independent copy of its argument. So in the above listing, a new dynamic character array is created, and character sequence is copied into it from the source array. A copy constructor is automatically called whenever C++ needs to make a copy of an object. Particularly in the following circumstances the copy constructor is called automatically:

1) When a class object is being defined and is initialized by another object of the same type.
2) When a function returns a value of the class type.
3) Whenever an argument of the class type is supplied for a call-by-value parameter.

If a class definition involves pointers and dynamically allocated memory using the new operator, then you need to include a copy constructor. Other classes do not need a copy constructor.

Destructors:

A destructor is a member function of a class that is called automatically when an object of the class goes out of scope. Destructors are used to eliminate any dynamic variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the heap. The name of a destructor must consist of the tilde symbol ~ followed by the name of the class. A destructor takes no arguments and returns no value. The code given above for the string class also includes destructor for it. It contains the following lines.

```
StringVar::~StringVar( )
{
   delete [] value;
}
```

From the above function it is clear that the *stringvar* destructor deletes the dynamic character array value.

8.6. Summary:

Pointers are variables that hold addresses in C and C++. We can define a variable in C++ to store a *memory address*. A **pointer** in **C++** is said to "point to" the memory address that is stored in it. Also, when defining a ***C++*** pointer variable, we must specify the type of variable to which it is pointing. For example, to define a pointer, which will store a memory address at which exists an int, we can do the following: The other feature of c++ pointers is that they can be "re-seated", which means that you can change their value, you can change what they're pointing to, as in the following: // c++ pointer program for modifying values/re-seating.

An int* c++ pointer points to an int, so an int** points to an int. At that memory address exists a variable of type int*. This int* variable also stores a memory address, at which exists an int.

Arrays of built-in and user-defined data types may be dynamically allocated. User-defined data types include classes. We'll see dynamically allocated arrays of classes in a latter lesson, so for now let's look at built-in data types.

A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy. By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created.

A destructor is a member function of a class that is called automatically when an object of the class goes out of scope. Destructors are used to eliminate any dynamic

variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the heap.

8.7. Technical Terms:

Pointer: The memory address of a variable or object.

Pointer Variable: A variable that contains a memory address.

Constructor: A constructor is a method that has the same name as its class.

Destructor: A destructor is a method that has as its name the class name prefixed by a tilde, ~.

Copy Constructor: A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy.

Exception: An exception is an error or anomaly that occurs as a program is executing. It can be due to a lack of system resources, such as a lack of memory or unavailability of a file, or raised by program design.

8.8. Model Questions:

1. What a Pointer? Explain different types of pointers?
2. What is a dynamic array? Explain how to create and use it?

3. Explain application of dynamic arrays with string class?
4. What is a copy constructor?
5. How do you delete dynamically allocated variables from memory in an object?

8.9. References:

Problem Solving With C++ by Walter Savitch, **Pearson Education Asia**

C++ by Balagurusamy, **BPB Publications.**

Let Us C++ by Y. Kanitkar.

AUTHOR:

Y. VENKATESWARA RAO, M.C.A.,
Lecturer,
Dept.Of Computer Science,
JKC College,
GUNTUR