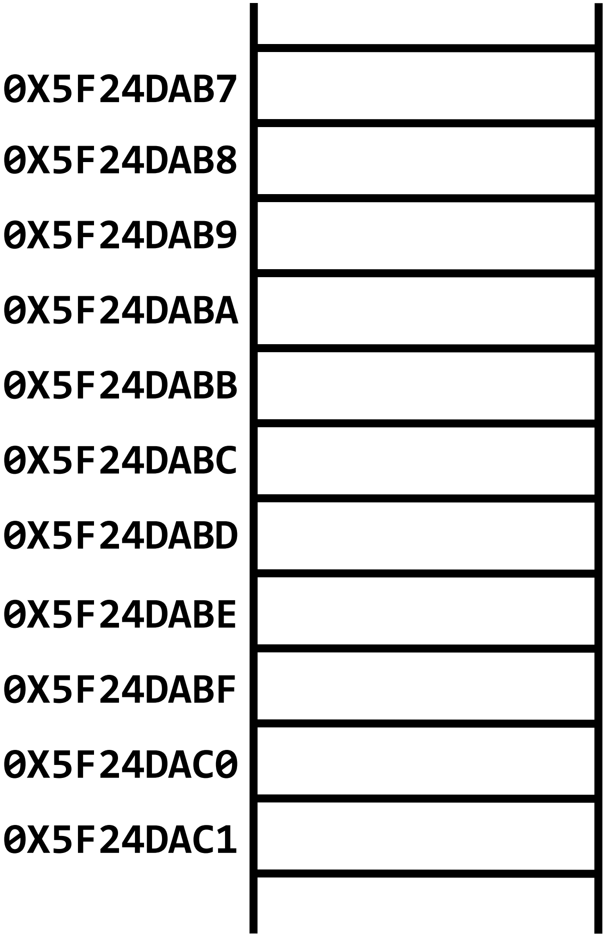
C++ Pointers
Memory Management

In this topic

- Memory address
- L-values and R-values
- The address operator
- Pointer variables
- Parameter passing by pointer
- The typedef Specifier
- Dynamic arrays and memory management
- Pointer arithmetic

Memory Address

- The memory unit of a computer is a series of byte memories stacked together and that have identification numbers assigned to them by the operating system
- The identification number of a memory unit is known as the **memory address** of the memory unit
- Memory addresses are made up of hexadecimal numbers (such as `0X5F24DABA`)
- The number of hexadecimal digits in a memory address is typically 8 digits or 16 digits depending on the operating system and the hardware



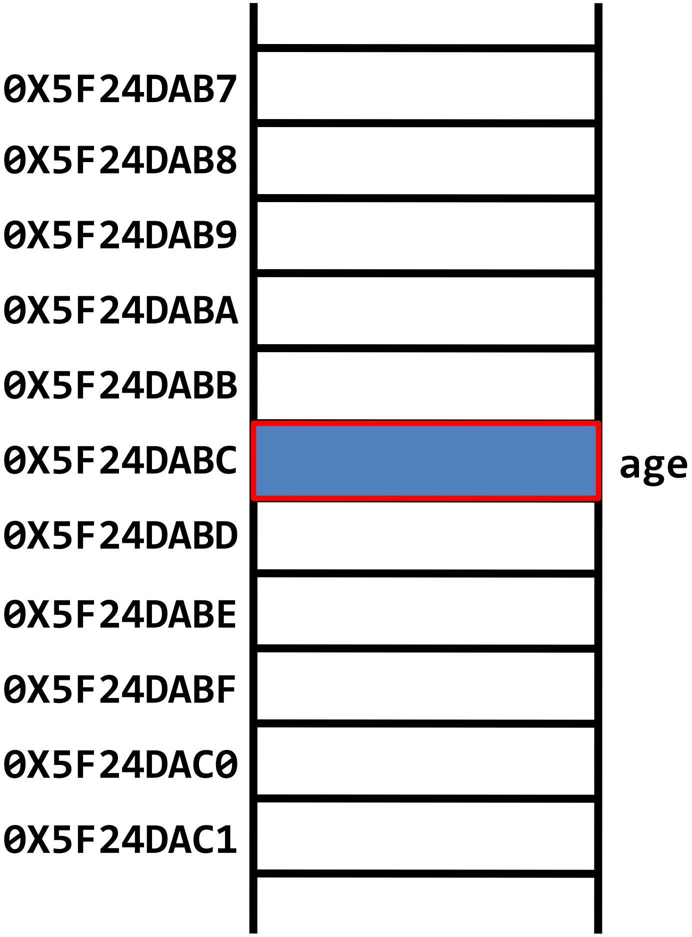
0X5F24DAB7
0X5F24DAB8
0X5F24DAB9
0X5F24DABA
0X5F24DABB
0X5F24DABC
0X5F24DABD
0X5F24DABE
0X5F24DABF
0X5F24DAC0
0X5F24DAC1

Memory Address

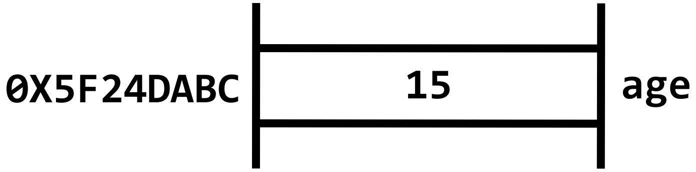
- Declaring a variable such as

  `int age;`

  therefore searches for a freely available memory space enough to store ***int*** value (4 bytes), reserves the memory space for the variable, and finally gives the memory space the programmer chosen name ***age***
- Thus the memory space will be identified by the programmer with the name ***age*** and with the memory address ***0X5F24DABC*** by the operating system



L-values and R-values



- Now consider the statement

  ```
  age = 15; //Here age refers to the memory space whose address is 0X5F24DABC
  ```

- The statement stores the value 15 in the memory space named age
- In this statement, ***age refers to the identifiable memory space*** whose address is **0X5F24DABC**
- On the other hand, the statement

  ```
  cout << age << endl; //Here age refers to the value stored in the memory space
  ```

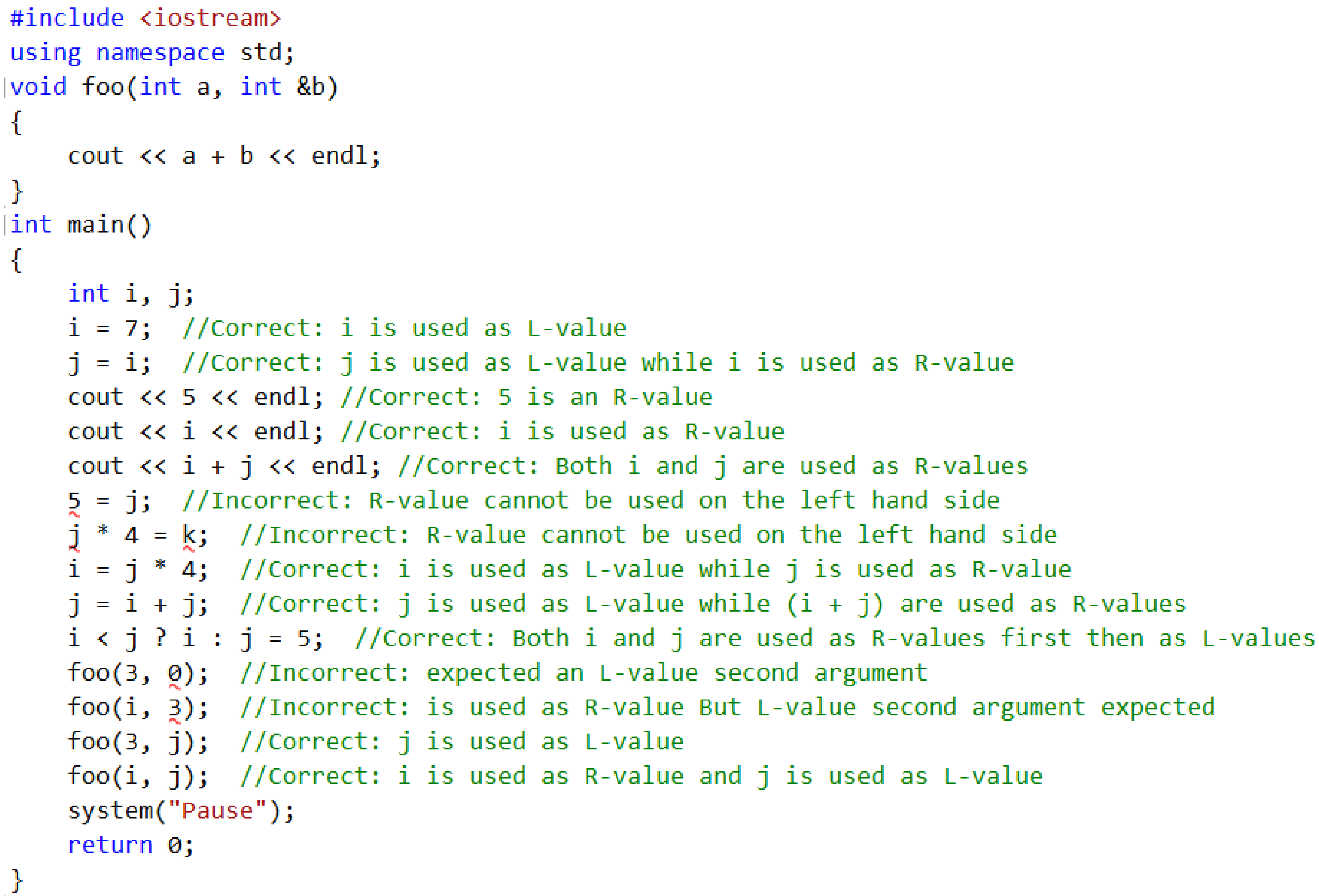
  prints the value stored in the memory space named age
- In this statement, ***age refers to the value 15 stored in the memory***
- In C++, an expression that refers to an identifiable memory space is known as an **L-value**; and an expression that is not an L-value is known as an **R-value**
- Literal values such as 8, -1.6, true, 'h', or "CMPT" are all **R-values**
- A variable name behaves as an **L-value** when it refers to an identifiable memory space but it behaves as an **R-value** when it refers to the literal value stored in a memory space

L-values and R-values

- Typically **R-values** are used
  - On the right hand side of assignment operator,
  - In cout statements,
  - In expressions (arithmetic, Boolean, logical), and
  - As arguments when passing arguments by value
- **R-values** cannot be used
  - On the left hand side of assignment operator, and
  - As arguments when passing by reference
- On the other hand **L-values** can be used in all places where **R-values** can be used (when they refer to the value stored in memory) as well as on those cases where **R-values** can't be used (when they refer to an identifiable memory)

L-values and R-values

- To better understand **L-values** and **R-values**, analyze the C++ program given below and determine how the variables are being used (**L-value** or **R-value**), identify all the syntax errors in the program, and explain the cause of the syntax errors

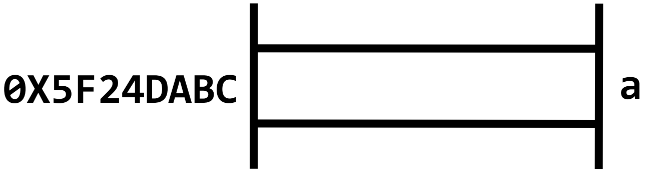


```
#include <iostream>
using namespace std;
void foo(int a, int &b)
{
    cout << a + b << endl;
}
int main()
{
    int i, j;
    i = 7;  //Correct: i is used as L-value
    j = i;  //Correct: j is used as L-value while i is used as R-value
    cout << 5 << endl; //Correct: 5 is an R-value
    cout << i << endl; //Correct: i is used as R-value
    cout << i + j << endl; //Correct: Both i and j are used as R-values
    5 = j;  //Incorrect: R-value cannot be used on the left hand side
    j * 4 = k;  //Incorrect: R-value cannot be used on the left hand side
    i = j * 4;  //Correct: i is used as L-value while j is used as R-value
    j = i + j;  //Correct: j is used as L-value while (i + j) are used as R-values
    i < j ? i : j = 5;  //Correct: Both i and j are used as R-values first then as L-values
    foo(3, 0);  //Incorrect: expected an L-value second argument
    foo(i, 3);  //Incorrect: is used as R-value But L-value second argument expected
    foo(3, j);  //Correct: j is used as L-value
    foo(i, j);  //Correct: i is used as R-value and j is used as L-value
    system("Pause");
    return 0;
}
```

The Address Operator &

- The address operator is used to compute the memory address of an **L-value** expression (such as a variable name of any data type)
- For example, given

```
int a;
```



- The print statement

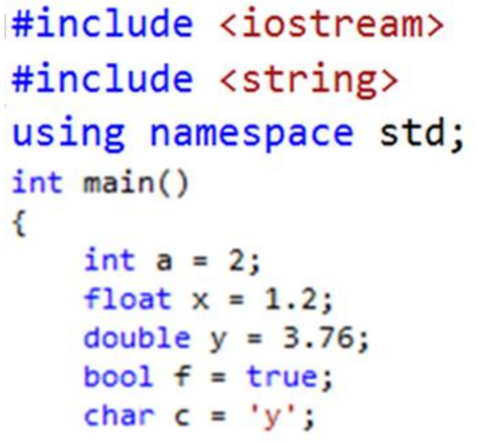
```
cout << &a << endl;
```

prints the memory address of the memory space that the variable **a** refers to which is **0X5F24DABC**

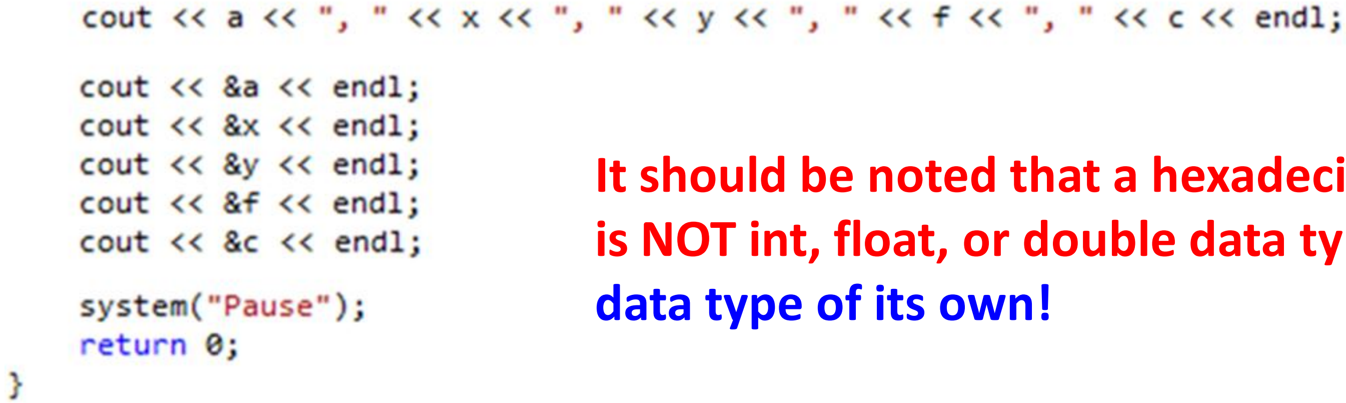
- Please note that the variable **a** does not need to be initialized in order to compute its memory address
- See the following program for more examples

The Address Operator &

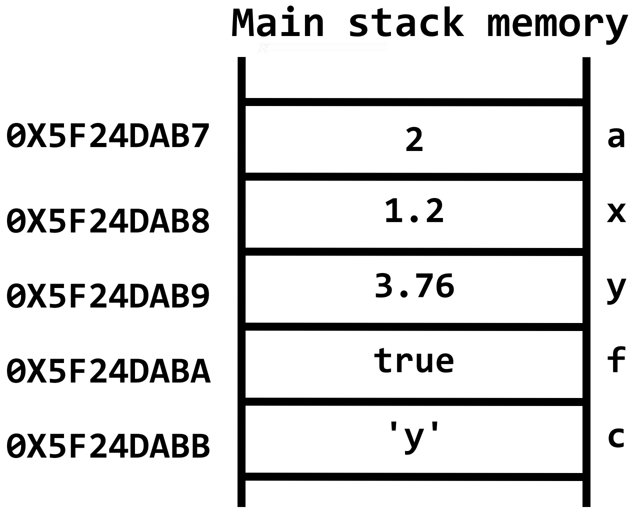
- The following program demonstrates the usage of the address operator (**&**) to print the address of memory spaces



```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    int a = 2;
    float x = 1.2;
    double y = 3.76;
    bool f = true;
    char c = 'y';
```



```
    cout << a << ", " << x << ", " << y << ", " << f << ", " << c << endl;

    cout << &a << endl;
    cout << &x << endl;
    cout << &y << endl;
    cout << &f << endl;
    cout << &c << endl;

    system("Pause");
    return 0;
}
```



Main stack memory

0X5F24DAB7	2	a
0X5F24DAB8	1.2	x
0X5F24DAB9	3.76	y
0X5F24DABA	true	f
0X5F24DABB	'y'	c

It should be noted that a hexadecimal number is NOT int, float, or double data type. It is a data type of its own!

Pointer Variables
Declaration, Initialization, and Definition

- C++ pointers are variables designed to store memory address values (which are hexadecimal numbers)
- **Syntax**

```
data_type *p; //pointer declaration
p = memory_address; //pointer initialization
data_type *p = memory_address; //pointer definition
```

- Thus a C++ pointer variable does not store data; instead it stores the address of a memory space that stores data
- Some pointer variable declarations are shown below

```
float *p1; //space before asterisk OK
double* p2; //space after asterisk OK
string * p3; //space before and after asterisk OK
char   *   p4; //many spaces OK but bad coding!
```

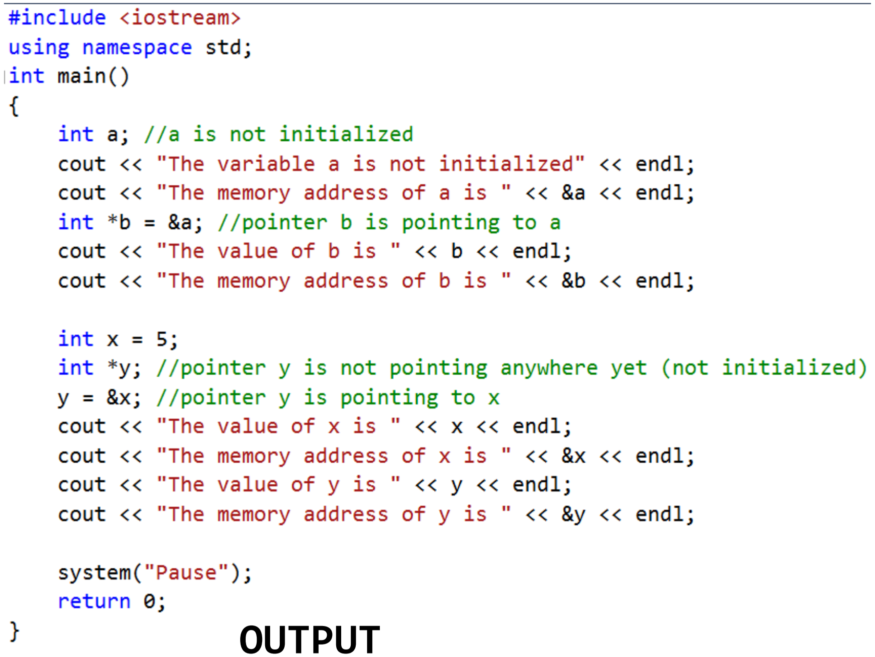
Pointer Variables
Declaration, Initialization, and Definition

- The memory address operator (**&**) is a handy tool in order to assign a pointer variable a memory address value
- Example

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

- In this case, the memory address of the memory space that the variable ***a*** refers to is assigned to the pointer variable ***b***
- When a pointer variable is assigned a memory address value of a certain memory space, we say the pointer is pointing to the memory space
- In the example above, we say the pointer **b** is pointing to the memory space named **a**
- See the example program below…

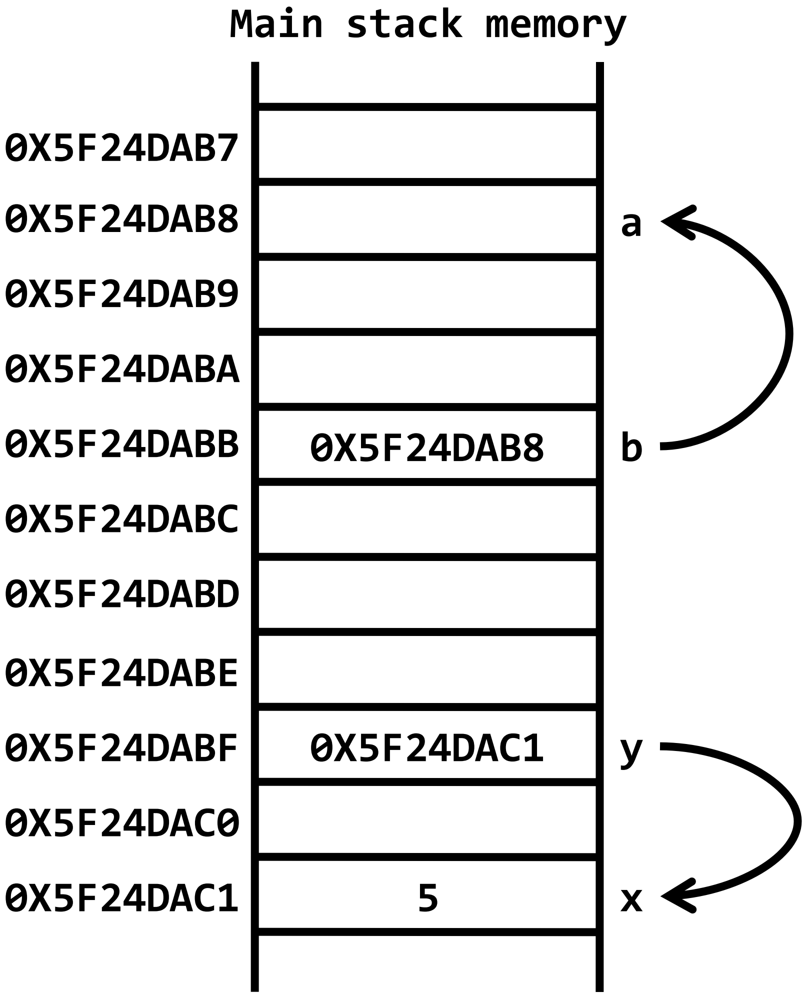
Pointer Variables
Declaration, Initialization, and Definition



```
#include <iostream>
using namespace std;
int main()
{
    int a; //a is not initialized
    cout << "The variable a is not initialized" << endl;
    cout << "The memory address of a is " << &a << endl;
    int *b = &a; //pointer b is pointing to a
    cout << "The value of b is " << b << endl;
    cout << "The memory address of b is " << &b << endl;

    int x = 5;
    int *y; //pointer y is not pointing anywhere yet (not initialized)
    y = &x; //pointer y is pointing to x
    cout << "The value of x is " << x << endl;
    cout << "The memory address of x is " << &x << endl;
    cout << "The value of y is " << y << endl;
    cout << "The memory address of y is " << &y << endl;

    system("Pause");
    return 0;
}
```

OUTPUT

```
The variable a is not initialized
The memory address of a is 0X5F24DAB8
The value of b is 0X5F24DAB8
The memory address of b is 0X5F24DABB
The value of x is 5
The memory address of x is 0X5F24DAC1
The value of y is 0X5F24DAC1
The memory address of y is 0X5F24DABF
Press any key to continue . . .
```



Main stack memory

0X5F24DAB7		
0X5F24DAB8		a
0X5F24DAB9		
0X5F24DABA		
0X5F24DABB	0X5F24DAB8	b
0X5F24DABC		
0X5F24DABD		
0X5F24DABE		
0X5F24DABF	0X5F24DAC1	y
0X5F24DAC0		
0X5F24DAC1	5	x

Incorrect use of pointers

- Pointer variables require strict data type matching between a pointer and the memory space whose memory address is assigned to the pointer
- The following code segment gives correct and incorrect uses of pointer variables

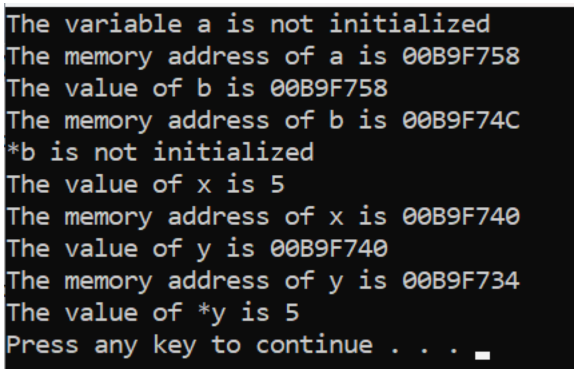
```
int a = 7;
float b = 3.5;
float* c = &b; ← Correct!
int* d = &b; ← Error! Data type mismatch
```

Pointer Variables
The dereference operator

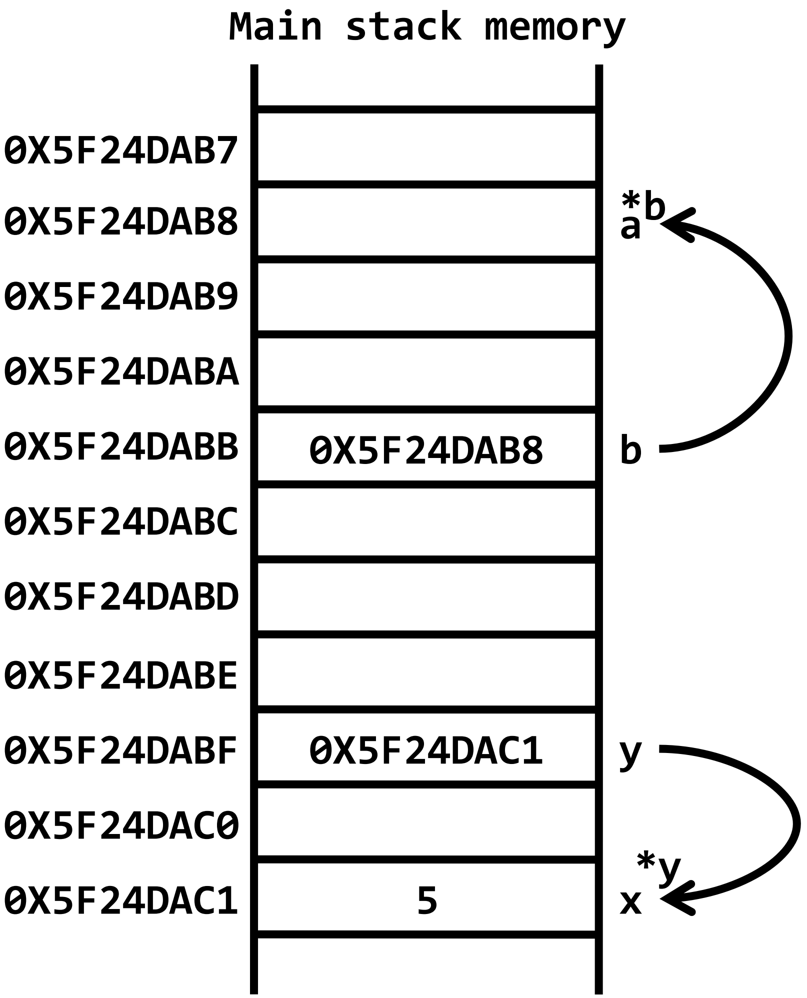
- The main purpose of pointer variables is that a pointer variable is able to refer to the memory space that it is pointing to
- Given a pointer variable ***b*** that is already declared and initialized, the dereference of the pointer variable denoted as ****b*** refers to the memory space that the pointer is pointing
- See the example program below

Pointer Variables
The dereference operator

```
int main()
{
    int a;
    cout << "The variable a is not initialized" << endl;
    cout << "The memory address of a is " << &a << endl;
    int *b = &a;  //pointer b is pointing to a
    cout << "The value of b is " << b << endl;
    cout << "The memory address of b is " << &b << endl;
    cout << "*b is not initialized" << endl;

    int x = 5;
    int *y; //pointer y is not pointing anywhere yet (not initialized)
    y = &x;  //pointer y is pointing to x
    cout << "The value of x is " << x << endl;
    cout << "The memory address of x is " << &x << endl;
    cout << "The value of y is " << y << endl;
    cout << "The memory address of y is " << &y << endl;
    cout << "The value of *y is " << *y << endl;

    system("Pause");
    return 0;
}
```

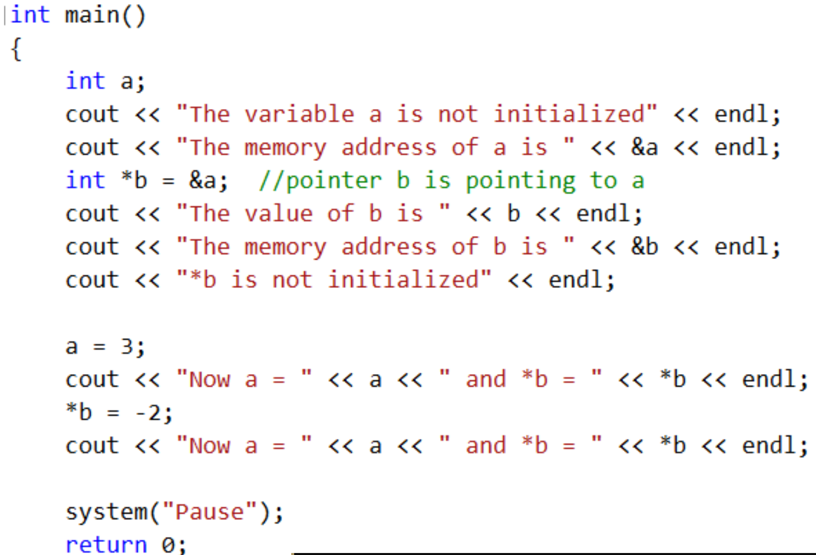


```
The variable a is not initialized
The memory address of a is 00B9F758
The value of b is 00B9F758
The memory address of b is 00B9F74C
*b is not initialized
The value of x is 5
The memory address of x is 00B9F740
The value of y is 00B9F740
The memory address of y is 00B9F734
The value of *y is 5
Press any key to continue . . .
```

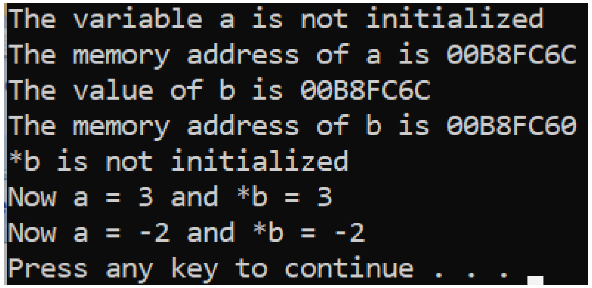


Pointer Variables
The dereference operator

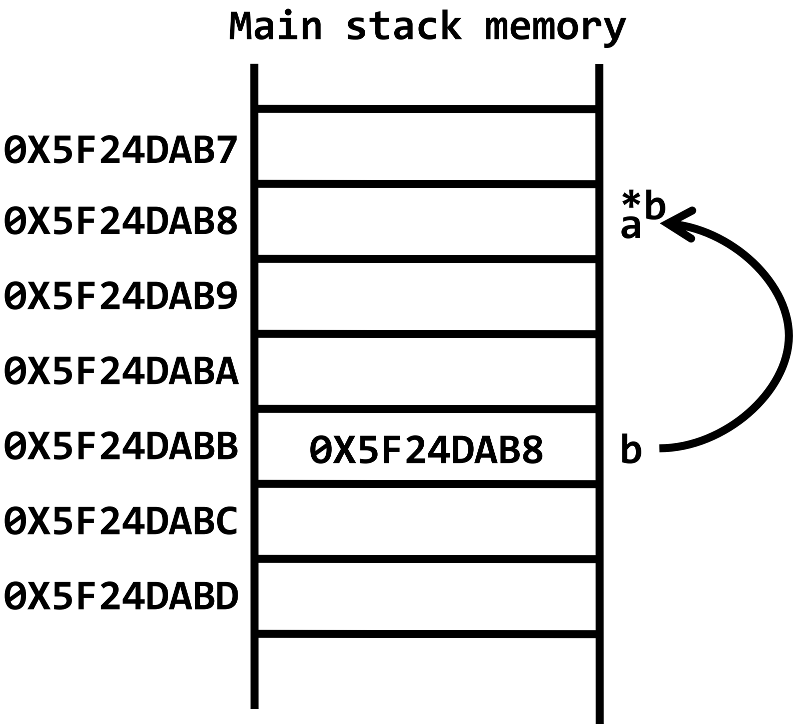
- The dereference of a pointer is an **L-value** and can also be used to access the memory space that the pointer is pointing to as shown below



```
int main()
{
    int a;
    cout << "The variable a is not initialized" << endl;
    cout << "The memory address of a is " << &a << endl;
    int *b = &a;  //pointer b is pointing to a
    cout << "The value of b is " << b << endl;
    cout << "The memory address of b is " << &b << endl;
    cout << "*b is not initialized" << endl;

    a = 3;
    cout << "Now a = " << a << " and *b = " << *b << endl;
    *b = -2;
    cout << "Now a = " << a << " and *b = " << *b << endl;

    system("Pause");
    return 0;
}
```

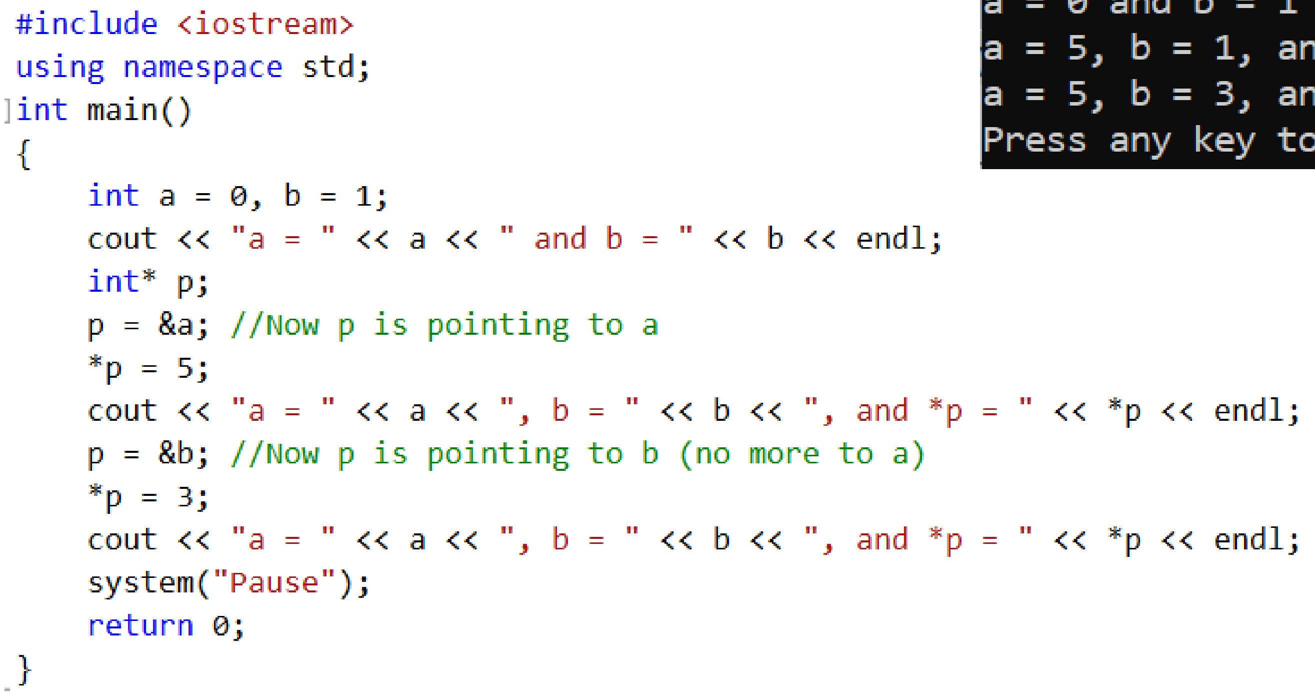


```
The variable a is not initialized
The memory address of a is 00B8FC6C
The value of b is 00B8FC6C
The memory address of b is 00B8FC60
*b is not initialized
Now a = 3 and *b = 3
Now a = -2 and *b = -2
Press any key to continue . . .
```



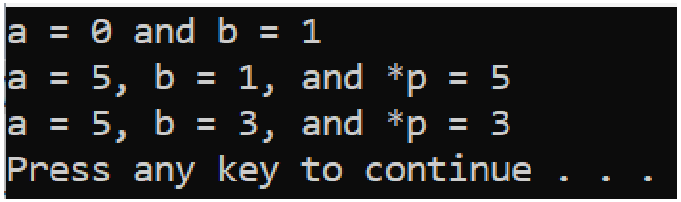
Pointer Variable Modification

- When a pointer variable is modified, it points to a different memory space
- The dereference of the pointer will also refer to the different memory space
- See example below



```cpp
#include <iostream>
using namespace std;
int main()
{
    int a = 0, b = 1;
    cout << "a = " << a << " and b = " << b << endl;
    int* p;
    p = &a; //Now p is pointing to a
    *p = 5;
    cout << "a = " << a << ", b = " << b << ", and *p = " << *p << endl;
    p = &b; //Now p is pointing to b (no more to a)
    *p = 3;
    cout << "a = " << a << ", b = " << b << ", and *p = " << *p << endl;
    system("Pause");
    return 0;
}
```

OUTPUT



```
a = 0 and b = 1
a = 5, b = 1, and *p = 5
a = 5, b = 3, and *p = 3
Press any key to continue . . .
```

The Heap Memory
The new operator

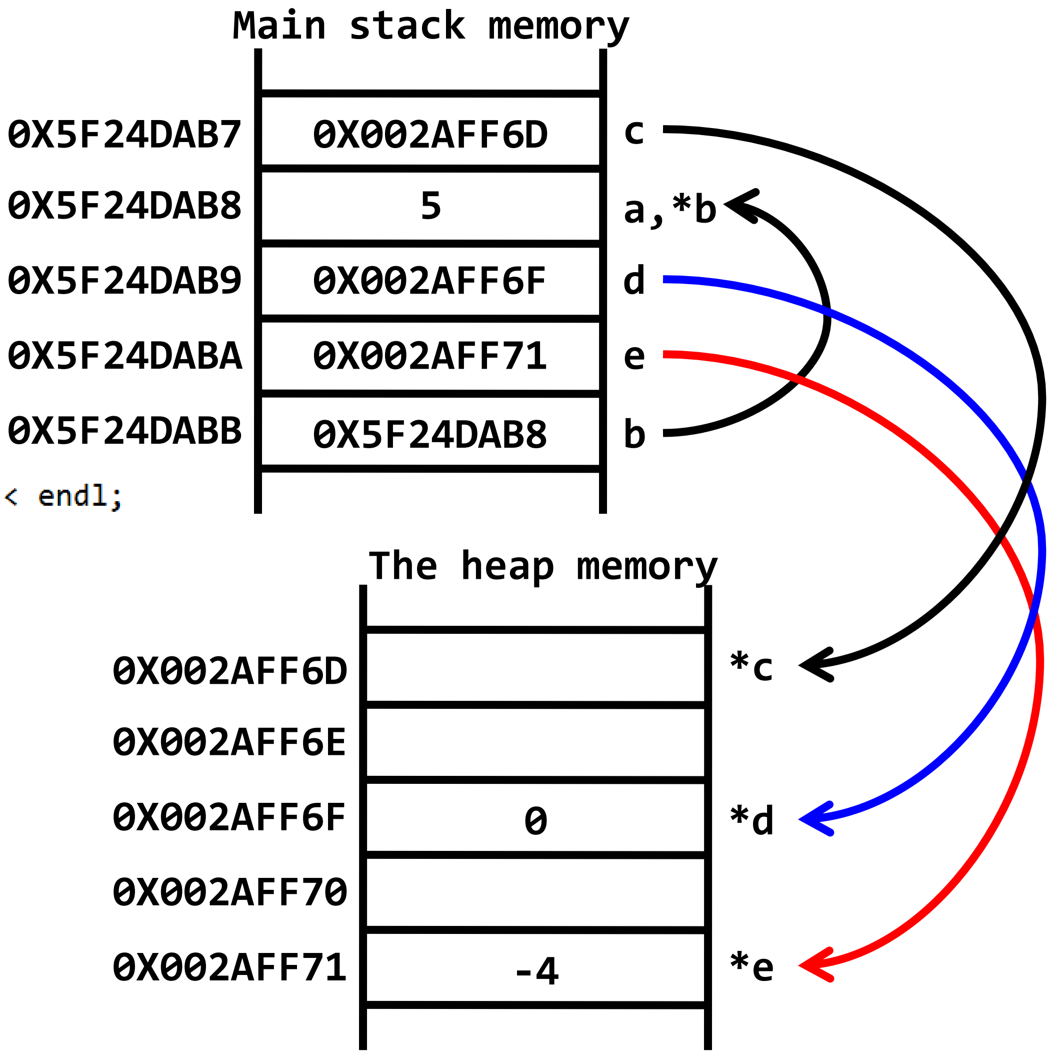
- Pointer variables can be assigned the memory address of existing variables (memory spaces) as shown in the previous examples
- In addition, it is also possible to allocate a new memory space with a pointer variable using the ***new*** operator
- When a new memory space is allocated using a pointer variable, the new memory space will be allocated from a special memory space known as the **heap memory** (or **the free store**)
- A memory space allocated in the heap memory is also known as a ***dynamic memory***
- See the example program given below…

The Heap Memory

```
#include <iostream>
using namespace std;
int main()
{
    int a = 5;
    int *b = &a;
    int *c = new int;
    int *d = new int();
    int *e = new int(-4);
    cout << "a = " << a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c is not initialized" << endl;
    cout << "*d = " << *d << endl;
    cout << "*e = " << *e << endl;
    system("Pause");
    return 0;
}
```

OUTPUT

```
a = 5
*b = 5
*c is not initialized
*d = 0
*e = -4
Press any key to continue . . .
```



Main stack memory

0X5F24DAB7	0X002AFF6D	c
0X5F24DAB8	5	a,*b
0X5F24DAB9	0X002AFF6F	d
0X5F24DABA	0X002AFF71	e
0X5F24DABB	0X5F24DAB8	b

The heap memory

0X002AFF6D		*c
0X002AFF6E		
0X002AFF6F	0	*d
0X002AFF70		
0X002AFF71	-4	*e

The Heap Memory Access Rule

- The heap memory is accessible from any block of a program (such as main program block and function block)
- Therefore it is common memory to all parts of a program
- This means we can use the new operator inside a main program or inside any function to allocate memory in the heap memory

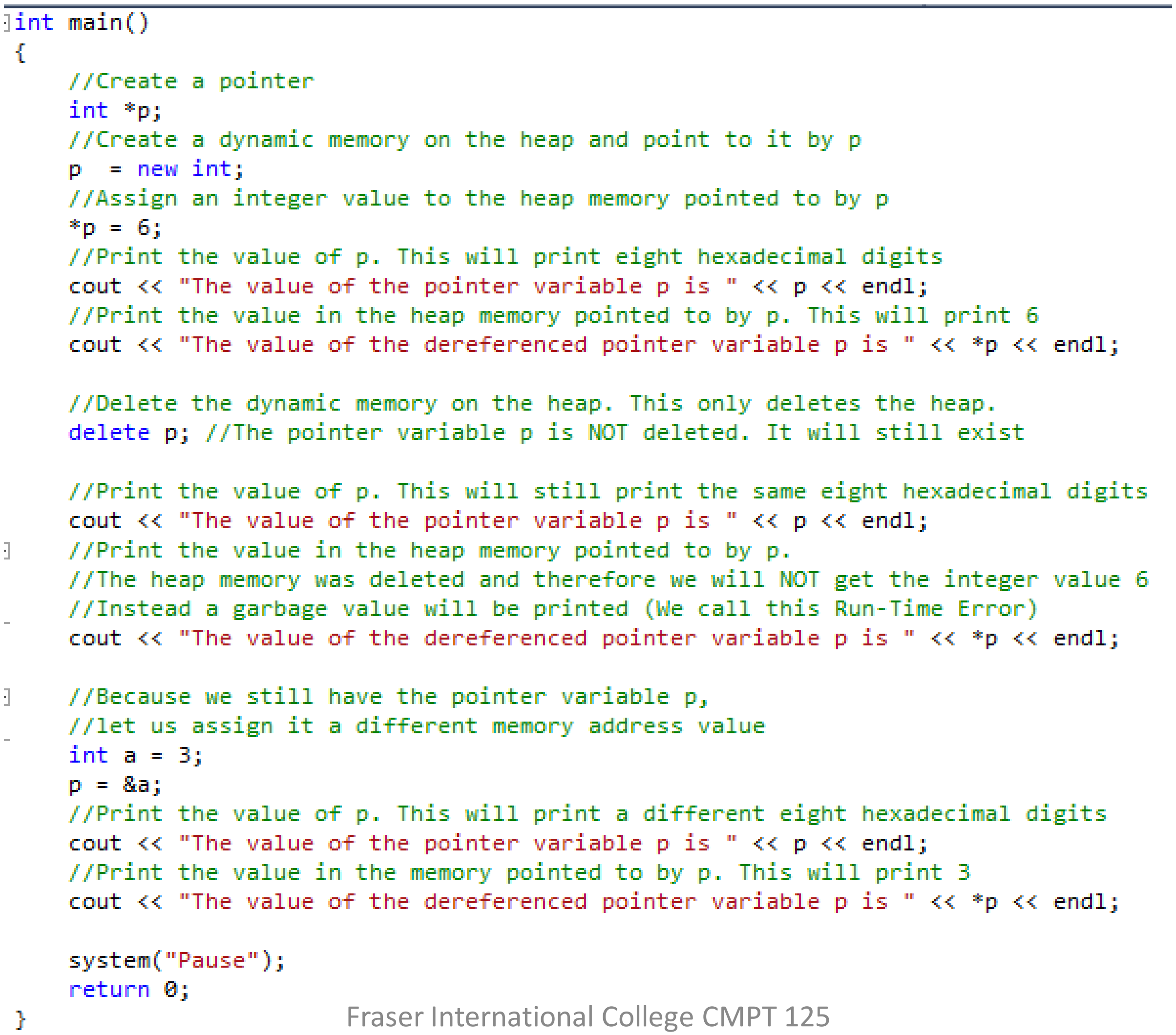
The Heap Memory
The `delete` operator

- Memory spaces allocated in the main stack and call stack are automatically de-allocated (cleared) at the moment a function completes its execution or when a program finishes execution
- The **heap memory** is however an exception to this rule
- When our program terminates, ***any memory that was allocated in the heap memory still remains tied to our program even after our program terminates***
- For this reason it is important for us to delete (or de-allocate or clear) any memory that was allocated on the heap memory otherwise the memory space will be wasted (***memory leak***)
- In order to de-allocate a heap memory, we use the ***`delete`*** operator
- Give a pointer p that is pointing to a memory space in the heap memory

  ```
  delete p;
  ```

  deletes the memory in the heap pointed by the pointer variable p
- Once a pointer is deleted, it can no more be used to access the memory that was pointed by the pointer variable. See below…

The Heap Memory



```
int main()
{
    //Create a pointer
    int *p;
    //Create a dynamic memory on the heap and point to it by p
    p  = new int;
    //Assign an integer value to the heap memory pointed to by p
    *p = 6;
    //Print the value of p. This will print eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the heap memory pointed to by p. This will print 6
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    //Delete the dynamic memory on the heap. This only deletes the heap.
    delete p; //The pointer variable p is NOT deleted. It will still exist

    //Print the value of p. This will still print the same eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the heap memory pointed to by p.
    //The heap memory was deleted and therefore we will NOT get the integer value 6
    //Instead a garbage value will be printed (We call this Run-Time Error)
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    //Because we still have the pointer variable p,
    //let us assign it a different memory address value
    int a = 3;
    p = &a;
    //Print the value of p. This will print a different eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the memory pointed to by p. This will print 3
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    system("Pause");
    return 0;
}
```

Pointer to a de-referenced Pointer

- Consider the following code segment

```
int a = 9;
int *p = &a;
int *q = &(*p);
```

- Now **q** is a pointer to ***p**
- But ***p** is **a**
- Therefore **q** is a pointer to **a**

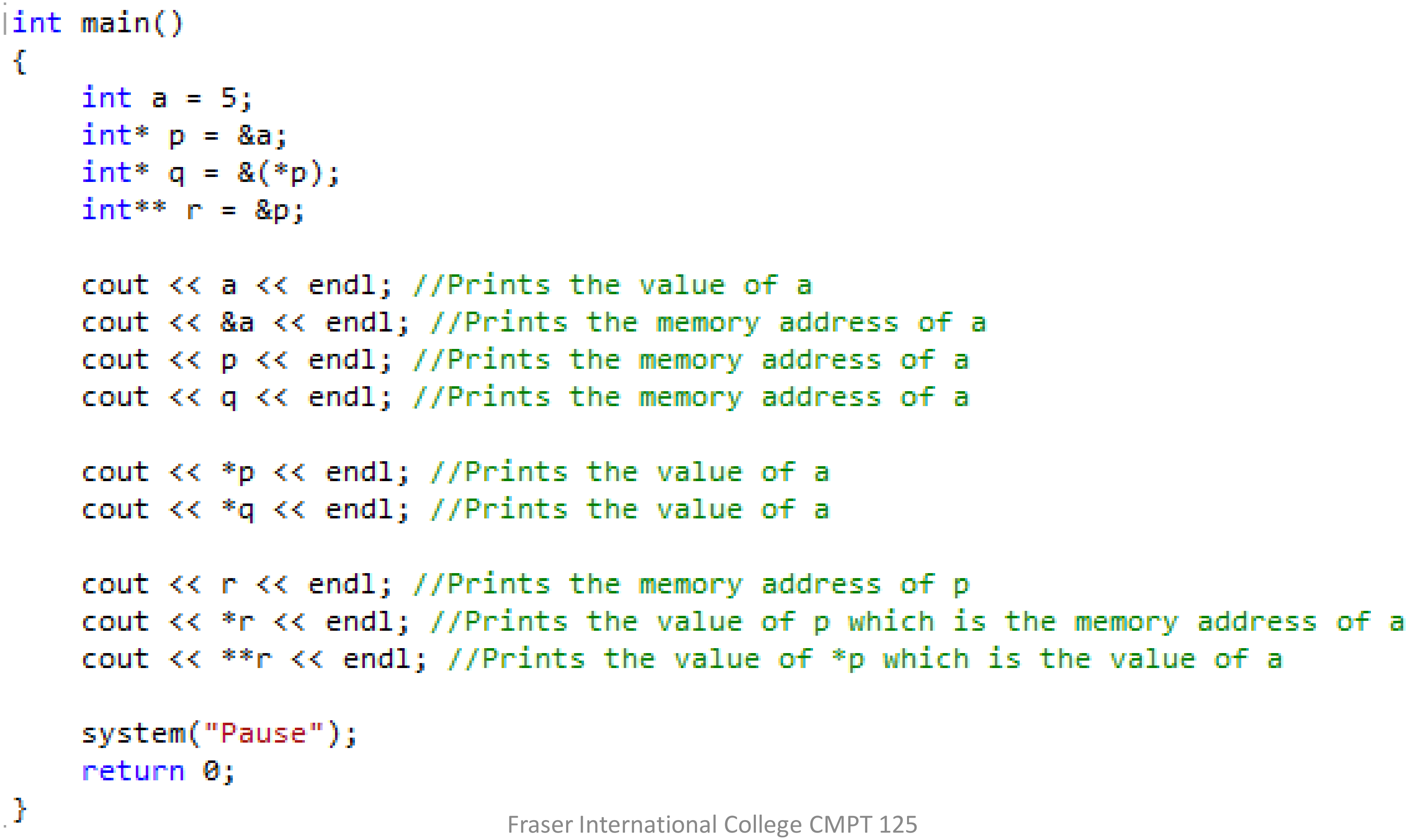
Pointer to a Pointer

- Consider the following code segment

```
int a = 5;
int *b = &a;
int **c = &b;
```

- Now **c** is a pointer to **b**
- But **b** is itself a pointer to **a**
- The variable **c** is therefore a pointer to the pointer variable **b**
- We say the variable **c** is a pointer to a pointer (or simply, a double pointer)
- Moreover, ***c** refers to **b**
- While ****c** refers to **a**

Pointer to de-referenced pointer and Pointer to Pointer Example



```
int main()
{
    int a = 5;
    int* p = &a;
    int* q = &(*p);
    int** r = &p;

    cout << a << endl; //Prints the value of a
    cout << &a << endl; //Prints the memory address of a
    cout << p << endl; //Prints the memory address of a
    cout << q << endl; //Prints the memory address of a

    cout << *p << endl; //Prints the value of a
    cout << *q << endl; //Prints the value of a

    cout << r << endl; //Prints the memory address of p
    cout << *r << endl; //Prints the value of p which is the memory address of a
    cout << **r << endl; //Prints the value of *p which is the value of a

    system("Pause");
    return 0;
}
```

Reference to a Pointer

- Similarly, we can create a reference to a pointer variable
- A reference to a pointer variable must itself be a pointer
- For example

```
int a = 5;
int *c = &a; //A pointer to a value variable
int* &d = c;//A reference to a pointer
```

- Now **d** is a reference to the pointer variable **c**

Reference to a de-referenced Pointer

- We may also create a reference variable to a de-referenced pointer as follows

```
int a = 9;
int *p = &a;
int &b = *p; //b is an int variable
```

- Now **b** is a reference to the de-reference of **p** which is the same as saying **b** is a reference to the variable **a**

Parameter Passing in C++
Pass by Pointer

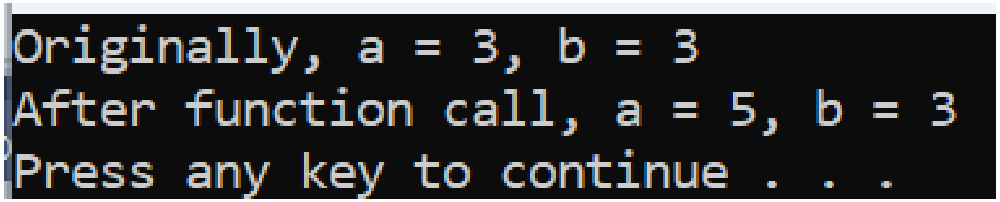
- A pointer can also be passed to a function as an argument
- The corresponding function parameter is then required to be a pointer as well
- In this case, the value of the pointer argument **(which is a memory address value)** will be copied to the function parameter
- This means both the pointer argument and the pointer parameter will point to the same memory space
- Therefore the pointer parameter of the function together with de-referencing can then be used to access the memory space pointed by the parameter (and also by the argument)
- See the example below…

Parameter Passing in C++

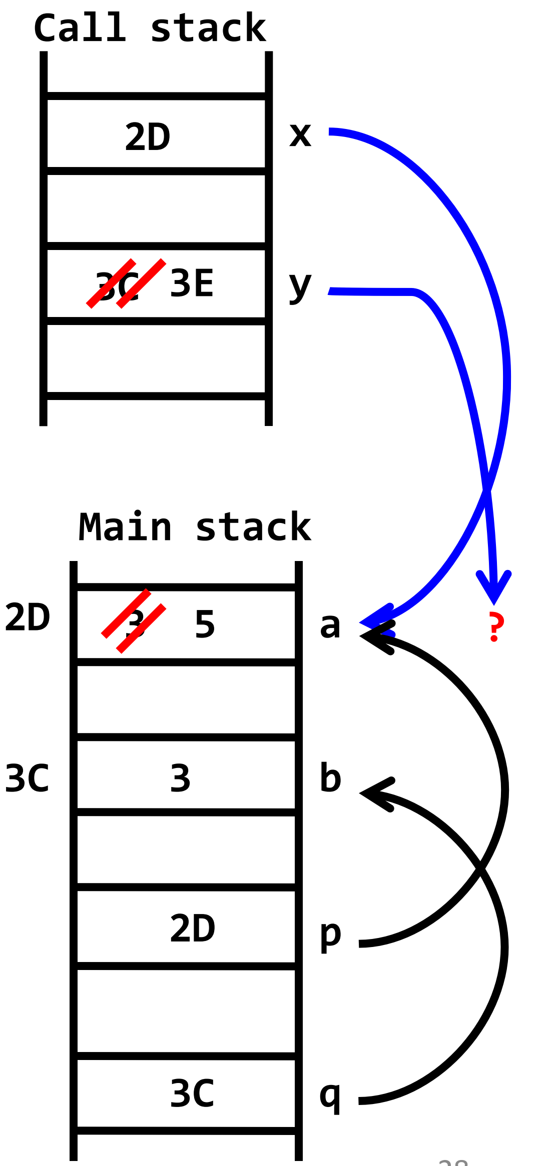
Pass by Pointer

```
void foo(int* x, int *y)
{
    *x += 2;
    y += 2;
}
int main()
{
    int a = 3, b = 3;
    int *p = &a;
    int *q = &b;
    cout << "Originally, a = " << a << ", b = " << b << endl;
    foo(p, q);
    cout << "After function call, a = " << a << ", b = " << b << endl;
    system("Pause");
    return 0;
}
```

OUTPUT



```
Originally, a = 3, b = 3
After function call, a = 5, b = 3
Press any key to continue . . .
```



Parameter Passing in C++

Pass by Pointer

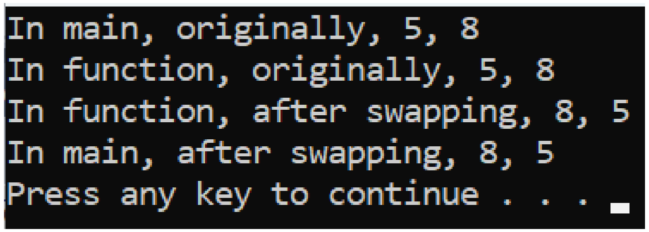
- **When passing pointers to functions; although the parameter passing is strictly speaking pass by value, de-referencing makes it possible for the pointer parameter to work outside the function call stack**
- This is known as **parameter passing by pointer**
- We may also pass the memory address of a variable to a function as an argument
- Once again this will be parameter passing by pointer as well
- See below...

Parameter Passing in C++
Pass by Pointer

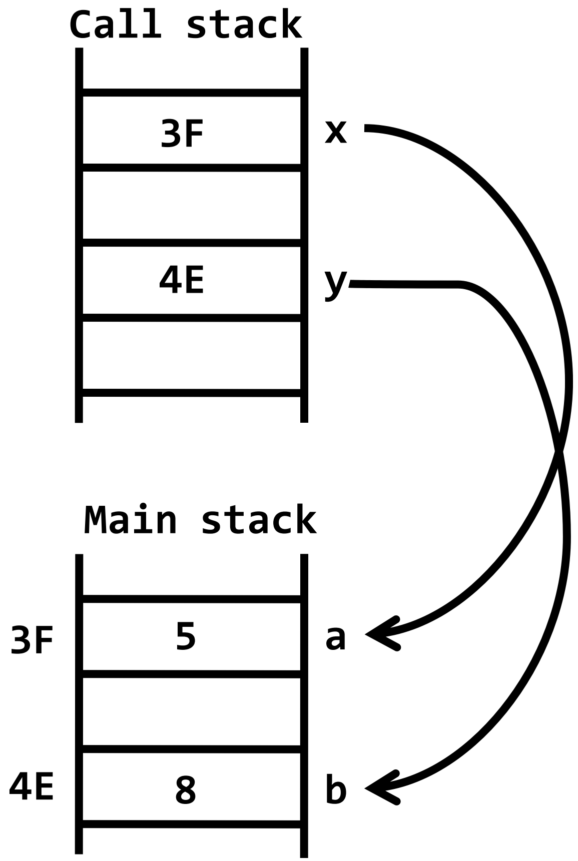
- This program swaps the values of the variables a and b defined in the main program from within the function using **parameter passing by pointer**

```cpp
#include <iostream>
using namespace std;
void swap(int* x, int* y)
{
    cout << "In function, originally, " << *x << ", " << *y << endl;
    int temp = *x;
    *x = *y;
    *y = temp;
    cout << "In function, after swapping, " << *x << ", " << *y << endl;
}
int main()
{
    int a = 5, b = 8;
    cout << "In main, originally, " << a << ", " << b << endl;
    swap(&a, &b);
    cout << "In main, after swapping, " << a << ", " << b << endl;
    system("Pause");
    return 0;
}
```

OUTPUT



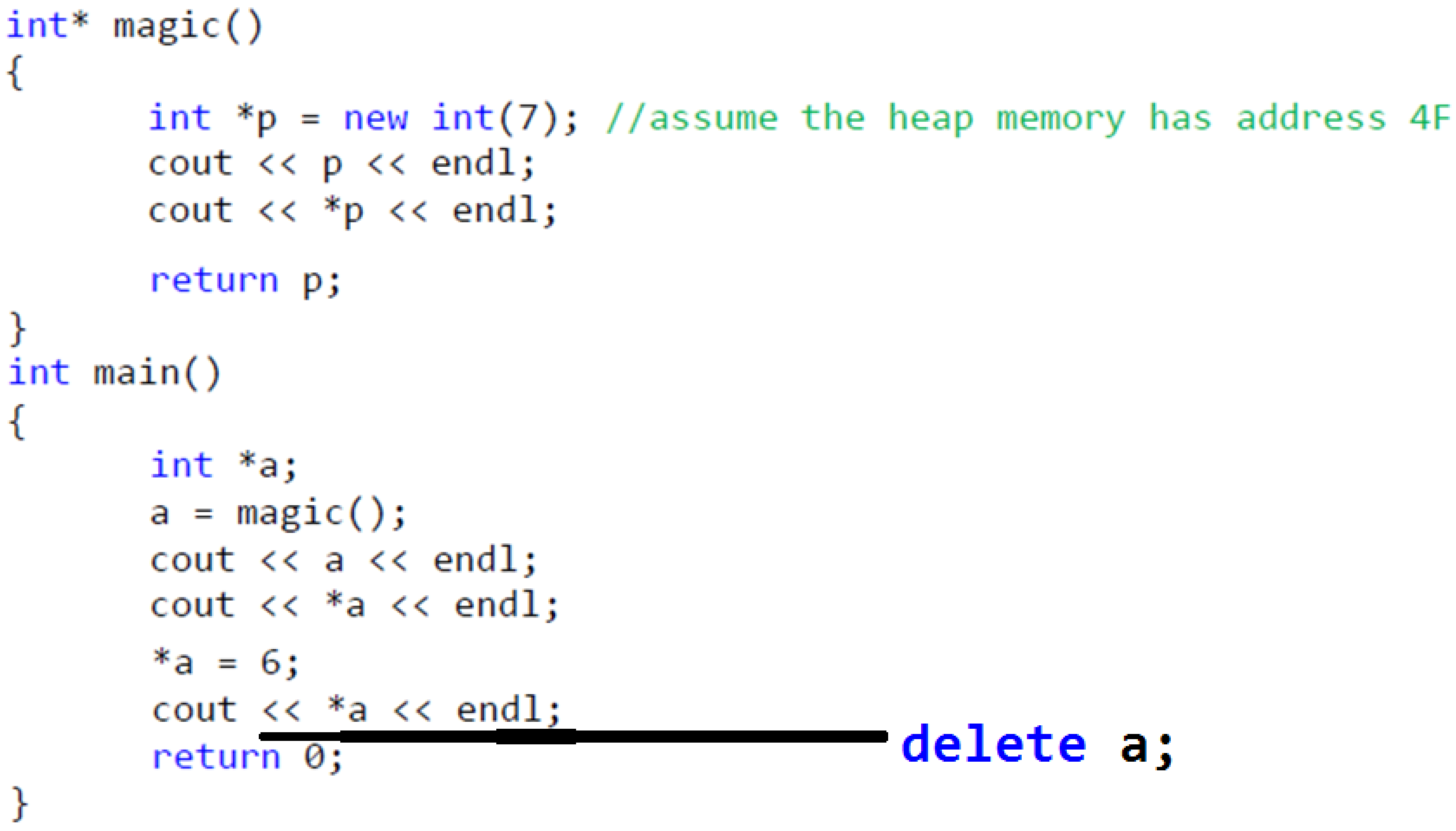
```
In main, originally, 5, 8
In function, originally, 5, 8
In function, after swapping, 8, 5
In main, after swapping, 8, 5
Press any key to continue . . .
```



Function Return
Returning a Pointer

- A C++ function may also return a pointer in which case the function will effectively return a memory address value
- Thus the returned value from the function may be stored in a pointer variable and subsequently de-referenced to access the memory space the pointer is pointing to

Function Return
Returning a Pointer

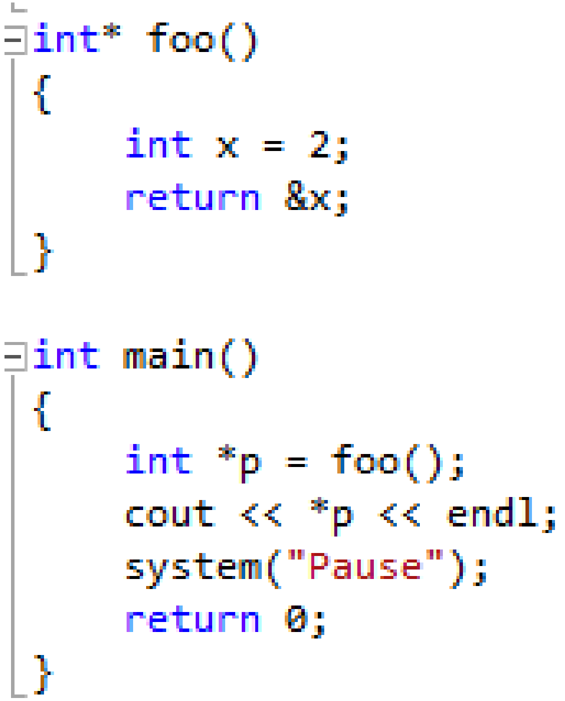


```
int* magic()
{
        int *p = new int(7); //assume the heap memory has address 4F
        cout << p << endl;
        cout << *p << endl;

        return p;
}
int main()
{
        int *a;
        a = magic();
        cout << a << endl;
        cout << *a << endl;

        *a = 6;
        cout << *a << endl;
        return 0;
}
```

```
delete a;
```

Function Return
Returning a Pointer

- It should be noted that when returning a pointer, a function should ensure that the memory location whose address is returned will still have a scope outside the function; for otherwise it will cause a runtime error when we try to access the memory afterwards

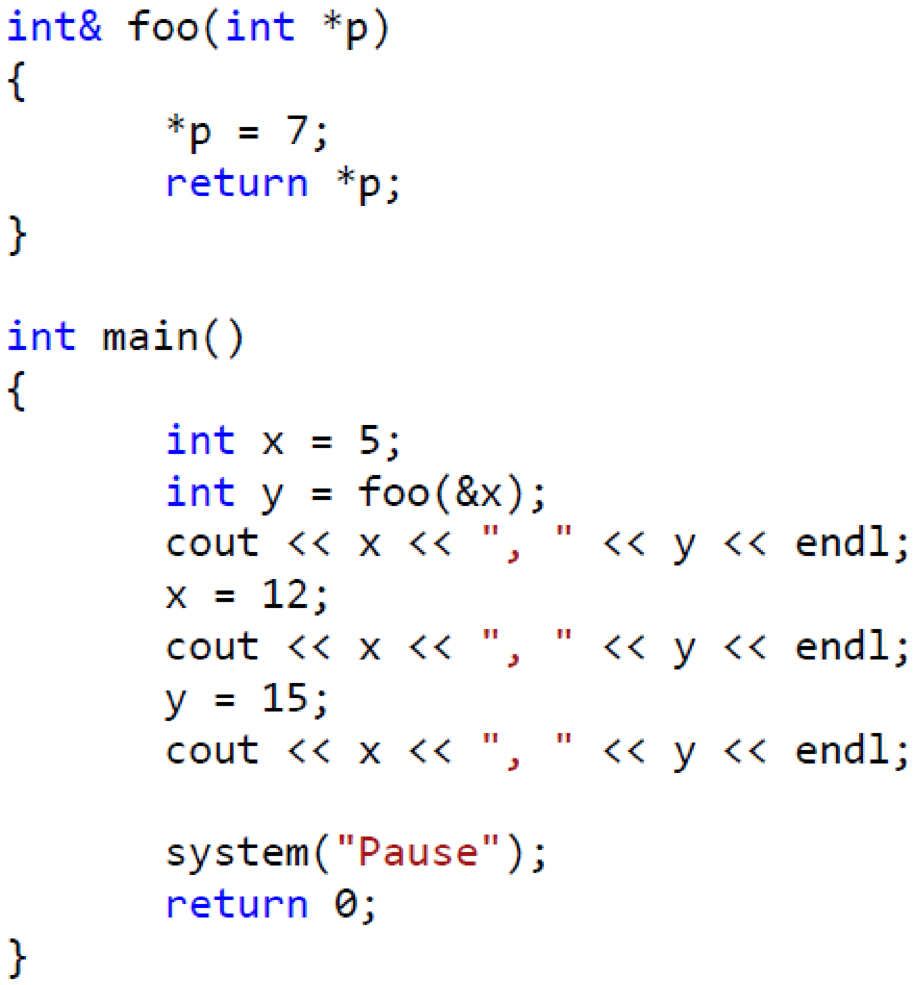


```
int* foo()
{
    int x = 2;
    return &x;
}

int main()
{
    int *p = foo();
    cout << *p << endl;
    system("Pause");
    return 0;
}
```

Function Return
Returning a Reference

- A function may also return by reference
- In this case, the function will not return a value stored in a memory space; neither will it return the memory address of a memory space
- Instead it will return the memory space itself
- This means the returned value of the function will be an **L-value**
- See example below

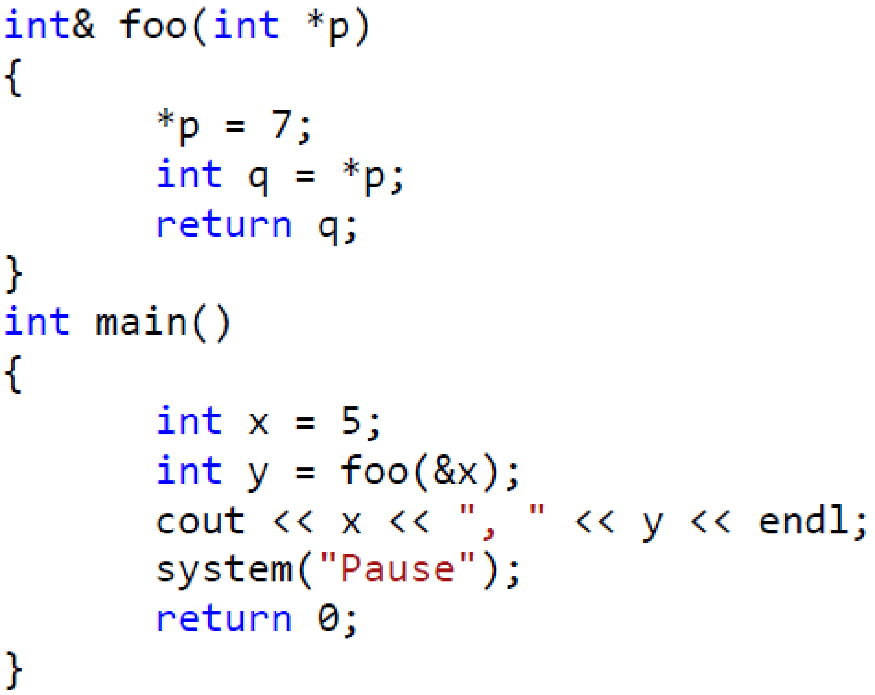
Function Return
Returning a Reference



```
int& foo(int *p)
{
        *p = 7;
        return *p;
}

int main()
{
        int x = 5;
        int y = foo(&x);
        cout << x << ", " << y << endl;
        x = 12;
        cout << x << ", " << y << endl;
        y = 15;
        cout << x << ", " << y << endl;

        system("Pause");
        return 0;
}
```

Function Return
Returning a Reference

- It should be noted a function should never return a local memory location by reference because the memory space will be cleared once control goes out of the function and thus the memory can not be referenced any longer. See example below.



```
int& foo(int *p)
{
        *p = 7;
        int q = *p;
        return q;
}
int main()
{
        int x = 5;
        int y = foo(&x);
        cout << x << ", " << y << endl;
        system("Pause");
        return 0;
}
```

C++ typedef Specifier

- Consider the following declaration

```
int a, b, c;
```

- What is the data type of **a**, what is the data type of **b**, and what is the data type of **c**?

  Answer:- All **a**, **b**, and **c** are **int** type

- Now consider the following declaration

  int* a, b, c;

- What is the data type of **a**, what is the data type of **b**, and what is the data type of **c**?
- **Answer**
  - The variable **a** is an **integer pointer** data type
  - But both **b** and **c** are **integer** data type!!!

C++ typedef Specifier

- Why so?
- Because in C++ the pointer * designation applies only to the first variable in the declaration; all the other variables in the same declaration will not get the pointer designation!
- So how can we declare two or more variables in the same declaration statement and yet we would like all of them to be pointers?
- **Answer**:- We have two ways
  - ➢ One way is to have a pointer designation for each of the variables as follows

    ```
    int *a, *b, *c;
    ```

  - ➢ Another way is to **define a new named data type** of our interest **using typedef** specifier as follows

C++ typedef Specifier

- In order to define a data type name, we proceed as follows

```
typedef int* intPtr;
```

- Now `intPtr` is a data type name
- Moreover `intPtr` is another name to **int***
- We can choose any valid identifier name for the new data type name
- Then we can declare integer pointer variables as follows

```
intPtr a, b, c;
```

- Now all the variables **a**, **b**, and **c** are **intPtr** data type; that is to say, they are **int*** data type variables
- Normally, we put **typedef** statement **at the top of our C++ programs just below the include directives!!!**

C++ Dynamic Arrays

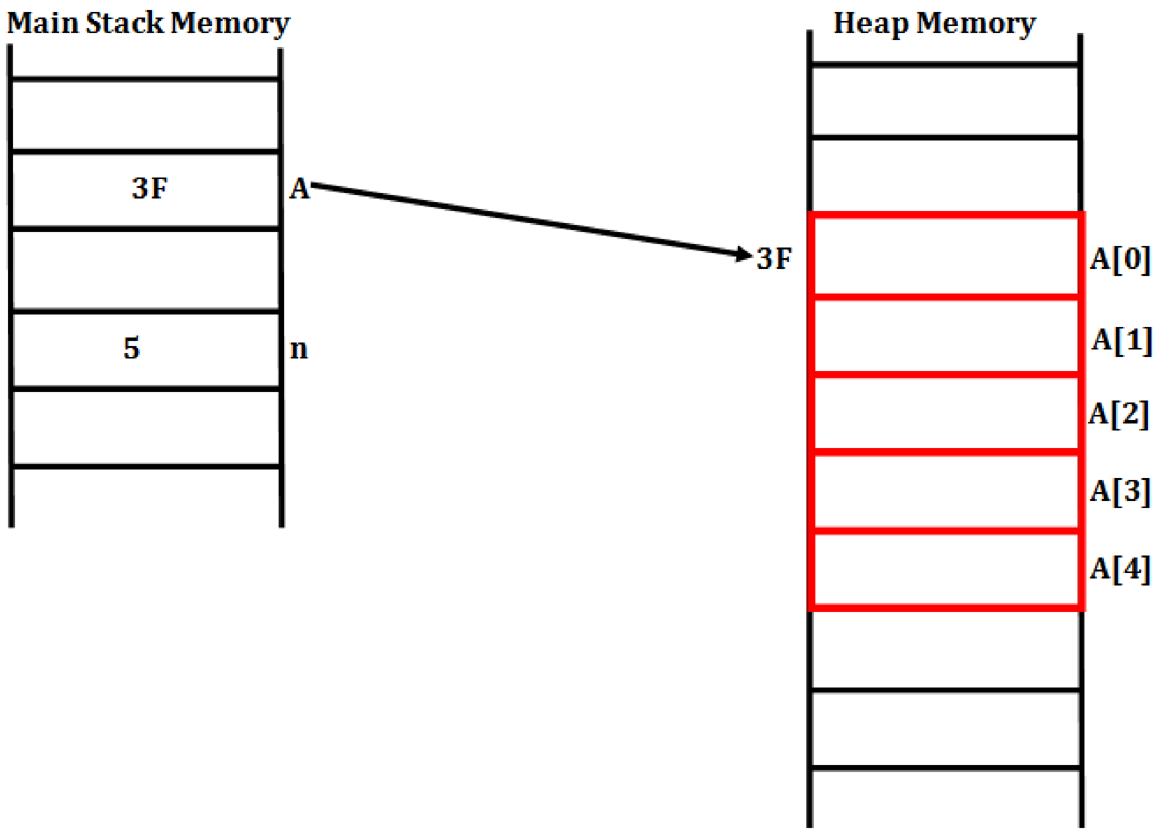
- Pointers also allow us to allocate more than one memory spaces on the heap as follows

```
int n;
cout << "How many elements would you like to store? ";
cin >> n;
double *A = new double[n];
```

- The pointer variable **A** is called a dynamic array
- In this case, **n** consecutive memory spaces will be allocated on the heap memory and the pointer **A** will point to the first memory space
- We say the memory spaces allocated are the elements of the dynamic array **A** and can be accessed by indexing in the same way we access elements of C++ static arrays
- See the diagram below…

C++ Dynamic Arrays

```
double *A = new double[n];
```



C++ Dynamic Arrays

- The creation of one dimensional C++ dynamic arrays follows the following syntax
- **Syntax**

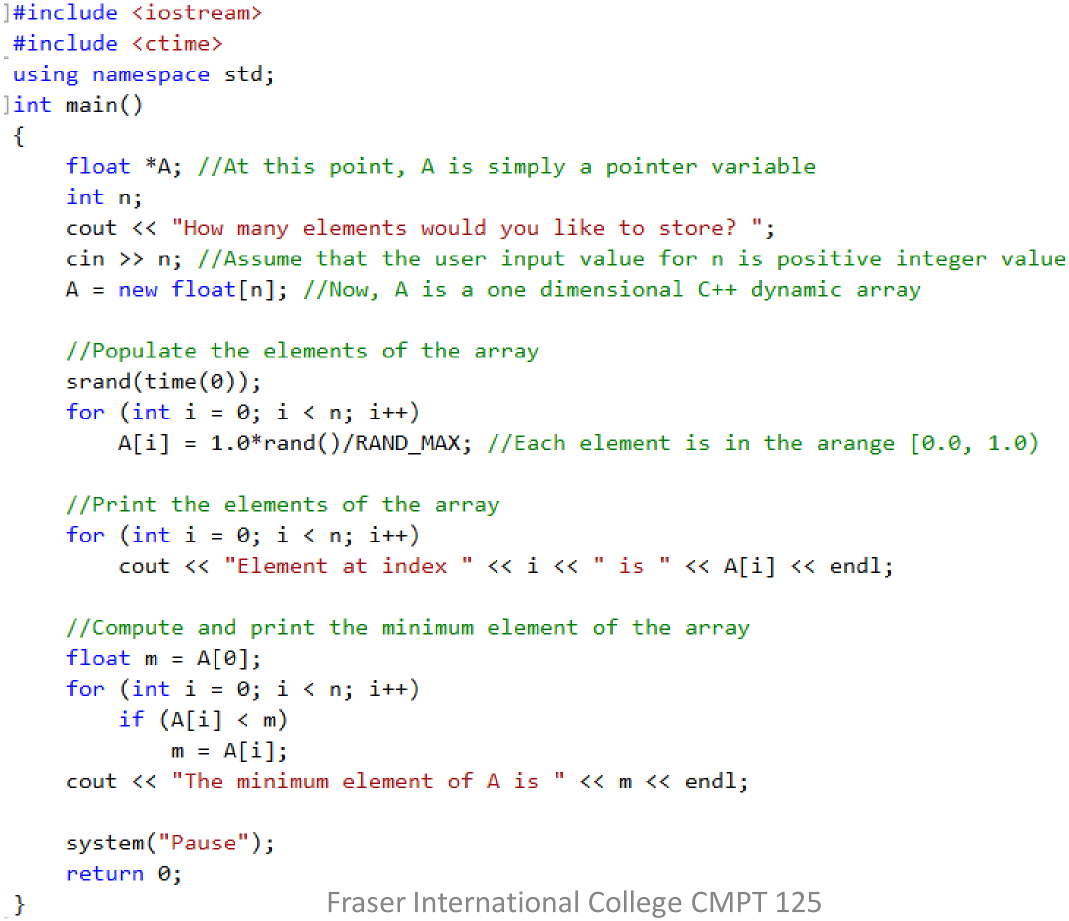
```
data_type *A = new data_type[n];
```

- Alternatively,

```
data_type *A;
⋮
//read value of n
 ⋮
A = new data_type[n];
```

C++ Dynamic Arrays

- The following program demonstrates the creation of one dimensional C++ dynamic arrays



```cpp
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
    float *A; //At this point, A is simply a pointer variable
    int n;
    cout << "How many elements would you like to store? ";
    cin >> n; //Assume that the user input value for n is positive integer value
    A = new float[n]; //Now, A is a one dimensional C++ dynamic array

    //Populate the elements of the array
    srand(time(0));
    for (int i = 0; i < n; i++)
        A[i] = 1.0*rand()/RAND_MAX; //Each element is in the arange [0.0, 1.0)

    //Print the elements of the array
    for (int i = 0; i < n; i++)
        cout << "Element at index " << i << " is " << A[i] << endl;

    //Compute and print the minimum element of the array
    float m = A[0];
    for (int i = 0; i < n; i++)
        if (A[i] < m)
            m = A[i];
    cout << "The minimum element of A is " << m << endl;

    system("Pause");
    return 0;
}
```

The delete operator

- Recall that the **new** operator reserves memory space from the heap memory
- Therefore the memory spaces allocated for a dynamic array are also allocated on the heap memory
- However, we also know that the heap memory is not automatically cleared by C++ and requires to be cleared by the programmer
- Thus the memory spaces allocated to the dynamic array in the previous example will not be cleared when the program terminates
- ***We say the program has a memory leak***
- A memory leak is very dangerous in that if a program with a memory leak is executed repeatedly; then certainly at some point all the memory spaces in the computer will be used up and will cause a runtime error
- Therefore we must clear the heap memory before our program is terminated

The delete operator

- In order to de-allocate (delete or free) the consecutive memory spaces allocated for dynamic arrays, we use the delete operator as follows

  Syntax

  ```
     delete[] A;
  ```

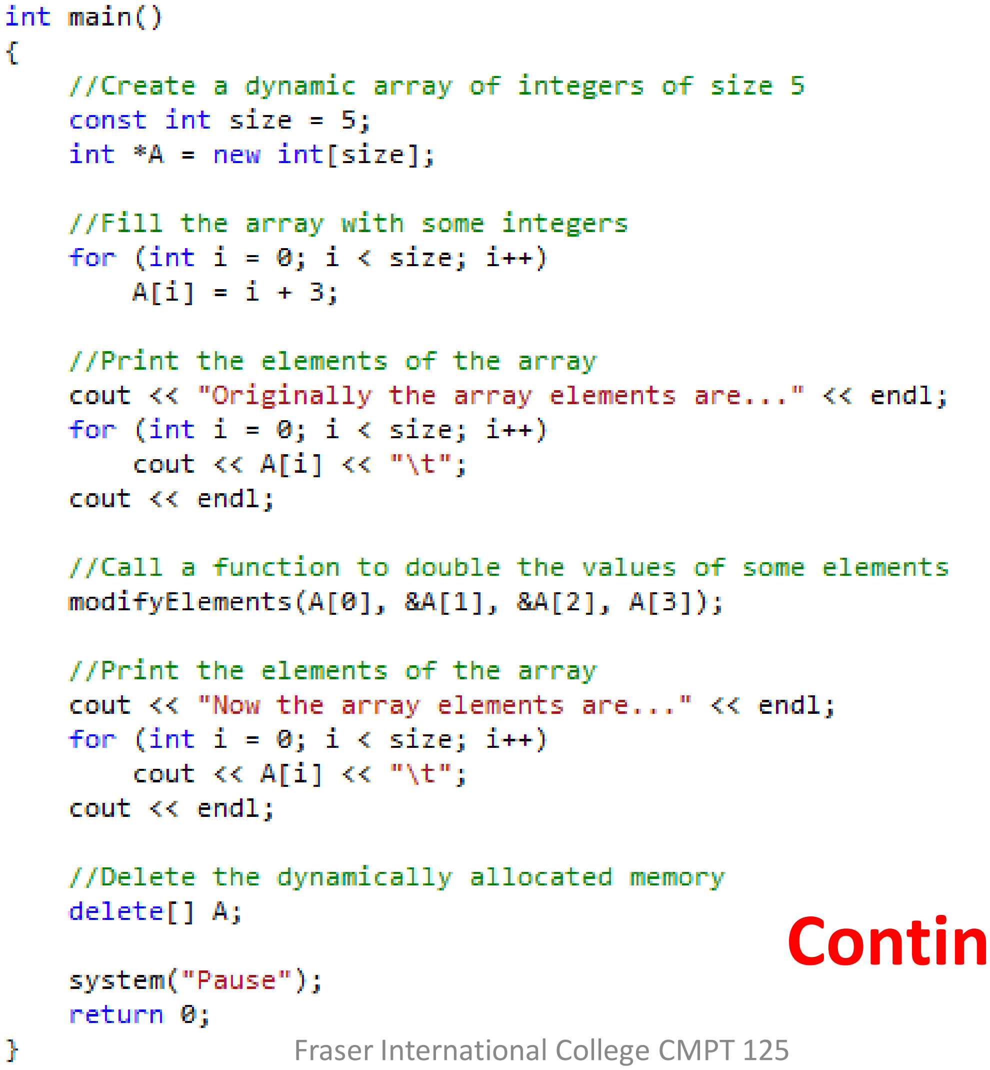
- The square bracket indicates we are deleting consecutive memory spaces that were allocated together
- When the square bracket is not used with the delete operator, then ONLY the first memory memory space will be de-allocated

Passing Element of Dynamic Array to function

- An element of an array (static array or dynamic array) is simply the same as any simple data type variable
- Thus an element of a dynamic array can be passed to a function just like any simple data type variables
- ***Thus parameter passing by value, by pointer or by reference can be used***
- In the case of parameter passing by reference, any modification made to the parameter of the function will also modify the element of the array
- Similarly in the case of parameter passing by pointer, any modification to the parameter of the function (together with de-referencing) will also modify the element of the array

Passing Element of Dynamic Array to function

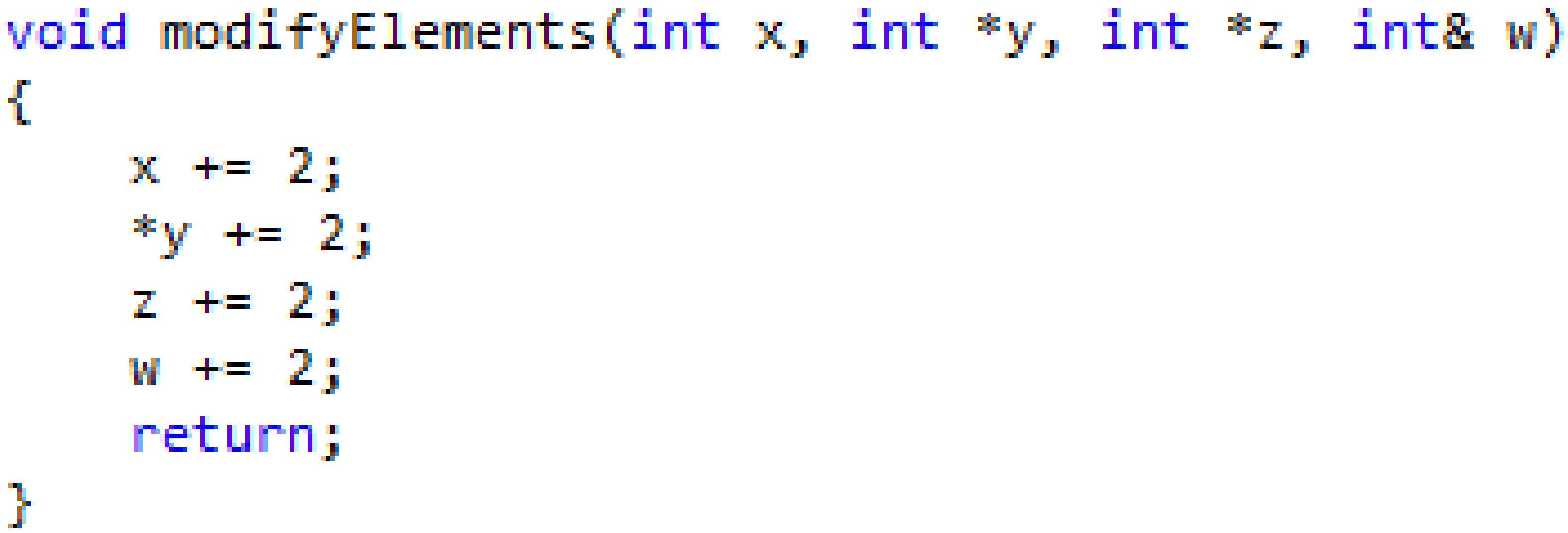
- **Example 2.** Consider the following program and determine its output



```
int main()
{
    //Create a dynamic array of integers of size 5
    const int size = 5;
    int *A = new int[size];

    //Fill the array with some integers
    for (int i = 0; i < size; i++)
        A[i] = i + 3;

    //Print the elements of the array
    cout << "Originally the array elements are..." << endl;
    for (int i = 0; i < size; i++)
        cout << A[i] << "\t";
    cout << endl;

    //Call a function to double the values of some elements
    modifyElements(A[0], &A[1], &A[2], A[3]);

    //Print the elements of the array
    cout << "Now the array elements are..." << endl;
    for (int i = 0; i < size; i++)
        cout << A[i] << "\t";
    cout << endl;

    //Delete the dynamically allocated memory
    delete[] A;

    system("Pause");
    return 0;
}
```

Continues →→→

Passing Element of Dynamic Array to function



```
void modifyElements(int x, int *y, int *z, int& w)
{
    x += 2;
    *y += 2;
    z += 2;
    w += 2;
    return;
}
```

Remark:- De-referencing in the function will affect the element of the array but not without dereferencing

Passing Dynamic Arrays to functions

- Dynamic arrays can also be passed to functions
- Since a dynamic array is effectively a pointer, passing a dynamic array to a function is equivalent to passing a pointer to a function
- In passing a dynamic array to a function, any modification made to an element of the array inside the function will be reflected back to the main program
- Suppose we have a one dimensional dynamic array variable **A** of float of size **size**
- Then we can pass this array to a function that prints the elements of the array as

```
    printArray(A, size);
```

- Now the function needs to be defined to take two arguments: a pointer and an integer

Passing Dynamic Arrays to functions

- Also, it is obvious the function does not return anything, hence it is a void function
- Therefore the function declaration should look like

```
void printArray(const int *p, const int s)
```

- Notice that the size parameter is better made constant for it will not be modified; similarly for the dynamic array as well because the print function does not modify any of the elements of the array

Passing Dynamic Arrays to functions

- Similarly, we can also populate the array inside a function and call the function as follows:

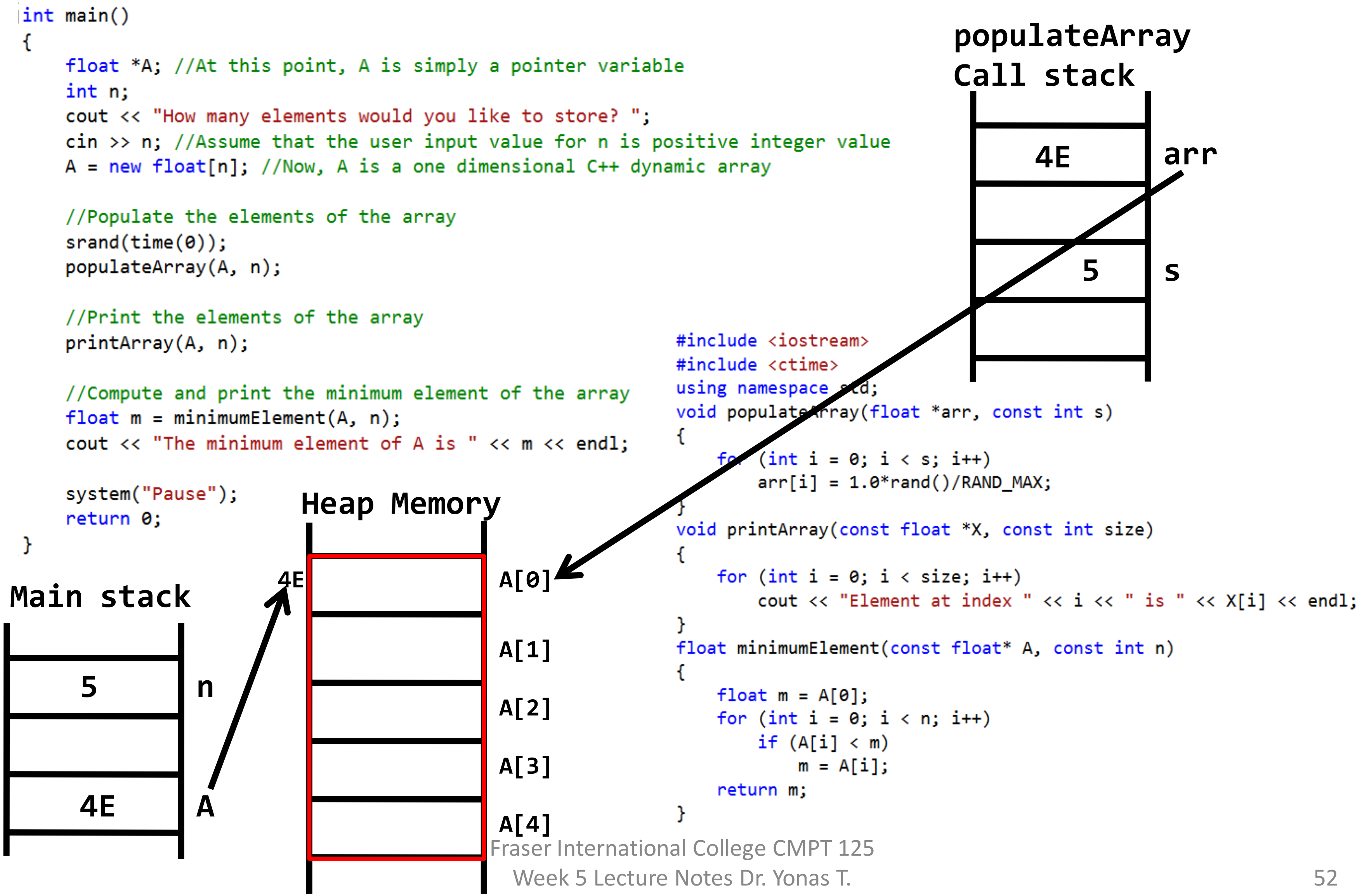
```
populateArray(A, size);
```

- The function declaration will be

```
void populateArray(float *p, const int s)
```

- Observe that the function declaration does not have a **constant** for the array pointer parameter because this function will fill the elements of the array with some values, hence it will modify them
- See the example below…

Passing Dynamic Arrays to functions



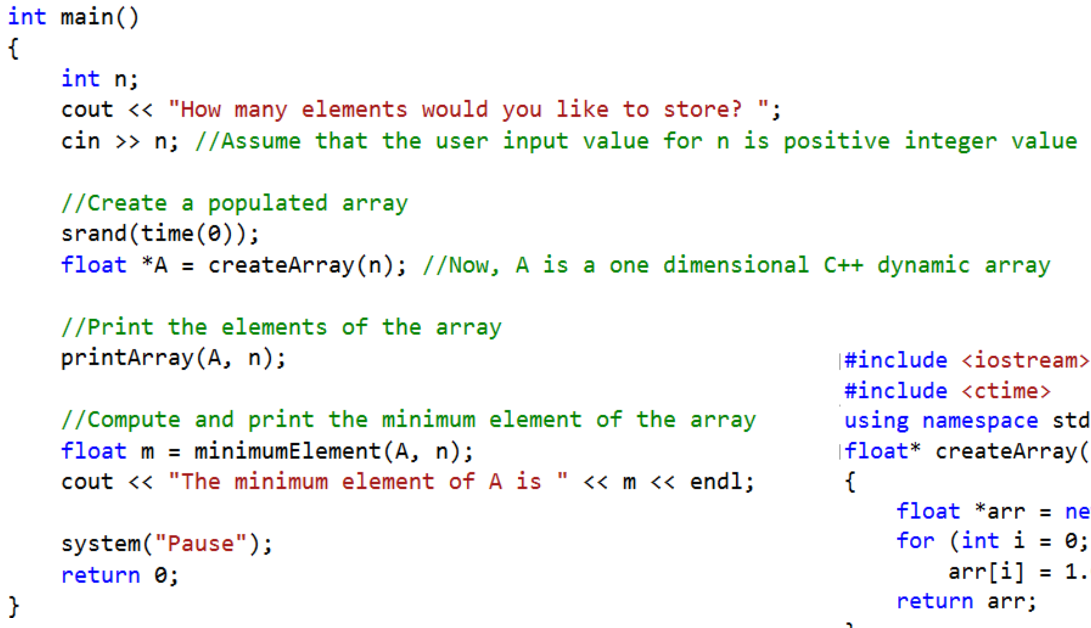
```
int main()
{
    float *A; //At this point, A is simply a pointer variable
    int n;
    cout << "How many elements would you like to store? ";
    cin >> n; //Assume that the user input value for n is positive integer value
    A = new float[n]; //Now, A is a one dimensional C++ dynamic array

    //Populate the elements of the array
    srand(time(0));
    populateArray(A, n);

    //Print the elements of the array
    printArray(A, n);

    //Compute and print the minimum element of the array
    float m = minimumElement(A, n);
    cout << "The minimum element of A is " << m << endl;

    system("Pause");
    return 0;
}
```

```
#include <iostream>
#include <ctime>
using namespace std;
void populateArray(float *arr, const int s)
{
    for (int i = 0; i < s; i++)
        arr[i] = 1.0*rand()/RAND_MAX;
}
void printArray(const float *X, const int size)
{
    for (int i = 0; i < size; i++)
        cout << "Element at index " << i << " is " << X[i] << endl;
}
float minimumElement(const float* A, const int n)
{
    float m = A[0];
    for (int i = 0; i < n; i++)
        if (A[i] < m)
            m = A[i];
    return m;
}
```

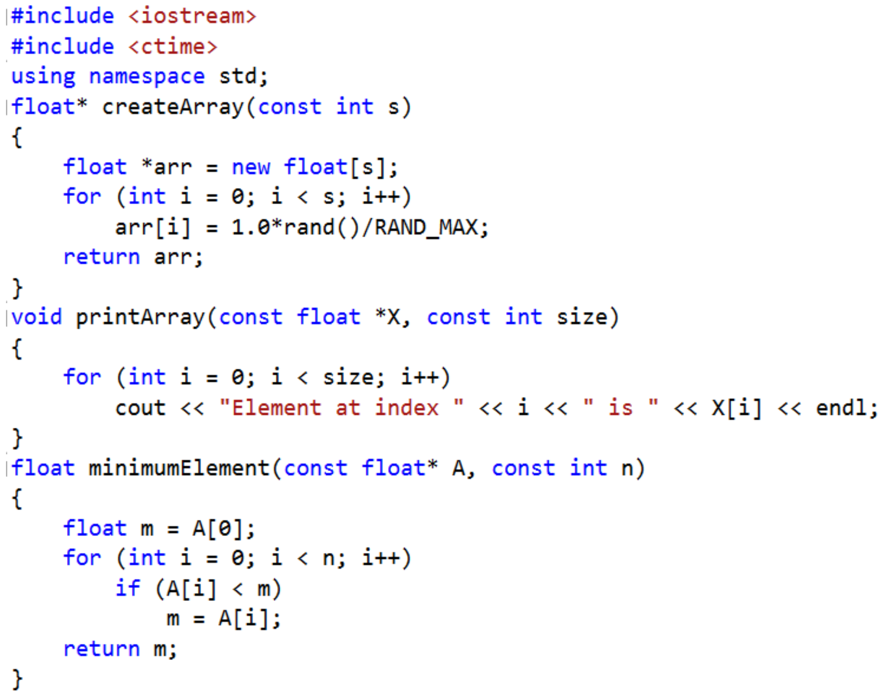
Returning Dynamic Arrays from functions

- **The most important feature of C++ dynamic arrays and that makes them different from static arrays is the fact that we can create them inside functions and return them from the functions!!!**
- **This was not possible with static arrays!!!**
- In order to return a dynamic array from a function
  - ➢Declare the function such that the return type is a pointer
  - ➢Create the dynamic array inside the function
  - ➢Return the pointer dynamic array
- See the example below…

Returning Dynamic Arrays from functions



```
int main()
{
    int n;
    cout << "How many elements would you like to store? ";
    cin >> n; //Assume that the user input value for n is positive integer value

    //Create a populated array
    srand(time(0));
    float *A = createArray(n); //Now, A is a one dimensional C++ dynamic array

    //Print the elements of the array
    printArray(A, n);

    //Compute and print the minimum element of the array
    float m = minimumElement(A, n);
    cout << "The minimum element of A is " << m << endl;

    system("Pause");
    return 0;
}
```



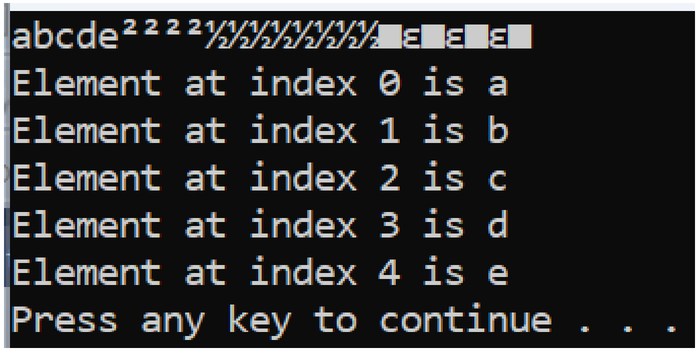
```
#include <iostream>
#include <ctime>
using namespace std;
float* createArray(const int s)
{
    float *arr = new float[s];
    for (int i = 0; i < s; i++)
        arr[i] = 1.0*rand()/RAND_MAX;
    return arr;
}
void printArray(const float *X, const int size)
{
    for (int i = 0; i < size; i++)
        cout << "Element at index " << i << " is " << X[i] << endl;
}
float minimumElement(const float* A, const int n)
{
    float m = A[0];
    for (int i = 0; i < n; i++)
        if (A[i] < m)
            m = A[i];
    return m;
}
```

C++ char data type dynamic arrays and cstrings

- We may also create a dynamic array of char data type as with any other data type, see below

```
#include <iostream>
using namespace std;
int main()
{
    //Create a char data type dynamic array
    char *C = new char[5];

    //Initialize the elements of the array
    for (int i = 0; i < 5; i++)
        C[i] = 'a' + i;

    //Print the array
    cout << C << endl; //This is a run time error. Explain!

    //Print the elements of the array
    for (int i = 0; i < 5; i++)
        cout << "Element at index " << i << " is " << C[i] << endl; //This is OK

    //Delete the dynamic array
    delete[] C;

    system("Pause");
    return 0;
}
```

OUTPUT



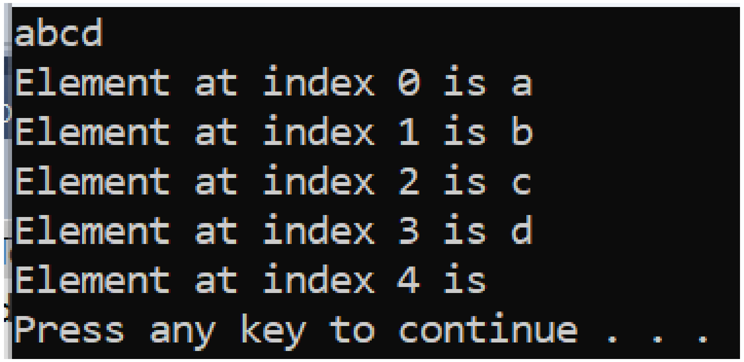
```
abcde²²²²½½½½½½½½█ε█ε█ε█
Element at index 0 is a
Element at index 1 is b
Element at index 2 is c
Element at index 3 is d
Element at index 4 is e
Press any key to continue . . .
```

C++ char data type dynamic arrays and cstrings

- We can also create a cstring using dynamic arrays of char data types as follows

```
#include <iostream>
using namespace std;
int main()
{
    //Create a char data type dynamic array for a cstring
    char *C = new char[5];

    //Initialize the cstring
    for (int i = 0; i < 4; i++)
        C[i] = 'a' + i;
    C[4] = '\0';

    //Print the cstring
    cout << C << endl; //This is OK

    //Print the elements of the cstring
    for (int i = 0; i < 5; i++)
        cout << "Element at index " << i << " is " << C[i] << endl; //This is OK

    //Delete the cstring
    delete[] C;

    system("Pause");
    return 0;
}
```

OUTPUT



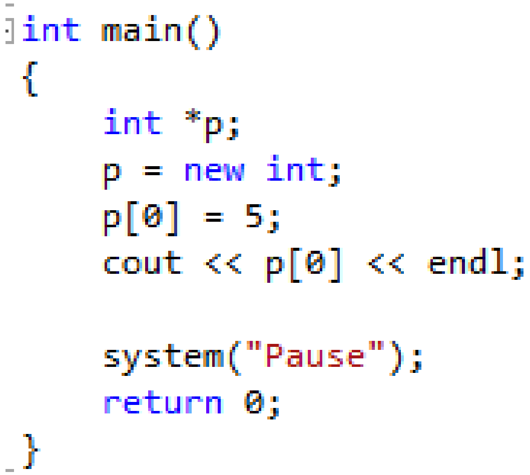
```
abcd
Element at index 0 is a
Element at index 1 is b
Element at index 2 is c
Element at index 3 is d
Element at index 4 is
Press any key to continue . . .
```

Some Remarks
Dynamic Array of Size 1

- When we introduced pointers, we have seen that we can declare a pointer and point it to one memory location on the heap as follows:

  float *p = new float;

- Now, we may also think of the pointer variable **p**, as if it was a dynamic array of size **1** defined as **float *p = new float[1];**
- In fact the following program shows we can initialize and access the heap memory using the pointer variable p together with indexing



```
int main()
{
    int *p;
    p = new int;
    p[0] = 5;
    cout << p[0] << endl;

    system("Pause");
    return 0;
}
```

Some Remarks
Array Parameters

- We have presented the function declarations for our dynamic arrays to use pointer data type as

```
void printArray(const float *A, const int size)
```

- Some people rather like to use the style presented for static arrays

```
void printArray(const float A[ ], const int size)
```

- For all practical purposes, these two are identical!
- Moreover we can use either of these function declarations for static arrays as well as for dynamic arrays!

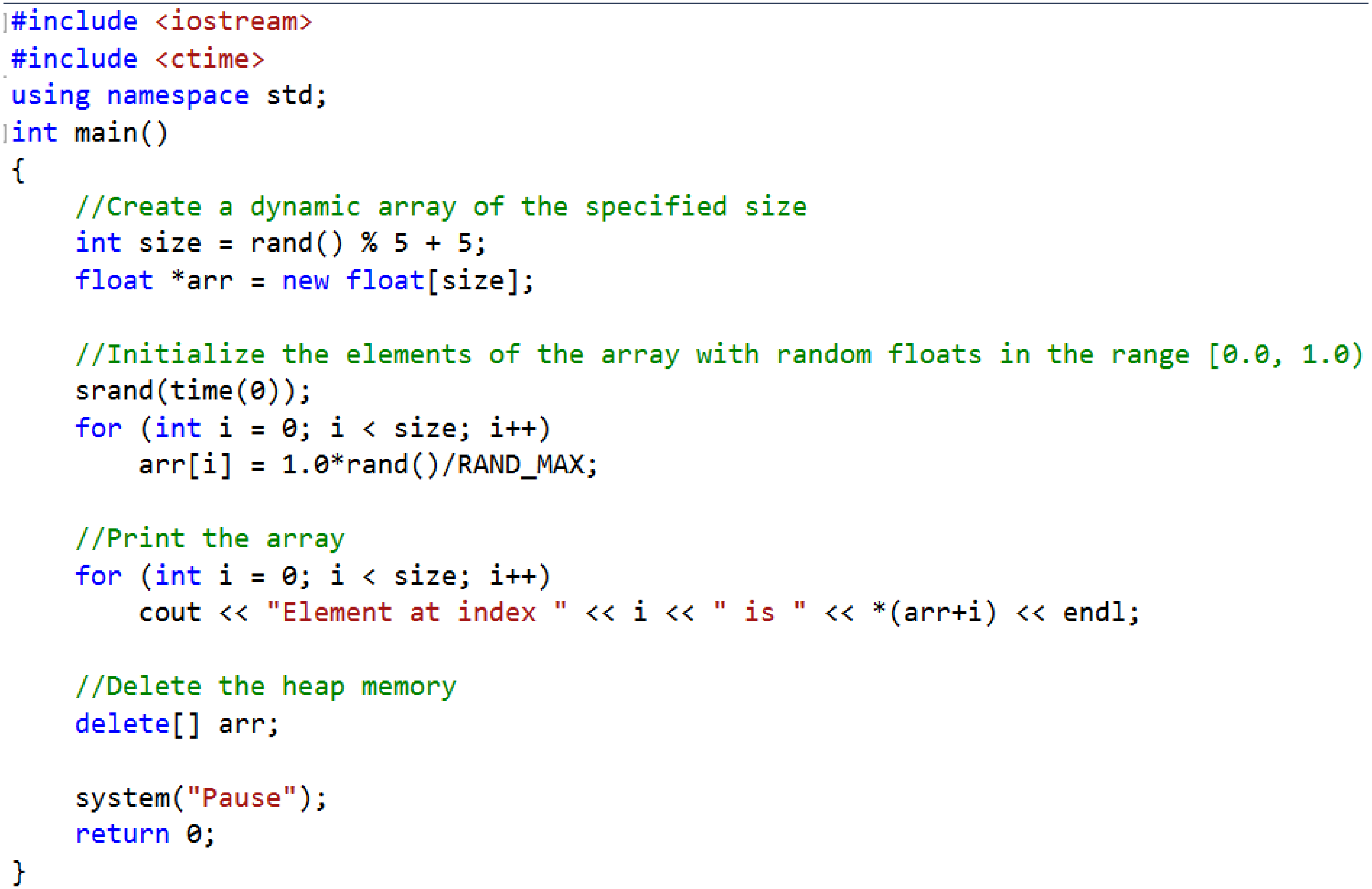
Some Remarks
Pointer Arithmetic

- Consider the statement

```
float *A = new float[5];
```

- Then A is a dynamic array of float data type elements with size 5
- Moreover, the elements of A are accessed with indexing as A[0], A[1], A[2], A[3], and A[4]
- But also we note that A is a pointer pointing to A[0] which means dereference of A (given by *A) is actually A[0]
- Similarly, (A+1) is a pointer pointing to A[1] which means *(A+1) is actually A[1]
- Generally speaking, **`A[i]`** is the same as **`*(A+i)`**
- Thus we can access elements of the dynamic array either as **`A[i]`** or **`*(A+i)`**
- Accessing elements with **`*(A+i)`** is known as pointer arithmetic
- See the example below…

Some Remarks
Pointer Arithmetic



```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
    //Create a dynamic array of the specified size
    int size = rand() % 5 + 5;
    float *arr = new float[size];

    //Initialize the elements of the array with random floats in the range [0.0, 1.0)
    srand(time(0));
    for (int i = 0; i < size; i++)
        arr[i] = 1.0*rand()/RAND_MAX;

    //Print the array
    for (int i = 0; i < size; i++)
        cout << "Element at index " << i << " is " << *(arr+i) << endl;

    //Delete the heap memory
    delete[] arr;

    system("Pause");
    return 0;
}
```