C++ Programming

03- References and Pointers,
Arrays and Strings



Pointers

- Pointers keep track of where data and functions are stored in memory, allowing powerful memory management.
- They allow the creation and manipulation of dynamic data structures.
- A pointer with value 0 points to nothing (and is said to be a null pointer).
- It is not necessary to initialise pointers when they are declared.
- Hence we can use pointers to objects in a class, specially if they are self-referenced.



Pointers

- Uninitialized pointers are potentially dangerous.
- They could point to any location in memory and can lead to program crashes or totally unexpected behaviour.
- When writing our class we hence need to make sure that the pointers are always initialised inside the constructor(s).
- Inaccurate use of operations on pointers may also be dangerous.



Pointers

- A pointer is declared by preceding the variable or class member name with the * character.
- To access the data item that a pointer p points to we use the expression *p.
- A pointer may be initialised to hold the memory address of an existing data item using the & (addressof) operator, e.g.

```
int x = 7;
int *p = &x;
```

 The above code would make the pointer p point to the data item associated with the variable x.



Address Arithmetic

- If **p** is a pointer and **n** is an integer **p+n** denotes the memory address obtained by adding **n** times the size of the data item to which **p** refers to the address stored in **p**.
- Hence p+1 will point to the memory address immediately following the last byte of the item to which p points.
- If the pointer had not been initialised correctly this may not be the case.
- We can move pointers step-by-step through blocks of memory using statements such as p++.



Variables and References

- References are used to access a block of memory that holds an object or a variable.
- Suppose s1 and s2 are variables whose type is a class called Student.
- The assignment "s2 = s1;" makes a copy of s1 and store it in the memory block associated with s2.
- We need to make s2 hold a reference to s1.



Variables and References

- To create a reference variable we precede its name with & in the declaration.
- Student &s2 = s1;
- Note that when declaring a reference variable we must initialise such as

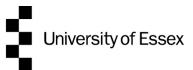
Student &s2;

with no initialisation not allowed.



Pointers – an Example and Exercise

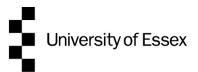
```
#include <iostream>
   using namespace std;
                                       What would be
                                        the outputs?
   int main()
 6
        int a:
        int *aPtr; // a is an int
        a = 7; // assign 7 to a
        aPtr = \&a; // assign the address of a to aPtr
10
        cout << "The address of a is " << &a << endl;</pre>
11
12
        cout << "The value of aPtr is " << aPtr << endl;</pre>
        cout << "The value of a is " << a << endl;</pre>
13
14
        cout << "The value of *aPtr is " << *aPtr << endl;</pre>
15
        cout << "&*aPtr = " << &*aPtr << endl;
16
        cout << "*&a = " << *&a << endl;
17 }
```



References and Arguments

- When passing arguments to functions in C++, copies of the values supplied in the function calls are used unless the argument is declared to be a reference.
- We can write a function to swap the values held in two variables by passing them as reference arguments:

```
1  void swap(int &x, int &y)
2  {
3     int temp = y;
4     y = x;
5     x = temp;
6  }
```



References and Arguments

- Suppose we write a function to display the marks of a student, that may be used by other programmers in programs that use our **Student** class.
- If we declare the function in our header file as

void displayMarks(Student &s);

- the user cannot be sure that our function will not modify the contents of the object he supplies as an argument.
- We should instead declare it as a constant reference argument:

void displayMarks(const Student &s);



Arrays and Arguments

- When passing an array as an argument to a method the actual value passed is the memory address of the start of the array.
- Note that in C++ there is no way to obtain the length of an array.
- We must supply the length of the array as an extra argument.



Returning Objects From Functions

- A **return** statement will always return a copy of the value to be returned unless the return type of the function is declared to be a reference.
- If we wanted to write a function to find and return the student with the highest mark from an array of objects of type **Student** and want to avoid the need to make a copy we should use a declaration of the form

Student &getBest(Student a[], int size)



Returning Objects From Functions

- Care must be taken to avoid returning references to local variables as function results.
- Suppose we wanted to write a function to read from a file a line of input that contains the details of a student and return a **Student** object containing those details we may attempt to use something of the form

```
1 Student &getStudent()
2 {
3    Student s;
4    // read student's attributes and
5    // store them in s
6    return s;
7 }
```



Returning Objects From Functions

- The code on the previous slide will not work correctly.
- The lifetime of the variable s expires when the call to the function terminates and the stack memory that was used to store the variable will get re-used when another function is called so we are returning a reference to an object whose value will not persist.
- Hence we should **not** return a reference in such circumstances but instead use

```
1 Student getStudent()
2 {
3    Student s;
4    // read student's attributes and
5    // store them in s
6    return s;
7 }
```



Self-Referential Classes

- Suppose we wished to write a **Person** class for use in a family tree.
- The parents are themselves objects of type **Person** so the members of the class include other objects.
- We cannot write

```
1 class Person // incomplete - needs functions
2 {
3     private:
4     string name; // etc
5     Person mother, father;
6 };
```

 since the mother and father members would be objects of type Person and it would be necessary to allocate memory for other objects of type Person inside the memory allocated for each object of that type.



Self-Referential Classes

• Furthermore, we cannot use

```
1 class Person
2 {
3     private:
4     string name; // etc
5     Person &mother, &father;
6 };
```

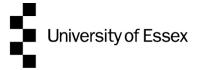
- since references have to be initialised when they are declared, so we would need to have existing **Person** objects before we can initialise the first such object in a program.
- Hence when using self-referential classes (classes whose attributes include other objects of the same class) we need to use a different approach, using *pointers*.



Pointers inside a class

• Hence our **Person** class should be declared as

```
1 class Person
2 {
3    private:
4    string name; // etc
5    Person *mother, *father;
6 };
```



Pointers to Objects

- We often need to access members of objects that are pointed to by pointers.
- If **fred** was an object of type **Person** and we wished to print Fred's mother's name we could use

cout << (*fred.mother).name;</pre>

- We need the parentheses around *fred.mother; without these the expression would mean the object that fred.mother.name pointed to.
- To print the name of Fred's father's mother we would have to use cout << (*(*fred.father).mother).name;
- The parentheses are cumbersome and make the code hard to read.



Pointers to Objects

- To avoid the syntactic complications caused by the need to use parentheses when accessing members of objects pointed to by pointers C++ has an extra operator, ->.
- The expression p->q is equivalent to (*p).q;

 Using this operator we can print the name of Fred's father's mother using

cout << fred.father->mother->name;



Pointers as Arguments

- It is possible to use pointers instead of references when passing arguments to functions.
- We could write an alternative version of our function "swap" to swap the values of two variables

```
1  void swap2(int *x, int *y)
2  {
3     int temp = *y;
4     *y = *x;
5     *x = temp;
6  }
```

 The necessary calls to swap the values of a and b will be swap(a, b) but swap2(&a, &b).



Arrays

- An array variable is stored as a constant pointer that points to the first element of the array.
- Hence it is possible to use pointers instead of subscripts to access the elements of an array;
- a[n] is actually a shorthand notation for *(a+n).
- In a declaration the size of an array must be specified
- e.g. int a[5]; or int a[] = {3,7,4,9,6};



Arrays

 Here are some examples of declaration and initialisation of arrays.

```
int b[5]; // contents unspecified
int c[5] = {1,2,3,4,5}; // c[0]=1, c[1]=2, etc
int d[] = {1,2,3,4,5}; // will have size 5
int e[5] = {0}; // all elements are 0
int f[5] = {1}; // last four elements are 0
int g[5] = {1,2,3}; // last two elements are 0
int h[5] = {1,2,3,4,5,6}; // error
```



Array

 No range-checking is performed when accessing array elements, so the following code would compile and run.

```
int a[] = {1,2,3,4,5};
cout << a[20];
```

- The value output would be the contents of a memory address beyond the end of the array.
- The impact of a[20] = 42; is unpredictable.
- There is a class called vector in the STL which provides arrays with range checking.

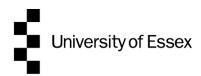


Multi-dimensional Arrays

We can declare multi-dimensional arrays. e.g.

```
int h[2][3] = \{\{1,2,3\},\{2,4,6\}\};
```

- When declaring multi-dimensional arrays we must specify the sizes of all but the first dimension,
- for example a declaration of the form int j[][4] = ...;
 would be permitted,
- but int k[4][] = ...; and int m[][] = ...; would not be.



Strings and Character Arrays

- Although C++ has a string class, a string constant such as "Hello, world" is not an object of type string; It is a character array.
- It is essential that the length of a string can be obtained;
- otherwise all functions that take strings as arguments would need an extra length argument.
- Hence a string is always terminated with a special character '\0'
- And the array must be at least one character longer than the string.



Strings and Character Pointers

- Many classes have string attributes;
- If we choose to use a character array for the name in our **Person** class we must either specify the size of the array in which the name will be stored or instead use a character pointer.
- Additionally when setting the name we would have to copy the name into this space;
- we cannot write something like fred.name = "Fred"; since fred.name is a constant pointer.



Strings and Character Pointers

- Instead of using char name[50]; we may use char
 *name as a member of the Person class.
- It is now possible to set the name using something like **fred.name = "Fred"**; .
- The name member will be made to point to a string that has already been created so no copying is necessary.



Accessing Command-Line Arguments

 In week 2 we saw that the main function in a program may have arguments:

int main(int argc, char *argv[]) { ... }

- The first argument specifies the number of words on the command line used to run the program
- The second is an array of pointers to the beginning of strings containing these words.



Accessing Command-Line Arguments

• If a program is run using a command:

myprog Mike Wazowski;

- the argc argument to the main function will have the value 3, and the three elements of argv will point to the beginning of the strings "myprog", "Mike" and " Wazowski".
- argv[0] holds the name used to invoke the program so the command-line arguments start in argv[1] and argc will always be at least 1.



Accessing Command-Line Arguments

 The following program is a personalised version of "Hello World" where the name of the person to be greeted may be supplied as the command-line arguments:

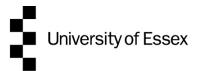
```
#include <iostream>
   using namespace std;
    main(int argc, char *argv[])
        cout << "Hello,";</pre>
        if (argc==1)
             cout << " world";</pre>
        else
           for (int i = 1; i<argc; i++)
11
             cout << ' ' << argv[i];</pre>
12
13
        cout << endl;</pre>
14
```



Pointers to Functions

• In C++ we can pass a pointer to a function as an argument.

```
1  void applyAll(int (*fun)(int), int arr[], int len)
2  {
3     for (int i = 0; i<len; i++)
4          cout << i << ':' << (*fun)(arr[i]) << endl;
5  }</pre>
```

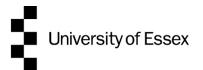


Pointers to Functions

 When a function-name is used in an expression without any parentheses it denotes a pointer to the function so we can call our function using statements of the form

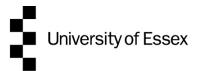
applyAll(myFun, myArr, 10);

Note that no & character is used.



Dynamic Memory Allocation

- It is often the case that the programmer does not know how large an array or string will need to be.
- Since all data items on the stack must have a fixed size we need to use the heap when creating data items whose size is not known until run-time.
- Additionally we have seen that functions should not return references to data items stored in local variables;
- The same applies to pointers.



Dynamic Memory Allocation

- Dynamic memory allocation is performed using the new operator,
- In C++ we may create any data item, not just a class object.
- Furthermore the result of applying the operator is a pointer to the object, e.g.

```
int *anArray = new int[10]; // 10 is size
int *ip = new int(10); //10 is initial value
```

 C++ has no garbage collector so when a data item on the heap is no longer needed the memory space it occupied should be released using the delete operator, e.g.

```
delete [] anArray; // need [] for arrays
delete ip;
```





CONTACT YOUR LECTURER

m.barros@essex.ac.uk

