CE221
Programming in C++
Part 2
References and Pointers,
Arrays and Strings
Variables and References 1

In Java a variable of primitive type is associated with a memory location that holds the value of the variable, but a variable of class type is associated with a location that holds a reference to an object of that type.

However, in C++ (unless it is explicitly declared to be a reference) a variable of class type is associated with a block of memory that holds an object.

This difference has a significant impact on many aspects of programming.
Suppose \( s_1 \) and \( s_2 \) are variables whose type is a class called \texttt{Student}. In Java the statement

\[
\text{s2} = \text{s1};
\]

would make \( s_2 \) refer to the same object as \( s_1 \).

However in C++ such an assignment would make a copy of \( s_1 \) and store it in the memory block associated with \( s_2 \).

Suppose our \texttt{Student} class has methods/functions called \texttt{addMark} to add a mark for a student and \texttt{updateAverage} to update an average-mark attribute for the student and \( a \) is an array of 10 students.
Variables and References 3

If we wish to add marks and update the average for each student in the array, in Java we might use a loop of the form

```java
for (i = 0; i<10; i++)
    { Student s = a[i];
      s.addMark(...);
      s.updateAverage();
    }
```

In C++ this would not work since in each iteration of the loop the variable `s` will hold a copy of one of the objects in the array, and only this copy will be updated. Furthermore, the lifetime of this copy will expire at the end of the iteration.

We need to make `s` hold a reference to the array element.
To create a reference variable we precede its name with & in the declaration. Hence for a correctly-working C++ version of the loop on the previous slide we would need to use

```cpp
Student &s = a[i];
```

No other changes are necessary – reference variables are used in the same way as any other variables within expressions.

Note that when declaring a reference variable we must initialise it to refer to some existing data item or object; a declaration such as

```cpp
Student &s;
```

with no initialisation is not allowed.
References and Arguments 1

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:

```c++
void swap(int &x, int &y)
{
    int temp = y;
    y = x;
    x = temp
}
```

(Note that there is no way to do this in Java since primitive-value arguments are never references.)
When passing an object as an argument to a function that does not change the contents of that object a reference argument is not necessary; however using a non-reference argument means that an unnecessary copy of the object is made and this could have an impact on performance, particularly if the object is large.

Hence when passing objects as arguments references are almost invariably used.
Suppose we write a function to display the marks of a student, that may be used by other programmers in programs that use our \texttt{Student} class.

If we declare the function in our header file as

\begin{verbatim}
  void displayMarks(Student &s);
\end{verbatim}

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:

\begin{verbatim}
  void displayMarks(const Student &s);
\end{verbatim}

The compiler will then check that the function does not modify the contents of \texttt{s} and the user can have confidence in our function.
Arrays As Arguments

We shall discuss arrays in C++ in more detail later, but it is important to note that when passing an array as an argument to a method the actual value passed is the memory address of the start of the array. Hence copies are never made when passing arrays so we do not need to use reference arguments.

Note that in C++ there is no way to obtain the length of an array – arrays are not objects so they cannot have a `length` attribute. Hence unless the length of an array supplied as an argument is known at compile-time or can be detected by the use of a special terminator value as the last element we must supply the length of the array as an extra argument.
Returning Objects From Functions 1

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

```c
Student &getBest(Student a[], int size)
```
Returning Objects From Functions 2

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

```cpp
Student &getStudent()
{
    Student s;
    // read student's attributes and
    // store them in s
    return s;
}
```
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

```c
Student getStudent()
{
    Student s;
    // read student's attributes and
    // store them in s
    return s;
}
```
Suppose we wished to write a `Person` class for use in a family tree. Each person has attributes such as name, date and place of birth, and also a mother and father. The parents are themselves objects of type `Person` so the members of the class include other objects.

We cannot write

```cpp
class Person // incomplete - needs functions
{
    private:
        string name; // etc
        Person mother, father;
};
```

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 – this is impossible.
Self-Referential Classes 2

Furthermore, we cannot use

```cpp
class Person
{
 private:
    string name; // etc
    Person &mother, &father;
};
```
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 1

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). 0 is the only integer that can be used to initialise a pointer. (Note that a reference can never be null.)

It is not necessary to initialise pointers when they are declared. Hence we can use pointers to `Person` objects for the mother and father members of our `Person` class. We can use a null pointer when the parent of someone in the family tree is unknown.
Pointers 2

Uninitialised pointers are potentially dangerous. They could point to any location in memory and attempts to manipulate the contents of the data item to which the pointer points could result in arbitrary memory contents being overwritten leading 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. For example it is possible to perform arithmetic on pointers to step through the values in an array; when doing so we must make sure that we do not go beyond the end of the array.
A pointer is declared by preceding the variable or class member name with the * character. Hence our `Person` class should be declared as

```cpp
class Person
{
    string name; // etc
    Person *mother, *father;
};
```

To access the data item that a pointer `p` points to we use the expression `*p`. In this context the operation performed by the * operator is known as **dereferencing**. (As a unary operator * denotes dereferencing but as a binary operator it denotes multiplication.)
A pointer may be initialised to hold the memory address of an existing data item using the `&` (address-of) operator, e.g.

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

The above code would make the pointer `p` point to the data item associated with the variable `x`. (We must be careful to ensure that pointers that will exist beyond the lifetime of a local variable do not point to that variable at the end of its lifetime.)
Pointers – an Example and Exercise

#include <iostream>
using namespace std;
int main()
{
    int a;     // a is an int
    int *aPtr; // aPtr is an int*
        // (pointer to an int)
    a = 7;     // assign 7 to a
    aPtr = &a; // assign the address of a to aPtr
    cout << "The address of a is " << &a << endl;
    cout << "The value of aPtr is " << aPtr << endl;
    cout << "The value of a is " << a << endl;
    cout << "The value of *aPtr is " << *aPtr << endl;
    cout << "&*aPtr = " << *&aPtr << endl;
    cout << "*&a = " << *&a << endl;
}
Pointers to Objects 1

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

\begin{verbatim}
cout << (*fred.mother).name;
\end{verbatim}

Since the \texttt{*} operator has a lower precedence\(^1\) than \texttt{.}, we need the parentheses around \texttt{*fred.mother}; without these the expression would mean the object that \texttt{fred.mother.name} pointed to. To print the name of Fred's father's mother we would have to use

\begin{verbatim}
cout << (**(fred.father).mother).name;
\end{verbatim}

The parentheses are cumbersome and make the code hard to read.

1 We shall see the operator precedence table in part 3.
Pointers to Objects 2

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 \((*p).q\); i.e. it denotes the \( q \) member of the object to which \( p \) points. Using this operator we can print the name of Fred's father's mother using

```cpp
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 from slide 6 to swap the values of two variables:

```c
void swap2(int *x, int *y)
{
    int temp = *y;
    *y = *x;
    *x = temp
}
```

Note the difference between the bodies of the two versions – references are dereferenced implicitly but pointers must be explicitly dereferenced.

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

Note that when calculating the size of the data item the compiler assumes that the pointer points to an object of the type specified in its declaration; 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 $p++$. 
References v Pointers – Summary

References and pointers are implemented in a very similar way – pointers are always stored as memory addresses and references are usually stored as memory addresses.

However, there are significant differences in their usage. A reference must be initialised and null references are not allowed; furthermore, once established, a reference cannot be made to refer to a different object.

References are implicitly dereferenced, whereas we must explicitly dereference pointers using the * operator.
Arrays 1

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; this can be done explicitly, e.g. `int a[5];` or by initialisation of all of the members, e.g. `int a[] = {3,7,4,9,6};`. In the first declaration no initialisation is performed; the initial values in the array will be the values that were already in the memory locations it uses.
Here are some examples of declaration and initialisation of arrays.

```c
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
```

Observe that if a size is specified and some but not all of the elements are initialised the rest will be initialised to 0 (or the equivalent value of the appropriate type). An attempt to initialise too many elements will result in a compilation error.
Arrays 3

No range-checking is performed when accessing array elements, so the following code would compile and run (assuming the value obtained by calculating $a+20$ is a valid memory address).

```cpp
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 – another variable will probably get overwritten, leading to the program behaving incorrectly at some later stage.

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.

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

The above declaration introduces an array containing two elements, each of which are arrays containing 3 integers.

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. This style of representation of strings is inherited from C so a string stored in a character array is often referred to as a C-string. [ Until we describe the `string` class later, any occurrences of the word string in the standard font in the slides will refer to C-strings]

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 1

Many classes have string attributes; if we choose to use a character array for the name in our `Person` class to avoid the overheads associated with the `string` class we must either specify the size of the array in which the name will be stored or instead use a character pointer.

Using the first approach (e.g. `char name[50];`) the space for the name is allocated within the `Person` object and if a person has a short name much of that space would be unused. 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.
Instead of using `char name[50];` we may use `char *name` as a member of the `Person` class. The space for the string will then not be allocated when the object is created; instead the object will just contain a pointer to the beginning of a string that will be elsewhere in memory.

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. The space overhead of the four extra bytes for the pointer is likely to be more than compensated for by the fact that the lengths of the names of most people will be much less than the arbitrary maximum size (50) used in the first approach.
Accessing Command-Line Arguments 1

In part 1 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 and the second is an array of pointers to the beginning of strings containing these words. (How do we know that this is an array of pointers rather than a pointer to an array? The precedence rules state `*argv[1]` means `*(argv[1])` so `argv` must be an array and `argv[1]` must be a pointer.)
Accessing Command-Line Arguments 2

If a program is run using a command:

```
myprog Mike Sanderson
```

the `argv` 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 "Sanderson".

Note that `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.

A program might need to know the name that was used to invoke it since it may be invoked through a shortcut to the program file; this feature is widely used in the UNIX/Linux operating systems where several commands that behave similarly are implemented as links (shortcuts) to the same file.
Accessing Command-Line Arguments 3

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:

```cpp
#include <iostream>
using namespace std;
main(int argc, char *argv[])
{
    cout << "Hello,";
    if (argc==1) cout << " world";
    else
        for (int i = 1; i<argc; i++)
            cout << ' ' << argv[i];
    cout << endl;
}
```
Pointers to Functions 1

In CE151 we saw that Python has higher-order functions; that is a function can be passed as an argument to another function. Java does not directly support this feature although similar behaviour can be obtained by supplying a class that implements an interface as an argument.

In C++ we can pass a pointer to a function as an argument. The function on the following slide will apply the function supplied as the first argument to each of the elements of the array supplied as the second argument in turn outputting the returned values; the third argument is the length of the array.
Note the syntax of the function-pointer argument – parentheses are needed in the same place as they are needed when the argument is used.

```c
void applyAll(int (*fun)(int), int arr[], int len)
{ for (int i = 0; i<len; i++)
    cout << i << ':' << (*fun)(arr[i]) << endl;
}
```

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

```c
applyAll(myFun, myArr, 10);
```

Note that no `&` character is used.
Dynamic Memory Allocation 1

It is often the case that the programmer does not know how large an array or string will need to be – the information to determine this may not be available until the program is run. 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. (Some compilers now allow the creation of an array with a variable as the size.)

Additionally we have seen that functions should not return references to data items stored in local variables since the variables will no longer exist after returning from the function; the same applies to pointers, so if a function creates an object that is to be returned by reference it should use the heap.
Dynamic Memory Allocation 2

As in Java dynamic memory allocation is performed using the `new` operator, but the usage of this operator is significantly different to Java. 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.

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

Unlike Java, 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.

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