

C++ Programming

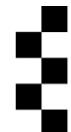
03- References and Pointers,
Arrays and Strings



University of Essex

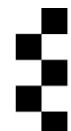
Pointers

- 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.

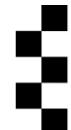


Pointers

- 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 **&** (address-of) operator, e.g.

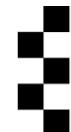
```
int x = 7;  
int *p = &x;
```

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



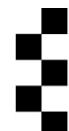
Address Arithmetic

- 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

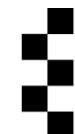
- To create a reference variable we precede its name with **&** in the declaration.
- **Student &s2 = s1;**
- Note that when declaring a reference variable we must initialise such as
Student &s2;
- with no initialisation not allowed.



Pointers – an Example and Exercise

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int a;
8     int *aPtr;      // a is an int
9     a = 7;         // assign 7 to a
10    aPtr = &a; // assign the address of a to aPtr
11    cout << "The address of a is " << &a << endl;
12    cout << "The value of aPtr is " << aPtr << endl;
13    cout << "The value of a is " << a << endl;
14    cout << "The value of *aPtr is " << *aPtr << endl;
15    cout << "&*aPtr = " << &*aPtr << endl;
16    cout << "*&a = " << *&a << endl;
17 }
```

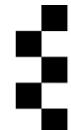
What would be
the outputs?



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.

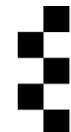
- If we declare the function in our header file as

```
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.

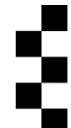
- We should instead declare it as a constant reference argument:

```
void displayMarks(const Student &s);
```



Arrays and Arguments

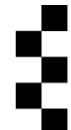
- When passing an array as an argument to a method the actual value passed is the memory address of the start of the array.
- Note that in C++ there is no way to obtain the length of an array.
- We must supply the length of the array as an extra argument.



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

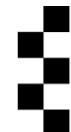
Student &getBest(Student a[], int size)



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

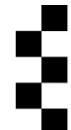
```
1 Student &getStudent()
2 {
3     Student s;
4     // read student's attributes and
5     // store them in s
6     return s;
7 }
```



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 }
```

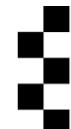


Self-Referential Classes

- 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.

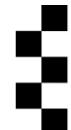


Self-Referential Classes

- 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

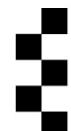
- Hence our **Person** class should be declared as

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



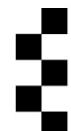
Pointers to Objects

- 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;
- 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

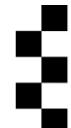
```
1 void swap2(int **x, int *y)
2 {
3     int temp = *y;
4     *y = *x;
5     *x = temp;
6 }
```

- 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

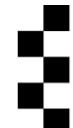
int d[] = {1,2,3,4,5}; // will have size 5

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

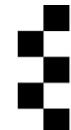


Array

- 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.

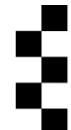


Multi-dimensional Arrays

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

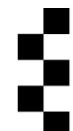
```
int h[2][3] = {{1,2,3},{2,4,6}};
```

- 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.



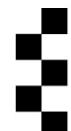
Strings and Character Arrays

- 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.

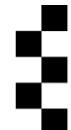


Accessing Command-Line Arguments

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

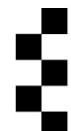
```
int main(int argc, char *argv[]) { ... }
```

- 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

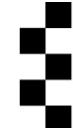
- If a program is run using a command:
myprog Mike Wazowski;
- the **argc** argument to the main function will have the value 3, and the three elements of **argv** will point to the beginning of the strings "**myprog**", "**Mike**" and "**Wazowski**".
- **argv[0]** holds the name used to invoke the program 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:

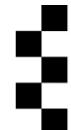
```
1 #include <iostream>
2
3 using namespace std;
4
5 main(int argc, char *argv[])
6 {
7     cout << "Hello,";
8     if (argc==1)
9         cout << " world";
10    else
11        for (int i = 1; i<argc; i++)
12            cout << ' ' << argv[i];
13    cout << endl;
14 }
```



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 }
```

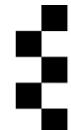


Pointers to Functions

- 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

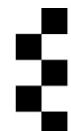
applyAll(myFun, myArr, 10);

- 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.

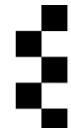
```
int *anArray = new int[10]; // 10 is size
```

```
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.

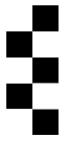
```
delete [] anArray; // need [] for arrays
```

```
delete ip;
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk

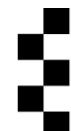


C++ Programming

04 - Control Structures,
Operators, Classes and Objects

Control Structures

- The control structures in C++ are essentially the same as in Java.
- Selection structures: **if** (with optional **else**) and **switch**
- Repetition structures: **while**, **do** and **for**
- Exception handling: **try/catch**
- However the declaration of user-defined exception classes differs from Java. (Exception-handling in C++ will be covered later.)



Control Structures

- The **break** and **continue** statements can be used to break out of control structures.
- The **break** statement causes the program to jump out of the nearest enclosing loop or **switch** statement
- The **continue** statement (which is used only in loops) causes the program to jump to the end of the current iteration
- It proceeds by checking the condition for continuing in the loop



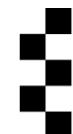
break and continue – an Example

```
for (init; condition; update) {  
    // code  
    if (condition to break) {  
        break;  
    }  
    // code  
}
```

```
for (init; condition; update) {  
    // code  
    if (condition to break) {  
        continue;  
    }  
    // code  
}
```

```
while (condition) {  
    // code  
    if (condition to break) {  
        break;  
    }  
    // code  
}
```

```
while (condition) {  
    // code  
    if (condition to break) {  
        continue;  
    }  
    // code  
}
```



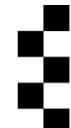
Default Argument To Functions

- A function can have arguments with default values, e.g.

```
int volume(int length = 1, int width = 1,  
int height = 1) { ..... }
```

- **volume(3,4)** would be treated as **volume(3,4,1)**. (The supplied values are used for the first arguments.)
- If some but not all of the arguments of a function have default values they must appear at the end of the list, e.g.

```
void printArea(float length, float width, int  
decimalPlaces = 2) { ..... }
```



Function Overloading

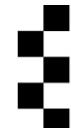
- It is permissible to write more than one function with the same name as long as they have different numbers of arguments or different argument types.
- The compiler will determine which version to use for each call by looking at the arguments.
- The return types of the overloaded functions may be the same or may be different, e.g.

int square(int x);

float square(float x);

float area(float height, float width);

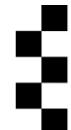
float area(float radius);



More about Functions

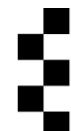
- In C++, it is not possible to declare a function inside the body of another function.
- Short simple functions are sometimes declared as **inline**, e.g.

```
inline int cube(int i) { return i*i*i; }
```



Operators and Precedence

- C++ has many operators.
- It has all of the operators of Java apart from **instanceof**.
- In addition there are the operators `->`, `::` and **delete** and the unary versions of `*` and `&` that we have already encountered, and there are `.*`, `->*`, several cast operators and **sizeof**.
- The operator precedence table is shown on the following slides, with the highest precedence first, so for example `*p.q` means `*(p.q)` .



Operator Precedence Table 1

Operators	Function	Associativity
::	binary scope resolution	left to right
::	unary scope resolution	
()	function call	left to right
[]	Array subscript	
.	Member selection via object	
->	Member selection via pointer	
++	Unary postfix increment	
--	Unary postfix decrement	
<i>typeid</i>	Runtime type information	
<i>dynamic_cast<type></i>	Runtime type-checked cast	
<i>static_cast<type></i>	Compile-time type-checked cast	
<i>reinterpret_cast<type></i>	Cast for nonstandard conversions	
<i>const_cast<type></i>	Cast away const-ness	



Operator Precedence Table 2

Operators	Function	Associativity
<code>++</code>	Unary prefix increment	Right to left
<code>--</code>	Unary prefix decrement	
<code>+</code>	Unary plus	
<code>-</code>	Unary minus	
<code>!</code>	Unary logical negation	
<code>~</code>	Unary bitwise complement	
<code>sizeof</code>	Determine size in bytes	
<code>&</code>	address	
<code>*</code>	dereference	
<code>new</code>	Dynamic memory allocation	
<code>new[]</code>	Dynamic array allocation	
<code>delete</code>	Dynamic memory deallocation	
<code>delete[]</code>	Dynamic array deallocation	
<code>(type)</code>	C-style unary cast	Right to left
<code>.*</code>	Pointer to member via object	Left to right
<code>->*</code>	Pointer to member via pointer	



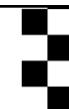
Operator Precedence Table 3

Operators	Function	Associativity
*	multiplication	left to right
/	division	
%	modulus	
+	addition	
-	subtraction	
<<	Bitwise left shift	
>>	Bitwise right shift	
<	Less than	
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Is equal to	
=!	Is not equal to	
&	Bitwise AND	
^	Bitwise exclusive OR	
/	Bitwise inclusive OR	



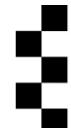
Operator Precedence Table 4

Operators	Function	Associativity
&&	Logical AND	left to right
//	Logical OR	
?:	Ternary conditional	Right to left
=	Assignment	Right to left
+=	Addition assignment	
-=	Subtraction assignment	
*=	Multiplication assignment	
/=	Division assignment	
%=	Modulus assignment	
&=	Bitwise AND assignment	
^=	Bitwise exclusive OR assignment	
=	Bitwise inclusive OR assignment	
<<=	Bitwise left-shift assignment	
>>=	Bitwise right-shift assignment	
,	sequencing	Left to right



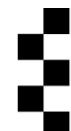
Order of Evaluation

- It is important to note that the operator precedence table does not always specify the precise order of evaluation.
- It tells us that $i++ * j--$ means $(i++) * (j--)$
- In this example it does not matter – the outcome will be the same.
- In some cases the order of evaluation is specified – the operands of the logical operators, the ternary conditional and the comma operator are always evaluated from left to right



Order of Evaluation

- In an expression of the form $e1 ? e2 : e3$ only one of $e2$ and $e3$ will be evaluated;
- If $e1$ evaluates to true (or a non-zero value) the value of the expression is the value of $e2$, otherwise it is the value of $e3$.
- The logical operators are evaluated lazily: $e1 \&& e2$ must be false if $e1$ is false, so $e2$ will not be evaluated, and similarly $e1 | | e2$ must be true if $e1$ is true, so again $e2$ will not be evaluated.



The Increment and Decrement Operators

- Both **i++** and **++i** may be used to increment the value held in the variable **i**.
- However the values of the two expressions are different based on their precedence rule
- This difference is significant only if the increment is used as a sub-expression: **a[i++] = 5;** and **a[++i] = 5;** will update different elements of the array.
- The same applies to the **--** operator.
- Note that expressions such as **++i++** and **++i--** are not allowed;



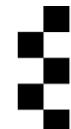
Classes and Objects – an Example 1

```
1 // Time.h - declaration of class Time.  
2 // member functions are defined in Time.cpp  
3 #ifndef _TIME_H  
4 #define _TIME_H  
5  
6 class Time  
7 {  
8     public:  
9         Time(); //constructor  
10        void setTime(int, int, int);  
11        // set hour, minute and second  
12        void printUniversal();  
13        // print time in universal-time format  
14        void printStandard();  
15        // print time in standard-time format  
16    private:  
17        int hour; // 0 - 23 (24-hour clock format)  
18        int min; //0-59  
19        int sec; //0-59  
20    }; //end of class Time  
21 #endif
```



Classes and Objects – an Example 2

```
1 //Time.cpp - member-function definitions for class Time.
2 #include <iostream>
3 #include <iomanip>
4 #include "Time.h"
5
6 using namespace std;
7 // constructor initializes each data member to zero.
8 // ensures all Time objects start in a consistent state.
9 Time::Time()
10 {
11     hour = min = sec = 0;
12 }
13 // set new Time value using universal time;
14 // ensure that the data remains consistent by setting // invalid values to zero
15 void Time::setTime(int h, int m, int s)
16 {
17     hour = (h>=0 && h<24) ? h : 0; // validate hour
18     min = (m>=0 && m<60) ? m : 0; // validate minute
19     sec = (s>=0 && s<60) ? s : 0; // validate second
20 }
```



Classes and Objects – an Example 3

```
1 // Time.cpp (continued)
2 // print time in universal-time format (HH:MM:SS)
3 void Time::printUniversal()
4 {
5     cout << setfill('0') << setw(2) << hour << ":"
6     << setw(2) << min << ":" << setw(2) << sec;
7 }
8 // print time in standard-time format
9 // ([H]H:MM:SS am/pm)
10
11 void Time::printStandard()
12 {
13     cout << (hour==0 || hour==12 ? 12 : hour%12) << ":"
14     << setfill('0') << setw(2) << min << ":"
15     << setw(2) << sec << (hour<12 ? " am" : " pm");
16 }
17
18 /* setw sets the field width of the next item to
19 * be output
20 * setfill sets the fill character to be used for
21 * all subsequent outputs – we need leading zeroes */
```



Classes and Objects – an Example 4

```
1 // program to test Time class
2 #include <iostream>
3 #include "Time.h"
4
5 using namespace std;
6
7 int main()
8 {
9     Time t; //instantiate object t of class Time
10    // output t's initial value in both formats
11    cout << "The initial universal time is ";
12    t.printUniversal(); // 00:00:00
13    cout << "\nThe initial standard time is ";
14    t.printStandard(); // 12:00:00 am
15    t.setTime(13, 27, 6); //change time
16    // output t's new value in both formats
17    cout << "\n\nUniversal time after setTime is ";
18    t.printUniversal(); // 13:27:06
19    cout << "\nStandard time after setTime is ";
20    t.printStandard(); // 1:27:06 pm
21 }
```



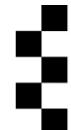
Invoking Constructors

- When a class object is declared with no initialisation (e.g. **Time t;**)
- If we wish to use a different constructor we must supply the arguments in parentheses after the variable name, e.g. **Time t2(12,15);** .
- To create objects on the heap we would use statements such as

```
Time *tPtr1 = new Time;
```

```
Time *tPtr2 = new Time(15,30,40);
```

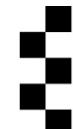
- Note that parentheses are not used if no arguments are to be supplied to the constructor.



Accessing Class Members

- We have already seen that outside of the member functions of a class its members are accessed using the . operator or the -> operator.
- The left-hand operand of the . operator must be a class object or a reference to an object, whereas the left-hand operand of -> must be a pointer to a class object, e.g.

```
1 Time t;
2 Time tArray[5];
3 Time &tRef = t;
4 Time *tPtr = &t;
5 t.setTime(13, 27, 6);
6 tArray[2].setTime(13, 27, 11);
7 tRef.printStandard();
8 tPtr->setTime(13, 27, 19);
9 (tArray+2)->printUniversal();
```



Constant Objects and Members Functions

- It is possible to declare constant class objects. e.g.
const Time noon(12); // or (12, 0, 0)
- The only functions that can be applied to such objects are ***constant member functions***.
- Hence a statement such as **noon.printStandard();** would be rejected by the compiler when using the current version of our class.
- To allow such a statement to be able to be used we should have defined **printStandard** as a constant member function by adding the **const** keyword at the end of the declaration:

void printStandard() const;



Initialising Members of Objects

- Prior to C++11 it was not permissible to initialise a member of a class in its declaration, unless it is a constant static member. Hence

```
1  class X
2  {
3      int y = 3;
4      .....
5 }
```

- would not have been allowed.
- There is an alternative to the initialisation by assignment seen in the constructors for our **Time** class; we can initialise members using an *initialiser list*:

```
Time::Time() : hour(0), sec(0), min(0) {}
```



Initialising Members of Objects

- If one of the members of a class is an object of another class we must invoke one of that class's constructors to initialise it. Consider the following incomplete class.

```
1 class Lecture
2 {
3     private:
4         Time startTime;
5         .....
6     public:
7         Lecture(....., int startHour);
8 }
```

- In the constructor we need to initialise the **startTime** member. We cannot simply write in the body

startTime(startHour);

- since the compiler would try to treat this as a function call.



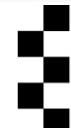
Initialising Members of Objects

- One option for the initialisation of the **startTime** member would be to allow the no-argument constructor to be invoked implicitly and then use the **setTime** function:

```
startTime.setTime(startHour, 0, 0);
```

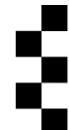
- This is not an ideal approach
- Instead we should perform the initialisation as part of an initialiser list:

```
1 Lecture::Lecture(....., startHour):  
2 | | | | | | | | startTime(startHour);  
3 { .....
```



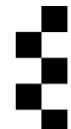
Initialising Members of Objects

- If a class has as a member an object of another class and has a constructor in which that object is not initialised in an initialiser list, the no-argument constructor for the object is invoked implicitly.
- If the compiler generates a default no-argument constructor for a class, this will initialise any object members using their no-argument constructors.



Static Members of Classes

- A class may have static members; these belong to the class rather than to individual objects of the class and exist even if no objects of the class exist.
- Consider a class to store information about employees.
- Assume that each employee needs to be given a unique payroll number, with the first being given the number 1, the second 2 and so on.
- Hence we need to keep a count of how many **Employee** objects have been created.



Static Members of Classes

- The following gives an outline of how we would initialise and use the static member:

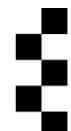
```
1 // Employee.h (incomplete)
2 class Employee:
3 {
4     private:
5         int payRollNum;
6         static int count = 0;
7         // = 0 allowed only since C++11 .....
8     public: E
9     mpolyee(.....);
10    .....
11 };
```

```
1 // Employee.cpp (incomplete)
2 // int Employee::count = 0;  need this before C++11
3 Employee::Employee(.....)
4 {
5     payRollNum = ++count;
6     .....
7 }
```



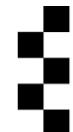
Using this

- If within a member function we need to supply the object to which the function is being applied as an argument to another function we use the keyword **this**, as in Java.
- However, in C++, **this** is a pointer to the object, not a reference.
- If the function being called expects an object or a reference to an object as its argument we have to use ***this**.



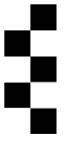
Using this

```
1 #include<iostream>
2 using namespace std;
3
4 /* local variable is same as a member's name */
5 class Test
6 {
7 private:
8     int x;
9 public:
10    void setX (int x)
11    {
12        // The 'this' pointer is used to retrieve the object's x
13        // hidden by the local variable 'x'
14        this->x = x;
15    }
16    void print() { cout << "x = " << this->x << endl;
17    cout << "x = " << (*this).x << endl;
18 }
19 };
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk

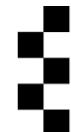


C++ Programming

05 – Operator Overloading,
Friend Functions, the “Big Three”

Operator Overloading

- Operator overloading enables the C++ operators to work with objects of user-defined classes.
- Overloading cannot create new operators and cannot change the precedence, associativity or of an operator.
- The four operators ., .* , :: and ...?.... cannot be overloaded.
- The use of << and >> for output and input is an example of operator overloading.



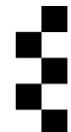
Operator Overloading

- When the compiler sees an expression of the form $e1\#e2$ where $\#$ is a binary operator and at least one of $e1$ and $e2$ is an expression denoting an object
- It will try to treat the expression as either $e1.\text{operator}\#(e2)$ or $\text{operator}\#(e1,e2)$.
- The value of an overloaded operator is the value returned by the $\text{operator}\#$ function.



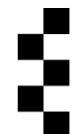
Operator Overloading

- If we need to overload any of the operators (), [] and -> or any assignment operator for use with a class,
- the function ***must*** be written as a member of the class.
- Note that if we overload an operator such as + so that its first argument can be an object of a particular class this does not give a meaning to the corresponding assignment operator (in this case +=);



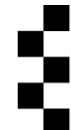
Operator Overloading

- When the compiler sees an expression of the form ***e#*** or **#*e*** where **#** where is a unary operator and ***e*** is an expression denoting an object
- It will try to treat the expression as either ***e.operator#()*** or ***operator#(e)***.
- If we wish to overload either of the **++** or **--** operators for use with objects of a class some method is needed to distinguish between the prefix and postfix versions.
- The compiler will try to treat ***++t*** as either ***t.operator++()*** or ***operator++(t)***, but will try to treat ***t++*** as either ***t.operator++(0)*** or ***operator++(t, 0)***.



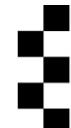
Operator Overloading - Example

- To give examples of the writing of functions to perform operator overloading we shall provide `+`, `+=` and `++` operators for our time class.
- The value of `t+n` where `n` is a non-negative integer should be a time `n` seconds after the time held in `t`; `t+=n` should increment the time stored in `t` by `n` seconds and `t++` and `++t` should increment the time stored in `t` by one second.



Operator Overloading - Example

- The **operator+** function must take a parameter of type **unsigned int** and return either an object of type **Time** or a reference to such an object.
- The function must not change the contents of the object to which it is applied so it should be defined as a constant member function.
- The value of the **+ =** operator should be a reference to the object that has been assigned to in order to allow it to be used as part of a larger expression such as **(t + = 7).printStandard()** or **myt = t + = 7**.



Operator Overloading - Example

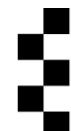
- Since the value of **++t** is the value of **t** after the increment the prefix version of **operator++** can simply return a reference to the object to which it has been applied, avoiding the need for copying.
- However the value of **t++** is the old value of **t** so we must save a copy of the time in a local variable in the postfix version of the function before incrementing and, since it is unsafe to return a reference to a local variable, we must return a copy of this saved object.
- We should hence add to the public part of the class declaration (in the **.h** file) the lines

```
Time operator+(unsigned int) const;  
Time& operator+=(unsigned int);  
Time& operator++(); // prefix version  
Time operator++(int); // postfix version
```



Operator Overloading - Example

- To avoid duplication of code we should write the code to add to times in one of the functions **operator+** and **operator+=**
- The **+=** operator can be written directly without performing any copying, whereas a copy of the object being returned is made by the **+** operator (since it returns a value rather than a reference).

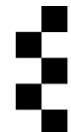


Operator Overloading - Example

- Here is the complete **operator+=** function.

```
1 Time& Time::operator+=(unsigned int n)
2 {
3     sec += n;
4     if (sec >= 60)
5     {
6         min += sec/60;
7         sec %= 60;
8         if (min >= 60)
9         {
10            hour = (hour + min/60)%24;
11            min %= 60;
12        }
13    }
14    return *this;
15 }
```

- Recall that the value of **this** is a pointer to the object to which the function has been applied; we need to return a reference to the object so we have to use ***this**.



Operator Overloading - Example

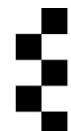
- In the **operator+** function we need to make a copy of the object to which the time has been applied, increment this using **+ =** and then return it.
- We could make a copy using.

Time tCopy = *this;

- However this would result in the no-argument constructor being used to initialise an object and the immediate overwriting of the default values using assignment. To be more efficient we should create a copy using

Time tCopy(*this);

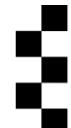
- This invokes a **copy constructor** to initialise the new object to be a copy of an existing one.



Operator Overloading - Example

- Here is the code for the **operator+** function and the two versions of **operator++**.

```
1 Time Time::operator+(unsigned int n) const
2 {
3     Time tCopy(*this);
4     tCopy += n;
5     return tCopy;
6 }
7
8 Time& Time::operator++()
9 {
10    *this += 1;
11    return *this;
12 }
13
14 // prefix version
15 Time Time::operator++(int n) // postfix version
16 {
17     Time tCopy(*this);
18     *this += 1;
19     return tCopy;
20 }
```



Friend Functions

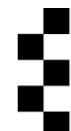
- A function defined outside the class but allowed to access the private and protected members of the given class
- Prototype is declared within the class leading with the keyword **friend**

```
3  class A
4  {
5  private:
6  |  int a, b;
7
8  public:
9  void getdata(int x, int y)
10 {
11     a = x;
12     b = y;
13 }
14 friend void fun1(A);
15 };
16 void fun1(A obj)
17 {
18     cout << "The value of 1st private member is-" << obj.a << endl;
19     cout << "The value of 2nd private member is-" << obj.b << endl;
20 }
```



Friends Functions

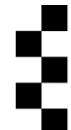
- To allow the user to output times using expressions such as **cout<<t** we wish to overload the **<<** operator.
- The compiler will try to treat the above expression as either **cout.operator<<(t)** or **operator<<(cout, t)** so we cannot perform this overloading by writing a member function that is applied to a **Time** object.
- Hence we need to write an **operator<<** function outside the **Time** class.
- In order for this function to be able to access the private data from the class it must be declared as a ***friend function*** inside the class.
- The function will need to take two parameters, references to **ostream** and **Time** objects.
- To allow **cout << t1 << t2** to work as expected the value of **cout << t1** must be **cout**, so the function should return a reference to its stream argument.



Friends Functions

- The **friend** declaration can be made anywhere inside the class declaration, since a friend is not a member of this class and is hence neither private nor public.
- It is however conventional to declare friend functions at either the beginning or end of class declarations:

```
1 class Time
2 {
3     public:
4     .....
5     private:
6     .....
7     friend ostream& operator<<(ostream&, const Time&);
8 }
```

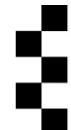


Friends Function

- The **operator<<** function is simply written as a global function in the **.cpp** file.
- The body of the function should be very similar to that of one of the **print** functions written in part 3; we choose to use the universal version.

```
1 ostream &operator<<(ostream &o, const Time &t)
2 {
3     o << setfill('0') << setw(2) << t.hour << ":" 
4     << setw(2) << t.min << ":" << setw(2) << t.sec; return o;
5 }
6
```

- We might wish to overload the **>>** operator to allow for input of times but to provide a robust version of this



Constant References

- Assume that the following three statements appear in separate parts of a program.

const int months = 12;

int &a = months;

a = 13;

- This cannot be allowed since it enables the value of a constant to be changed.
- Hence only a constant reference may refer to a constant data item so we need

const int &a = months;



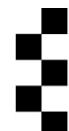
Constant References

- A constant reference may refer to a non-constant data item so the following code is allowed.

```
int x = 99;
```

```
const int &a = x;
```

- If this code does appear in a program we are not allowed to change the contents of the data item by using the reference, so although **x = 101;** would be allowed **a = 101;** would not.



Constant References

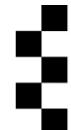
- If a function returns a non-constant reference a call to that function may be used as the left-hand operand of an assignment, e.g.

f(a) = 99;

- To prevent a statement such as

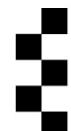
s.regNo() = 123;

- being used to change the value of an attribute of a constant object, a constant member function is not allowed to return a non-constant reference to the object to which it is applied or any of the members of that object.



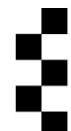
The “Big Three”

- All classes must have a copy constructor, an assignment operator and a ***destructor***.
- These three functions are often referred to as the “big three”.
- If the programmer fails to provide any of these functions for a class the compiler will generate a default version.



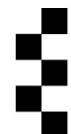
The “Big Three”

- The assignment operator for a class **X** is a public member function declared as **X& operator=(const X&)**.
- It is called implicitly whenever a programmer uses an assignment of the form **x = e**, where **x** refers to an object of the class and **e** is an expression whose value is an object of the class.
- Note that it is permissible to write other **operator=** functions with different argument types.



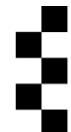
The “Big Three”

- A destructor for a class **X** must be a public member function called **$\sim X$** .
- It must have no arguments and has no return type.



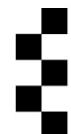
The “Big Three”

- The copy constructor for the class takes as an argument another object of the class and is invoked implicitly in all circumstances where a copy of a class object is needed other than by assignment.
- The argument to a copy constructor must be a reference; otherwise we would have to use the copy constructor to create an object to be passed as an argument to the copy constructor.
- It must not change the contents of the object that is being copied so the copy constructor for a class **X** should be declared as **X(const X&)**.



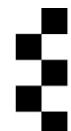
The “Big Three”

- The default copy constructor generated by the compiler will use the copy constructors for any members of the class which are objects of other classes and initialise all other members using simple copying.
- The default assignment operator will just perform assignments to all the members.
- The default destructor will invoke the destructors for any members of the class which are objects of other classes and do nothing else.



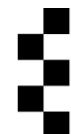
The “Big Three”

- The most usual circumstances in which non-default versions are required are when one of the members of the class is a pointer.
- The copy of the pointer will point to the same data item as the original pointer.
- This is sometimes referred to as *shallow copying*; in many cases *deep copying* is required, i.e. the pointer in the copy should point to a copy of the data item to which the pointer in the original object pointed.



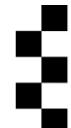
Example – an Array Class

- The standard library provides a class for arrays with range- checking.
- This is called **vector** and allows for arrays of objects of any type.
- To illustrate the use of the “big three” we shall show how to write a similar, but simpler class, which just supports arrays of **int**.
- When we make a copy of an **Array** object we need to create a copy of the dynamic array so the default copy constructor and assignment operator are not appropriate.



Example – an Array Class

- Note that to allow the use of **a[i]** in different contexts we need to provide two versions of the **operator[]** function.
- If **a** is a constant array we need to allow the use of expressions such as **x = a[i]**.
- To allow the operator to be used in an assignment expression such as **a[i] = 3** when **a** is not a constant array we also need a version that returns a non-constant reference.

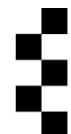


Example – an Array Class

```
1 // // Array.h
2 #ifndef _ARRAY_H_
3 #define _ARRAY_H_
4
5 #include <iostream>
6
7 using namespace std;
8
9 class Array
10 {
11 public:
12     Array(int = 10); // size; default argument
13     Array(const Array&); // copy constructor
14     ~Array(); // destructor
15     const Array& operator=(const Array&);
16     int getSize() const;
17     int& operator[](int);
18     int operator[](int) const;
19 private:
20     int size;
21     int *ptr; // pointer to first element
22     friend ostream &operator<<(ostream&, const Array&);
23     friend istream &operator>>(istream&, Array&);
24         // will not be robust
25 }; #endif
```

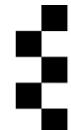
Example - an Array Class

- The assignment operator should return a reference to the object to which it is applied.
- We chose to return a constant reference in order to prevent attempts to use expressions such as **(a1 = a2) = a3**.
- We also provided **operator==** and **operator!=** functions which checked whether two arrays had the same contents.



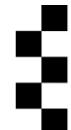
Example – an Array Class

```
1 // Array.cpp
2 #include <iostream>
3 #include <iomanip>
4 #include "Array.h"
5
6 using namespace std;
7
8 Array::Array(int arrSize)
9 {
10    size = arrSize>0 ? arrSize : 10;
11    ptr = new int[size];
12    for (int i = 0; i<size; i++)
13        ptr[i] = 0;
14 }
15
16 Array::Array(const Array &a): size(a.size)
17 {
18    ptr = new int[size];
19    for (int i = 0; i<size; i++)
20        ptr[i] = a.ptr[i];
21 }
22
23 Array::~Array()
24 {
25    delete [] ptr;
26 }
27
28 const Array& Array::operator=(const Array &a)
29 {
30    if (&a != this)
31    {
32        if (size != a.size)
33        {
34            delete [] ptr;
35            size = a.size;
36            ptr = new int[size];
37        }
38        for (int i = 0; i<size; i++)
39            ptr[i] = a.ptr[i];
40    }
41    return *this;
42 }
```



Example – an Array Class

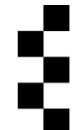
- If the argument supplied to the assignment operator is a reference to the same array as the object to which the function is applied (e.g. $a = a$) no copying is necessary.
- If the argument has the same size as the object to which the function is applied there is no need to allocate a new dynamic array;



Example – an Array Class

```
1 int& Array::operator[](int subscript)
2 {
3     assert(0 << subscript && subscript< size);
4     return ptr[subscript];
5 }
6
7 int Array::operator[](int subscript) const
8 {
9     assert(0 << subscript && subscript< size);
10    return ptr[subscript];
11 }
12 /* the assert function takes a boolean argument and
13 * throws an exception if value of the argument
14 * is not true; otherwise it does nothing
15 */
```

```
16 istream& operator>>(istream &in, Array &a)
17 {
18     for (int i = 0; i<a.size; i++)
19         in >> a.ptr[i];
20     return in;
21 }
22
23 ostream& <<(ostream &out, const Array &a)
24 // displays 4 items per line in fixed-width fields
25 {
26     int i;
27     for (i = 0; i<a.size; i++)
28     {
29         out << setw(12) << a.ptr[i];
30         if (i%4 == 3) out << endl;
31     }
32     if (i%4 != 0)
33         out << endl;
34 }
35 }
```



Type Conversion

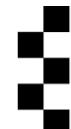
- If a class has a member function **operator $T()$** where T is the name of a type, this function may be used to convert an object of the class to an object of type T .
- Such a function must return a result of type T .
- We can use this, for example, to add to our **Time** class something similar to Java's **toString** method.
- We would add to the public part of the class declaration

operator string();

or

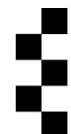
operator char*();

- The type is not specified in the declaration since the compiler can infer that **operator string()** must return a string.



Type Conversion

- To invoke the functions on our previous slide to convert a time **t** into a string we would simply use **char*(t)** or **string(t)**.
- Type conversion operator functions can be invoked implicitly – if we provided an expression denoting an object of type **Time** in some context where an object of type **string** is required the function **operator string** would be applied to the object.
- Hence having written this function permissible to write code such as **string s = t;** or supply a time as an argument to a function that has a **string** parameter.
- Only one conversion will be performed implicitly; if there are conversion operators to convert from **T1** to **T2** and **T2** to **T3** we cannot use an object of type **T1** where **T3** is expected.



Type Conversion Constructors

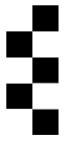
- A constructor with a single argument of a type other than the class of which it is a member can be used as a implicit type conversion constructor.
- Hence, since the **Array** class has a constructor with an **int** parameter we could use an **int** in a program in any context in which an array is expected.
- To prevent implicit conversions using a constructor we should declare that constructor as explicit in the class declaration:

```
explicit Array(int=10);
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk



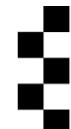
C++ Programming

06 – The string class, Files and Streams

The string Class

- C++ provides two ways of storing strings. They may be stored as C-strings in arrays using a terminator character '\0' to denote the end of a string or as objects of type **string**.
- A string literal, such as "**Hello**", is treated as a character array, the string being represented as a C-string.
- For simple string usage C-strings are normally more efficient than string objects;
- However, for robust programming and complex string manipulation the **string** class is more useful.
- Any file that uses the **string** class must include the appropriate header file:

```
#include <string>
```



The string Class

- The string class has several constructors.
- There is a constructor with no arguments which initialises a string to be empty,

```
#include <string>
string s1; // initially empty
string s2("Hello"); // initially holds Hello
string s3(s2); // initially a copy of s2
```

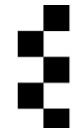
- There is also a constructor with arguments of type **int** and **char**:

```
string s4(5,'x'); // initially holds xxxxx
```



The string Class

- The constructor with a **const char*** argument can be used as a type conversion constructor
- There is no constructor with just a character argument
- We cannot write a declaration such as **string s('a');** (although we could use **string s(1, 'a');** .
- In addition to the assignment operator there are overloaded versions of **operator=** with argument types of **const char*** and **char** so assignments such as **s1 = "xx"** and **s1 = 'a'** are permitted.



The string Class

- Characters in a string object can be accessed using a member function called **at** or the **[]** operator which has been overloaded.

```
1 string s("Hello");
2 cout << s.at(1); // outputs e
3 s.at(2) = 'e'; // s now holds Heelo
4 cout << s[4]; // outputs o
5 s[4] = 's' // s now holds Heels
```

- The **at** function will throw an exception if its argument is out of range but the **[]** operator does not perform range-checking so its use can lead to runtime errors.



The string Class

- The **string** class has many functions and overloaded operators.
- We can use the **append** function or **`+=`** to append to the end of a string a copy of another string or a C-string:

```
string s1("Hello"), s2(s1);
s1 += " world"; // now holds Hello world
s2.append(s2); // now holds HelloHello
```

- The second operand of **`+=`** may also be a character and there is a version of the **append** function which appends multiple copies of a character:

```
s1 += '!'; s2.append(3, '!');
```



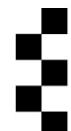
The string Class

- There is another version of the **append** function which appends a copy of part of another string:
s2.append(s1, 3, 2);
- The + operator may be used to concatenate strings.

```
s3 = s1+s2;
```

```
s4 = s3+'!';
```

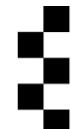
```
s5 = "***"+s4;
```



The string Class

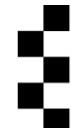
- The **insert** function may be used to insert content at any place in a string;
- **s1.insert(m, x)** will insert a copy of **x** (which may be a **string** object or a C-string) into **s1** in front of the character at location **m**.
- As with **append** there is a version with a **char** parameter that takes an extra number-of-copies argument:

```
String s("C21");
s.insert(1, "E2"); // now holds CE221
s.insert(5, 2, '+'); // now holds CE221++
```



The string Class

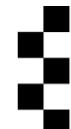
- The length of a string may be obtained using either of the **length** and **size** member functions.
- The **substr** function may be used to extract a substring from a string.
- It is a member function with two arguments: **s.substr(m, n)** will generate a substring starting of length **n** with the character at location **m**.



The string Class

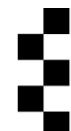
- We can replace part of a string using the **replace** function:
s1.replace(m, n, x) will replace the substring of length **n** starting at location **m** with a copy of **x**, which may be a character, another **string** object or a C-string.
- To delete a substring from a string we can use the **erase** function which takes two integer arguments analogous to those of **substr**.

```
String s("Hello");
s.replace(2, 2, "xxy"); // now holds Hexxyo
s.erase(1, 3) // now holds Hyo
// could also have used s.replace(1, 3, "")
```



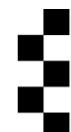
The string Class

- Strings may be compared using the six comparison operators.
- The `<`, `<=`, `>` and `>=` operators compare strings lexicographically using the ASCII ordering.
- The six operators also allow a **string** object to be compared with a C-string.
- There is also a **compare** function: `s1.compare(s2)` returns 0 if the strings are equal, a negative number if `s1` lexicographically precedes `s2` and a positive number otherwise.
- There are also several other versions of **compare** with extra arguments that allow comparisons of substrings.



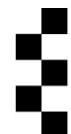
The string Class

- There are several member functions for searching in strings.
- If the item was not found the special value **string::npos** is returned.
- A second argument may be supplied, indicating the position in the string where searching should start.
- The **find** function when used with one argument will attempt to locate the first occurrence in the string; the **rfind** function will attempt to locate the last occurrence.



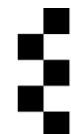
The string Class

- If the string `s` held **Hello** then `s.find('l')` would return the value 2 but `s.rfind('l')` would return 3.
- `s.find('l', 3)` would search for the first occurrence of l at or after location 3 and hence return 3, and `s.rfind('l', 2)` would search for the last occurrence of l at or before location 2 and hence return 2.



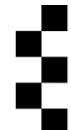
The string Class

```
1 using namespace std;
2
3 #include <iostream>
4 #include <string>
5
6 int countOccs(string s, char c)
7 {
8     int occs = 0;
9     int pos = s.find(c);
10    while (pos!=string::npos)
11    {
12        occs++;
13        pos = s.find(c, pos+1);
14    }
15    return occs;
16 }
17
18 int main
19 {
20     String s("abcabcbcab");
21     cout << countOCCS(s, 'b'); // should output 4
22     cout << countOCCS(s, 'd'); // should output 0
23 }
```



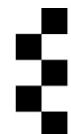
The string Class

- The functions **find_first_of** and **find_first_not_of** can be used to find the first occurrence of any character from an argument string.
- For example we could use something like `s.find_first_not_of(" \t\n")` to find the first non-white-space character in a string.
- There are also functions **find_last_of** and **find_last_not_of**.



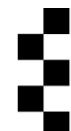
Inputting Strings

- Use of the overloaded `>>` operator to input a **string** object will cause the next sequence of non-white-space characters to be read from the input stream, so it cannot be used to input strings that contain spaces.
- To input a line of text that may contain white space into a string we need to use the function **getline**.
- This takes two arguments, a reference to the input stream and a reference to a **string** object in which the line is to be stored. e.g. **getline(cin, s2)**.
- The newline character at the end of the line will be consumed but will not be stored in the **string** object.



String Streams

- A ***string stream*** allows “input” and “output” to be performed from and to **string** objects.
- This allows us to use the stream formatting facilities such as **setw** and **setfill** provided when generating strings from data and to convert data in strings into objects of other types.
- An output string stream can be created using the no-argument constructor of the **ostringstream** class; this will create a **string** object to store the “output”.
- We can then use the **str** function from the class to access the string.
- We could use an output string stream to write an **operator string** function for our **Time** class (this would be declared in the class as **operator string() const**).



String Streams

```
1 using namespace std;
2
3 #include <iostream>
4 #include <string>
5 #include <sstream> // for string streams #include "Time.h"
6
7 Time::operator string() const
8 {
9     ostringstream s;
10    s << setfill('0') << setw(2) << hour << ":" << setw(2)
11    << min << ":" << setw(2) << sec;
12    return s.str();
13 }
```



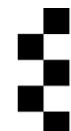
String Streams

- An input string stream is created by supplying a **string** object as an argument to the constructor of the **istringstream** class.
- We can then extract data items from the string using the **>>** operator.
- We could use this to extract numeric data from a string containing digits; assuming that **s** is such a string we could write code such as

```
istringstream str(s);
int i;
str >> i;
```

- We can check whether the extraction was successful by applying the **good** function to the stream, e.g.

```
if (!str.good())
    cout << "error reading from string";
```

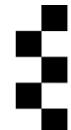


File Processing

- Programs may access files for input and output using file streams.
- The class **ifstream** inherits from **istream** so we can use **>>** to obtain input from a file; and similarly the class **ofstream** inherits from **ostream** so we can use **<<** to output to a file.
- A file stream object may be associated with a file using its constructor (e.g. **ifstream str("data.txt");**) or using the open function e.g.

```
ofstream str;  
str.open("output.txt");
```

- Programs that use file streams need **#include <fstream>**.



File Processing

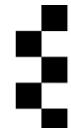
- The `!` operator has been overloaded for file streams so `!str` will be true if the file could not be opened.
- The first argument to file stream constructors and the `open` function must be a C-string; they can take a second argument which specifies an *access mode* (if this is not supplied it defaults to `ios::in` for input streams and `ios::out` for output streams).
- When `ios::out` is used for an output stream any contents of the file (if it already exists) will be overwritten. To append output to the end of an existing file `ios::app` should be used instead:

```
ofstream str;  
str.open("output.txt", ios::app);
```



File Processing

- Streams may be associated with binary files; in this case the appropriate access mode (for both input and output) is **ios::binary**.
- A file will be closed by the destructor when the lifetime of the file stream object is about to expire; we sometimes need to close files before this happens,
- A file associated with a stream **str** may be explicitly closed using **str.close()**; after doing this any attempts to perform input or output on that file will fail.

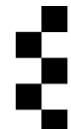


Input Stream Functions

- As an alternative to the use of `>>` we can obtain input from an input stream using member functions.
- The **get** function will return the next character in the file; this may be white space.
- If the end of the file has been reached the special value **EOF** would be returned.
- To process the characters from a stream one by one we could use a loop of the form

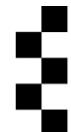
```
char c;  
while ((c = str.get()) != EOF) // process c
```

- It is also possible to check whether the end of a file has been reached using **str.eof()**, which returns a boolean result.



Input Stream Functions

- It is also possible to input the contents of a file line by line, using the **getline** member function of the **istream** class.
- This takes two arguments, the address of the beginning of a character array which will be used to store the line as a C-string, and an integer indicating a maximum size of the string.
- (This function is not the same as the **getline** function described on slide 16; that is used to input a line into a **string** objects and is not a member of the **istream** class.)

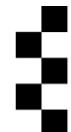


Input Stream Functions

- Consider the following code.

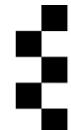
```
1 ifstream str("data.txt");
2 const int SIZE = 80;
3 char buffer[SIZE];
4 while (!str.eof())
5 {
6     str.getline(buffer, SIZE);
7     // process buffer
8 }
9
10 // using namespace std;
```

- If a line in the input file contained more than 79 characters only the first 79 would be input (the size argument includes space for the end-of-string delimiter); the rest of the line would remain in the input stream to be read by the next call to the function.



Formatting Output

- It is possible to output to a file one character at a time using the **put** member function; this is rarely used since **str.put(c)** is equivalent to **str << c**.
- The **width** function can be used to specify a minimum field width for the next item to be output; if the item requires less space fill characters are inserted as padding (the default fill character is a space).
- To output a number **n** in a field of width 6 we could use **cout.width(6); cout << n;**



Formatting Output

- The **fill** function may be used to change the fill character.
- To display a number in a fixed-width field with leading zeroes instead of spaces we could use

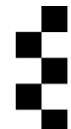
```
cout.fill('0');
```

```
cout.width(6);
```

```
cout << n;
```

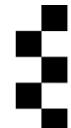
```
cout.fill(' ');
```

- The last line would of course not be necessary if the next item to be output in a fixed-width field was another number where leading zeroes are required.



Formatting Output

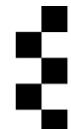
- Real numbers may be output in fixed-point (e.g. **0.34**) or scientific format (e.g. **3.4e-01**).
- To specify a particular format for use for all subsequent real number outputs we should use the **setf** function with an argument of **ios::fixed** or **ios::scientific**.
- We can specify a precision using the **precision** function.



I/O Manipulators

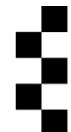
- It is usually more concise to place format control inside a chain of `<<` operations.
- This can be done using *i/o manipulators*, e.g.
`cout << setfill('0') << setw(5) << n << setfill(' ') << endl;`
- The **setw** and **setfill** manipulators perform the same tasks as the **width** and **fill** functions.
- There is also a **setprecision** manipulator.
- When using i/o manipulators it is necessary to use the line

#include <iomanip>



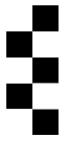
I/O Manipulators

- The manipulators for selecting the format for real-number output are simply **fixed** and **scientific**.
- These are used without parentheses and are not preceded by **ios::**:
`cout << fixed << setprecision(5) << pi << endl;`
- We can also select left- or right- alignment for items in fixed-width fields using **left** or **right**.
`cout << left << setw(5) << n << endl;`
- (It is also possible to set alignment using functions but these are more complicated than the examples we have seen and hence rarely used.)



CONTACT YOUR LECTURER

m.barros@essex.ac.uk



C++ Programming

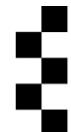
07 – Templates, Containers, STL
and Vectors

Template Functions

- Consider the following function.

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

- If we wished to swap the values of two variables of type **float** or of type **string** we would have to write extra versions of the function with different argument types.
- It is possible to avoid the need to write overloaded versions of functions by using a ***template*** function.



Template Functions

- Here is a template version of the function from the previous slide:

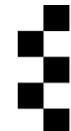
```
1 template <class T>
2 void swap(T &a, T &b)
3 {
4     T temp = a;
5     a = b;
6     b = temp;
7 }
```

- The template declaration says that **T** is a type parameter that can be instantiated with any type (it doesn't have to be a class).
- If we make a call **swap(x, y)** where **x** and **y** are variables of type **float**, the compiler will generate a version of the function with **T** instantiated to **float**.



Template Functions

- The compiler will generate a separate instance of the template function for each type for which a call is made, so if we made calls **swap(x, y)** and **swap(a, b)** where **x** and **y** are of **float** and **a** and **b** are of **int** two separate versions of the function will be created.
- The machine code would hence be the same as if the programmer had written overloaded versions of the function.
- The function on the previous slide cannot be called with two arguments of different types even if a conversion between the two types exists.



Template Functions

- We could write a template function to print the contents of an array in a desired format:

```
1 template <class T>
2 void printArray(const T array[], int count)
3 {
4     for (int i = 0; i<count; i++)
5         cout << array[i] << " ";
6 }
```

- If a call is made to this function with the first argument being an array of objects of some class for which the `<<` operator has not been overloaded the compiler will report an error.

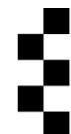


Template Functions

- A template function can have more than one type parameter, e.g.

```
1 template <class S, class T>
2 void printPair(const S &s, const T &t)
3 {
4     cout << '<' << s << ',', ' ' << t << '>';
```

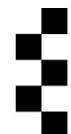
- Here **S** and **T** may be instantiated with the same type or with different types.



Template Classes

- A *template class* is a class in which one or more of the members has a parameterised type, e.g.
- The following class can store pairs of objects.

```
1 template <class S, class T>
2 class Pair
3 {
4     private:
5         S s;
6         T t;
7     public:
8         Pair(S a, T b): s(a), t(b) { };
9         S getFirst() const { return s; };
10        T getSecond() const { return t; };
11 }
```

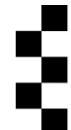


Template Classes

- A variable to hold an object of this class would be declared using the syntax

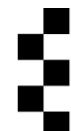
```
Pair<string, int> a("fred", 35);
```

- If the complete definitions of **getFirst** and **getSecond** were in a file called **Pair.cpp** the compiler when compiling this file would not know what instantiations of **S** and **T** are required by code in other files that uses the class and hence could not generate appropriate versions of the functions.



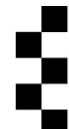
The Standard Template Library

- The ***standard template library*** (STL) provides a number of template classes for various kinds of collections of objects, and algorithms that can be applied to these classes.
- Each of these template classes is known as a ***container***.
- There are three ***sequence containers*** to store sequential data: **vector**, **deque** and **list**.
- In addition there are three ***container adaptors***: **stack**, **queue** and **priority_queue**;



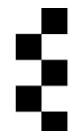
The Standard Template Library

- There are four ***associative containers*** to store non-sequential data with rapid searching: **set**, **multiset**, **map** and **multimap**.
- The STL also contains two other non-template classes, **bitset** and **valarray**, that have some of the functionality of containers, but we will not use these in this module.



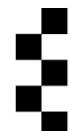
The Standard Template Library

- All container classes and container adaptors have member functions called **size** and **empty**.
- There is also a function **max_size** which returns the maximum possible size for the container.
- Each class has a **swap** function, which takes as an argument a reference to another container of the same type: after a call to **c1.swap(c2)**, **c1** will contain the previous contents of **c2** and **c2** will contain the previous contents of **c1**.



The Standard Template Library

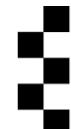
- Each class other than **priority_queue** also has the usual six comparison operators, although the behaviour of operators such as `<` depends on the container.
- The **clear** function (which takes no argument and returns no result) resets the container to be empty.
- There are also functions **begin**, **end**, **rbegin** and **rend** which can be used to obtain **iterators** to traverse through the elements of a collection and **erase** to remove one or more elements (using an iterator).



The vector Class

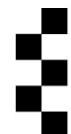
- The **vector** template class provides an implementation of arrays with range-checking and should be used when we want to access elements by position.
- Programs that use this class should contain the line **#include <vector>**.
- The no-argument constructor will initialise a vector to have no contents and a default capacity.
- There is also a constructor with an argument of type **int** which initialises the vector to contain a fixed number of elements, each initialised to the same value – this value may be specified in a second argument which defaults to 0 (or the equivalent for any other type):

```
vector<string> v1;  
vector<int> v2(20); // will contain 20 zeroes  
vector<char> v3(10, 'x'); // will contain 10 'x's
```



The vector Class

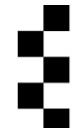
- For a vector of class objects the use of the single-argument **vector** constructor will cause initialisation of the objects in the vector to be performed by the class's no-argument constructor.
- In addition some other **vector** methods make use of the no-argument constructor so when writing a class that is expected to be used in collections a no-argument constructor should normally be provided.



The vector Class

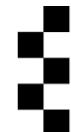
- It is possible to initialise the contents of a vector to be a copy of the contents of an array.
- This is done using a constructor with two arguments – the first is the address of the beginning of the array and the second is the address of the memory location immediately after the end of the array.
- We can use a smaller second argument or a larger first argument to copy part of the array into the vector.

```
int a[] = { 1, 2, 3, 4, 5, 6, 7 };
vector<int> v4(a, a+7);
vector<int> v5(a, a+3); // will contain 1,2,3
vector<int> v6(a+4, a+7); // will contain 5,6,7
```



The vector Class

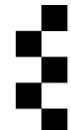
- As with the **string** class, elements of a vector can be accessed using a member function **at** or via subscripting; only the **at** function provides range-checking so for safe access to the elements of a vector an expression such as **v.at(n)** should be used.
- There are member functions **front** and **back** which return references to the first and last element; these can be used as the left-hand operand of an assignment statement:
v2.front() = 7; cout << v2.back();
- If the vector is empty a call to either of these two functions will throw an exception.



The vector Class

- There are member functions **push_back** which will append an element to the end of the vector and **pop_back** which will remove the last element (or throw an exception if the vector is empty).
- If a call to **push_back** results in the size of the vector becoming greater than the current capacity the capacity will be doubled.
- Hence if we know that many calls to **push_back** will be made it can sometimes be more efficient to increase the capacity once using **reserve** before making any of the calls:

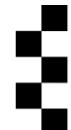
```
v.reserve(100); // increase capacity to 100
```



The vector Class

- The **reserve** function can also be useful if the size of a large vector is about to reach its capacity and we know that we are going to add only a small number of new elements.
- We can prevent the automatic doubling of the capacity (which would result in a large amount of unused space being allocated) by reserving enough space for the extra elements, e.g.

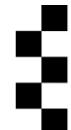
```
v.reserve(v.size()+10);
```



The vector Class

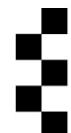
- It is possible to change the size of a vector using **resize**.
- If the new size is smaller than the original size the vector will be truncated; if it is greater multiple copies of an extra value will be appended to the end of the vector.
- The second argument, if provided, specifies this value; the default is 0 or its equivalent, (so we must provide the second argument for a vector of objects of a class without a no-argument constructor):

```
v2.resize(5);  
// elements after v2[4] will be deleted  
  
v1.resize(10);  
// empty strings appended to vector  
// if its size is less than 10  
  
v1.resize(15, "hello")  
// 5 copies of "hello" appended to vector
```



The vector Class

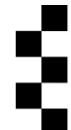
- It is possible to insert values into locations other than the end and remove values from arbitrary locations.
- These operations involve shifting of the existing elements to the right of the insertion or deletion point and are hence rather inefficient.
- Hence if they are to be performed frequently it would be more appropriate to use a different container class such as **list** or **deque**.



Range-based For Loops

- Range-based for loops similar to the enhanced for loops of Java were introduced in C++11. The syntax is
 - **for (*T i: range*) *statement***
- The range may be an array or a container (including a string), or any other class that has been written that supports iterators.
- The statement may of course be a sequence inside { and }
 -
- We could output the contents of a vector one item per line using

```
for (int i: v2) cout << i << endl;
```



Range-based For Loops

- To convert all the characters in the vector **v1** to upper-case we would use

```
for (char &c: v1) c = toupper(c);
```

- To output details of all students in a vector of students (assuming a `<<` operator has been written for the **Student** class) we could use a loop such as

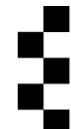
```
for (Student s: studs) cout << s << endl;
```

- However this will result in copies of the student objects being made, so it would be more efficient to use

```
for (Student &s: studs) cout << s << endl;
```

- or

```
for (const Student &s: studs) cout << s << endl;
```



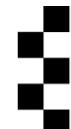
Range-based For Loops

- It is common practice to use **auto** in range-based for loops since the variable type will almost invariably be the type of the data items in the range:

```
for (auto &s: studs) cout << s << endl;
```

- There will be rare occasions when it is useful to use a variable whose type differs from the type of the data items when a conversion is available.
- One example could be using a **string** variable with a vector of C-strings to allow **string** member functions to be used:

```
vector<const char*> v(.....);
for (string s: v)
if (s.find('s')!=string::npos)
    cout << s << endl;
```



Range-based For Loops

- Consider an attempt to output the contents of a vector with position prefixes:

```
int i = 0;  
for (int a: v3)  
{  
    cout << i << ':' << a << endl;  
    i++;  
}
```

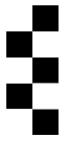
- The traditional for loop would be more concise:

```
for (int i = 0; i<v3.size(); i++)  
    cout << i << ':' << v3[i] << endl;
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk

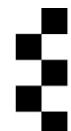


C++ Programming

08 – Iterators, STL algorithms

Iterators

- An **iterator** can be regarded as a smart pointer that points to each element in a collection in turn;
- Programs that use iterators should contain the line **#include <iterator>**.
- An iterator is not actually a pointer but the unary ***** operator has been overloaded so it can be used to refer to the element of the collection that the iterator “points” to.
- In addition the **++** operator has been overloaded to allow the pointer to be advanced to the next element in the collection.
- Some iterators also have overloaded version of operators such as **--** and **+=**.
- An iterator is declared using syntax such as **vector<int>::iterator it**. (The more natural **iterator<vector<int>>** cannot be used since iterators cannot be implemented using a template class.)

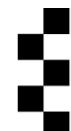


Iterators

- There are three different types of iterator: unidirectional (with just `++`), bidirectional (with `++` and `--`) and random-access (with full "pointer" arithmetic);
- To obtain an iterator that starts at the beginning of a collection the **begin** function from the container class should be used, e.g.

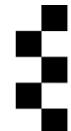
```
vector<int>::const_iterator it = myvec.begin();
```

- The function returns an object of type ***T*::iterator** where ***T*** is the type of the object to which it is applied.



Iterators

- The **end** function from the container class returns an iterator whose pointer position is just after the last element of the collection so we can compare the current state of our iterator with **myvec.end()** to determine whether all items have been traversed.
- We should use **==** or **!=** to perform the comparison since not all iterators have **<** or **<=** operators.
- The **end** function also returns a non-constant iterator, but the **==** and **!=** operators regard a constant iterator and a non-constant iterator that point to the same element in a collection as being equal.



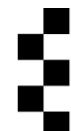
Iterators

- We can use the following loop to print the contents of a vector **v** with element type **int** one item per line.

```
vector<int>::const_iterator it;
for (it = v.begin(); it != v.end(); it++)
    cout << *it << endl;
```

- The following loop will replace all values greater than 100 in the vector with the value 0. (This time we cannot use a constant iterator.)

```
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
    If (*it > 100)
        *it = 0;
```

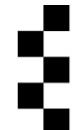


Iterators

- If we wish to traverse a collection in reverse order and the container supports only unidirectional iterators we need to use a ***reverse iterator***.
- The following loop will print the contents of **v** in reverse order.

```
vector<int>::const_reverse_iterator rit;  
for (rit = v.rbegin(); rit != v.rend(); rit++)  
    cout << *rit << endl;
```

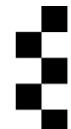
- Observe that we need to use the functions **rbegin** and **rend** to obtain reverse iterators, and **++** (not **--**) is used to move to the previous element.



Iterators

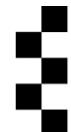
- The **vector** and **string** classes support random-access iterators so it is not in fact necessary to use a reverse iterator to access the contents in reverse order; we could use `--` to step through the elements from **v.end()** to **v.begin()**.
- Since we need to access the element referred to by **v.begin()** but not the one referred to by **v.end()** we cannot use a for loop with a similar structure to the previous slides.

```
vector<int>::const_iterator it = v.end();
while (it != v.begin())
{
    it--;
    cout << *it << endl;
}
```



The **erase** Function

- The **erase** function of a container class may be used to remove one or more elements from a collection.
- The single-argument version simply removes the element currently pointed to by the iterator; an exception will be thrown if the iterator does not point to an element of the collection.
- We can remove the first element from a vector **v** (or any other container) using **v.erase(v.begin())**.
- To remove the element at location **n** we could use **v.erase(v.begin() + n)**; this technique can be used only with containers that have random-access iterators.



The erase Function

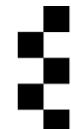
- We could use the following code to locate and remove the first occurrence of 0 in the vector **v**.

```
vector<int>::iterator it = v.begin();
while (it != v.end() && *it != 0)
    it++;
if (it != v.end())
    v.erase(it);
```



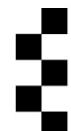
The erase Function

- The two-argument version of `erase` will remove a sequence of elements starting with the element pointed to by the first argument and ending immediately before the element pointed to by the second argument so to remove the elements at locations 3, 4 and 5 from the vector `v` we could use `v.erase(v.begin() + 3, v.begin() + 6)`.
- and to remove all but the first six elements we could use `v.erase(v.begin() + 6, v.end())`.
- The two arguments must have the same type so we cannot use a combination of a normal iterator and a reverse iterator.
- An exception will be thrown if the iterator does not point to an element in the container, e.g. if we try to use something like `v1.erase(v2.begin())`.



Inserting into Vectors

- The **vector** class has three **insert** member functions allowing new elements to be inserted at a position specified by an iterator.
- In all of these, the element(s) will be inserted in front of the position pointed to be the first argument;
- An exception will be thrown if the iterator does not "point" either to an element of the collection or to the position immediately after the last element.



Inserting into Vectors

- The simplest **insert** function has just two arguments, the second being the item to be inserted.

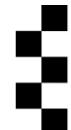
```
v.insert(v.begin()+2, 4);
```

// inserts 4 in front of v[2]

- The second **insert** function allows multiple copies of the same element to be inserted.
- This takes the number of copies as its second argument and the element as its third argument.

```
v.insert(v.begin()+3, 2, 5);
```

// inserts 2 copies of 5 in front of v[3]



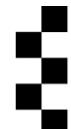
Inserting into Vectors

- The third **insert** function allows a sequence of elements from another collection (which does not have to be a vector) or from an array to be inserted at a position specified by an iterator.
- The second argument specifies the location of the beginning of the sequence and the third the location immediately after the end of the sequence.

int a[] =

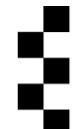
v.insert(v.begin(), a, a+5);

// inserts copies of first 5 elements of a



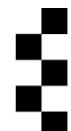
typedef

- If several functions in a program used constant reverse iterators for **vector<int>** it would be inconvenient have to use **vector<int>::const_reverse_iterator** in many declarations.
- To make the code more concise we can make use of **typedef**.
- After defining the type **CRI** using
typedef vector<int>::const_reverse_iterator CRI;
- we could declare constant reverse iterators by simply using declarations of the form **CRI it;**



typedef

- The position of the name of a type in a **typedef** statement is always the same position as that of a variable of that type in a declaration.
- Hence, for example, since **int *p[40];** would define **p** to be an array of 40 pointers to integers,
- **typedef int *PA[40];** would define the type **PA** to be "array of 40 pointers to integers".
- [We can tell that the variable declaration gives an array of pointers rather than a pointer to an array since the precedence rules state that the expression ***p[39]** means ***(p[39])** so **p** must be an array and **p[39]** must be a pointer.]



typedef

- It is possible to include **typedef** statements inside class declarations. Consider for example

```
class C
{
    typedef ...X...
    ....
}
```

- Inside the class declaration we can use the name **X** to refer to the type specified by the **typedef** statement; outside the class we would have to use **C::X**.
- This is in fact how the writers of the standard template library were able to define types like **vector<int>::iterator**.



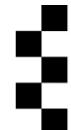
Using auto in C++11

- C++ 11 introduced a facility for the compiler to infer types of variables from their initialisation, avoiding the need to explicitly use lengthy type names in declarations.
- In a declaration of the form

auto *x* = *e*;

- the type of **x** will be the type of the expression **e**.
- Note that if **e** is a call to a function that returns a reference the type of **x** will **not** be a reference;
- if we want a reference we must explicitly use the **&** symbol:

auto &*x* = myfun(*y*);



Using auto in C++11

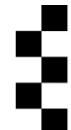
- Using C++ 11 we could have written on slide 7

```
auto it = v.end();
```

- instead of

```
vector<int>::const_iterator it = v.end();
```

- The inferred type of **it** would be **vector<int>::iterator**, but we may be willing to accept this if we know that our code will not change the contents of **v**, and do not require the compiler to check this.



Using auto in C++11

- Note that **auto** can be used only when the variable is given an initial value in its declaration. We could not replace

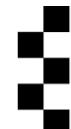
```
vector<int>::const_reverse_iterator rit;
```

- from slide 6 with

```
auto rit;
```

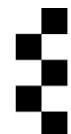
- However, we could have used

```
auto rit = v.begin();  
for (; rit != v.rend(); rit++)  
    cout << *rit << endl;
```



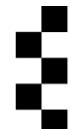
STL Algorithms

- The standard template library has a large collection of functions known as ***algorithms*** that can be used to search and manipulate the contents of containers or strings.
- The STL has about 70 algorithms, including **find**, **replace**, **search**, **sort**, **copy**, and **remove**.
- Not all algorithms work with all container classes; for example a set is an unordered collection of items (`{1, 2, 3}`) is the same set as (`{3, 1, 2}`) so there is no concept of being sorted for a set.



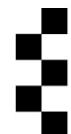
STL Algorithms

- All algorithms take two iterator objects as arguments, specifying the start and end of the portion of the collection to which the algorithm is applied.
- (If this is the whole collection the results returned by **begin** and **end** or **rbegin** and **rend** should be used.)
- There will often be additional arguments.
- As usual the end argument should be an iterator that references the element after the last item in the portion.
- To use most algorithms a program must contain the line **#include <algorithm>**.
- However there are some numerical algorithms (which work only with collections of numbers or objects that have appropriate operators, e.g. **operator+**) for which the line **#include <numeric>** should be used instead.



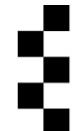
find

- The **find** algorithm will return an iterator that references some occurrence of a specific value, supplied as a third argument.
- In the case of a sequence container or string this will be the first occurrence (or last occurrence if a reverse iterator is being used).
- If the value does not occur in the collection the end iterator for the collection or portion of the collection (i.e. that supplied as the second argument) will be returned.
- The code fragment on the next slide will print the contents of the vector **v** (of items of type **int**) from the first occurrence of 0 to the end of the vector and also the substring of the string **s** extending from the character following the first occurrence of '*' to the end of the string.



find

```
1 typedef vector<int>::const_iterator VecIter;
2 typedef string::const_iterator StrIter;
3 VecIter zero = find(v.begin(), v.end(), 0)
4 for (VecIter it1 = zero; it1!=v.end(); it1++)
5     cout << *it1 << ' ';
6 cout << endl;
7 StrIter star = find(s.begin(), s.end(), '*')
8 if (star != s.end())
9     for (StrIter it2 = star+1; it2!=s.end(); it2++)
10    cout << *it2;
11 cout << endl;
```



find

- The following function will return the number of occurrences of the character **c** in the string **s**

```
1 int count(char c, const string& s)
2 {
3     int occs = 0;
4     string::const_iterator it = find(s.begin(), s.end(), c);
5     while (it!=s.end())
6     {
7         occs++;
8         it = find(it+1, s.end(), c);
9     }
10    return occs;
11 }
```



find

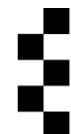
- We could also write a template version that will count the number of occurrences of an item in any collection with the correct item type.

```
1 template <class T, class C>
2 int count(T val, const C& s)
3 {
4     int occs = 0;
5     typename C::const_iterator it = find(s.begin(), s.end(), val);
6     while (it!=s.end())
7     {
8         occs++;
9         it = find(it+1, s.end(), val)
10    }
11    return occs;
12 }
```



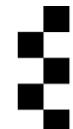
find

- There is nothing in the code on the previous slide to indicate that **C** should denote a container class and the objects in **C** should have type **T**.
- However if an attempt were made to call the function with classes that do not satisfy this a type error would be raised when trying to generate a call to **find**.
- Since the compiler does not know that **C** should denote a container class it would not know that **C::const_iterator** is a type rather than a class member and would try to treat it as the latter unless it is informed that it is the name of a type; hence we have to use the keyword **typename**.



sort

- The two-argument **sort** algorithm will sort the contents of a collection (or part of a collection) into ascending order (i.e. smallest first).
- It can be used only with containers that support random-access iterators.
- Comparison of items within the function is performed using the `<` operator so if the items in the collection are objects they must belong to a class that has an **operator<** function.
- The function changes the order of the contents within the container so it does not make a copy and does not return a result.
- If the arguments are reverse iterators the contents of the collection will be sorted into descending order.

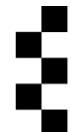


sort

```
1 #include <vector>
2 #include <iterator>
3 #include <algorithm>
4 #include <iostream>
5
6 using namespace std;
7
8 template <class T>
9 void printvec(vector<T> v)
10 {
11     cout << '<';
12     vector<T>::const_iterator it = v.begin();
13     while (it!=v.end()) {
14         cout << *it;
15         if (++it != v.end()) cout << ',';
16     }
17     cout << '>' << endl;
18 }
19
20 int main()
21 {
22     vector<int> v;
23     v.push_back(7);
24     v.push_back(17);
25     v.push_back(3); // v now holds [7,17,3]
26     vector<int> v2(v); // v2 also holds [7,17,3]
27     sort(v.begin(), v.end());
28     printvec(v); // outputs <3,7,17>
29     sort(v2.rbegin(), v2.rend());
30     printvec(v2); // outputs <17,7,3>
31 }
```

sort

- There is a three-argument **sort** algorithm that will sort the contents using a programmer-supplied comparison function instead of <.
- This is useful when we wish to sort a sequence of objects and the value to be used for sorting is one of the members of the class but either the **operator<** function of the class compares a different member or the class has no **operator<** function.
- The third argument to **sort** should be a pointer to a boolean function that takes two arguments (constant references to the items to be compared) and returns true if the value of the first argument is less than the value of the second argument using the ordering that we wish to use.



sort

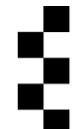
- The code below will sort a vector **v** of type **Vector<student>** by descending average mark (assuming that **aveMark** is a public member of the **Student** class).
- [Recall that to pass a pointer to a function as an argument to another function we simply supply the function's name.]

```
1 bool compareMark(const Student& s1, const Student& s2)
2 {
3     return s1.aveMark < s2.aveMark;
4 }
5 // need a reverse iterator for descending order
6 sort(v.rbegin(), v.rend(), compareMark);
```



for_each

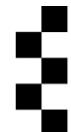
- The **for_each** algorithm provides in C++ some of the functionality of Java's **for int i:myList** or Python's **for i in myList**.
- Its use is not required since the introduction of range-based for loops in C++11.
- However a programmer may have to adapt code that was written before C++11 so should know how to use it.
- As **for_each** is a function rather than a loop construct we have to provide the "loop body" as an argument.



for_each

- We can print the contents of a vector of integers one item per line by supplying a function that will print a single integer and a newline as an argument to **for_each**:

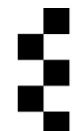
```
1 void print(int i)
2 {
3     cout << i << endl;
4 }
5 vector<int> v;
6 .....
7 for_each(v.begin(), v.end(), print);
```



for_each

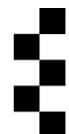
- Since the "loop body" in C++ has to be written as a function it cannot access any local variables from the function that makes the call to **for_each**

```
1 int sum;
2 void addtoSum(int i)
3 {
4     sum += i;
5 }
6
7 int main()
8 {
9     .....
10    sum = 0;
11    for_each(v.begin(), v.end(), addtoSum);
12 }
```



accumulate

- There are ways to overcome the need to use a global variable by using class objects but these are complicated.
- Since the task of obtaining value such as sums, products, minimum and maximum values is often needed and could not be done easily before the introduction of range-based for loops an algorithm to perform such tasks is provided in the STL. It is called **accumulate** (and declared in the **<numeric>** header.)
- We could obtain the sum of the items in a vector using
sum = accumulate(v.begin(), v.end(), 0)
- The third argument is a starting value. Items from the collection are repeatedly added to this with the final value being returned.

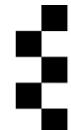


accumulate

- Note that the return type of **accumulate** is determined by the type of the third argument since it is being used to build up the result. If we used

```
sum = accumulate(v.begin(), v.end(), 0.0)
```

- the sum would be a real number.
- To use **accumulate** to generate anything but the sum we need to supply as a fourth argument a pointer to a two-argument function to be performed instead of addition
- Its first argument and return type need to have the same type as the third argument to **accumulate** and the type of its second argument should be the type of the items in the collection.



accumulate

- A call to **accumulate(it, end, x, f)** will effectively perform a loop of the form

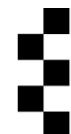
```
while (it != end)
```

```
    x = f(x, *it++);
```

- and return the final value of **x**.
- To use **accumulate** to find the minimum value in a collection of integers we could write a function

```
int minimum(int m, int n) { return m < n ? m : n; }
```

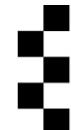
- The first argument represents the smallest item found so far and the second represents the current item in the collection; the value returned is the updated smallest item found so far.



accumulate

- The initial value for the smallest item so far needs to be supplied as the third argument to **accumulate** – we should use the largest **int** value supported by the C++ implementation
- This can be obtained using **numeric_limits<int>::max()** (declared in the header file **<limits>**).
- Assuming our program contains the **minimum** function on the previous slide, a call to **accumulate** to find the minimum value in a vector **v** of integers should be of the form

```
min = accumulate(v.begin(), v.end(),
                 numeric_limits<int>::max(),
                 minimum);
```

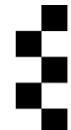


accumulate

- Suppose we have a collection of objects of type **Student** with a member called **mark** (of type **float**) and wish to find the highest mark in the collection.
- This time we need to write a function which extracts the mark from the current object and compares it with the maximum so far.
- Assuming no student can have a negative mark the appropriate start value for the maximum so far is 0.

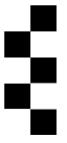
```
float maxMark(float m, const Student &s)
{
    return s.mark > m ? s.mark : m;
}

.....
max = accumulate(studs.begin(), studs.end(), 0.0, maxMark);
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk

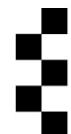


C++ Programming

09 – The list and deque classes,
Container Adapter,
Associative Containers

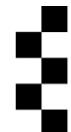
The list class

- The **list** container is a sequence container that stores the sequence as dynamically-allocated cells each containing a pointer to a data item and pointers to the previous and next in the list.
- However accessing items non-sequentially is not efficient so no subscripting or **at** functions are provided.
- Iterators for list objects are bidirectional – they support **++** and **--**, but not **+** and **+=**.



The list class

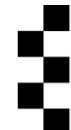
- The no-argument **list** constructor initialises the list to be empty.
- There is also a two-argument constructor – the first argument is a repetition count and the second a value: **list<int> l(3, 4)** will initialise **l** to hold 3 copies of the value 4.
- The first or last item of a list may be accessed using **front** or **back** and removed using **pop_front** or **pop_back**.
- There is no member function to remove items at other specified locations; to do this we need to use the iterator **erase** function.



The list class

- In addition to **push_back** the **list** class has a **push_front** member function for insertion at the front.
- There are **insert** methods that takes an iterator as the first argument; additional arguments are supplied in the same way as the methods from the **vector** class.

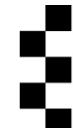
```
1 #include <list>
2
3 ....
4 list<int> l; //inside a main of course
5 l.push_back(3);
6 l.push_front(4);
7 l.insert(++l.begin(), 44);
8 for (int i:l) // assuming C++ 11
9   cout << i << " ";
10  cout << endl;
11  // output will be 4 44 3
```



The list class

- All occurrences of a particular value may be removed using remove function

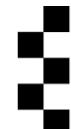
```
1 list<int> l(4, 5); // 5 5 5 5
2 l.insert(++l.begin(), 7); // 5 7 5 5 5
3 list<int>::iterator pos = find(l.begin(), l.end(), 7);
4 l.insert(pos, 8); // 5 8 7 5 5 5
5 l.push_front(7); // 7 5 8 7 5 5 5
6 l.remove(7); // 5 8 5 5 5
7 cout << l.back(); // 5 will be output
8 l.pop_front(); // 8 5 5 5
```



The list class

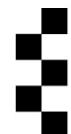
- There is also a **remove_if** function that removes all values that satisfy a condition; the argument is a pointer to a function that takes a data item as its argument and returns a boolean value indicating whether that item satisfies the condition.
- To remove all upper-case letters from a list **l1** of type **list<char>** we could use **l1.remove_if(isupper)**.
- The following code will remove all negative values from the list **l2** of type **list<int>**.

```
1  bool isneg(int i)
2  {
3      return i<0;
4  }
5  l2.remove_if(isneg);
```



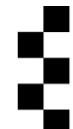
The list class

- We cannot use the sort algorithm to sort lists, since they do not support random-access iterators.
- Hence there are two **sort** member functions to sort a list into ascending order; a no- argument version that sorts using `<` and a version with one argument similar to the last argument of the three-argument **sort** algorithm.
- The function **unique** will remove from the list all but one of any sequence of adjacent equal items.
- If we want to eliminate all duplicate items from a list we need to sort it first so that equal items are adjacent.



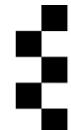
The list class

- The **splice** function can be used to move all of the elements of one list into another.
- **I1.splice(pos, I2)** moves all of the elements of **I2** into **I1** immediately in front of the element referenced by the iterator **pos**.
- After this **I2** will be empty. Note that no copying of elements takes place – only the pointers in the list cells are changed.
- The function can take extra arguments to allow part of **I2** to be moved.
- The **merge** function can be used to move all of the elements of a sorted list into another, with the result being sorted.
- **s1.merge(s2)** moves all of the elements of **s2** into **s1** in appropriate positions to maintain correct ordering.
- If either of the lists is not sorted the order of elements is unpredictable.
- The functions can take an extra comparator argument to specify the ordering.



The deque class

- The **deque** container (pronounced as "deck" and being an abbreviation of double-ended queue) is similar to **vector** but supports efficient addition and removal at both ends.
- The **deque** class provides all of the functionality of **vector** and additionally **push_front**, **pop_front**, **front** and **back** functions.
- Programs that use this container need to include the header file **<deque>**.



The stack and queue adaptors

- The **stack** and **queue** container adaptors provide wrappers around sequence containers to allow them to be used as stacks and queues.
- A stack is a first-in-last-out sequence; items can be added to and removed from the top only and the only item that can be accessed directly is the top item.
- On the other hand a queue is a first-in-first-out sequence; items are added at the back but removed from the front and the only items that can be accessed directly are the front and back items.
- A program that uses one of these adaptors requires the use of **#include <stack>** or **#include <queue>**.



The stack class

- The **stack** class by default is implemented using a deque, but we may if we wish obtain a stack that is implemented using a vector or a list:

```
stack<int> s1; // uses deque<int>
```

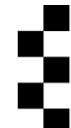
```
stack<int, vector<int>> s2; // uses vector<int>
```

- We may initialise a stack to contain the contents of an existing sequence object of the same type as the class being used to implement the stack; the first item in the sequence will be at the bottom of the stack:

```
list<int> l;
```

```
.....
```

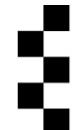
```
stack<int, list<int>> s3(l);
```



The stack class

- The main member functions of the **stack** class are **push**, **pop** and **top**. These are implemented using the functions **push_back**, **pop_back**, and **back**.
- **pop** and **top** have undefined behaviour if the stack is empty.
- In addition member functions **empty** and **size** (as described in part 7) are provided.

```
1 | #include <stack>
2 |
3 | .....
4 | stack<int> st;
5 | st.push(5);
6 | st.push(4);
7 | while (!st.empty())
8 | {
9 |     cout << st.top() << endl;
10|    st.pop();
11| }
```

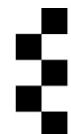


The queue class

- The **queue** class by default also uses a deque, but we may if we wish obtain a queue that is implemented using a list (but not a vector since the **vector** class does not support **pop_front**):

```
#include <queue> queue<char> q1;  
// uses deque<char>  
queue<char, list<char>> q2;  
// uses list<char>
```

- As with stacks we may initialise a queue to contain of the contents of an existing sequence object of the appropriate type; the first item in the sequence will be at the front of the queue.



The queue class

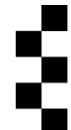
- The main member functions of the **queue** class are **push**, **pop**, **front** and **back**.
- The last three have undefined behaviour if the queue is empty.
- The **push** function adds to the back and the **pop** function removes the front element.
- Once again member functions **empty** and **size** are provided.

```
1 queue<char> q;
2 q.push('x');
3 q.push('y');
4 cout << "back is " << q.back() << endl; // y
5 while (!q.empty())
6 {
7     cout << q.front() << endl;
8     q.pop();
9 } // x will be output, then y
```



The priority_queue class

- A ***priority queue*** is like a queue but items have priorities and when an item is added to the back it will advance ahead of any items with lower priority.
- The order in which items with equal priority will reach the head of the queue not defined.
- The **priority_queue** class by default uses a vector, but we may obtain a queue that is implemented using a deque.



The priority_queue class

- Programs that use the **priority_queue** adaptor need to use **#include <queue>**.
- Priorities are by default compared using the **<** operator; in most applications the items in the queue will be class objects with the priority being a member of the class;
- we would need to write an **operator<** function for the class such that **a<b** returns true if **a** has lower priority than **b**.
- If the item class already has a **<** operator that compares something other than the priorities, we need to supply a comparator class;
- this class should be specified in the declaration of the queue, as a third template argument e.g.

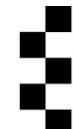
```
priority_queue<Job, deque<Job>, MyComp> pq;
```



The priority_queue class

- The comparator class needs to be a class with an **operator()** function that takes as arguments two data items **a** and **b** (or constant references to such items) and returns the value of **a<b** under the ordering to be used.
- For example if we wish to use a priority queue of objects of type **Job**, where **Job** is a class with a priority stored in a public instance variable **pri**, but has a **<** operator that does not compare priorities, we could write a class of the form

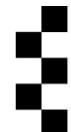
```
1 class MyComp
2 {
3     bool operator()(const Job &a, const Job &b)
4     {
5         return a.pri < b.pri;
6     }
7 }
```



The priority_queue class

- The member functions of the **priority_queue** class have the same names as those of the **stack** class.
- In the following example we assume that the **YellowPerson** class has a priority that is set using the last argument of its constructor, the `<` operator compares priorities and the `<<` operator has been overloaded for the class.

```
1 priority_queue<YellowPerson> pq;
2 pq.push(YellowPerson("homer", ..... ,2));
3 pq.push(YellowPerson("marge", ..... , 3));
4 cout << pq.top(); // will be marge pq.pop();
5 pq.push(YellowPerson("bart", ..... ,2));
6 cout << pq.top(); // could be homer or bart
```



The pair Template Class

- C++ provides a template class called **pair** for storing pairs of objects. e.g.

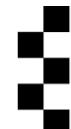
```
pair<int, bool> p(3, false);
```

- We can access the elements of such a pair using the public data members **first** and **second**:

```
cout << p.first << ' ' << p.second << endl;
```

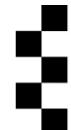
- If we wanted to supply a **pair** object as an argument to a function it would be inconvenient to specify its type explicitly so a template function called **make_pair** is provided

```
myfun(make_pair("bart", 45));
```



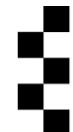
Associative Containers

- An associative container is used to store non-sequential data.
- The STL provides four such containers: **set**, **multiset**, **map** and **multimap**.
- In addition to the member functions listed in part 6 that are common to all containers, the associative containers all have member functions called **find**, **count**, **insert**, **lower_bound** and **upper_bound**.



Associative Containers

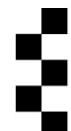
- Associative containers have bi-directional iterators but not random-access iterators.
- The **find** method performs the same task as the **find** algorithm (but in a more efficient way), i.e. **c.find(x)** will give the same result as **find(c.begin(), c.end(), x)**.
- In the case of a multi-map or multi-set the iterator that is returned will reference the first occurrence in the underlying implementation;
- We can increment the iterator to reach other occurrences.
- The **count** method returns the number of occurrences of its argument; for a set or map this will always be 0 or 1.



Associative Containers

- The **insert** method will insert an element into the collection; in the case of a set or map it will leave the collection unchanged if the element is already present.
- This method returns a result of type **pair<C::iterator, bool>** (where **C** is the instantiated container class, e.g. **set<int>**).
- The second element indicates whether anything was inserted; the first element is an iterator object that refers to the inserted element or the existing element with the same value if the element was not inserted.

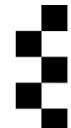
```
1 set<int> s;
2 .....
3 if (s.insert(7).second)
4     cout << "7 inserted";
5 else
6     cout << "7 already present";
```



Associative Containers

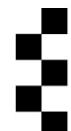
- The **lower_bound** and **upper_bound** methods can be used to obtain start and end iterators for traversing a range of elements from a container.
- **s.lower_bound(x)** will return an iterator referencing the first occurrence of an element greater than or equal to x whereas **s.upper_bound(y)** will return an iterator referencing the first occurrence of an element greater than y.
- Both will return an iterator equal to **s.end()** if no such element exists.
- To visit all of the elements in the range x to y inclusive we can use a loop of the form

```
1 C::iterator end = s.upper_bound(y);
2 for (C::iterator it = s.lower_bound(x); it != end; it++)
3     // process *it
```



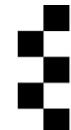
The set class

- The **set** container is used to store sets – a set may not contain duplicate elements.
- It has a no-argument constructor that will initialise a set to be empty, a copy constructor and a two-argument constructor that will initialise the contents of the set using all or part of an array or container object.
- The arguments are either start and end iterators or pointers
- (If the array or container object contains duplicate values only one occurrence of each value will be stored in the set.)
- The **<set>** header file needs to be included in programs that use sets (or multi-sets).



The set class

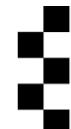
```
1 #include <set>
2 #include <string>
3
4 string a[6] = { "a", "b", "c", "d", "e", "f" };
5 vector<string> ( { "g", "h", "e", "i", "c", "a", "*****", "+++" } ) v;
6 set<string> A(a, a+6); // load A from a
7 set<string> B(v.begin(), v.begin()+6);
8 // load B from first 6 elements of b
9 set<string> C;           // initially empty
```



The set class

- Inserting an element into a set does not invalidate any iterators that already refer to elements of the set and removing an element using the `erase` method does not invalidate any iterators that do not refer to the deleted element.
- Hence we could use the following code to remove all numbers greater than `n` from the set `s` (of type `set<int>`).

```
1 set<int>::iterator it = s.begin(), it2;
2 while (it != s.end())
3 {
4     it2 = it++;
5     // need to advance it before deleting element
6     if (*it2 > n)
7         s.erase(it2);
8 }
```



The set class

- The ***intersection*** of two sets **s1** and **s2** is a set containing all elements that occur in both **s1** and **s2**, whereas the ***union*** of **s1** and **s2** is a set containing all elements that occur in **s1** or **s2** (or both).
- There are STL algorithms called **set_intersection** and **set_union** to obtain the intersection and union of two sets.
- The first four arguments for each of these functions are iterators referencing the beginning and end of the two sets; a fifth argument is used to specify a set in which the result should be stored.
- This takes the form of a special sort of iterator called an ***inserter***.
- A function call of the form **inserter(s, s.begin())** is used to generate an inserter for a set **s**.
- (This function is declared in the **<iterator>** header file.)



The set class

- In the following code we assume that the sets **A**, **B** and **C** are as declared on slide 25.

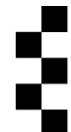
```
1 set_intersection(A.begin(), A.end(),
2                   B.begin(), B.end(),
3                   inserter(C, C.begin())); // C contains "a", "c", "e"
4 C.clear();
5 set_union(A.begin(), A.end(), B.begin(), B.end(),
6           inserter(C, C.begin()));
7 // C contains "a", "b", "c", "d", "e", "f", "g", "h ", "i",
```



The set class

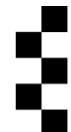
- There is also an algorithm called **set_difference**; this is used to obtain a set containing all of the elements that appear in the set specified by the first two arguments but do not appear in the set specified by the third and fourth arguments.

```
1 C.clear();
2 set_difference(A.begin(), A.end(), B.begin(), B.end(),
3 |                         inserter(C, C.begin()));
4 |                         // C contains "b", "d", "f"
5 C.clear();
6 set_difference(B.begin(), B.end(), A.begin(), A.end(),
7 |                         inserter(C, C.begin()));
8 |                         // C contains "g", "i", "h"
```



The multiset class

- The **multiset** class is used for collections of elements which may contain duplicates;
- its methods are the same as those for **set** but some behave slightly differently.
- For example, the **insert** function will always succeed so the second element of the pair returned by the function will always be **true**.
- The number of occurrences of an item in the union of two multisets is the larger of its occurrence counts from the two sets whereas in the intersection it is smaller of the two occurrence counts.

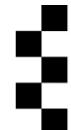


The map class

- A **map** is a collection of keys with associated values, where each key must be unique (but values may occur many times).
- The **map** container in C++ is similar to Python's **dict** type – maps can be viewed as associative arrays and the values associated with keys can be accessed using subscripting.
- Since the keys and values are normally of different types the **map** template class takes two type parameters, e.g.

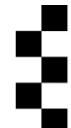
```
map<string, int> age;
```

- Programs that use this class (or **multimap**) need to include the header file **<map>** .



The map class

- The insert method for the map class takes a pair as its argument – it is possible to write code such as
pair<string, int> p("lisa", 14); age.insert(p)
- Or
age.insert(pair<string, int>("bart", 15));
- but it is inconvenient to have to explicitly specify the type parameters. A more concise approach is to use the template function **make_pair** described on slide 18:
age.insert(make_pair("homer", 55));
- The insert method will not insert a pair if the key is already present in the map;
- it cannot be used to update the value associated with an existing key.



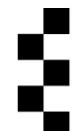
The map class

- An iterator for a map will reference a pair so we could print the contents of the **age** map using

```
1 map<string, int>::const_iterator it;
2 for (it = age.begin(); it != age.end(); it++)
3     cout << it->first << ":" << it->second << endl;
```

- We could also use the **for_each** algorithm, supplying a function that takes a pair as an argument:

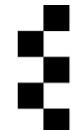
```
1 void printPair(const pair<string, int> &p)
2 {
3     cout << p.first << ":" << p.second << endl;
4 }
5 for_each(age.begin(), age.end(), printPair);
```



The map class

- The **find** and **count** methods for maps take a key as an argument:

```
1 map<string,int>::iterator it = age.find("bart");
2 if (it != age.end())
3     cout << "Bart's age is " << it->second << endl;
4 if (age.count("lisa") == 1)
5     cout << "Lisa is " << age.find("lisa")->second;
```



The map class

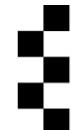
- As in Python we can access the value associated with a key using subscripting:

```
int bartsAge = age["bart"];
```

- If the key is not present in the map the value of the subscript operation will be 0 (or equivalent) – no exception will be thrown.
- Subscripting is also used to update the value associated with a key :.

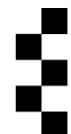
```
age["maggie"] = 1; age["bart"]++;
```

- If the key is not present in the map a new pair will be added.



The multimap class

- In a ***multi-map*** a key may appear more than once. The methods for the **multimap** class are the same as those for **map**, apart from the absence of the subscripting operator, which cannot be supported since a key does not necessarily have a unique value.
- To retrieve the value(s) associated with a key we have to use the **find** method.
- Since this returns a reference to the first occurrence of the key we can increment the iterator to find subsequent occurrences.

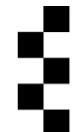


The multimap class

- Assuming **tel** is a multi-map for the storage of telephone numbers we could use the following code to print all of Bart's telephone numbers:

```
1 multimap<string,int>::iterator it = tel.find("bart");
2 while (it != tel.end() && it->first == "bart")
3 {
4     cout << it->second << endl;
5     it++;
6 }
```

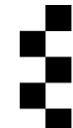
- Note that in the underlying implementation only the keys are sorted so we cannot determine in which order the values will be printed.



The multimap class

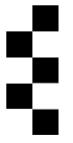
- To update a value associated with a key we must locate the pair using an iterator and then change the second element of the pair.
- Assuming one of Bart's numbers needs to be changed from 1234 to 5678 we might use

```
1 multimap<string,int>::iterator it = tel.find("bart");
2 while (it != tel.end() && it->first == "bart")
3 {
4     if (it->second == 1234)
5         { it->second = 5678;
6          break;
7      }
8     it++;
9 }
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk

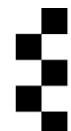


C++ Programming

10 – Inheritance, Abstract
Classes and Virtual Functions

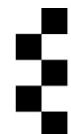
Inheritance

- Inheritance is a form of software reuse: we create a new class that absorbs the data and behaviours of an existing class and enhances them with new capabilities (new members, redefined members).
- The terms superclass and subclass are commonly used to describe the new and old class, but C++ uses the terms ***derived class*** and ***base class***.
- Although a derived class possesses all of the members of its base class the private members of the base class cannot be accessed directly in the methods or friend functions of the derived class;



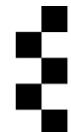
Inheritance

- C++ provides three kinds of inheritance: public, protected and private.
- When using public inheritance the public and protected members of the base class are regarded as public and protected members of the derived class,
- When using protected inheritance the public and protected members of the base class are regarded as protected members of the derived class and when using private inheritance all members of the base class are regarded as private members of the derived class.



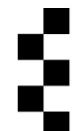
Inheritance

- The use of protected and private inheritance in C++ is relatively rare; public inheritance is required in most applications.
- If a class has a member with the same name (and in the case of a member function the same argument types) as a member of its base class, it will replace the inherited member.
- Data members with the same names should usually be avoided but it is quite reasonable to have methods with the same name,



Inheritance

- If a class A is a subclass of B, which is in turn a subclass of C then C is said to be an *indirect* base class of A whereas B is a *direct* base class of A.
- In C++ (unlike Java) multiple inheritance is allowed – a class may have more than one direct base class.
- For example the class **ifstream** has as base classes both **istream** and **fstream**.
- If two base classes have members with the same name it is necessary to avoid ambiguity – we would have to redefine the member in the derived class.



Inheritance

- The syntax used to indicate inheritance in C++ differs from that of Java; there is no **extends** keyword.
- We indicate that **Student** is a derived class of **Person** using a declaration of the form

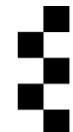
```
1 class Student: public Person
2 {
3     private:
4         int year, regNo;
5     public:
6         Student(string name, int year, int regNo) : Person(name)
7         {
8             this->year = year; this->regNo = regNo;
9         }
10 }
```

- The use of **public** indicates that public inheritance is being used.



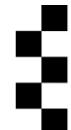
Inheritance

- A constructor for a derived class must invoke the constructor(s) of its direct base class(es).
- This may be done explicitly as part of an initialiser list, as in the example on the previous slide (where we have assumed that the **Person** class has a one-argument constructor);
- otherwise the no-argument constructor from the base class (if it exists) will be invoked implicitly before execution of the body of the derived class constructor.
- If a constructor needs to explicitly initialise inherited members to values that differ from those that would be set by a base class constructor this must be done by assignment in the function body; inherited members cannot be initialised in an initialiser list.



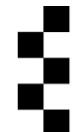
Inheritance

- A destructor for a derived class will always implicitly invoke the destructor(s) of its direct base class(es);
- The body of a destructor written by the programmer will be executed before the base class destructor(s) .
- A call to a copy constructor for a base class may supply a derived class object as its argument. For example in a declaration such as **Person p(s);**
- (where **s** is an object of type **Student**) the **Person** object will be initialised to be a copy of the inherited attributes stored in the object **s**.
- The same applies to assignment operator: **p = s** will perform assignment using the inherited attributes.



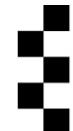
Inheritance

- A member function of a derived class will sometimes need to invoke a member function of the base class that has been redefined in the derived class.
- In particular a method will often need to invoke the method that it replaces.
- For example if the **Student** class has a **print** member function this may wish to invoke the **print** function from the **Person** class to print the values of the inherited members.
- To invoke a method that has been redefined the call must be preceded by the name of the base class, followed by ::.



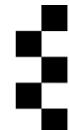
Inheritance

```
1 void Person::print(ostream &o) const
2 {
3     o << "Name: " << name;
4     // would normally expect to print some other
5     // attributes as well
6 }
7 void Student::print(ostream &o) const
8 {
9     Person::print(o);
10    o << "; Year: " << year
11    << "; Registration number: " << regNo;
12 }
```



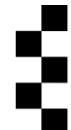
Inheritance – an example

- We now present a detailed example of the use of inheritance.
- A commission employee earns only commission on the sales that he makes, whereas a base-plus-commission employee also earns a basic flat rate salary in addition to his commission on sales.
- We show on the following slides base class **CommissionEmployee** and a derived class **BasePlusCommissionEmployee** written with separate header files.



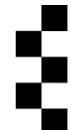
Inheritance – an Example

```
1 #ifndef _COMMISSION_H_
2 #define _COMMISSION_H_
3
4 #include <string>
5
6 using namespace std;
7
8 class CommissionEmployee
9 {
10     public:
11         CommissionEmployee(const string &, const string &,
12                             const string &, double = 0.0, double = 0.0);
13         void setFirstName(const string &);
14         string getFirstName() const;
15         void setLastName(const string &);
16         string getLastName() const;
17         void setSocialSecurityNumber(const string &);
18         string getSocialSecurityNumber() const;
19         void setGrossSales(double);
20         double getGrossSales() const;
21         void setCommissionRate(double);
22         double getCommissionRate()
23             const; double earnings() const;
24         void print() const; // prints object to stdout
25     private: // could also use protected
26         string firstName, lastName, socialSecurityNumber;
27         double grossSales; //gross weekly sales
28         double commissionRate; //commission percentage
29 }; #endif
```



Inheritance – an example

```
1 #include <iostream>
2 #include "CommissionEmployee.h"
3
4 CommissionEmployee::CommissionEmployee(const string &first,
5     const string &last, const string &ssn,
6     double sales, double rate): firstName(first), lastName(last)
7 { // use member functions since validation needed
8     setGrossSales(sales); setSocialSecurityNumber(ssn);
9     setCommissionRate(rate);
10 }
11 double CommissionEmployee::earnings() const
12 {
13     return commissionRate * grossSales;
14 }
15 // other member function definitions needed
16 // (set/get functions, print)
```



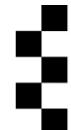
Inheritance – an example

```
1 // BasePlusCommissionEmployee.h
2 #ifndef _BASEPLUS_H_
3 #define _BASEPLUS_H_
4 #include <string>
5 #include "CommissionEmployee.h"
6
7 class BasePlusCommissionEmployee:
8     public CommissionEmployee
9 {
10     public: BasePlusCommissionEmployee(const string &,
11             const string &, const string &, double = 0.0,
12             double = 0.0, double = 0.0);
13     void setBaseSalary(double);
14     double getBaseSalary() const;
15     double earnings() const; // overrides inherited member
16     void print() const; // overrides inherited member
17     private:
18         double baseSalary;
19 };
20#endif
```



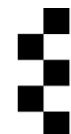
Inheritance – an example

```
1 // BasePlusCommissionEmployee.cpp
2 #include <iostream>
3 #include "BasePlusCommissionEmployee.h"
4
5 BasePlusCommissionEmployee::BasePlusCommissionEmployee( const string &first,
6     const string &last, const string &ssn, double sales,
7     double rate, double salary):
8
9     CommissionEmployee(first, last, ssn, sales, rate)
10    {
11        setBaseSalary(salary);
12    }
13
14 double BasePlusCommissionEmployee::earnings() const
15 {
16     return getBaseSalary() + CommissionEmployee::earnings();
17 }
```



Inheritance – an example

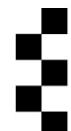
```
1 // main.cpp
2 #include <iostream>
3 #include "BasePlusCommissionEmployee.h"
4
5 using namespace std;
6
7 int main()
8 {
9     BasePlusCommissionEmployee employee("Bob", "Lewis",
10        "333-33-3333", 5000, .04, 300);
11
12    cout << "First name is " << employee.getFirstName()
13    << "\nLast name is " << employee.getLastName()
14    << "\nSocial Security number is "
15    << employee.getSocialSecurityNumber()
16    << "\nGross sales is " << employee.getGrossSales()
17    << "\nCommission rate is "
18    << employee.getCommissionRate() << endl;
19    cout << "Base salary is " << employee.getBaseSalary() << endl;
20    cout << "Employees earnings: $"
21    << employee.earnings() << endl;
22 }
```



Static and Dynamic Binding

- In the code on the previous slide the **earnings** method from the **BasePlusCommissionEmployee** class will be invoked on the penultimate line since the variable **employee** has that type.
- If we wrote a similar class in Java and then wrote code such as

```
BasePlusCommissionEmployee be = .....;  
CommissionEmployee ce = be;  
System.out.println(ce.earnings());
```
- the **BasePlusCommissionEmployee** method would be invoked since **ce** refers to an object of the subclass. Java uses ***dynamic binding*** and decides which method to invoke at run time.
- However C++ normally uses ***static binding***; the choice of which method to invoke is made by the compiler according to the type of the variable, not the type of the object.

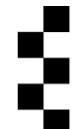


Static and Dynamic Binding

- The C++ equivalent of the Java code on the previous slide is

```
BasePlusCommissionEmployee be(.....);  
CommissionEmployee &ce = be;  
cout << ce.earnings();
```

- Note the use of a reference variable; if we had written **CommissionEmployee ce = be** we would be making a copy of the inherited part of **be**.
- The above code will invoke the **earnings** member function from the **CommissionEmployee** class since the type of the variable is a reference to this class.
- The fact that the variable refers to an object of the derived class plays no part in the decision as to which function to invoke.



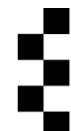
What's not inherited

- The following items are not inherited from a base class:
 - constructors and destructors (the names of these use the name of the class)
 - assignment operators (if the programmer does not provide an assignment operator for a derived class the compiler will generate a default one – this will invoke the assignment operator from the base class)
 - friend functions (they are not members of the class!)



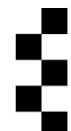
When to use inheritance

- A programmer will often have to make a choice of whether to use inheritance or ***composition*** (i.e. using a class object as a member of another class).
- The general rule is that inheritance should be used for "is-a" relationships, e.g. a student ***is*** a person so it is sensible to write a **Student** class as a subclass of **Person**, but a car ***has*** an engine so a class **Car** should normally have an **Engine** object as one of its members rather than being written as a subclass of **Engine**.



When to use inheritance

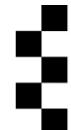
- In some circumstances an "is-a" relationship should not be represented using inheritance.
- It could be argued that a stack is a list but it is not appropriate to write a stack class as a subclass of **List** since the latter class has methods that should not be applied to stacks.
- A square is a rectangle but problems may occur if we try to write a class called **Square** as a subclass of **Rectangle**.
- If the latter has methods **setWidth** and **setHeight** a user may change either the width or the height of a square so that it is no longer square.
- The programmer may redefine these methods in the **Square** class so that both will adjust both the width and height, but because of the use of static binding it is not possible to prevent a user from invoking the base class versions.



When to use inheritance

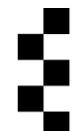
- It would be possible to write **Square** as a subclass of **Rectangle** using protected or private inheritance but this would not allow the user to invoke other methods of the base class, such as **getArea**, and the benefits of inheritance would be lost.
- A wrapper class would probably be more appropriate:

```
1  class Square
2  {
3      private:
4          Rectangle r;
5      public:
6          Square(int size): r(size, size) {}
7          void setSize(int s)
8          {
9              r.setWidth(s);
10             r.setHeight(s);
11         }
12         int getArea() const
13         {
14             return r.getArea();
15         }
16     };
```



Virtual Functions

- Consider a class for the storing information about shapes that are to be displayed on some graphical display.
- Each shape will have some attributes such as size, screen position and colour.
- To allow all of the shapes that are to be displayed to be processed uniformly we need to store them in a list or set of objects of the same type; hence we shall need a **Shape** class.
- However, some of the properties of shapes are dependent on the individual shapes: the area of a square is the square of its sides, but the area of a circle is πr^2 .
- Consequently we will need a subclass of the **Shape** class for each type of shape.

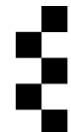


Virtual Functions

- Here is an outline of a **Shape** class.

```
1  class Shape
2  {
3      public:
4          Shape(int size, int x, int y);
5          void changeSize(int newSize);
6          void move(int newX, int newY);
7          int getSize(), getX(), getY();
8      protected:
9          int size, xpos, ypos;
10 }
```

- We assume that the shapes being used are squares, circles, equilateral triangles and other regular polygons, so that we do not have to consider the size in terms of width and length.

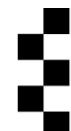


Virtual Functions

- A **Circle** class can be written as a derived class of **Shape**:

```
1 class Circle: public Shape
2 {
3     public:
4         Circle(int diam, int x, int y): Shape(diam, x, y) { }
5         float area() const
6         {
7             int radius = size/2;
8             return M_PI * radius * radius;
9         }
10    };
```

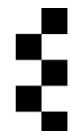
- Note that the header file **<cmath>** needs to be included in order to use **M_PI**.



Virtual Functions

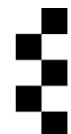
- We can write similar classes **Square** and **Triangle**; although the areas of the squares will be integers, all of the **area** methods should return results of type **float** so that all shape classes have similar functionality.
- To calculate the total area of all of the shapes in a collection **c** of pointers to objects of type **Shape** (assuming that all of the objects belong to derived classes that have **area** methods) we would wish to be able to write a loop of the form

```
float totalArea = 0.0;  
for (it = c.begin(); it != c.end(); it++)  
    totalArea += (*it)->area();
```



Virtual Functions

- The code on the previous slide will not compile since the **Shape** class does not have an **area** method so the compiler will not accept `(*it)->area()`.
- We could write a version that tries out several different dynamic casts (to be covered in part 10) but this would be cumbersome and we would have to know the names of all of the subclasses so the code would have to be modified if new subclasses were created.
- In Java we could simply give the **Shape** class an **area** method that returns 0.0;
- due to dynamic binding the appropriate subclass method would get invoked for each object in the collection.
- This would not work in C++ since static binding results in the method from the **Shape** class being invoked for each object.

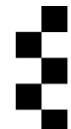


Virtual Functions

- To get dynamic binding in C++ we have to declare a member function in a base class to be a ***virtual function***.
- This is done by preceding its name with the keyword **virtual**.
- Hence we should add to the public part of the **Shape** class the function definition

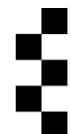
```
virtual float area() const { return 0.0; }
```

- The function in the derived class should not be declared as virtual unless we expect to further extend this class with other subclasses that will need different versions of the function, so we should not change the declaration of the **area** function in the **Circle** class.



Virtual Functions

- When a function declared in a base class as virtual is applied to an object of a derived class accessed using a pointer or reference to the base class, the derived class version of that function overrides the inherited version and will be invoked.
- Note that the dynamic behaviour of virtual functions is only achieved when pointers or references are used since, for example, a variable of type **Shape** cannot hold a **Circle** object.
- Also note that if a derived class has a function with the same name as a virtual function of the base class, but with different argument types, the derived-class version will not override the virtual function.

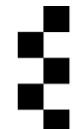


Virtual Functions

- Consider the following code.

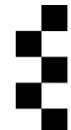
```
1 Circle c(12, 4, 8);
2 Shape s1 = c;
3 Shape &s2 = c;
4 Shape *p = &c;
5 cout << s1.area() << endl;
6 cout << s2.area() << ',' << p->area() << endl;
```

- In the first assignment only the inherited part of **c** is copied into **s1** so **s1** is not a circle, and the base class **area** function will be invoked in the first output statement.
- The variable **s2** refers to a circle and the pointer **p** points to a circle, so since **area** is a virtual function the derived class version will be invoked twice in the second output statement.



Abstract Classes

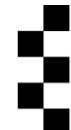
- An ***abstract class*** is one that is used purely as a base class; no instances of it are allowed that are not instances of derived classes.
- In C++ a different technique is used: a class is abstract if it has a ***pure virtual function***.
- This is a function that has no implementation in the base class and is declared using the syntax
`virtual float area() const = 0;`
- As in Java all concrete subclasses of an abstract class must provide versions of the function to override the pure virtual version.



Abstract Classes

- Here is an abstract version of the **Shape** class.

```
1  class Shape
2  {
3      public:
4          Shape(int size, int x, int y);
5          void changeSize(int newSize);
6          void move(int newX, int newY);
7          int getSize(), getX(), getY();
8          virtual float area() const = 0;
9      protected:
10         int size, xpos, ypos;
11     };
```



Abstract Classes

- Since no instances of an abstract class that are not instances of derived classes can be created it is not possible to have variables whose type is the abstract class;
- we must use references and/or pointers, so a declaration such as **Shape s;** would not be allowed. The following would, however, be permissible:

Shape &s = Circle(6, 10, 10)

Shape *p = new Square(5, 20, 20);

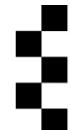
- A declaration such as **Shape s[10];** is also not allowed and the type of objects in an STL collection cannot be an abstract class.



Abstract Classes

- Pointers to abstract classes may be used as template arguments for the STL containers so it is possible to declare collections of **Shape** objects via pointers:

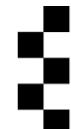
```
1 Vector<Shape*> v;
2 v.push_back(new Circle(10, 20, 20));
3 v.push_back(new Triangle(5, 30, 30));
4 .....
5 for (int i = 0; i < v.size(); i++)
6     cout << v[i]->area() << endl;
```



Abstract classes and output

- Suppose we want to use `cout << x`, where `x` is a reference variable of type `Shape&`.
- Since `x` must refer to an object of a class derived from `Shape`, we would probably want the output to depend on the class of this object.
- However, since `operator<<` cannot be written as a member function of `Shape` we cannot make it virtual.
- Instead we need to write an `operator<<` function that calls a virtual function to perform the actual output:

```
1 ostream &operator<<(ostream &o, const Shape &s)
2 {
3     s.put(o);
4     return o;
5 }
```



Abstract classes and output

- In the class **Shape** the function **put** would be declared as a pure virtual function:

```
virtual void put(ostream&) const = 0;
```

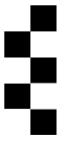
- Each subclass would contain a version of **put** that outputs the contents of that class to the stream.
- For example in the **Square** class we might use something like

```
1 void put(ostream &o) const
2 {
3     o << "Square of size " << size << " at (" 
4     << x, << ',', << y << ')';
5 }
```



CONTACT YOUR LECTURER

m.barros@essex.ac.uk



C++ Programming

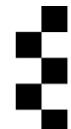
11 – Type Conversions,
Exceptions

Type Conversion

- We have seen that it is possible to convert between types by creating new objects using type conversion constructors.
- However we often wish to treat a pointer or a reference to an object of one type as a pointer or reference to an object of a subtype.
- In Java we might use a loop of the form

```
for (Person p:myList)  
if (p instanceof Student)  
    System.out.println(((Student)p).regNo());
```

- to print the registration numbers of all students in a list of persons; we use down-casting to treat a reference to a person as a reference to a student.



Type Conversion

- In C++ we can use a similar syntax; this is known as **C-style casting**. If we know that all of the members of **myList** are students (where **myList** is of type **List<Person*>**) we could print their registration numbers using

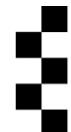
```
1 for (List<Person*>::Iterator it = myList.begin();  
2       it != myList.end(); it++)  
3 {  
4     Person *p = *it;  
5     cout << (Student*)p->regNo() << endl;  
6 }
```

- Note that this type of casting only works with pointers and references; also we had to use a list of pointers since a list of objects of type **Person** cannot hold objects of derived classes.



Type Conversion

- When C-style casting is used no type-checking whatsoever is performed either by the compiler or at runtime, so if any of the objects in the list in the code on the previous slide was not a student the behaviour of the program would be unpredictable and almost certainly incorrect.
- Because of the total lack of type-checking the use of C-style casting in C++ is discouraged;
- C++ instead provides type- casting operators that do perform some type-checking;
- these include **static_cast** and **dynamic_cast**.



static_cast

- The **static_cast** operator can be used to convert between types in situations where the compiler regards it as reasonable to do so.
- Permitted static casts include down-casting from pointers or references to base class objects to pointers or references to derived class objects, and also conversions between numeric types
- The syntax for a static cast is **static_cast<T>(e)**, where **T** is a type and **e** is an expression.
- We could replace the output line of the code fragment on slide 3 with

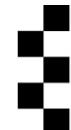
```
cout << static_cast<Student*>(p)->regNo() << endl;
```

static_cast

- When **static_cast** is used no runtime checking is performed so, as before, if the pointer **p** on the previous slide did not point to a student the behaviour of the program would be unpredictable and almost certainly incorrect.
- The compiler will however check that the type conversion is reasonable; for example, an attempt to cast a pointer to a person to a pointer to a date would be rejected.
- To obtain a real-number average from an integer total and an integer count we could use

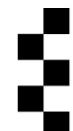
double average = static_cast<double>(sum)/count;

- The compiler will generate code to convert between integer and real-number representations.



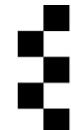
dynamic_cast

- The **dynamic_cast** operator can be used to treat a pointer or reference to a base class object as a pointer or reference to a derived class object, with run-time type checking being performed.
- In the case of pointer conversion the result of the operation will be a null pointer if the cast is invalid;
- In the case of reference conversion an exception of type **bad_cast** would be thrown.
- C++ has no **instanceof** operator, but we can check if a person is a student by examining the outcome of an attempted dynamic cast.
- Two versions of C++ code to perform the same task as the Java code on slide 2 are presented on the next slide; one uses pointers and the other uses references.



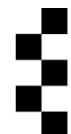
dynamic_cast

```
1 // first version
2 for (List<Person*>::Iterator it = myList.begin();
3     it != myList.end(); it++)
4 {
5     Student* s = dynamic_cast<Student*>(*it);
6     if (s != 0)
7         cout << s->regNo() << endl;
8 }
9
10 // second version
11 for (List<Person*>::Iterator it = myList.begin();
12    it != myList.end(); it++)
13 {
14     try
15     { cout << dynamic_cast<Student&>(*(*it)).regNo()
16         << endl;
17     }
18     catch (bad_cast b) {}
19 }
```



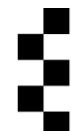
Exceptions

- It is often the case that the handling of an error needs to be done in a different part of the program than the place where it is detected.
- For example if something goes wrong when a function is called the caller may want to display an appropriate message.
- The detector of the error needs to indicate in some way that an error has occurred.
- This could be done using a global flag (e.g. **if (x<0) error = true;**) or by returning a special value (e.g. **if (x<0) return -99999;**), but the preferred approach is to throw an exception (e.g. **if (x<0) throw tantrum();**)

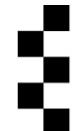
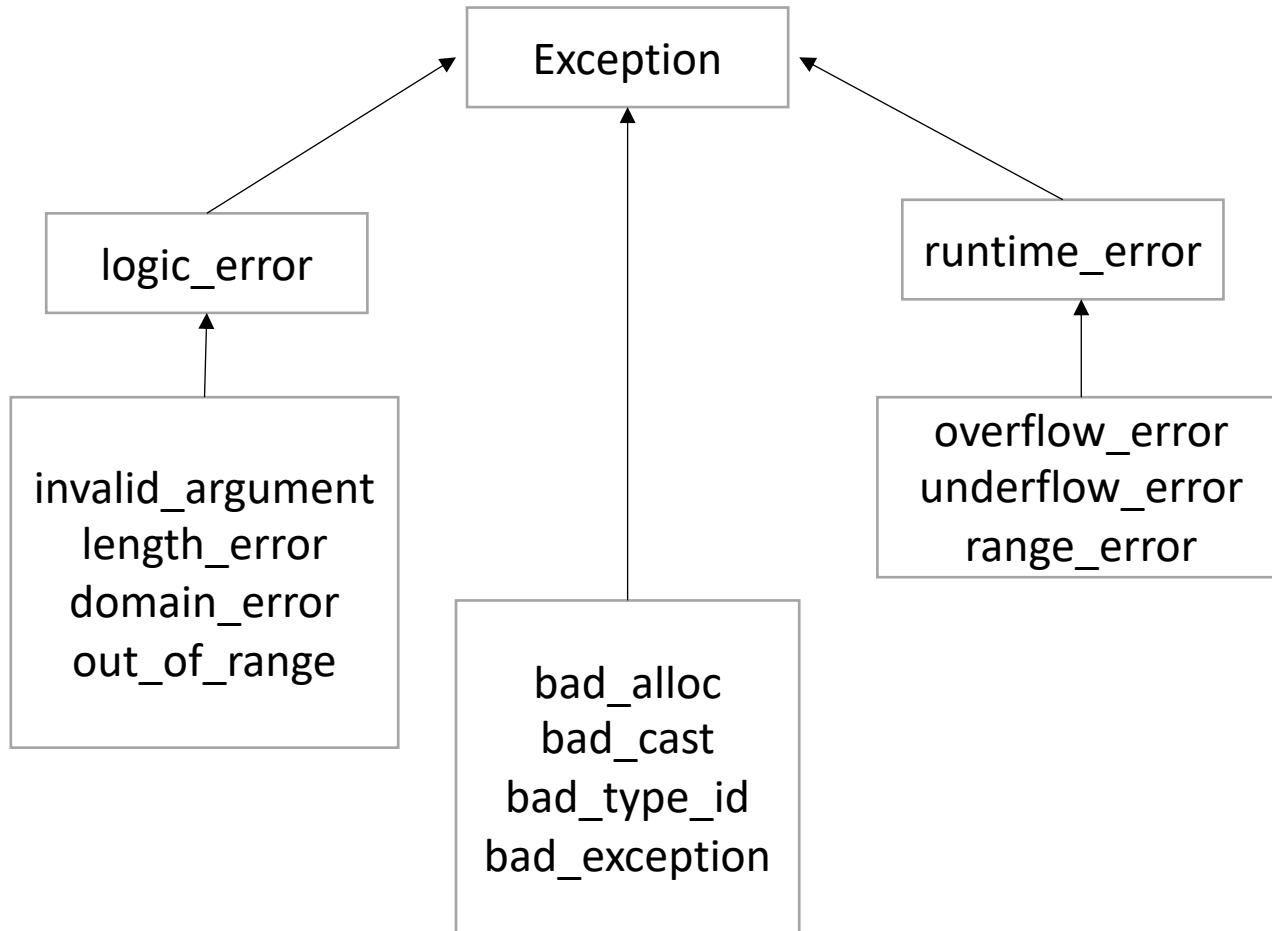


Exception

- C++'s approach to exception handling uses **try** and **catch** blocks in the same way as Java.
- An exception should normally be an object whose type is either **exception** (as defined in the header file **<stdexcept>**) or some subclass of **exception**, but (unlike Java) it is permissible to use a class that does not inherit from **exception**.
- The **exception** class has a public member function called **what** that will return a C-string containing a description of the exception.
- The header file **<stdexcept>** defines several exception classes that may be thrown by standard library functions; the hierarchy shown on the next slide shows most of these classes.

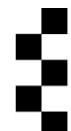


Exception



Exception

- The two subclasses of the **exception** class, **runtime_error** and **logic_error**, have constructors that take a constant reference to a string as an argument;
- The inherited **what** function will return, as a C-string, the string which was supplied when the object was created.
- When a user wishes to create his own exception classes the usual practice is to use one of these as the base class.
- The former should be used when the error is of a nature that could not be prevented by the programmer (e.g. failure to open a file) whereas the latter should be used if the programmer could have prevented it (e.g. index out of range).



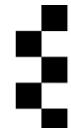
Exception

- If we wish to write a named exception class that behaves in the same way as one of these classes we would simply use something like

```
1 class MyException: public runtime_error
2 {
3     public:
4         MyException(const string &s): runtime_exception(s) {}
5 }
```

- This allows users to write **catch** blocks that just catch occurrences of **MyException**, for example.

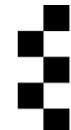
```
1 try
2 {
3     .....
4 }
5 catch (MyException e)
6 {
7     cout << "Caught exception: " << e.what();
8 }
```



Exception

- In many applications we may wish to generate message strings containing some fixed text and additional text supplied as an argument to the exception constructor.

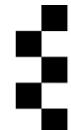
```
1 class StudentException: public logic_error
2 {
3     public:
4         StudentException(const string& s):
5             logic_error("Student problem: " + s)
6     {
7     }
8 }
```



Exception

- If we need to generate a message string that cannot be created in an expression that can be used in an initialiser list we should instead store the string as a private data member and redefine the **what** function to return it.

```
1 class StudentException: public logic_error
2 {
3     private char mess[100];
4     public:
5         StudentException(const string &s):
6         {
7             // generate message in mess
8         }
9         const char *what() const
10        {
11            return mess;
12        }
13 }
```

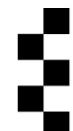


Exception

- We can specify which classes of exceptions could potentially be thrown by a function. C++, however, has no **throws** keyword, so **throw** is used with the following syntax:

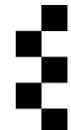
```
1 int myFunc(int n) throw (MyException, HisEx)
2 {
3     if (n==0)
4         throw MyException("n is zero");
5     if (n<0)
6         throw HisEx(n);
7     .....
8 }
```

- Observe that parentheses are always used, even if there is only one exception class in the list.
- Unlike Java there is no concept of checked and unchecked exceptions so the choice of whether to use an exception-list in a declaration is left to the programmer.



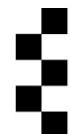
Exception

- Since a function with a throw list can call another function defined in a separate file that has no throw list the compiler is unable to check that the function throws only the listed exceptions.
- Hence the use of throw lists is now generally discouraged.



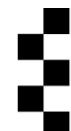
Exception

- We can use **throw()** in the declaration to indicate that the function will not throw an exception.
- Again this does not however guarantee that an exception will not be thrown since the compiler is unable to check whether any calls made inside the function body to functions written in a separate file can potentially throw exceptions.
- If a function does try to propagate an exception that was thrown by a called function but is not listed in its throw-clause an exception of type **bad_exception** will be thrown instead.



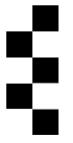
Exception

- We can use **catch(exception e)** to catch all exceptions that derive from **exception**;
- This will not however catch any exceptions that are not objects derived from this class.
- The variable **e** would just hold the inherited part of the class, so if the exception is of a class that has, for example, overwritten
 - the **what** member of **exception** a call to **e.what()** would use the base class version, not the derived class version.
- A catch block headed **catch(...)** will catch anything that is thrown, but since this may be an object of any type, or even a primitive or pointer, we cannot examine it within the block. (The ... is part of the syntax, i.e. exactly three dots without any spaces.)



CONTACT YOUR LECTURER

m.barros@essex.ac.uk



C++ Programming

01- Introduction

Zoom webinar guidelines

A screenshot of a Zoom meeting window titled "Zoom Meeting ID: 844-414-128". The window is displayed in a web browser with the URL "conferencing.psu.edu". The browser's title bar shows the meeting ID. The main content area features the Penn State logo and the word "zoom". Below this, there is descriptive text about Zoom video conferencing and a note about a three-part program. At the bottom, there is a "Zoom Features" section with a list of items and a blue button with a white icon. The bottom navigation bar includes "Audio Settings", "Chat", "Raise Hand", "Q&A", and "Leave Meeting".

Zoom Meeting ID: 844-414-128

PennState

zoom

Zoom video conferencing allows you to engage in multi-person video or audio meetings using software installed on your computer, without the need for dedicated video conferencing hardware. Zoom combines cloud video conferencing, simple online meetings, group messaging, and a software-defined conference room solution into one easy-to-use platform. For more information, visit the [Zoom website](#).

Zoom is the first piece of a three part video management program that will take place over the next year. For more information about the program and timeline, visit the [program section](#).

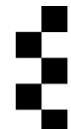
Zoom Features

- Screen share documents, photos, and video clips
- iPhone/iPad screen share with iOS mirror
- High-quality desktop and application sharing

Leave Meeting

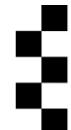
Module Schedule

- Two lectures (**Monday 13.00-13.50 and Tuesday 16.00-16.50**) each week in the autumn term
- We have pre-recorded lectures focused on the theory
- The live lectures will hold tutorial-style lectures plus Q&As
- One 2-hour lab at **13.00 on Thursdays.**
(Note that the labs start in week 3 – there are none in week 2)
- There will also be two revision lectures in the summer term.



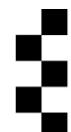
Assessment

- One two-hour examination in May/June (60% of the module credit)
- Two programming assignments to be submitted by 23/11/2020 and 20/01/2021 (20% each)
- **(Note that this module has no week 6 test.)**



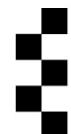
Recommended Reading

- The main recommended text for this module is
 - *Thinking in C++*, B. Eckel, Volume 1, 2nd edition (Prentice Hall, 2000)
 - available for free download at
 - <https://www.micc.unifi.it/bertini/download/programmazione/TICPP-2nd-ed-Vol-one-printed.pdf>
- Alternative books include
- *C++: How to Program*, P.J. Deitel and H.M. Deitel, 10th edition (Pearson, 2016)
 - *C++ for Java Programmers*, T. Budd (Addison Wesley, 1999), and the definitive reference to the original version of C++
 - *The C++ Programming Language*, B. Stroustrup, 3rd edition (Addison Wesley, 2000) .



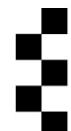
How to succeed in this module?

- Watch all pre-recorded videos before the lectures
- Go through the examples given in the tutorials
- Complete the labs within a few days (mostly)
- Practice, practice, practice!!!
- Ask questions
- Communicate (with me and colleagues)!!
- Work on your assignments weekly (don't leave to do it right before the deadline)
- Make sure you attend the review lectures!!
- Make sure to revisit all available material (videos, slides, etc) before the exam



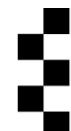
Why learn C++

- C++ is more powerful and efficient than other high-level programming languages (although more complicated).
- C and C++ are the most widely used programming languages for robotics and games, and for writing compilers and operating systems and drivers of hardware devices.
- C++ is built based on C, and C syntax is used in C++
- Consequently a knowledge of C++ is a big advantage in the jobs market.



Application written in C++

- OS
 - Macos, Linux, Windows
- Business and web applications
 - Microsoft Office
- Large-scale graphics applications
 - Adobe photoshop
- Web browsers and web applications
 - Firefox, google chrome
- Other programming languages
 - Java, Python, PhP, Pearl, etc



All about C++

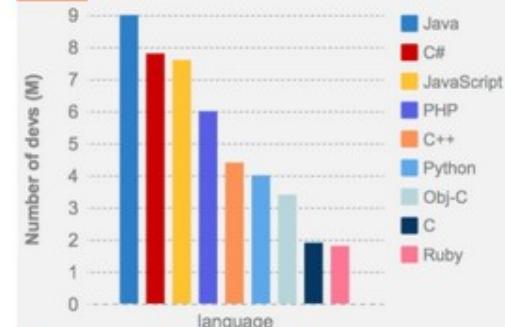
~4.4 million C++ devs
~1.9 million C devs

There are 4.4 million C++ developers and nearly 2 million C developers in the world.



#1

#2



As many C++ developers as Python developers

We've analyzed a range of sources to estimate the number of worldwide developers using the most popular languages.

C++ is on par with Python, while the adoption of C is similar to that of Ruby.

C++ developers by world region

EMEA and Asia-Pacific are the most densely populated regions with regard to developers in general and C++ developers in particular.



#3

#4

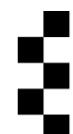
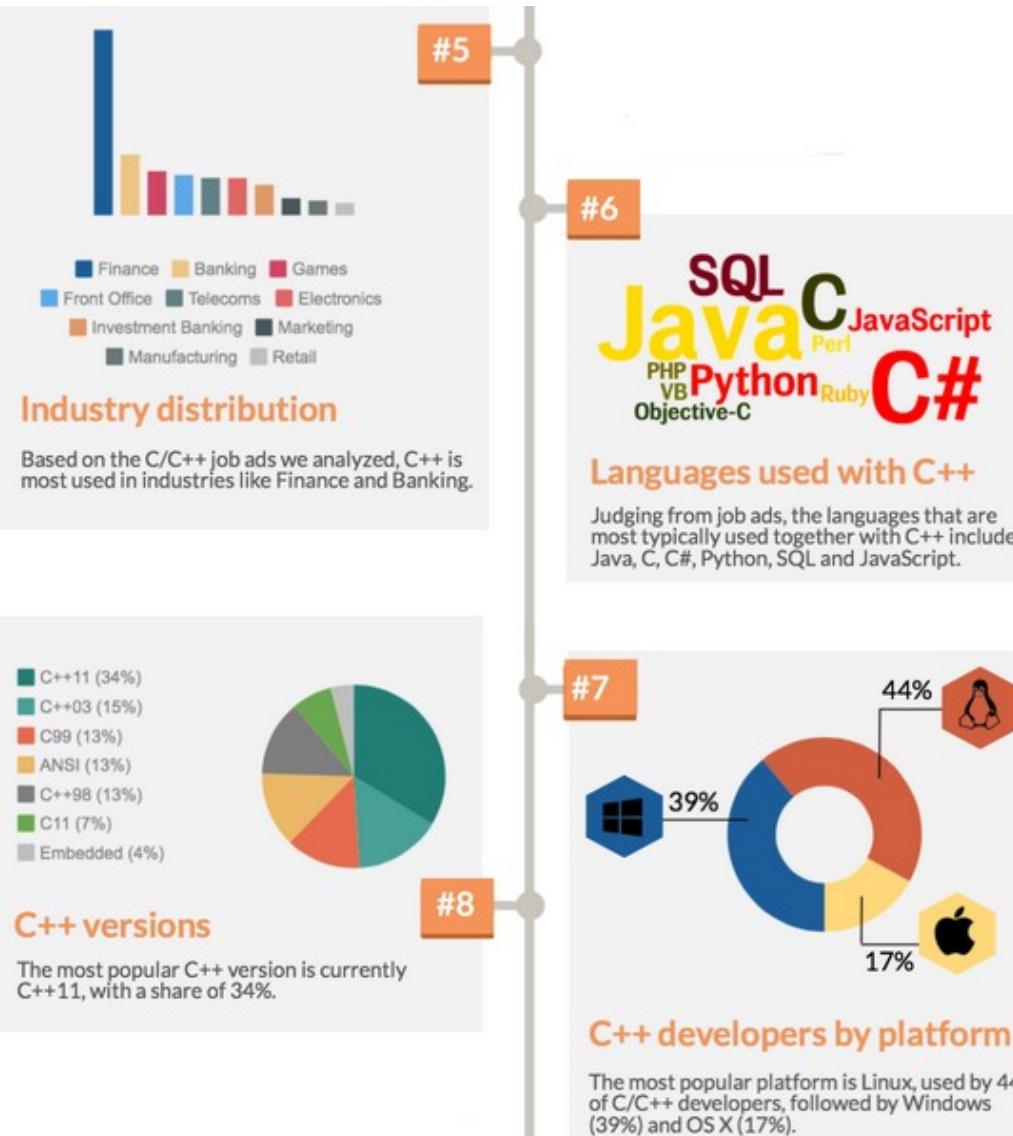
Where C++ is relatively ahead of other languages

C++ is relatively more popular than other languages and technologies in Russia, Czech Republic, Hungary, France, Singapore, Finland, Israel and Germany.

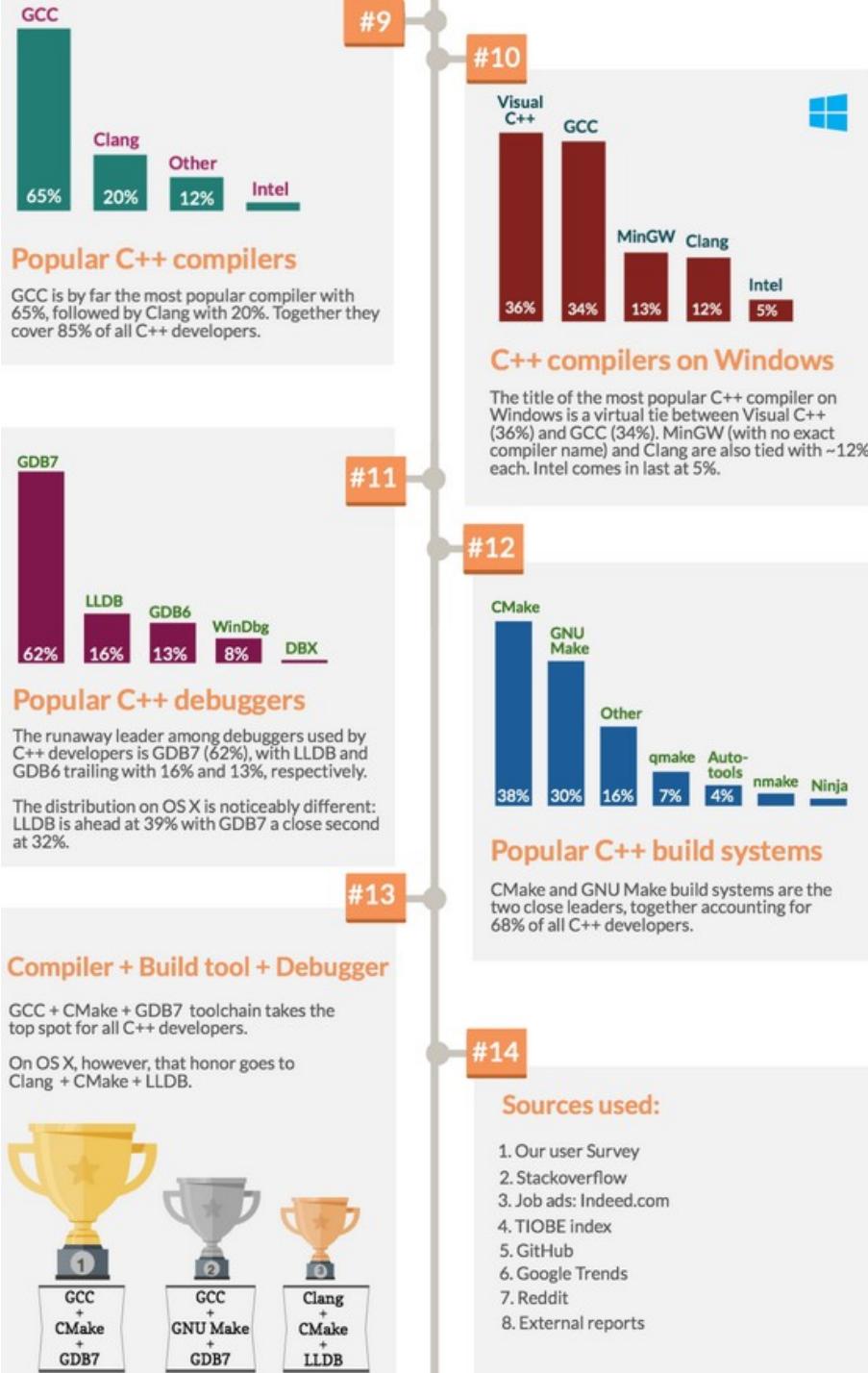


University of Essex

All about C++



All about C++



Learning Outcomes

After completing this module, students will be expected to be able to

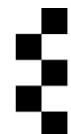
- explain the basic concepts and features of C++.
- describe the underlying memory model and explain the role of the
- execution stack and the heap.
- write object-oriented programs that incorporate basic C++ features such as pointers, references, inheritance, function overriding, operator overloading, exceptions, etc.
- make effective use of the C++ Standard Template Library.



Lecture Outline

The lectures for this module are divided into three main parts:

- Basics: fundamental types and variables, memory management, references, pointers, arrays, control structures, functions, classes and objects, operator overloading, an Array class, the string class, file processing, comparison of C++ and Java.
- Libraries: templates (function templates and class templates), containers, iterators, algorithms, the Standard Template Library (STL).
- Advanced topics: inheritance, polymorphism, exception handling.



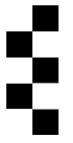
Moodle



University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

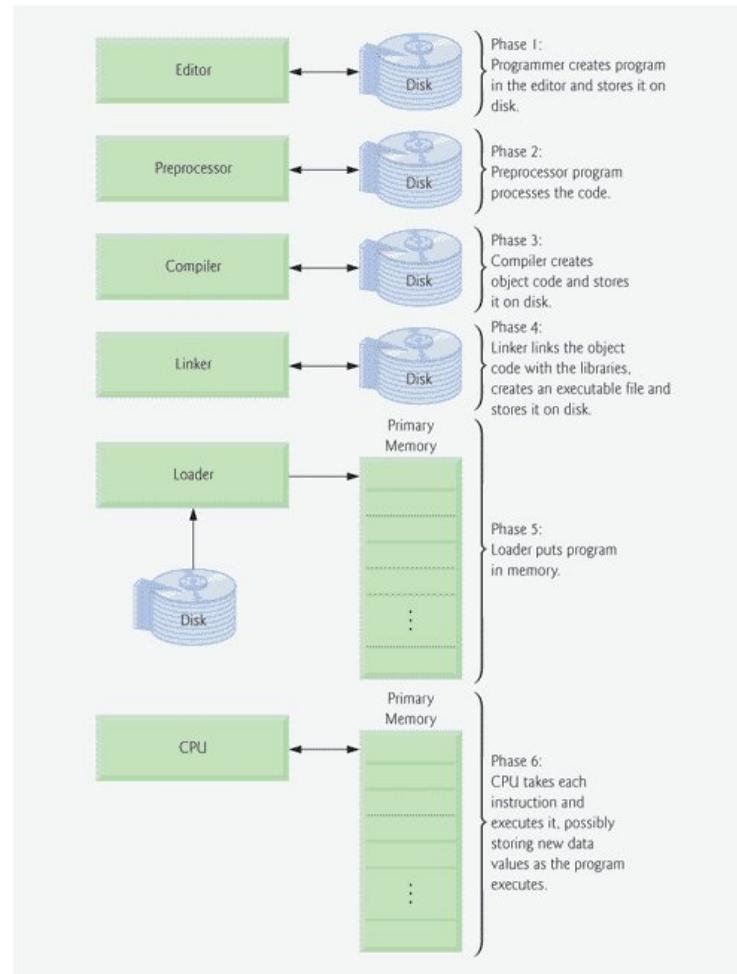


C++ Programming

02- Hello C++

C++ Development Environments

- Development of a C++ program involves six phases:



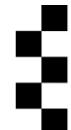
Integrated Development Environment

- Popular IDEs include Visual C++, NotePad++. **However, in the labs we shall be using Sublime, Visual Studio or TextPad.**
- There is a video on how to use Sublime on Moodle.



You first application is the “Hello World”

```
1 // C++  
2 #include <iostream>  
3 //  
4 using namespace std;  
5 // could use int main(int argc, char* argv[]) or  
6 // int main(int argc, char* argv[], char **env)  
7 main() // default return type is int  
8 {  
9     // cout << "\nHello, World!\n";  
10    std::cout << "\nHello, World!\n";  
11    // return 0;  
12 }
```



You first application is the “Hello World”

```
1 // C++  
2 #include <iostream>  
3 //  
4 using namespace std;  
5 // could use int main(int argc, char* argv[]) or  
6 // int main(int argc, char* argv[], char **env)  
7 main() // default return type is int  
8 {  
9     // cout << "\nHello, World!\n";  
10    std::cout << "\nHello, World!\n";  
11    // return 0;  
12 }
```

The main function:

- not written inside a class
- may take arguments to access command-line arguments, but optional;
- its return type is **int**. When a function is declared without a return type it is by default given the type **int**
- not regarded as an error if a function fails to return a result

You first application is the “Hello World”

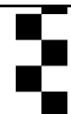
```
1 // C++  
2 #include <iostream>  
3 //  
4 using namespace std;  
5 // could use int main(int argc, char* argv[]) or  
6 // int main(int argc, char* argv[], char **env)  
7 main() // default return type is int  
8 {  
9     // cout << "\nHello, World!\n";  
10    std::cout << "\nHello, World!\n";  
11    // return 0;  
12 }
```

- Screen output is sent to a stream called **cout**, defined in a **namespace** called **std**.
- We have to inform the compiler of the namespace: we can do this explicitly using **std::cout**,
- or we can use a **using** statement to indicate that the namespace **std** resolves any otherwise undeclared identifiers and then simply use **cout**.
- The **<<** operator is used to send values to be output to a stream; we may use a chain of these

You first application is the “Hello World”

```
1 // C++  
2 #include <iostream>  
3 //  
4 using namespace std;  
5 // could use int main(int argc, char* argv[]) or  
6 // int main(int argc, char* argv[], char **env)  
7 main() // default return type is int  
8 {  
9     // cout << "\nHello, World!\n";  
10    std::cout << "\nHello, World!\n";  
11    // return 0;  
12 }
```

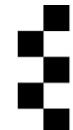
In C++ (and C) information about external classes is stored in the form of declarations in a text file called a **header** and to make these available to the compiler we must include the contents of appropriate header files within the source code using **#include**.



C++ vs JAVA

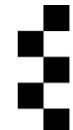
```
1 // C++
2 #include <iostream>
3 //
4 using namespace std;
5 // could use int main(int argc, char* argv[])
6 // int main(int argc, char* argv[], char **env)
7 main() // default return type is int
8 {
9     // cout << "\nHello, World!\n";
10    std::cout << "\nHello, World!\n";
11    // return 0;
12 }
```

```
1 // Java
2 public class HelloWorld
3 {
4     public static void main(String args[])
5     {
6         System.out.print("\nHello, World!\n");
7     }
8 }
```



“Hello World” using a class

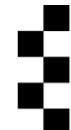
```
1 #include <iostream>
2 using namespace std;
3 // HelloWorld class definition // (a user-defined type)
4 class HelloWorld
5 {
6     public:
7     void displayMessage()
8     {
9         cout << endl << "Hello, World!" << endl;
10    }
11}; // semicolon to terminate class definition
12
13 int main()
14 {
15     HelloWorld my_hello;
16     my_hello.displayMessage();
17 }
```



“Hello World” using a class

“Hello World” program in which the output is produced in a class and the main function makes a call to a function in this class

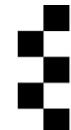
```
1 #include <iostream>
2 using namespace std;
3 // HelloWorld class definition // (a user-defined type)
4 class HelloWorld
5 {
6     public:
7     void displayMessage()
8     {
9         cout << endl << "Hello, World!" << endl;
10    }
11 };// semicolon to terminate class definition
12
13 int main()
14 {
15     HelloWorld my_hello;
16     my_hello.displayMessage();
17 }
```



“Hello World” using a class

conventional to start class names with upper-case letters and variables and function names with lower-case letters

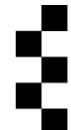
```
1 #include <iostream>
2 using namespace std;
3 // HelloWorld class definition // (a user-defined type)
4 class HelloWorld
5 {
6     public:
7         void displayMessage()
8     {
9         cout << endl << "Hello, World!" << endl;
10    }
11 } // semicolon to terminate class definition
12
13 int main()
14 {
15     HelloWorld my_hello;
16     my_hello.displayMessage();
17 }
```



“Hello World” using a class

```
1 #include <iostream>
2 using namespace std;
3 // HelloWorld class definition // (a user-defined type)
4 class HelloWorld
5 {
6     public:
7     void displayMessage()
8     {
9         cout << endl << "Hello, World!" << endl;
10    }
11 };// semicolon to terminate class definition
12
13 int main()
14 {
15     HelloWorld my_hello;
16     my_hello.displayMessage();
17 }
```

We have used **endl** instead of '\n';
this makes the program easier to
read.



“Hello World” using a class

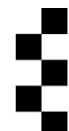
```
1 #include <iostream>
2 using namespace std;
3 // HelloWorld class definition // (a user-defined type)
4 class HelloWorld
5 {
6     public:
7     void displayMessage()
8     {
9         cout << endl << "Hello, World!" << endl;
10    }
11 };// semicolon to terminate class definition
12
13 int main()
14 {
15     HelloWorld my_hello;
16     my_hello.displayMessage();
17 }
```

- This is the class instantiation
- You can also use:
`HelloWorld my_hello = HelloWorld();`
- `my_hello` is this the object



Using Separate Files

- Most classes written in C++ will be expected to be reused
- It is common practice to write each class in a separate file, into
 - A header file with a **.h** extension
 - And the respective **.cpp** extension.



Using separate files

The header file for our **HelloWorld** class will be

```
1 // HelloWorld.h
2 class HelloWorld
3 {
4     public:
5         void displayMessage();
6 }
```

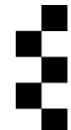
The file containing the **main** function will be

```
1 // HelloMain.cpp
2 #include "HelloWorld.h"
3 int main()
4 {
5     HelloWorld myHelloWorld;
6     myHelloWorld.displayMessage();
7     return 0;
8 }
```



Using Separate Files

- We must specify the filename(s), including the `.h` extension, inside quotes in the `#include` directive(s).
- The angled brackets are used for header files that are found in the C++ system folders.
- Assuming the `HelloWorld.h` file is to be included in the `HelloWorld.cpp` file its contents will be copied into the latter file at compilation time.



Using Separate Files

- We have to write the function body outside of the class definition

```
1 #include <iostream>
2 #include "HelloWorld.h"
3
4 using namespace std;
5
6 void HelloWorld::displayMessage()
7 {
8     cout << endl << "Hello, World!" << endl;
9 }
```



Avoiding Duplicate Header Inclusion

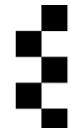
- Many header files need to include other header files
- With multiple files from multiple sources, it is reasonable to assume that there will be repetition of includes.
- This do NOT result in errors!
- The header file for the string class has been written in such a way that if the class declaration has already been included it will not be included again.



Avoid Duplicate Header Inclusion

- The contents of header files are inserted into the temporary copy of the source file by the *preprocessor*
- To prevent class declarations being duplicated, header files should use preprocessor *directives* :

```
1 // myClass.h
2 #ifndef _MYCLASS_H_
3 #define _MYCLASS_H_
4 // use names like _MYCLASS_H_ to avoid
5 // potential clashes with program variable
6 // names
7 // body of header file comes here
8 #endif
```



Avoiding Duplicate Header Inclusion

- Here is a version of **HelloWorld.h** that uses preprocessor directives to avoid any possibility of duplicate inclusion.

```
1 // HelloWorld.h
2 #ifndef _HELLOWORLD_H_
3 #define _HELLOWORLD_H_
4 class HelloWorld
5 {
6     public:
7         void displayMessage();
8     };
9 #endif
```

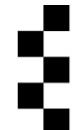


A Typical C++ Program Framework

```
1 // Class1.h
2 #ifndef _CLASS1_H_
3 #define _CLASS1_H_ // include headers
4
5 class Class1
6 {
7     public:
8         Class1(/* args */);
9         // other members
10    private:
11        // members
12 };
13#endif
```

```
1 // Class1.cpp
2 // include headers
3 Class1::Class1(/* args */)
4▼ {
5     // constructor
6     // implementation
7 }
8 // implementation of other
9 // members
```

```
1 // main.cpp
2 // include headers
3 main(int argc, char *argv[])
4 {
5     // create objects, use them
6 }
```



Primitive Types

- C++ has a large number of primitive types:

bool

char

short int (or short) int

long int (or long)

long long int (or long long)

unsigned char

unsigned short int (or unsigned short) unsigned int (or unsigned)

unsigned long int (or unsigned long)

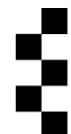
unsigned long long int (or unsigned long long)

float double long double



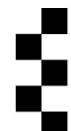
Primitive Types

- The value of a data item of type **bool** is either **true** or **false**.
- Any numeric value may be used in a context where a boolean is required; non-zero values are interpreted as **true** and 0 as **false**.
- The next 10 types listed on the previous slide are all regarded as integer types and it is possible to freely assign between them – overflow may of course occur.
- The unsigned versions cannot hold negative values: an initialisation such as **unsigned int i = -1;** will result in the value of **i** being the integer that has the same value as the two's complement representation of -1, i.e. the maximum value for the type **int**.



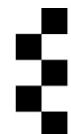
Primitive types

- The range of values that may be stored in a data item of each type is hardware-dependent.
- A **char** will almost invariably have 8 bits, a **short** will have at least 16 bits and a **long** will have at least 32 bits.
- The unsigned versions will always have the same number of bits as their signed counterparts.
- In most modern implementations the **short** type uses 16 bits, both **int** and **long** use 32 bits and **long long** uses 64 bits.
- The **float** type usually uses 32 bits, **double** 64 bits and **long double** 128 bits.
- Since in some older implementations the **int** type has only 16 bits it is wise to use **long** for any variable or class member that is expected to sometimes take values larger than 32767.



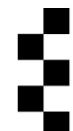
Memory and Storage

- The computer memory for a C++ program has four areas.
- There is an area which stores the code and constants. This is pretty much invariant – the code shouldn't change.
- Data items are stored in three areas:
 - a ***static*** area for items whose lifetime is the duration of the program.
 - a ***stack*** area for local variables in functions, whose lifetime is the invocation of a function; this area is also used to pass arguments to functions and return results, and to store return addresses.
 - a ***heap*** area for dynamic items, whose lifetime is controlled by the programmer



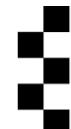
Variables

- A variable in a programming language has three major attributes.
- It has a ***type*** (e.g. **int**, **bool**, **string**, **Student**). Types may be primitive or classes.
- It has ***lifetime*** (or extent). It is born, it is used, it dies.
It has ***scope*** (or visibility). When it is in existence, what other things can see it?
- It also has a name, a size (i.e. how much memory it occupies), and a value.



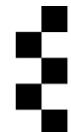
Storage Classes and Variable Lifetime

- There are five ***storage classes***:
 - ***automatic***: for local variables only, existing only when the block in which the variables are defined is active.
 - ***register***: for local variables only, for fast access, rarely used nowadays.
 - ***mutable***: for class members only, to make them always modifiable even in a **const** member function or as a member of a **const** object.
 - ***external***: for identifier linkage of global variables and functions, lasting for the duration of the program.
 - ***static***: for variables and class members that last for the duration of the program and can be initialised once only
- There are five keywords that can (or could) be used to indicate that a variable should have one of these classes: **auto**, **register**, **mutable**, **extern** and **static**.



Storage Classes and Variable Lifetime

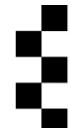
- Since local variables are usually stored on the stack by default the use of **auto** keyword to a variable should be automatic is obsolete and since C++11 is no longer used for this purpose but has a totally different meaning.
- The register class is used to indicate that a frequently-accessed local variable should be stored in a register instead of memory in order to optimise performance
- Such optimisations are rarely necessary on modern machines.
- A **const** member function cannot normally change the values of members of the object to which it is applied.
- There may be rare occasions when we wish to allow the value of a particular member to be changed in such a function (for example, during program development it may be desirable to add an access counter when analysing for possible optimisations), hence the need for mutable variables.



Storage Classes and Variable Lifetime

- A static data item exists for the duration of the program and can be initialised only once.
- Classes can have static members; in this case one copy is shared by all of the objects of that class.
- If a local variable in a function is declared to be static it is initialised when the program starts running and retains its value between calls to that function.
- A static variable must be initialised with a value when it is declared, e.g.

static int x = 77; *// not static int x;*



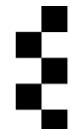
Storage Classes and Variable Lifetime

- A ***global*** variable is one that is defined outside any class or function.
- If a program is split between several files it may be necessary to access the same global variable in more than one file.
- The variable can however be declared in only one of the files so in all of the others it must be declared as external.
- For example, if one file contains the global declaration

```
int a = 5;
```

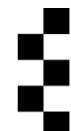
- and we need to access **a** in another file we would need to include the declaration

```
extern int a;
```
- in that file. This is often done by placing the line in a header file associated with the first file and included in the second file.



Scope

- The scope of a variable specifies where in the program it can be accessed.
- A variable declared locally in a block of code can be accessed only between its declaration and the end of that block.
- If, within the block we make a call to a function, the variable cannot be accessed within the function.
- A private member of a class can be accessed only within that class and its member functions and ***friend*** functions (we shall see what a friend function is later).
- The member functions may be defined outside the class body as seen in our example on slide 20.
- If our **HelloWorld** class had any private members they would have been accessible in the **displayMessage** function.

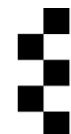


Scope

- A local variable may have the same name as a variable that is already in scope; the following code fragment is valid.

```
1 int a = 0; // global .....
2
3 int myFunc()
4 {
5     char a = 'x';
6     .....
7 };
```

- Within the body of **myFunc** any occurrence of **a** will refer to the local variable. If we need to access the global variable we must use **::a**.



Exercise

```
1 #include <iostream>
2
3 using namespace std;
4 int a = 1;
5
6 void f()
7 {
8     int b = 1;
9     static int c = a; // initialised once only
10    int d = a; // initialised at each call of f()
11    cout << "a=" << a++ << " b=" << b++ <<
12    " c=" << c << " d=" << d << endl;
13    c+=2;
14 }
15
16 int main()
17 {
18     while (a<4)
19         f();
20 }
```

What are the scope and lifetime of **a**, **b**, **c** and **d**?

What would be the outputs?



CONTACT YOUR LECTURER

m.barros@essex.ac.uk