I/O Streams

```
#include <iostream>
using namespace std;

        or

#include <iostream.h>
```

State-of-Stream Member Functions

- **good()** - Returns true if stream is ok
- **eof()** - Returns true if end-of-file hit
- **fail()** - Returns true if error has occurred. In Boolean expressions **cin** or **cout** can be used on it's own to test for an error occurring.
- **bad()** - Returns true if fatal error has occurred
- **rdstate()** - Returns current flags
- **clear()** - Clears all flags

Stream Input

- **getline(***string***,** *length***)** - Member function, reads length – 1 characters into string.
- **>>** - Manipulator, reads up to first space or return, used with **cin**
- **<<** - Manipulator, prints out string, used with **cout**

I/O Stream Manipulators

- **endl** - Outputs '\n' and flushes the output buffer
- **ends** - Outputs '\0'
- **flush** - Flushes output buffer

I/O Stream Format Flag Member Functions

- **setf(***flags***)** - Sets *flags* as additional format flags and returns previous flags, multiple flags are separated by |
- **unsetf(***flags***)** - Clears *flags*
- **flags()** - Return all set format flags
- **flags(***flags***)** - Sets flags as the new format flags and returns the previous flags

I/O Stream Format Flags

Flags	Manipulators	Member Functions
	setiosflags(*flags*)	setf(*flags*)
	setw(int) † left † right †	width(int) †
boolalpha	boolalpha noboolalpha	
uppercase	uppercase nouppercase	
showpos	showpos noshowpos	
showpoint	showpoint noshowpoint	
fixed scientific	fixed scientific	
	setprecision(int)	precision(int)
showbase	showbase noshowbase	
hex oct dec		

† This requires #include <**iomanip**> or #include <**iomanip.h**>

File Streams

#include <**fstream**>

using namespace std;

or

#include <**fstream.h**>

Creating File Stream Objects

ifstream *in_stream*;

ofstream *out_stream*;

File Stream Member Functions

- **open(***name***)** - Opens stream's file (default mode)
- **close()** - Closes stream's file
- **is_open()** - Returns whether the file is opened
- **get(***character***)** - Read single character
- **put(***character***)** - Write single character
- **putback(***character***)** - Doesn't alter file but the program acts as if the file had been altered

open(*name*, *flags***)**:

flags:

in	- Reads (file must exist)
out	- Empties (or creates) and writes
out | app	- Appends (creates if necessary)
in | out	- Reads and writes (file must exist)
in | out | trunk	- Empties, (or creates) reads and writes

Maths Functions

#include <**cmath**>
#include <**cstlib**>
using namespace std;

or

#include <**ctype.h**>
#include <**stdlib.h**>

- **abs(***x***)** - Returns magnitude of *x*
- **floor(***x***)** - Returns floating point value equal to smallest integer that represents x
- **ceil(***x***)** - Returns floating point value equal to largest integer that represents x
- **pow(***x*, *y***)** - Returns x^y, x and y are doubles
- **sqrt(***x***)** - Returns $\sqrt{x}$
- **atoi(***string***)** - Returns integer equal to *string*

Switch Statements

switch (*expression***)**
{
case *value1***:** *statements 1*
...
break;
...
...
...

case *valueN***:** *statements N*
...
break;

default: *statements default*
}

Enumerations & Type Definitions

- **enum** *New_Type_Name* **{***value1 = 0***,** *value2***,** *value3***, ... };**
- **typedef** *Known_Type_Definition New_Type_Definition***;**

If-Then-Else Shorthand

If *w = x* **then** *y* **else** *z*: **(***w* **==** *x***) ?** *y* **:** *z*

Array Declaration

Single Dimension:

Type Array_Name[*Size*];

i.e. int bigArray[100];

Multi Dimension:

Type Array_Name[*SizeD1*][*SizeD2*]…[*SizeDN*];

i.e. char page[30][100];

Dynamic:

Type* Array_Pointer_Name;
Array_Pointer_Name = new Type[Size]

i.e. int *number_ptr;
number_ptr = new int[10];

Multidimensional Dynamic:

Type** Array_Pointer_Name;
Array_Pointer_Name = new Type*[SizeD1];
for(int i = 0 ; i < SizeD1 ; i++)
Array_Pointer_Name = new Type[SizeD2];

Array Parameters

Single Dimension:

Type_Returned Function_Name(Type *Array_Name*[])

i.e. int sumArray(int bigArray[])

Multi Dimension:

Type_Returned Function_Name(Type *Array_Name*[][*SizeD2*])

i.e. int sumArray(int bigArray[][100])

Dynamic:

Type_Returned Function_Name(Type *Array_Name*[])

i.e. int sumArray(int bigArray[])

or

Type_Returned Function_Name(Type* *Array_Name*)

i.e. int sumArray(int* bigArray)

Multidimensional Dynamic:

Type_Returned Function_Name(Type *Array_Name*[][SizeD2])

i.e. int sumArray(int bigArray[][100])

or

Type_Returned Function_Name(Type** *Array_Name*)

i.e. int sumArray(int** bigArray)

To prevent arrays being altered by functions use **const**

i.e. int sumArray(const int bigArray[])

Array Arguments

Calling functions array arguments are passed without their size, even for multidimensional arrays.

i.e. cout << sumArray(bigArray);

Deleting Arrays

Dynamic:

```
delete []Array_Pointer_Name
```

i.e. delete []number_ptr;

Multidimensional Dynamic:

```
for(int i = 0 ; i < SizeD1 ; i++)
    delete []Array_Pointer_Name[i];
delete []Array_Pointer_Name;
```

MAX INDEX = (DECLARED SIZE – 1)

Vectors

```
#include <vector>
using namespace std;
```

or

```
#include <vector.h>
```

Creating Vectors — **Vector is a template**

vector<*Type***>** *Name*	- Creates vector of type *Type* and name *Name*
vector<*Type***>** *Name***(***OtherVector***)**	- Creates vector which is copy of OtherVector
vector<*Type***>** *Name***(***n***)**	- Creates vector which has *n* elements

e.g. *(functions and iterators explained below)*

```
vector<int> myVector;
...
list<int>::iterator myIterator;

for (myIterator = myVector.begin() ;
        myIterator != myVector.end() ; myIterator ++) {
   cout << *myIterator << ", ";  // Prints out numbers in vector
   }
```

Vector Functions

vector1.**swap(***vector2***)** or **swap(***vector1***,** *vector2***)**	- Swap data
*vector***[***index***]** or *vector*.**at(***index***)**	- Returns element at *index*
vector.**front()**	- Returns first element
vector.**back()**	- Returns last element
vector.**insert(***iteratorpos***,** *n***,** *elem***)**	- Inserts *n* copies of *elem* at iterator position *iteratorpos*
vector.**push_back(***elem***)**	- Appends a copy of *elem* at the end
vector.**pop_back(***elem***)**	- Removes last element and returns nothing
vector.**erase(***iteratorpos***)**	- Removes element at iterator position *iteratorpos*
vector.**erase(***beg***,** *end***)**	- Removes element in range *beg* to *end*
vector.**clear()**	- Removes all elements
==, !=, <, <=, >, >=	- Can be used for vector comparisons
†**stable_sort(***iteratrpos1***,** *iteratrpos2***)**	- Sorts elements between two iterator positions

† This requires #include <**algorithm**> or #include <**algo.h**>

Iterators

Creating Vector Iterators

vector< *type***>::iterator** *iteratorName*;

Iterator Functions

vector.**begin()**	- Returns random access iterator for first element
vector.**end()**	- Returns random access iterator for last element
vector.**rbegin()**	- Returns reverse iterator for first element
vector.**rend()**	- Returns reverse iterator for last element
*iteratorName******	- Returns element at iterator position
*iteratorName***++**	- Steps iterator one position forward
*iteratorName***--**	- Steps iterator one position backwards
iterator1Name **==** *iterator2Name*	- Compares the position of two iterators (also !=)
iterator1Name **=** *iterator2Name*	- Assigns the position of one iterators to another iterator

String Functions

```
#include <cctype>
using namespace std;
```

or

```
#include <ctype.h>
```

Character Functions

- **isalpha(***character***)** - True if *character* is an upper or lower case letter ('a' to 'z' or 'A' to 'Z')
- **toupper(***character***)** - Returns uppercase of *character*

```
#include <cstring>
using namespace std;
```

or

```
#include <string.h>
```

String Functions

- **strcpy(***string1, string2***)** - True if *character* is an upper or lower case letter ('a' to 'z' or 'A' to 'Z')
- **strlen(***string***)** - Returns string length excluding '\0'
- **strcat(***strDestination***,** *strSource***)** - Copies strings behaviour undefined for overflow.
- **strcmp(***string1***,** *string2***)** - <0 for string1 < string2
0 for string1 = string2
>0 for string1 > string2
- **strstr(***strToSearch***,** *strToSchFor***)** - Returns a pointer to the first occurrence of *strToSearchFor* in *strToSearch*, or NULL if *strToSearchFor* does not appear in string.

Pointers & References

A reference declaration:

```
int i = 3;
int &ri = i;
```

Reference to a constant:

```
const int &ri = i;
```

is equivalent to

```
int const &ri = i;
```

All references are constant therefore they always need to be initialised when they are defined.

A pointer declaration:

```
int i = 3;
int *pi = &i;
```

Pointer to a constant:

```
const int *pi = i;
```

is equivalent to

```
int const *pi = i;
```

Constant pointer:

```
int *const pi = i;
```

Constant pointers also have to be initialised as the memory address pointed to can not be changed.

Quick Sort

```
// Sorts elements in list from list[first] to list[last] into alphabetical order
void quickSort(char list[], int first, int last)
{
    int left = first; // Left sorting position (left arrow)
    int right = last; // Right sorting position (right arrow)
    int pivot = list[(left + right)/2];

    do  {
       while (list[right] > pivot)
          right--;
       while (list[left] < pivot)
          left++;
       if (left <= right)
          swap(list[left++], list[right--]);
    } while (right >= left);

    if (first < right)
       quickSort(list, first, right);
    if (left < last)
       quickSort(list, left, last);
}

// Swap the values of two parameters
void swap(char& first, char& second)
{
    char temp = first;
    first = second;
    second = temp;
}
```


Binary Search

```
// Search for element between list[first] and list[last] with value value
int binarySearch(char value, char list[], int first, int last)
{
    int midPoint = (first + last)/2;;

    if (first > last)
       return -1;
    else {
       if (list[midPoint] == value)
          return midPoint;
       else if (list[midPoint] > value)
          return binarySearch(value, list, first, midPoint - 1);
       else
          return binarySearch(value, list, midPoint + 1, last);
    }
}
```

Reverse Character Array

```
// Copies str1 into str2 backwards
void reverse(const char str1[], char str2[])
{
 int length = strlen(str1); // length of str1
 int str1Pos = 0; // Position in str1
 int str2Pos = 0; // Position in str2

 for(str2Pos = length-1 ; str1[str1Pos] ; str1Pos++){
   str2[str2Pos] = str1[str1Pos];
   str2Pos--;
 }

 str2[length] = '\0';
}
```

Sub-String

```
// Returns 1 if subStr is a sub-string of string, 0 otherwise
int subString(const char subStr[], const char string[])
{
  if(*string == '\0')
    return 0;
  else if(prefix(subStr, string))
    return 1;
  else
    return subString(subStr, string+1);
}

// Returns 1 if prefix is a prefix of string, 0 otherwise
int prefix(const char strPrefix[], const char string[])
{
  if(*strPrefix == '\0') {
    return 1;
  } else if(*strPrefix != *string)
    return 0;
  else
    return prefix(strPrefix+1, string+1);
}
```


Copy Alphabetic Characters

```
// Copies alphabetic characters from string to alphaString
void alphaCopy(const char string[], char alphaString[])
{
  int stringPos = 0; // String position
  int alphaStringPos = 0; // alphaString position

  while(string[stringPos]) {
    if(isalpha(string[stringPos])) {
      alphaString[alphaStringPos] = string[stringPos];
      alphaStringPos++;
    }
    stringPos++;
  }

  alphaString[alphaStringPos] = '\0';
}
```

GNU

- **Command Line**

g++ *filename.cpp* **-o** *executablename*

- **Single File (m*akefile*)**

executablename: *filename.cpp*

g++ -Wall -g *filename.cpp* **-o** *executablename*

[TAB]

- **Multiple Files (m*akefile*)**

executablename: *objectOneName.o objectTwoName.o*

g++ -Wall -g *objectOneName.o objectTwoName.o* **-o** *executablename*

objectOneName.o: *fileOneName.cpp fileOneName.h*

g++ -Wall -g -c *fileOneName.cpp*

objectTwoName.o: *fileTwoName.cpp fileTwoName.h*

g++ -Wall -g -c *fileTwoName.cpp*

clean:

rm -f *.o *executablename*

GDB

- **r** - Continue / resume execution
- **c** - Continue / resume execution
- **s** - Step until next line (will step into functions)
- **n** - Continue until next line (steps over functions)
- **finish** - Continue current function returns
- **info break** - List breakpoints
- **delete** *n* - Delete breakpoint number *n*
- **watch** - Set watch point, argument is an expression; debugger will stop execution when changes.
- **print** *variable* - Display current value of *variable*
- **where** - Shows stack trace
- **up** - Move to a higher stack frame
- **down** - Move to lower stack frame
- **quit** - Quits the debugger

Object Oriented

Good programmers always:

- Use *const* when ever possible, for methods & variables.
- Initialise all member variables in constructor's initialiser.
- Have minimum visibility for any member variables or functions.
- Echo back input data
- Use defensive programming
- Use Top Down Design
  - Split up program into simple easy tasks
  - There should be no set of three consecutive lines or more that is repeated anywhere in a program.
- Comment as you go along to keep track of what you are doing & avoid errors.
- Declare a copy constructor, an assignment operator and an =operator for classes with dynamically allocated memory, to avoid memory problems such as memory leaks or double deletion of dynamic memory.
- Prefer initialization to assignment in constructors, particularly when using templates as the type may be *const*.
- List members in an initialization list in the order in which they are declared, as this is the order in which they are initialized.

Static variable initialisers require a type:

float *Class_Name*::*Static_Variable* **=** 10.1**;**

When creating an object using the default constructor DO NOT use empty brackets:

*Class_Name Object_Instance***;**

Default constructor used

Default values for parameters are only added to the function signature.

Copy constructors called implicitly by the compiler when ever an objects is used as pass-by-value parameters or the return type of a function. To avoid this pointers or references can be used. The copy constructor, the =operator, and the destructor are called the big three because experts say if you need any of them you need all three. If any of these is missing, the compiler will create it but it may not behave as you want. The copy constructor and the overloaded =operator that the compiler generates for you will work fine if all member variables are of predefined types such as *int* and *double*, but it may misbehave on classes that have class member variables. For any class that uses pointers and the new operator, it is safest to define your own copy constructor, overloaded =, and a destructor. When an object is copied using a copy constructor generated by the compiler the memory address of pointers are copied and NOT the value in the memory pointed to by the pointers. This can result in an objects dynamically

allocated memory being deleted when a copy of the original object's destructor is called.

If you declare a **constant object** you make all member variables const and remove non-const member functions from the object.

The base class of a derived class may be specified to be public, protected or private. The **base class access specifier** affects the extent to which the derived class may inherit from the derived class.

```
class Goldfish : public Animal {
    ...
};
```

Virtual Functions & Pointers

- Derived classes may *only* override virtual functions, but may overload any function.
- Non-virtual functions are bound statically
- Virtual functions *may* be bound dynamically. All other functions are bound statically.
- Virtual functions are bound according to the class of the object executing the function
- When a virtual function is called by a pointer, then the function is bound according to the class of the object pointed at by the pointer, and NOT according to the type of the pointer.
- Static binding results in faster programs. Dynamic binding allows for flexibility at run-time. Therefore programmers should only use dynamic binding only when absolutely necessary.

A constructor for a sub class (derived class) always calls the default constructor for its super class (base class). If a base class has a constructor then that constructor should be called through a base class initialiser, which executes the base class constructor before the body of the derived class constructor. Otherwise base class initialisers are implicitly introduced by the compiler, but the compiler always calls a default class initialiser (constructor).

You can only copy a derived class to a base class, and not the other way around.

```
Base_Class_Instance = Derived_Class_Instance
Derived_Class_Instance =/= Base_Class_Instance
```

Pointers to a derived class may be implicitly converted to pointers of a based class, and not the other way around. Derived class objects may appear wherever base class objects are expected i.e. derived classes can replace base classes.

Virtual functions may have different implementations in the derived classes. The keyword *virtual* indicates that a function is virtual. The *virtual* keyword is only added to the function signature.
A virtual function may have no implementation. If so it only serves to define an interface provided by all derived classes. This type of virtual function is called a pure virtual function and is indicated by adding =0 before the ; in the function signature.

```
virtual void draw() = 0 ;
```

Virtual Destructors should always be used when using dynamic binding. Non virtual destructors are always called using the static type

and NOT the dynamic type. This means that a derived class's destructors might not be called. If virtual destructors are used the destructor is called according to the dynamic type.

Overloaded Operators

Overloaded operators are of two types, global and local. Both types are used identically.

Binary operators may be defined through global operators with two arguments, or through member (local) functions with one argument.

Unary (prefix) operators may be defined through global operators with one argument, or through member (local) functions without any arguments.

Overloaded operator functions can only be defined to accept an argument of a class or enumeration type. A pointer doesn't count.

friend *Class_Name* **&operator++(***Class_Name* **&***obj_name***)**

or

Class_Name **&operator++()**

The << operator should always be overloaded globally:

friend *ostream&* **operator<<(***ostream& o***,** *Class_Name obj_name***)**

Constant Methods

Labelling a method as being const means that it will not change any of the data members within the object. If a method is declared constant it can not change any variable values, therefore data members can never

be l-values. A constant method is indicated by adding the const keyword before the ; in the function signature and between the) and { in the function definition.

```
friend ostream& operator<<(ostream& o, Class_Name obj_name) const {

    return ostream << obj_name.string_variable1;

}
```

Function Templates

The function definition and the function prototype (signature) for a function template are each prefaced with the following:

```
template<class Type_Parameter>
```

The prototype and definition are then the same as any ordinary function prototype and definition, except that the *Type_Parameter* can be used in place of a type.

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3);
```

The definition for this function template might be:

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3){
    cout  << stuff1 << endl
          << stuff2 << endl
          << stuff3 << endl;
}
```

The function template given in this example is equivalent to having one function prototype and one function definition for each possible type name. The type name is substituted for the type parameter. For instance, consider the following function call:

```
show_stuff(2, 3.3, 4.4);
```

When this function call is executed, the compiler uses the function definition obtained by replacing T with the type name *double*. A separate definition will be produced for each different type for which you use the template, but not for any types you do not use. Only one definition is generated for a specific type regardless of the number of times you use the template.

Class Templates Syntax

The class definition and the definitions of the member functions are prefaced with the following:

```
template<class Type_Parameter>
```

The class and member function definitions are then the same as any ordinary class, except that the *Type_Parameter* can be used in place of a type.

```
template<class T>
class Pair {
public:
    Pair();
    Pair(T first_value, T second_value);
        ...
};
```


Member functions and overloaded operators are then defined as function templates.

```
template<class T>
Pair<T>::Pair(T first_value, T second_value) {
    ...
}
```

Using Templates

You can specialize (use) a class template by giving a type argument to the class name:

```
Pair<int>
```

The specialized class name can then be used just like any class name. It can be used to declare objects or to specify the type of a formal parameter.

```
Pair<int> pair_instance1, pair_instance2;
```