UNIT 5

C++ Standard Input Output Stream

C++ programming language uses the concept of streams to perform input and output operations using the keyboard and to display information on the monitor of the computer.

What is a Stream?

A stream is an object where a program can either insert or extract characters to or from it. The standard input and output stream objects of C++ are declared in the header file *iostream*.

Standard Input Stream

- Generally, the device used for input is the keyboard. For inputting, the keyword *cin* is used, which is an object.
- The overloaded operator of extraction, >>, is used on the standard input stream, in this case: *cin* stream.

Syntax for using the standard input stream is cin *followed by the operator* >> *followed by the* variable *that stores the data extracted from the stream.*

For example:

```
int prog; cin
>> prog;
```

In the example above, the variable *prog* is declared as an integer type variable. The next statement is the *cin* statement. The *cin* statement waits for input from the user‘s keyboard that is then stored in the integer variable *prog*.

The input stream *cin* wait before proceeding for processing or storing the value. This duration is dependent on the user pressing the RETURN key on the keyboard. The input stream *cin* waits for the user to press the RETURN key then begins to process the command. It is also possible to request input for more than one variable in a single input stream statement. A single *cin* statement is as follows:

```
cin >> x >> y;
```

is the same as:

```
cin >> x;
cin >> y;
```

In both of the above cases, two values are input by the user, one value for the variable x and another value for the variable y.

```
// This is a sample program        This is a comment Statement

#include <iostream.h>                Header File Inclusion Statement
void main()
{
int sample, example;
cin >> sample;
cin >> example;
}
```

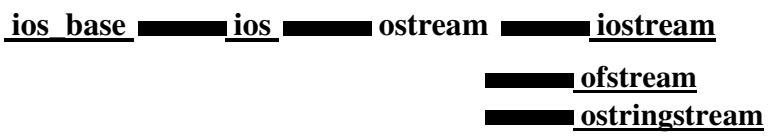
In the above example, two integer variables are input with values.

- The programmer can produce input of any data type.
- It is also possible to input strings in C++ program using *cin*. This is performed using the same procedures.
- The vital point to note is *cin* stops when it encounters a blank space.
- When using a cin, it is possible to produce only one word.

If a user wants to input a sentence, then the above approach would be tiresome. For this purpose, there is a function in C++ called *getline*.

Ostream

Output Stream



ostream objects are stream objects used to write and format output as sequences of characters. Specific members are provided to perform these output operations, which can be divided in two main groups:

- Formatted output
  - These member functions interpret and format the data to be written as sequences of characters.
  - These type of operation is performed using member and global functions that overload the insertion operator
- Unformatted output

- Most of the other member functions of the ostream class are used to perform unformatted output operations, i.e. output operations that write the data as it is, with no formatting adaptations.
- These member functions can write a determined number of characters to the output character sequence (put, write) and manipulate the *put pointer* (seekp, tellp).

Additionaly, a member function exists to synchronize the stream with the associated external destination of characters: sync.

The standard objects cout, cerr and clog are instantiations of this class.

The class inherits all the internal fields from its parent classes ios_base and ios:

Formatting information

- **format flags:** a set of internal indicators describing how certain input/output operations shall be interpreted or generated. The state of these indicators can be obtained or modified by calling the members flags, *setf* and *unsetf,* or by using manipulators.
- **field width:** describes the width of the next element to be output. This value can be obtained/modified by calling the member function width or parameterized manipulator *setw.*
- **display precision:** describes the decimal precision to be used to output floating-point values. This value can be obtained / modified by calling member precision or parameterized manipulator *setprecision.*
- **fill character:** character used to pad a field up to the *field width*. It can be obtained or modified by calling member function fill or parameterized manipulator *setfill*.
- **locale object:** describes the localization properties to be considered when formatting i/o operations. The locale object used can be obtained calling member *getloc* and modified using member imbue.

State information

- **error state:** internal indicator reflecting the integrity and current error state of the stream. The current object may be obtained by calling *rdstate* and can be modified by calling *clear* and *setstate*. Individual values may be obtained by calling *good*, *eof*, *fail* and *bad*.
- **exception mask:** internal exception status indicator. Its value can be retrieved/modified by calling member *exceptions*.

Formatted output:

operator<< Insert data with format (public member function)

Unformatted output:

put Put character (public member function)
write Write block of data (public member function)

Positioning:

tellp Get position of put pointer (public member function)
seekp Set position of put pointer (public member function)

Synchronization:

flush Flush output stream buffer (public member function)

Prefix/Suffix:

sentry Perform exception safe prefix/suffix operations (public member classes)

I/O Manipulators

What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by h rgamr‘ hieo display.

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

- hsmnpltrhstesm ucinlt ste_\‘ eln hrce.

For example:

```
cout << Exforsys << endl;
cout << Training;
```

produces the output:

Exforsys
Training

setw Manipulator:

This manipulator sets the minimum field width on output. The syntax is:

setw(x)

Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is <iomanip.h>

```
#include <iostream.h>
#include <iomanip.h>


void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout <st()< Efry‖< ew2)< Vle‖< nl
 ew8 <–1357 <st(0< 1< n
 ew8 <–1357 <st(0< 2< n
 ew8 <–1357 <st(0< 3< end;
}
```

The output of the above example
is: setw(8) setw(20)
Exforsys Values
E1234567 12345
S1234567 23456
A1234567 7892

setfill Manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

```
#include   <iostream.h>
#include   <iomanip.h>
void main( )
{
cout << setw(10) << setfill($‘) << 50 << 33 << endl;
}
```

The output of the above program is

$$$$$$$$5033

This is because the setw sets 10 width for the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

setprecision Manipulator:

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.

- The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.
- The keyword scientific before the setprecision manipulator prints the floating point number in scientific notation.

```
#include   <iostream.h>
#include   <iomanip.h>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x <<
endl; cout << sceintific << x << endl;
}
```

The above gives ouput as:

0.100
1.000000e-001

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

File I/O with Streams

Many real-life problems handle large volumes of data, therefore we need to use some devices such as floppy disk or hard disk to store the data.

The data is stored in these devices using the concept of files. A file is a collection of related data stored in a particular area on the disk.

Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

- Data transfer between the console unit and the program.
- Data transfer between the program and a disk file.
- The input/output system of C++ handles file operations which are very much similar to the console input and output operations.

It uses file streams as an interface between the programs and the files.

The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream.

In other words, the input stream extracts or reads data from the file and the output stream inserts or writes data to the file.

Classes for the file stream operations

The I/O system of C++ contains a set of classes that define the file handling methods.

These include ifstream, ofstream and fstream.

These classes, designed to manage the disk files, are declared in fstream.h and therefore we must include this file in any program that uses files.

Details of some useful classes :

filebuf

Its purpose is to set the file buffer to read and write. Contains openprot constant used in the open() of the filestream classes. Also contains close() and open() as member functions.

fstreambase

Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream classes. Contains open() and close() functions.

ifstream

Provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() functions from istream.

ofstream

Provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp(), and write() functions from ostream.

fstream

Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream.

The ifstream, ofstream and fstream classes are declared in the file fstream.h
The istream and ostream classes are also included in the fstream.h file.

Opening and closing a file

For opening a file, we must first create a file stream and than link it to the filename.

A filestream can be defined using the classes ifstream, ofstream, and fstream that are contained in the header file fstream.h

A file can be opened in two ways:

- Using the constructor function of the class. This method is useful when we open only one file in the stream.
- Using the member function open() of the class. This method is used when we want to manage multiple files using one stream.

Using Constructor

Create a file stream object to manage the stream using the appropriate class. That is, the class ofstream is used to create the output stream and the class ifstream to create the input stream.

Initialize the file object with the desired filename, e.g.:

ftemotie–apett)

The above statement creates an object outfile of class ofstream that manages the output stream. This statement also opens the file sample.txt and attaches it to the output stream for writing.
Similarly, the statement declared in as an ifstream object and attaches to the file sml.x‖frraig

ftemifl(sml.x‖;

Program: Writing and reading data into file, using constructors

```
# include <fstream.h>
void main()
{
ftemotie–apett) /cet iefrotu hrc  a‘
int i = 12;
float f = 4356.15;
hrar   hlo;
outfile << ch << endl <<< endl << f << endl << arr; //send the data to file
outfile.close();

ftemifl(sml.x‖;


infile >> ch >> i >> f >> arr; // read data from file

cout << ch << i << f << arr; // send data to screen
}
```

To write data into the file, character by character.

```
#include<fstream.h>
#include<string.h>
void main()
{
hrsr]–+ sspre fC ti nojc-oriented / rgamn agae‖

ftemotie–ape.x‖;/ pntefl nwiemd

for(int i = 0; i < strlen(str); i++)
outfile.put(str[i]); // write data into the file, character by character.
}
```

Writing and reading Objects of a class :

So far we have done I/O of basic data types. Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing and reading from the disk files objects directly.

The binary input and output functions read() and write() are designed to do exactly this job.

The write() function is used to write the object of a class into the specified file and

read() function is used to read the object of the class from the file.

Both these functions take two arguments:
 address of object to be written.
 size of the object.

The address of the object must be cast to the type pointer to char.

One important point to remember is that only data members are written to the disk file and the member functions are not.

Writing an object into the file

```
#include<fstream.h>
class Person
{
private:
char name[40];
int age; public:

void getData()
{
ot< \n Etrnm:;cn> ae ot< \nEtrae‖ i >ae
}
} ; // End of the class definition

void main()
{
Person per ; // Define an object of Person class

per.getData(); // Enter the values to the data members of the class.

ftemotie–esntt) /Oe h iei uptmd

outfile.write((char*)&per, sizeof(per)); // Write the object into the file
}
```

fstream object can be used for both input & output.

In the open() function we include several mode bits to specify certain aspects of the file object.
app -> To preserve whatever was in the file before. Whatever we write to the file will be appended to the existing contents.

We use in and out because we want to perform both input and output on the file.

eof() is a member function of ios class. It returns a nonzero value if EOF is encountered and a zero otherwise.

Parameters of open() function ios::app
Append to end of the file ios::ate Go to
end of the file on opening ios::in Open
file for reading only
ios::nocreate Open fails if the file does not exist
ios::noreplace Open fails if the file already
exists ios::out Open file for writing only
ios::trunc Delete contents of the file if it exists

File pointers and their manipulations

Each file has two associated pointers known as the file pointers.
One of them is called the **input pointer or get pointer**.
Other is called the **output pointer or put pointer.**

We can use these pointers to move through the files while reading or writing.

The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

Functions for manipulation of file pointers
seekg() Moves get pointer (input) to a specified location.
seekp() Moves put pointer (output) to a specified location.
tellg() Gives the current position of the get pointer.
tellp() Gives the current position of the put pointer.

infile.seekg(10);

Moves the file pointer to the byte number 10. The bytes in a file are numbered beginning from zero.Thus, the pointer will be pointing to the 11^{th} byte in the file.

Object Serialization

Object serialization consists of saving the values that are part of an object, mostly the value gotten from declaring a variable of a class. AT the current standard, C++ doesn‘t inherently support object serialization. To perform this type of operation, you can use a technique known as binary serialization.

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in sub-scopes, each one with its own name.

The format of namespaces is:

namespace identifier
{
entities
}

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
```

```
  cout << second::var <<
  endl; return 0;
 }
```

Output:

5
3.1416

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen.

using

The keyword using is used to introduce a name from a namespace into the current declarative region. For example:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

Output:

5
2.7183
10
3.1416

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

Output:

```
5
10
3.1416
2.7183
```

In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers where referring to their declarations in namespace first.

using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
// using namespace example
#include <iostream>
```

```
using namespace std;

namespace first
{
 int x = 5;
}

namespace second
{
 double x = 3.1416;
}

int main () {
 {
  using namespace first;
  cout << x << endl;
 }
 {
  using namespace second;
  cout << x << endl;
 }
 return 0;
}
```

Output:

5
3.1416

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

namespace new_name = current_name;

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

Advantages:

- A namespace defines a new scope.
- Members of a namespace are said to have namespace scope.

They provide a way to avoid name collisions (of variables, types, classes or functions) without some of the restrictions imposed by the use of classes, and without the inconvenience of handling nested classes

Standard Template Library

The Standard Template Libraries (STL's) are a set of C++ template classes to provide common programming data structures and functions such as doubly linked lists (list), paired arrays (map), expandable arrays (vector), large string storage and manipulation (rope), etc.

STL can be categorized into the following groupings:

- Container classes:
  - Sequences:
    - **vector**: (this tutorial) Dynamic array of variables, struct or objects. Insert data at the end.
    - (also see the YoLinux.com tutorial on using and STL list and boost ptr_list to manage pointers.)
    - **deque**: Array which supports insertion/removal of elements at beginning or end of array
    - **list**: (this tutorial) Linked list of variables, struct or objects. Insert/remove anywhere.
  - Associative Containers:
    - **set** (duplicate data not allowed in set), **multiset** (duplication allowed): Collection of ordered data in a balanced binary tree structure. Fast search.
    - **map** (unique keys), **multimap** (duplicate keys allowed): Associative key-value pair held in balanced binary tree structure.
  - Container adapters:
    - **stack** LIFO
    - **queue** FIFO
    - **priority_queue** returns element with highest priority.
  - String:
    - **string**: Character strings and manipulation
    - **rope**: String storage and manipulation
  - **bitset**: Contains a more intuitive method of storing and manipulating bits.
- Operations/Utilities:
  - **iterator**: (examples in this tutorial) STL class to represent position in an STL container. An iterator is declared to be associated with a single container class type.
  - **algorithm**: Routines to find, count, sort, search, ... elements in container classes
  - **auto_ptr**: Class to manage memory pointers and avoid memory leaks.

ANSI String Class

The ANSI string class implements a first-class character string data type that avoids many problems associated with simple character arrays (C-style strings). You can define a string object very simply, as shown in the following example:

```
#include <string>
using namespace std;
...
string first_name = Bjarne;
string last_name;

last_name = Stroustrup;

string names = first_name +   + last_name;
cout << names << endl;

names = last_name + ,  + first_name;
cout << names << endl;
```

Member functions

The string class defines many member functions. A few of the basic ones are described below:

Initialization (constructor)	A string object may defined without an initializing value, in which case its initial value is an empty string (zero length, no characters):
	string str1;
	
	A string object may also be initialized with
	
	• a string expression:
	• string str2 = str1;
	• string str3 = str1 + str2;
	string str4 (str2); // Alternate form
	
	• a character string literal:
	• string str4 = Hello there;
	string str5 (Goodbye); // Alternate form
	
	• a single character
	Unfortunately, the expected methods don‘t work:
	• string str6 = A‘; // Incorrect
	string str7 (A‘); // Also incorrect

	Instead, we must use a special form with two values: string str7 (1,'A'); // Correct The two values are the desired length of the string and a character to fill the string with. In this case, we are asking for a string of length one, filled with the character **A**. • a substring of another string object: • string str8 = ABCDEFGHIJKL; • // Initialize str9 as CDEFG • // Starts at character 2 (C') • // with a length of 5 • // (or the rest of the string, if shorter) string str9 (str8,2,5);

length
 	
Size
	size_type length() const;
	size_type size() const;
	Both of these functions return the length (number of characters) of the string. The **size_type** return type is an unsigned integral type. (The type name usually must be scoped, as in **string::size_type**.) string str = Hello;
	string::size_type len;
	len = str.length(); // len ==
	5 len = str.size(); // len == 5

c_str	const char* c_str() const;
	For compatibility with older code, including some C++ library routines, it is sometimes necessary to convert a string object into a pointer to a null-terminated character array (C-style string). This function does the conversion. For example, you might open a file stream with a user-specified file name:
	string filename;
	cout << Enter file name: ;
	cin >> filename;
	ofstream outfile (filename.c_str());
	outfile << Data << endl;

insert	string& insert(size_type pos, const string& str);
	Inserts a string into the current string, starting at the specified position.
	string str11 = abcdefghi;
	string str12 = 0123;
	str11.insert (3,str12);
	cout << str11 << endl; // abc0123defghi

	str12.insert (1,XYZ); cout << str12 << endl; // 0XYZ123

Erase	string& erase(size_type pos=0, size_type n=npos);
	Delete a substring from the current string. The substring to be deleted starts as position **pos** and is **n** characters in length. If **n** is larger than the number of characters between **pos** and the end of the string, it doesn't do any harm. Thus, the default argument values cause deletion of the rest of the string if only a starting position is specified, and of the whole string if no arguments are specified. (The special value **string::npos** represents the maximum number of characters there can be in a string, and is thus one greater than the largest possible character position.)
	string str13 = abcdefghi;
	str12.erase (5,3);
	cout << str12 << endl; // abcdei

replace	string& replace(size_type pos, size_type n, const string& str);
	Delete a substring from the current string, and replace it with another string. The substring to be deleted is specified in the same way as in **erase**, except that there are no default values for **pos** and **n**.
	string str14 = abcdefghi;
	string str15 = XYZ;
	str14.replace (4,2,str15);
	cout << str14 << endl; // abcdXYZghi
	
	Note: if you want to replace only a single character at a time, you should use the subscript operator ([]) instead.

find
 	
Rfind
	size_type find (const string& str, size_type pos=0) const;
	size_type find (char ch, size_type pos=0) const;
	size_type rfind (const string& str, size_type pos=npos) const;
	size_type rfind (char ch, size_type pos=npos) const;
	The **find** function searches for the first occurrence of the substring **str** (or the character **ch**) in the current string, starting at position pos. If found, return the position of the first character in the matching substring. If not found, return the value **string::npos**. The member function **rfind** does the same thing, but returns the position of the last occurrence of the specified string or character, searching backwards from **pos**.
	string str16 = abcdefghi;
	string str17 = def;
	string::size_type pos = str16.find (str17,0);
	cout << pos << endl; // 3
	pos = str16.find (AB,0);
	if (pos == string::npos) cout << Not found << endl;

find_first_of
find_first_not_of
find_last_of
find_last_not_of
	size_type find_first_of (const string& str, size_type pos=0) const; size_type find_first_not_of (const string& str, size_type pos=0) const;
	size_type find_last_of (const string& str, size_type pos=npos) const; size_type find_last_not_of (const string& str, size_type pos=npos) const;
	Search for the first/last occurrence of a character that appears or does not appear in the **str** argument. If found, return the position of the character that satisfies the search condition; otherwise, return **string::npos**. The **pos** argument specifies the starting position for the search, which proceeds toward the end of the string (for first searches) or toward the beginning of the string (for last searches); note that the default values cause the whole string to be searched.
	string str20 = Hello there;
	string str21 = aeiou;
	pos = str20.find_first_of (str21, 0); cout << pos << endl; // 1
	pos = str20.find_last_not_of (eHhlort); cout << pos << endl; // 5

substr	string substr (size_type pos, size_type n) const; Returns a substring of the current string, starting at position pos and of length n: string str18 = abcdefghi string str19 = str18.substr (6,2); cout << str19 << endl; // gh Note: if you want to retrieve only a single character at a time, you should use the subscript operator ([]) instead.
begin **End**	iterator begin(); const_iterator begin() const; iterator end(); const_iterator end() const; Returns an iterator that specifies the position of the first character in the string (**begin**) or the position that is just beyond the last character in the string (**end**). Iterators are special objects that are used in many STL algorithms.

Non-member functions and algorithms

In addition to member functions of the **string** class, some non-member functions and STL algorithms can be used with strings; some common examples are:

getline	istream& getline (istream& is, string& str, char delim = \n‘);
	Reads characters from an input stream into a string, stopping when one of the following things happens:
	• An end-of-file condition occurs on the input stream
	• When the maximum number of characters that can fit into a string have been read
	• When a character read in from the string is equal to the specified delimiter (**newline** is the default delimiter); the delimiter character is removed from the input stream, but not appended to the string.
	The return value is a reference to the input stream. If the stream is tested as a logical value (as in an **if** or **while**), it is equivalent to **true** if the read was successful and **false** otherwise (e.g., end of file). [Note: some libraries do not implement **getline** correctly for use with keyboard input, requiring that an extra carriage return be entered to terminate the read. Some patches are available.]
	The most common use of this function is to do line by line reads from a file. Remember that the normal extraction operator (>>) stops on white space, not necessarily the end of an input line. The **getline** function can read lines of text with embedded spaces.
	vector<string>
	vec1; string line;
	vec1.clear();
	ifstream infile (stl2in.txt);
	while (getline(infile,line,‘\n‘))
	{
	vec1.push_back (line);
	}

transform	OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op); Iterate through a container (here, a **string**) from **first** to just before **last**, applying the operation **op** to each container element and storing the results through the **result** iterator, which is incremented after each value is stored. This STL algorithm (from the **<algorithm>** library) is a little tricky, but simple uses are easy. For example, we can iterate through the characters in a string, modifying them in some way, and returning the modified characters back to their original positions. In this case, we set the **result** iterator to specify the beginning of the string. A common application is to convert a string to upper or lower case.

	string str22 = This IS a MiXed CaSE stRINg;
	transform (str22.begin(),str22.end(), str22.begin(), tolower);
	cout << [<< str22 <<] << endl; // [this is a mixed case string]
	
	Note that the **result** iterator must specify a destination that is large enough to accept all the modified values; here it is not a problem, since we're putting them back in the same positions. The **tolower** function (along with **toupper**, **isdigit**, and other useful stuff) is in the **<cctype>** library; for more general case conversions, take a look at the **<locale>** library, which is beyond the scope of the present discussion.

Operators

A number of C++ operators also work with string objects:

=	The assignment operator may be used in several ways:
	
	• Assigning one string object's value to another string object
	• string string_one = Hello;
	• string string_two;
	string_two = string_one;
	
	• Assigning a C++ string literal to a string object
	• string string_three;
	string_three = Goodbye;
	
	• Assigning a single character (char) to a string object
	• string string_four;
	• char ch = A';
	• string_four = ch;
	string_four = Z';

+	The plus operator concatenates:
	
	• two string objects
	• string str1 = Hello ;
	• string str2 = there;
	string str3 = str1 + str2; // Hello there
	
	• a string object and a character string literal
	• string str1 = Hello ;
	string str4 = str1 + there;
	
	• a string object and a single character

	• string str5 = The End; string str6 = str5 + !‘;
+=	The += operator combines the above assignment and concatenation operations in the way that you would expect, with a string object, a string literal, or a single character as the value on the right-hand side of the operator. string str1 = Hello ; str1 += there;
== **!=** **<** **>** **<=** **>=**	The comparison operators return a **bool** (true/false) value indicating whether the specified relationship exists between the two operands. The operands may be: • two string objects • a string object and a character string literal
<<	The insertion operator writes the value of a string object to an output stream (e.g., **cout**). string str1 = Hello there; cout << str1 << endl;
>>	The extraction operator reads a character string from an input stream and assigns the value to a string object. string str1; cin >> str1;

[ ]
 	
(subscript)
	The subscript operator accesses one character in a string, for inspection or modification:
	
	string str10 = abcdefghi;
	char ch = str10[3];
	cout << ch << endl; // d‘
	str10[5] = X‘;
	cout << str10 << endl; // abcdeXghi