Lập trình

**C++**

# FILES AND STREAMS

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

|  |  |
| --- | --- |
| **Sr.No** | **Data Type & Description** |
| 1 | **ofstream**  This data type represents the output file stream and is used to create files and to write information to files. |
| 2 | **ifstream**  This data type represents the input file stream and is used to read information from files. |
| 3 | **fstream**  This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. |

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

## **Opening a File**

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char \*filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

|  |  |
| --- | --- |
| **Sr.No** | **Mode Flag & Description** |
| 1 | **ios::app**  Append mode. All output to that file to be appended to the end. |
| 2 | **ios::ate**  Open a file for output and move the read/write control to the end of the file. |
| 3 | **ios::in**  Open a file for reading. |
| 4 | **ios::out**  Open a file for writing. |
| 5 | **ios::trunc**  If the file already exists, its contents will be truncated before opening the file. |

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

Similar way, you can open a file for reading and writing purpose as follows −

fstream afile;

afile.open("file.dat", ios::out | ios::in );

## **Closing a File**

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void close();

## **Writing to a File**

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

## **Reading from a File**

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

## **Read and Write Example**

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

[Live Demo](http://tpcg.io/MLhc7C)

#include <fstream>

#include <iostream>

using namespace std;

int main () {

char data[100];

// open a file in write mode.

ofstream outfile;

outfile.open("afile.dat");

cout << "Writing to the file" << endl;

cout << "Enter your name: ";

cin.getline(data, 100);

// write inputted data into the file.

outfile << data << endl;

cout << "Enter your age: ";

cin >> data;

cin.ignore();

// again write inputted data into the file.

outfile << data << endl;

// close the opened file.

outfile.close();

// open a file in read mode.

ifstream infile;

infile.open("afile.dat");

cout << "Reading from the file" << endl;

infile >> data;

// write the data at the screen.

cout << data << endl;

// again read the data from the file and display it.

infile >> data;

cout << data << endl;

// close the opened file.

infile.close();

return 0;

}

When the above code is compiled and executed, it produces the following sample input and output −

$./a.out

Writing to the file

Enter your name: Zara

Enter your age: 9

Reading from the file

Zara

9

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

## **File Position Pointers**

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg( n );

// position n bytes forward in fileObject

fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject

fileObject.seekg( n, ios::end );

// position at end of fileObject

fileObject.seekg( 0, ios::end );

# EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

* **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
* **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

## **Throwing Exceptions**

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

## **Catching Exceptions**

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

try {

// protected code

} catch( ExceptionName e ) {

// code to handle ExceptionName exception

}

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

try {

// protected code

} catch(...) {

// code to handle any exception

}

The following is an example, which throws a division by zero exception and we catch it in catch block.

[Live Demo](http://tpcg.io/Nuo9hc)

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

} catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

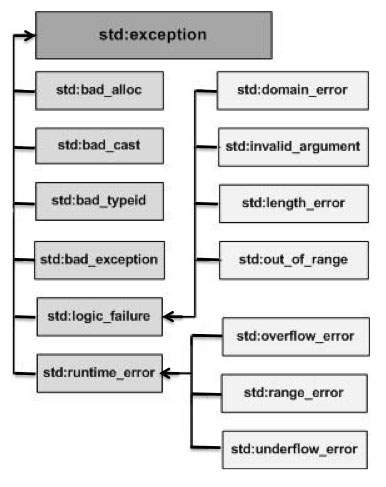
}

Because we are raising an exception of type **const char\***, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result −

Division by zero condition!

## **C++ Standard Exceptions**

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below −



Here is the small description of each exception mentioned in the above hierarchy −

|  |  |
| --- | --- |
| **Sr.No** | **Exception & Description** |
| 1 | **std::exception**  An exception and parent class of all the standard C++ exceptions. |
| 2 | **std::bad\_alloc**  This can be thrown by **new**. |
| 3 | **std::bad\_cast**  This can be thrown by **dynamic\_cast**. |
| 4 | **std::bad\_exception**  This is useful device to handle unexpected exceptions in a C++ program. |
| 5 | **std::bad\_typeid**  This can be thrown by **typeid**. |
| 6 | **std::logic\_error**  An exception that theoretically can be detected by reading the code. |
| 7 | **std::domain\_error**  This is an exception thrown when a mathematically invalid domain is used. |
| 8 | **std::invalid\_argument**  This is thrown due to invalid arguments. |
| 9 | **std::length\_error**  This is thrown when a too big std::string is created. |
| 10 | **std::out\_of\_range**  This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[](). |
| 11 | **std::runtime\_error**  An exception that theoretically cannot be detected by reading the code. |
| 12 | **std::overflow\_error**  This is thrown if a mathematical overflow occurs. |
| 13 | **std::range\_error**  This is occurred when you try to store a value which is out of range. |
| 14 | **std::underflow\_error**  This is thrown if a mathematical underflow occurs. |

## **Define New Exceptions**

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −

[Live Demo](http://tpcg.io/FUdUJO)

#include <iostream>

#include <exception>

using namespace std;

struct MyException : public exception {

const char \* what () const throw () {

return "C++ Exception";

}

};

int main() {

try {

throw MyException();

} catch(MyException& e) {

std::cout << "MyException caught" << std::endl;

std::cout << e.what() << std::endl;

} catch(std::exception& e) {

//Other errors

}

}

This would produce the following result −

MyException caught

C++ Exception

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

# DINAMIC MEMORY

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

* **The stack** − All variables declared inside the function will take up memory from the stack.
* **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

## **new and delete Operators**

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new**operator with the following statements −

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

double\* pvalue = NULL;

if( !(pvalue = new double )) {

cout << "Error: out of memory." <<endl;

exit(1);

}

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows −

delete pvalue; // Release memory pointed to by pvalue

Let us put above concepts and form the following example to show how ‘new’ and ‘delete’ work −

[Live Demo](http://tpcg.io/YFq73r)

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}

If we compile and run above code, this would produce the following result −

Value of pvalue : 29495

## **Dynamic Memory Allocation for Arrays**

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

char\* pvalue = NULL; // Pointer initialized with null

pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this −

delete [] pvalue; // Delete array pointed to by pvalue

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

double\*\* pvalue = NULL; // Pointer initialized with null

pvalue = new double [3][4]; // Allocate memory for a 3x4 array

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

delete [] pvalue; // Delete array pointed to by pvalue

## **Dynamic Memory Allocation for Objects**

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

[Live Demo](http://tpcg.io/qxQQ36)

#include <iostream>

using namespace std;

class Box {

public:

Box() {

cout << "Constructor called!" <<endl;

}

~Box() {

cout << "Destructor called!" <<endl;

}

};

int main() {

Box\* myBoxArray = new Box[4];

delete [] myBoxArray; // Delete array

return 0;

}

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

Constructor called!

Constructor called!

Constructor called!

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!

# NAMESPACES

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother’s or father’s name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

## **Defining a Namespace**

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

namespace namespace\_name {

// code declarations

}

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows −

name::code; // code could be variable or function.

Let us see how namespace scope the entities including variable and functions −

[Live Demo](http://tpcg.io/0xMQ1K)

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

int main () {

// Calls function from first name space.

first\_space::func();

// Calls function from second name space.

second\_space::func();

return 0;

}

If we compile and run above code, this would produce the following result −

Inside first\_space

Inside second\_space

## **The using directive**

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code −

[Live Demo](http://tpcg.io/8ggeE3)

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main () {

// This calls function from first name space.

func();

return 0;

}

If we compile and run above code, this would produce the following result −

Inside first\_space

The ‘using’ directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows −

using std::cout;

Subsequent code can refer to cout without prepending the namespace, but other items in the **std**namespace will still need to be explicit as follows −

[Live Demo](http://tpcg.io/J1UkeN)

#include <iostream>

using std::cout;

int main () {

cout << "std::endl is used with std!" << std::endl;

return 0;

}

If we compile and run above code, this would produce the following result −

std::endl is used with std!

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

## **Discontiguous Namespaces**

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one −

namespace namespace\_name {

// code declarations

}

## **Nested Namespaces**

Namespaces can be nested where you can define one namespace inside another name space as follows −

namespace namespace\_name1 {

// code declarations

namespace namespace\_name2 {

// code declarations

}

}

You can access members of nested namespace by using resolution operators as follows −

// to access members of namespace\_name2

using namespace namespace\_name1::namespace\_name2;

// to access members of namespace:name1

using namespace namespace\_name1;

In the above statements if you are using namespace\_name1, then it will make elements of namespace\_name2 available in the scope as follows −

[Live Demo](http://tpcg.io/2ydw4V)

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

}

using namespace first\_space::second\_space;

int main () {

// This calls function from second name space.

func();

return 0;

}

If we compile and run above code, this would produce the following result −

Inside second\_space

# TEMPLATES

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work −

## **Function Template**

The general form of a template function definition is shown here −

template <class type> ret-type func-name(parameter list) {

// body of function

}

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values −

[Live Demo](http://tpcg.io/6cvVGA)

#include <iostream>

#include <string>

using namespace std;

template <typename T>

inline T const& Max (T const& a, T const& b) {

return a < b ? b:a;

}

int main () {

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

If we compile and run above code, this would produce the following result −

Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World

## **Class Template**

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −

template <class type> class class-name {

.

.

.

}

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack −

[Live Demo](http://tpcg.io/ixzLqn)

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace std;

template <class T>

class Stack {

private:

vector<T> elems; // elements

public:

void push(T const&); // push element

void pop(); // pop element

T top() const; // return top element

bool empty() const { // return true if empty.

return elems.empty();

}

};

template <class T>

void Stack<T>::push (T const& elem) {

// append copy of passed element

elems.push\_back(elem);

}

template <class T>

void Stack<T>::pop () {

if (elems.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

}

// remove last element

elems.pop\_back();

}

template <class T>

T Stack<T>::top () const {

if (elems.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

// return copy of last element

return elems.back();

}

int main() {

try {

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate int stack

intStack.push(7);

cout << intStack.top() <<endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

} catch (exception const& ex) {

cerr << "Exception: " << ex.what() <<endl;

return -1;

}

}

If we compile and run above code, this would produce the following result −

7

hello

Exception: Stack<>::pop(): empty stack

# PREPROCESSOR

The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

You already have seen a **#include** directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives −

## **The #define Preprocessor**

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is −

#define macro-name replacement-text

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −

#include <iostream>

using namespace std;

#define PI 3.14159

int main () {

cout << "Value of PI :" << PI << endl;

return 0;

}

Now, let us do the preprocessing of this code to see the result assuming we have the source code file. So let us compile it with -E option and redirect the result to test.p. Now, if you check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows −

$gcc -E test.cpp > test.p

...

int main () {

cout << "Value of PI :" << 3.14159 << endl;

return 0;

}

## **Function-Like Macros**

You can use #define to define a macro which will take argument as follows −

[Live Demo](http://tpcg.io/4pUj1I)

#include <iostream>

using namespace std;

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {

int i, j;

i = 100;

j = 30;

cout <<"The minimum is " << MIN(i, j) << endl;

return 0;

}

If we compile and run above code, this would produce the following result −

The minimum is 30

## **Conditional Compilation**

There are several directives, which can be used to compile selective portions of your program's source code. This process is called conditional compilation.

The conditional preprocessor construct is much like the ‘if’ selection structure. Consider the following preprocessor code −

#ifndef NULL

#define NULL 0

#endif

You can compile a program for debugging purpose. You can also turn on or off the debugging using a single macro as follows −

#ifdef DEBUG

cerr <<"Variable x = " << x << endl;

#endif

This causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statment to comment out a portion of the program as follows −

#if 0

code prevented from compiling

#endif

Let us try the following example −

[Live Demo](http://tpcg.io/r3qAkV)

#include <iostream>

using namespace std;

#define DEBUG

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {

int i, j;

i = 100;

j = 30;

#ifdef DEBUG

cerr <<"Trace: Inside main function" << endl;

#endif

#if 0

/\* This is commented part \*/

cout << MKSTR(HELLO C++) << endl;

#endif

cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG

cerr <<"Trace: Coming out of main function" << endl;

#endif

return 0;

}

If we compile and run above code, this would produce the following result −

The minimum is 30

Trace: Inside main function

Trace: Coming out of main function

## **The # and ## Operators**

The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition −

[Live Demo](http://tpcg.io/54XMO7)

#include <iostream>

using namespace std;

#define MKSTR( x ) #x

int main () {

cout << MKSTR(HELLO C++) << endl;

return 0;

}

If we compile and run above code, this would produce the following result −

HELLO C++

Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line −

cout << MKSTR(HELLO C++) << endl;

Above line will be turned into the following line −

cout << "HELLO C++" << endl;

The ## operator is used to concatenate two tokens. Here is an example −

#define CONCAT( x, y ) x ## y

When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.

[Live Demo](http://tpcg.io/xadPS5)

#include <iostream>

using namespace std;

#define concat(a, b) a ## b

int main() {

int xy = 100;

cout << concat(x, y);

return 0;

}

If we compile and run above code, this would produce the following result −

100

Let us see how it worked. It is simple to understand that the C++ preprocessor transforms −

cout << concat(x, y);

Above line will be transformed into the following line −

cout << xy;

## **Predefined C++ Macros**

C++ provides a number of predefined macros mentioned below −

|  |  |
| --- | --- |
| **Sr.No** | **Macro & Description** |
| 1 | **\_\_LINE\_\_**  This contains the current line number of the program when it is being compiled. |
| 2 | **\_\_FILE\_\_**  This contains the current file name of the program when it is being compiled. |
| 3 | **\_\_DATE\_\_**  This contains a string of the form month/day/year that is the date of the translation of the source file into object code. |
| 4 | **\_\_TIME\_\_**  This contains a string of the form hour:minute:second that is the time at which the program was compiled. |

Let us see an example for all the above macros −

[Live Demo](http://tpcg.io/uO9yHk)

#include <iostream>

using namespace std;

int main () {

cout << "Value of \_\_LINE\_\_ : " << \_\_LINE\_\_ << endl;

cout << "Value of \_\_FILE\_\_ : " << \_\_FILE\_\_ << endl;

cout << "Value of \_\_DATE\_\_ : " << \_\_DATE\_\_ << endl;

cout << "Value of \_\_TIME\_\_ : " << \_\_TIME\_\_ << endl;

return 0;

}

If we compile and run above code, this would produce the following result −

Value of \_\_LINE\_\_ : 6

Value of \_\_FILE\_\_ : test.cpp

Value of \_\_DATE\_\_ : Feb 28 2011

Value of \_\_TIME\_\_ : 18:52:48

# SIGNAL HANDLING

Signals are the interrupts delivered to a process by the operating system which can terminate a program prematurely. You can generate interrupts by pressing Ctrl+C on a UNIX, LINUX, Mac OS X or Windows system.

There are signals which can not be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal. These signals are defined in C++ header file <csignal>.

|  |  |
| --- | --- |
| **Sr.No** | **Signal & Description** |
| 1 | **SIGABRT**  Abnormal termination of the program, such as a call to **abort**. |
| 2 | **SIGFPE**  An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow. |
| 3 | **SIGILL**  Detection of an illegal instruction. |
| 4 | **SIGINT**  Receipt of an interactive attention signal. |
| 5 | **SIGSEGV**  An invalid access to storage. |
| 6 | **SIGTERM**  A termination request sent to the program. |

## **The signal() Function**

C++ signal-handling library provides function **signal** to trap unexpected events. Following is the syntax of the signal() function −

void (\*signal (int sig, void (\*func)(int)))(int);

Keeping it simple, this function receives two arguments: first argument as an integer which represents signal number and second argument as a pointer to the signal-handling function.

Let us write a simple C++ program where we will catch SIGINT signal using signal() function. Whatever signal you want to catch in your program, you must register that signal using **signal** function and associate it with a signal handler. Examine the following example −

#include <iostream>

#include <csignal>

using namespace std;

void signalHandler( int signum ) {

cout << "Interrupt signal (" << signum << ") received.\n";

// cleanup and close up stuff here

// terminate program

exit(signum);

}

int main () {

// register signal SIGINT and signal handler

signal(SIGINT, signalHandler);

while(1) {

cout << "Going to sleep...." << endl;

sleep(1);

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Going to sleep....

Going to sleep....

Going to sleep....

Now, press Ctrl+c to interrupt the program and you will see that your program will catch the signal and would come out by printing something as follows −

Going to sleep....

Going to sleep....

Going to sleep....

Interrupt signal (2) received.

## **The raise() Function**

You can generate signals by function **raise()**, which takes an integer signal number as an argument and has the following syntax.

int raise (signal sig);

Here, **sig** is the signal number to send any of the signals: SIGINT, SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM, SIGHUP. Following is the example where we raise a signal internally using raise() function as follows −

#include <iostream>

#include <csignal>

using namespace std;

void signalHandler( int signum ) {

cout << "Interrupt signal (" << signum << ") received.\n";

// cleanup and close up stuff here

// terminate program

exit(signum);

}

int main () {

int i = 0;

// register signal SIGINT and signal handler

signal(SIGINT, signalHandler);

while(++i) {

cout << "Going to sleep...." << endl;

if( i == 3 ) {

raise( SIGINT);

}

sleep(1);

}

return 0;

}

When the above code is compiled and executed, it produces the following result and would come out automatically −

Going to sleep....

Going to sleep....

Going to sleep....

Interrupt signal (2) received.

# MULTITHREADING

Multithreading is a specialized form of multitasking and a multitasking is the feature that allows your computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Before C++ 11, there is no built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

This tutorial assumes that you are working on Linux OS and we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

## **Creating Threads**

The following routine is used to create a POSIX thread −

#include <pthread.h>

pthread\_create (thread, attr, start\_routine, arg)

Here, **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters −

|  |  |
| --- | --- |
| **Sr.No** | **Parameter & Description** |
| 1 | **thread**  An opaque, unique identifier for the new thread returned by the subroutine. |
| 2 | **attr**  An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. |
| 3 | **start\_routine**  The C++ routine that the thread will execute once it is created. |
| 4 | **arg**  A single argument that may be passed to start\_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed. |

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

## **Terminating Threads**

There is following routine which we use to terminate a POSIX thread −

#include <pthread.h>

pthread\_exit (status)

Here **pthread\_exit** is used to explicitly exit a thread. Typically, the pthread\_exit() routine is called after a thread has completed its work and is no longer required to exist.

If main() finishes before the threads it has created, and exits with pthread\_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

**Example**

This simple example code creates 5 threads with the pthread\_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread\_exit().

#include <iostream>

#include <cstdlib>

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

void \*PrintHello(void \*threadid) {

long tid;

tid = (long)threadid;

cout << "Hello World! Thread ID, " << tid << endl;

pthread\_exit(NULL);

}

int main () {

pthread\_t threads[NUM\_THREADS];

int rc;

int i;

for( i = 0; i < NUM\_THREADS; i++ ) {

cout << "main() : creating thread, " << i << endl;

rc = pthread\_create(&threads[i], NULL, PrintHello, (void \*)i);

if (rc) {

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

Compile the following program using -lpthread library as follows −

$gcc test.cpp -lpthread

Now, execute your program which gives the following output −

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Hello World! Thread ID, 0

Hello World! Thread ID, 1

Hello World! Thread ID, 2

Hello World! Thread ID, 3

Hello World! Thread ID, 4

## **Passing Arguments to Threads**

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example −

#include <iostream>

#include <cstdlib>

#include <pthread.h>

using namespace std;

#define NUM\_THREADS 5

struct thread\_data {

int thread\_id;

char \*message;

};

void \*PrintHello(void \*threadarg) {

struct thread\_data \*my\_data;

my\_data = (struct thread\_data \*) threadarg;

cout << "Thread ID : " << my\_data->thread\_id ;

cout << " Message : " << my\_data->message << endl;

pthread\_exit(NULL);

}

int main () {

pthread\_t threads[NUM\_THREADS];

struct thread\_data td[NUM\_THREADS];

int rc;

int i;

for( i = 0; i < NUM\_THREADS; i++ ) {

cout <<"main() : creating thread, " << i << endl;

td[i].thread\_id = i;

td[i].message = "This is message";

rc = pthread\_create(&threads[i], NULL, PrintHello, (void \*)&td[i]);

if (rc) {

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

When the above code is compiled and executed, it produces the following result −

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Thread ID : 3 Message : This is message

Thread ID : 2 Message : This is message

Thread ID : 0 Message : This is message

Thread ID : 1 Message : This is message

Thread ID : 4 Message : This is message

## **Joining and Detaching Threads**

There are following two routines which we can use to join or detach threads −

pthread\_join (threadid, status)

pthread\_detach (threadid)

The pthread\_join() subroutine blocks the calling thread until the specified 'threadid' thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

This example demonstrates how to wait for thread completions by using the Pthread join routine.

#include <iostream>

#include <cstdlib>

#include <pthread.h>

#include <unistd.h>

using namespace std;

#define NUM\_THREADS 5

void \*wait(void \*t) {

int i;

long tid;

tid = (long)t;

sleep(1);

cout << "Sleeping in thread " << endl;

cout << "Thread with id : " << tid << " ...exiting " << endl;

pthread\_exit(NULL);

}

int main () {

int rc;

int i;

pthread\_t threads[NUM\_THREADS];

pthread\_attr\_t attr;

void \*status;

// Initialize and set thread joinable

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

for( i = 0; i < NUM\_THREADS; i++ ) {

cout << "main() : creating thread, " << i << endl;

rc = pthread\_create(&threads[i], &attr, wait, (void \*)i );

if (rc) {

cout << "Error:unable to create thread," << rc << endl;

exit(-1);

}

}

// free attribute and wait for the other threads

pthread\_attr\_destroy(&attr);

for( i = 0; i < NUM\_THREADS; i++ ) {

rc = pthread\_join(threads[i], &status);

if (rc) {

cout << "Error:unable to join," << rc << endl;

exit(-1);

}

cout << "Main: completed thread id :" << i ;

cout << " exiting with status :" << status << endl;

}

cout << "Main: program exiting." << endl;

pthread\_exit(NULL);

}

When the above code is compiled and executed, it produces the following result −

main() : creating thread, 0

main() : creating thread, 1

main() : creating thread, 2

main() : creating thread, 3

main() : creating thread, 4

Sleeping in thread

Thread with id : 0 .... exiting

Sleeping in thread

Thread with id : 1 .... exiting

Sleeping in thread

Thread with id : 2 .... exiting

Sleeping in thread

Thread with id : 3 .... exiting

Sleeping in thread

Thread with id : 4 .... exiting

Main: completed thread id :0 exiting with status :0

Main: completed thread id :1 exiting with status :0

Main: completed thread id :2 exiting with status :0

Main: completed thread id :3 exiting with status :0

Main: completed thread id :4 exiting with status :0

Main: program exiting.

# WEB PROGRAMMING

## **What is CGI?**

* The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script.
* The CGI specs are currently maintained by the NCSA and NCSA defines CGI is as follows −
* The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
* The current version is CGI/1.1 and CGI/1.2 is under progress.

## **Web Browsing**

To understand the concept of CGI, let's see what happens when we click a hyperlink to browse a particular web page or URL.

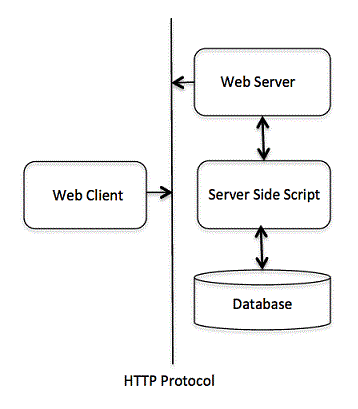
* Your browser contacts the HTTP web server and demand for the URL ie. filename.
* Web Server will parse the URL and will look for the filename. If it finds requested file then web server sends that file back to the browser otherwise sends an error message indicating that you have requested a wrong file.
* Web browser takes response from web server and displays either the received file or error message based on the received response.

However, it is possible to set up the HTTP server in such a way that whenever a file in a certain directory is requested, that file is not sent back; instead it is executed as a program, and produced output from the program is sent back to your browser to display.

The Common Gateway Interface (CGI) is a standard protocol for enabling applications (called CGI programs or CGI scripts) to interact with Web servers and with clients. These CGI programs can be a written in Python, PERL, Shell, C or C++ etc.

## **CGI Architecture Diagram**

The following simple program shows a simple architecture of CGI −



## **Web Server Configuration**

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI directory and by convention it is named as /var/www/cgi-bin. By convention CGI files will have extension as **.cgi**, though they are C++ executable.

By default, Apache Web Server is configured to run CGI programs in /var/www/cgi-bin. If you want to specify any other directory to run your CGI scripts, you can modify the following section in the httpd.conf file −

<Directory "/var/www/cgi-bin">

AllowOverride None

Options ExecCGI

Order allow,deny

Allow from all

</Directory>

<Directory "/var/www/cgi-bin">

Options All

</Directory>

Here, I assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell etc.

## **First CGI Program**

Consider the following C++ Program content −

#include <iostream>

using namespace std;

int main () {

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Hello World - First CGI Program</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<h2>Hello World! This is my first CGI program</h2>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Compile above code and name the executable as cplusplus.cgi. This file is being kept in /var/www/cgi-bin directory and it has following content. Before running your CGI program make sure you have change mode of file using **chmod 755 cplusplus.cgi** UNIX command to make file executable.

## **My First CGI program**

The above C++ program is a simple program which is writing its output on STDOUT file i.e. screen. There is one important and extra feature available which is first line printing **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and specify the content type to be displayed on the browser screen. Now you must have understood the basic concept of CGI and you can write many complicated CGI programs using Python. A C++ CGI program can interact with any other external system, such as RDBMS, to exchange information.

## **HTTP Header**

The line **Content-type:text/html\r\n\r\n** is a part of HTTP header, which is sent to the browser to understand the content. All the HTTP header will be in the following form −

HTTP Field Name: Field Content

For Example

Content-type: text/html\r\n\r\n

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

|  |  |
| --- | --- |
| **Sr.No** | **Header & Description** |
| 1 | **Content-type:**  A MIME string defining the format of the file being returned. Example is Content-type:text/html. |
| 2 | **Expires: Date**  The date the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. A valid date string should be in the format 01 Jan 1998 12:00:00 GMT. |
| 3 | **Location: URL**  The URL that should be returned instead of the URL requested. You can use this filed to redirect a request to any file. |
| 4 | **Last-modified: Date**  The date of last modification of the resource. |
| 5 | **Content-length: N**  The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file. |
| 6 | **Set-Cookie: String**  Set the cookie passed through the *string*. |

## **CGI Environment Variables**

All the CGI program will have access to the following environment variables. These variables play an important role while writing any CGI program.

|  |  |
| --- | --- |
| **Sr.No** | **Variable Name & Description** |
| 1 | **CONTENT\_TYPE**  The data type of the content, used when the client is sending attached content to the server. For example file upload etc. |
| 2 | **CONTENT\_LENGTH**  The length of the query information that is available only for POST requests. |
| 3 | **HTTP\_COOKIE**  Returns the set cookies in the form of key & value pair. |
| 4 | **HTTP\_USER\_AGENT**  The User-Agent request-header field contains information about the user agent originating the request. It is a name of the web browser. |
| 5 | **PATH\_INFO**  The path for the CGI script. |
| 6 | **QUERY\_STRING**  The URL-encoded information that is sent with GET method request. |
| 7 | **REMOTE\_ADDR**  The IP address of the remote host making the request. This can be useful for logging or for authentication purpose. |
| 8 | **REMOTE\_HOST**  The fully qualified name of the host making the request. If this information is not available then REMOTE\_ADDR can be used to get IR address. |
| 9 | **REQUEST\_METHOD**  The method used to make the request. The most common methods are GET and POST. |
| 10 | **SCRIPT\_FILENAME**  The full path to the CGI script. |
| 11 | **SCRIPT\_NAME**  The name of the CGI script. |
| 12 | **SERVER\_NAME**  The server's hostname or IP Address. |
| 13 | **SERVER\_SOFTWARE**  The name and version of the software the server is running. |

Here is small CGI program to list out all the CGI variables.

#include <iostream>

#include <stdlib.h>

using namespace std;

const string ENV[ 24 ] = {

"COMSPEC", "DOCUMENT\_ROOT", "GATEWAY\_INTERFACE",

"HTTP\_ACCEPT", "HTTP\_ACCEPT\_ENCODING",

"HTTP\_ACCEPT\_LANGUAGE", "HTTP\_CONNECTION",

"HTTP\_HOST", "HTTP\_USER\_AGENT", "PATH",

"QUERY\_STRING", "REMOTE\_ADDR", "REMOTE\_PORT",

"REQUEST\_METHOD", "REQUEST\_URI", "SCRIPT\_FILENAME",

"SCRIPT\_NAME", "SERVER\_ADDR", "SERVER\_ADMIN",

"SERVER\_NAME","SERVER\_PORT","SERVER\_PROTOCOL",

"SERVER\_SIGNATURE","SERVER\_SOFTWARE" };

int main () {

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>CGI Environment Variables</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<table border = \"0\" cellspacing = \"2\">";

for ( int i = 0; i < 24; i++ ) {

cout << "<tr><td>" << ENV[ i ] << "</td><td>";

// attempt to retrieve value of environment variable

char \*value = getenv( ENV[ i ].c\_str() );

if ( value != 0 ) {

cout << value;

} else {

cout << "Environment variable does not exist.";

}

cout << "</td></tr>\n";

}

cout << "</table><\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

## **C++ CGI Library**

For real examples, you would need to do many operations by your CGI program. There is a CGI library written for C++ program which you can download from <ftp://ftp.gnu.org/gnu/cgicc/> and follow the steps to install the library −

$tar xzf cgicc-X.X.X.tar.gz

$cd cgicc-X.X.X/

$./configure --prefix=/usr

$make

$make install

You can check related documentation available at [‘C++ CGI Lib Documentation](https://www.gnu.org/software/cgicc/doc/index.html).

## **GET and POST Methods**

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

## **Passing Information Using GET Method**

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows −

http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation and you can pass upto 1024 characters in a request string.

When using GET method, information is passed using QUERY\_STRING http header and will be accessible in your CGI Program through QUERY\_STRING environment variable.

You can pass information by simply concatenating key and value pairs alongwith any URL or you can use HTML <FORM> tags to pass information using GET method.

## **Simple URL Example: Get Method**

Here is a simple URL which will pass two values to hello\_get.py program using GET method.

[/cgi-bin/cpp\_get.cgi?first\_name=ZARA&last\_name=ALI](https://www.tutorialspoint.com/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI)

Below is a program to generate **cpp\_get.cgi** CGI program to handle input given by web browser. We are going to use C++ CGI library which makes it very easy to access passed information −

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Using GET and POST Methods</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("first\_name");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "First name: " << \*\*fi << endl;

} else {

cout << "No text entered for first name" << endl;

}

cout << "<br/>\n";

fi = formData.getElement("last\_name");

if( !fi->isEmpty() &&fi != (\*formData).end()) {

cout << "Last name: " << \*\*fi << endl;

} else {

cout << "No text entered for last name" << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Now, compile the above program as follows −

$g++ -o cpp\_get.cgi cpp\_get.cpp -lcgicc

Generate cpp\_get.cgi and put it in your CGI directory and try to access using following link −

[/cgi-bin/cpp\_get.cgi?first\_name=ZARA&last\_name=ALI](https://www.tutorialspoint.com/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI)

This would generate following result −

First name: ZARA

Last name: ALI

## **Simple FORM Example: GET Method**

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same CGI script cpp\_get.cgi to handle this input.

<form action = "/cgi-bin/cpp\_get.cgi" method = "get">

First Name: <input type = "text" name = "first\_name"> <br />

Last Name: <input type = "text" name = "last\_name" />

<input type = "submit" value = "Submit" />

</form>

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

Top of Form

First Name:  Last Name:  

Bottom of Form

## **Passing Information Using POST Method**

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

The same cpp\_get.cgi program will handle POST method as well. Let us take same example as above, which passes two values using HTML FORM and submit button but this time with POST method as follows −

<form action = "/cgi-bin/cpp\_get.cgi" method = "post">

First Name: <input type = "text" name = "first\_name"><br />

Last Name: <input type = "text" name = "last\_name" />

<input type = "submit" value = "Submit" />

</form>

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

Top of Form

First Name:  Last Name:  

Bottom of Form

## **Passing Checkbox Data to CGI Program**

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code for a form with two checkboxes −

<form action = "/cgi-bin/cpp\_checkbox.cgi" method = "POST" target = "\_blank">

<input type = "checkbox" name = "maths" value = "on" /> Maths

<input type = "checkbox" name = "physics" value = "on" /> Physics

<input type = "submit" value = "Select Subject" />

</form>

The result of this code is the following form −

Top of Form

 Maths  Physics 

Bottom of Form

Below is C++ program, which will generate cpp\_checkbox.cgi script to handle input given by web browser through checkbox button.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

bool maths\_flag, physics\_flag;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Checkbox Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

maths\_flag = formData.queryCheckbox("maths");

if( maths\_flag ) {

cout << "Maths Flag: ON " << endl;

} else {

cout << "Maths Flag: OFF " << endl;

}

cout << "<br/>\n";

physics\_flag = formData.queryCheckbox("physics");

if( physics\_flag ) {

cout << "Physics Flag: ON " << endl;

} else {

cout << "Physics Flag: OFF " << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

## **Passing Radio Button Data to CGI Program**

Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio button −

<form action = "/cgi-bin/cpp\_radiobutton.cgi" method = "post" target = "\_blank">

<input type = "radio" name = "subject" value = "maths" checked = "checked"/> Maths

<input type = "radio" name = "subject" value = "physics" /> Physics

<input type = "submit" value = "Select Subject" />

</form>

The result of this code is the following form −

Top of Form

 Maths  Physics 

Bottom of Form

Below is C++ program, which will generate cpp\_radiobutton.cgi script to handle input given by web browser through radio buttons.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Radio Button Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("subject");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Radio box selected: " << \*\*fi << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

## **Passing Text Area Data to CGI Program**

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box −

<form action = "/cgi-bin/cpp\_textarea.cgi" method = "post" target = "\_blank">

<textarea name = "textcontent" cols = "40" rows = "4">

Type your text here...

</textarea>

<input type = "submit" value = "Submit" />

</form>

The result of this code is the following form −

Top of Form

Bottom of Form

Below is C++ program, which will generate cpp\_textarea.cgi script to handle input given by web browser through text area.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Text Area Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("textcontent");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Text Content: " << \*\*fi << endl;

} else {

cout << "No text entered" << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

## **Passing Drop down Box Data to CGI Program**

Drop down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box −

<form action = "/cgi-bin/cpp\_dropdown.cgi" method = "post" target = "\_blank">

<select name = "dropdown">

<option value = "Maths" selected>Maths</option>

<option value = "Physics">Physics</option>

</select>

<input type = "submit" value = "Submit"/>

</form>

The result of this code is the following form −

Top of Form

Bottom of Form

Below is C++ program, which will generate cpp\_dropdown.cgi script to handle input given by web browser through drop down box.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc formData;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Drop Down Box Data to CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

form\_iterator fi = formData.getElement("dropdown");

if( !fi->isEmpty() && fi != (\*formData).end()) {

cout << "Value Selected: " << \*\*fi << endl;

}

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

## **Using Cookies in CGI**

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

## **How It Works**

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields −

* **Expires** − This shows date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
* **Domain** − This shows domain name of your site.
* **Path** − This shows path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
* **Secure** − If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
* **Name = Value** − Cookies are set and retrieved in the form of key and value pairs.

## **Setting up Cookies**

It is very easy to send cookies to browser. These cookies will be sent along with HTTP Header before the Content-type filed. Assuming you want to set UserID and Password as cookies. So cookies setting will be done as follows

#include <iostream>

using namespace std;

int main () {

cout << "Set-Cookie:UserID = XYZ;\r\n";

cout << "Set-Cookie:Password = XYZ123;\r\n";

cout << "Set-Cookie:Domain = www.tutorialspoint.com;\r\n";

cout << "Set-Cookie:Path = /perl;\n";

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Cookies in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "Setting cookies" << endl;

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

Here, it is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html\r\n\r\n**.

Compile above program to produce setcookies.cgi, and try to set cookies using following link. It will set four cookies at your computer −

[/cgi-bin/setcookies.cgi](https://www.tutorialspoint.com/cgi-bin/setcookies.cgi)

## **Retrieving Cookies**

It is easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form.

key1 = value1; key2 = value2; key3 = value3....

Here is an example of how to retrieve cookies.

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc cgi;

const\_cookie\_iterator cci;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>Cookies in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

cout << "<table border = \"0\" cellspacing = \"2\">";

// get environment variables

const CgiEnvironment& env = cgi.getEnvironment();

for( cci = env.getCookieList().begin();

cci != env.getCookieList().end();

++cci ) {

cout << "<tr><td>" << cci->getName() << "</td><td>";

cout << cci->getValue();

cout << "</td></tr>\n";

}

cout << "</table><\n";

cout << "<br/>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

Now, compile above program to produce getcookies.cgi, and try to get a list of all the cookies available at your computer −

[/cgi-bin/getcookies.cgi](https://www.tutorialspoint.com/cgi-bin/getcookies.cgi)

This will produce a list of all the four cookies set in previous section and all other cookies set in your computer −

UserID XYZ

Password XYZ123

Domain www.tutorialspoint.com

Path /perl

## **File Upload Example**

To upload a file the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type will create a "Browse" button.

<html>

<body>

<form enctype = "multipart/form-data" action = "/cgi-bin/cpp\_uploadfile.cgi"

method = "post">

<p>File: <input type = "file" name = "userfile" /></p>

<p><input type = "submit" value = "Upload" /></p>

</form>

</body>

</html>

The result of this code is the following form −

Top of Form

File:



Bottom of Form

**Note** − Above example has been disabled intentionally to stop people uploading files on our server. But you can try above code with your server.

Here is the script **cpp\_uploadfile.cpp** to handle file upload −

#include <iostream>

#include <vector>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <cgicc/CgiDefs.h>

#include <cgicc/Cgicc.h>

#include <cgicc/HTTPHTMLHeader.h>

#include <cgicc/HTMLClasses.h>

using namespace std;

using namespace cgicc;

int main () {

Cgicc cgi;

cout << "Content-type:text/html\r\n\r\n";

cout << "<html>\n";

cout << "<head>\n";

cout << "<title>File Upload in CGI</title>\n";

cout << "</head>\n";

cout << "<body>\n";

// get list of files to be uploaded

const\_file\_iterator file = cgi.getFile("userfile");

if(file != cgi.getFiles().end()) {

// send data type at cout.

cout << HTTPContentHeader(file->getDataType());

// write content at cout.

file->writeToStream(cout);

}

cout << "<File uploaded successfully>\n";

cout << "</body>\n";

cout << "</html>\n";

return 0;

}

The above example is for writing content at **cout** stream but you can open your file stream and save the content of uploaded file in a file at desired location.

Hope you have enjoyed this tutorial. If yes, please send us your feedback.

# TCP/IP