C++ is a general purpose, high level, compiler based object oriented programming language.

By using C++ different types of software can be designed.

Ex:

OS: DOS, Windows, Unix…

Editor: Notepad, Wordpad, Edit plus…

Commercial apps: Billing system in hotels, restaurants, super markets etc.,

Databases: Oracle, SQL server, My sql, mongo db etc.,

Translators:

1. Compiler and interpreter are used to convert high level program to machine code
2. Assembler is used to convert low level programs to machine code

Device drivers: A program that tells the OS how a device works.

PC and mobile games: Games are designed in c++. Ex: Snake game in mobiles – nokia

Protocols:

A protocol is a set of rules to be followed by two or more communicating bodies (computers) with each other.

ftp-file transfer protocol

dsp

http

tcp/ip

CDMA- code division Mulitple Access

We can design protocols using c++. FTP, HTTP are designed in c++. 90% of telecom protocols.

An Annotated Hello World  
The following dead simple C++ program shows all of the components of a C++ program:

// helloworld.cpp  
#include <iostream> // C++ standard headers don't use ".h"  
/\* main() in C++ takes argc (the number of arguments) and  
argv (an array of arguments) \*/  
int main(int argc, char\*\* argv)  
{  
// "std" is the standard C++ namespace, like a Java package  
std::cout << "Hello, World!" << std::endl;  
return 0;  
}

**Terminology**:

C++ - a **general-purpose programming language** with a bias towards systems programming that supports procedural programming, data abstraction, object-oriented programming, and generic programming. C ++ was designed and originally implemented by Bjarne Stroustrup. C++ is defined by ISO /IEC 14882 - Standard for the C++ Programming Language.

RTTI:

RTTI **(Run-time type Information)** in C++. In C++, RTTI (Run-time type information) is a **mechanism that exposes information about an object’s data type at runtime** and is available only for the classes which have at least one virtual function.

Current Standard:

The current ISO C++ standard is officially known as **ISO International Standard ISO/IEC 14882:2017 (E)** – Programming Language C++. Purchase the C++17 official standard. You can purchase the official standard at the ISO Store or at national body stores such as the ANSI store. The in-progress LaTeX source materials are maintained on GitHub.

**abstract class -**a class that can only be used as a base class for some other class. A class is abstract if it has at least one pure virtual function.   
  
**access control -**a C++ mechanism for prohibiting or granting access to individual members of a class.   
  
**access declaration -**a way of controlling access to a specified member of a base class when it is used in a derived class.   
  
**access specifier -**a way of labeling members of a class to specify what access is permitted.   
  
**aggregate -**an array or object of a class with no constructors, no private or protected members, no base classes, and no virtual functions.   
  
**allocation -**the process of giving memory space to an object.  
  
**ANSI -**acronym for American National Standards Institute, a standards body currently standardizing C++.   
  
**argument -**when calling a function, refers to the actual values passed to the function.   
  
**argument matching -**the process of determining which of a set of functions of a specified name matches given arguments in a function call.   
  
**ARM -**acronym for the book The C++ Annotated Reference Manual, a C++ reference book by Ellis and Stroustrup.   
  
**array -**an ordered and index-able sequence of values. C++ supports arrays of a single dimension (a vector) or of multiple dimensions.   
  
**asm -**C++ keyword used to specify assembly language in the middle of C++ code.   
  
**assignment -**the process of giving a value to a preexisting object.  
  
**assignment operator -**an operator for doing assignment.  
  
**auto -**a C++ keyword used to declare a stack-based local variable in a function. This is the default and is normally not needed.   
  
**base class -**a class that serves as a base for a derived class to inherit members from.   
  
**bit field -**a member of a class that represents small integral values.   
  
**bitwise copy -**to copy an object without regard to its structure or members.   
  
**bool -**C++ keyword used to declare a Boolean data type.   
  
**break -**C++ keyword used to specify a statement that is used to break out of a for or while loop or out of a switch statement.   
  
**browser -**a software development tool used for viewing class declarations and the class hierarchy.  
  
**built-in type -**see fundamental type.   
  
**C -**a programming language in widespread use. C++ is based on C.   
  
**C-style string -**refers to a char\* and to the contents of any dynamic storage it may point at. C++ does not have true strings as part of the language proper, though a standard string class library is envisioned as part of the ANSI standardization effort.   
  
**call by reference -**passing a pointer to an argument to a function. The function can then change the argument value.    
  
**call by value -**passing a copy of an argument to a function. The function cannot then change the argument value. C and C++ use call by value argument passing.   
  
**calling conventions -**refers to the system-specific details of just how the arguments to a function are passed. For example, the order in which they are passed on the stack or placed in machine registers.   
  
**case -**a C++ keyword used to denote an individual element of a switch statement.   
  
**cast -**a way of doing explicit type conversion via a cast operator.  
  
**catch -**a C++ keyword used to declare an exception handler.   
  
**cerr -**in C++ stream I/O, the standard error stream.   
  
**cfront -**a C++ front end that translates C++ source code to C code, which is then compiled via a C compiler. Originally developed by AT&T Bell Labs in the mid-1980s.   
  
**char -**a C++ keyword used to declare an object of character type. Often considered the same as a byte, though it is possible to have multi-byte characters.   
  
**cin -**in C++ stream I/O, the standard input stream.   
  
**class -**a C++ keyword used to declare the fundamental building block of C++ programs. A class has a tag, members, access control mechanisms, and so on.   
  
**class hierarchy -**see base class, derived class.   
  
**class layout -**the way in which data class members are arranged in a class object.   
  
**class library -**a set of related classes declared in header files and defined in object files.   
  
**class member -**a constituent member of a class, such as a data declaration, a function, or a nested class.   
  
**class template -**a template used for generating class types.   
  
**comments -**C++ has C-style comments delimited with /\* and \*/, and new C++-style line-oriented comments starting with //.   
  
**compilation unit -**see translation unit.   
  
**compiler -**a software tool that converts a language such as C++ into a different form, typically assembly language.   
  
**const -**a C++ keyword used to declare an object as constant or used to declare a constant parameter.   
  
**constant -**a literal or variable declared as const.

**constant expression -**a C++ expression that can be evaluated by the compiler. Used to declare bounds for an array among other things.   
  
**constructor -**a function called when a class object comes into scope. The constructor is used to initialize the object.    
  
**const\_cast -**a C++ keyword used as a style of cast for explicitly casting away const.   
  
**container class -**a type of class or template that is used to hold objects of other types. Lists and stacks would be examples of container classes.   
  
**continue -**C++ keyword used with for and while statements to continue the iteration at the top of the loop.   
  
**conversion -**to convert from one data type to another.   
  
**copy constructor -**a special type of constructor that is called when an object is copied.   
  
**cout -**in C++ stream I/O, the standard output stream.   
  
**data abstraction -**the idea of defining a data representation (for example, to represent a calendar date), and a set of operations to manipulate that representation, with no public access to the representation except via the operations.    
  
**deallocation -**the processing of freeing memory space previously used by an object.   
  
**debugger -**a tool for stepping through the execution of a program, examining variables, setting breakpoints, and so on.   
  
**declaration -**a C++ entity that introduces one or more names into a program.   
  
**declaration statement -**a declaration in the form of a statement that may be used in C++ where statements would normally be used.   
  
**declarator -**a part of a declaration that actually declares an identifier name. A declarator appears after a sequence of type and storage class specifiers.   
  
**default argument -**an optional argument to a function. A value specified in the function declaration is used if the argument is not given.   
  
**delete operator -**C++ keyword and operator used to delete dynamic storage.   
  
**delete[] operator -**See delete operator. Used to delete array objects.   
  
**demotion -**converting a fundamental type to another fundamental type, with possible loss of precision. For example, a demotion would occur in converting a long to a char.   
  
**deprecate -**to make obsolete (a language feature).   
  
**derived class -**a class that inherits members from a base class. See inheritance.   
  
**destructor -**a function called when a class object goes out of scope. It cleans up the object, freeing resources like dynamic storage. See constructor and deallocation.   
  
**dialect -**refers to a variant of a programming language, used by a subset of the software community. Can also refer to a particular style of programming.   
  
**do -**see while.   
  
**dominance -**refers to the case where one name is used in preference to another.See multiple inheritance.   
  
**double -**C++ keyword used to declare a floating point type.   
  
**dynamic storage -**refers to memory allocated and deallocated during program execution using the new operator and delete operator.   
  
**dynamic\_cast -**a C++ keyword that specifies a style of cast used with run-time type information. Using dynamic\_cast one can obtain a pointer to an object of a derived class given a pointer of a base class type. If the object pointed to is not of the specified derived class, dynamic\_cast will return 0.   
  
**else -**C++ keyword, part of the if statement.   
  
**embedded system -**a low-level software program that executes without much in the way of run-time services, such as those provided by an operating system.   
  
**encapsulation -**a term meaning to wrap up or contain within. Used in relation to the members of a class. See access control.   
  
**enum -**C++ keyword used to declare an enumeration.   
  
**enumeration -**a set of discrete named integral values. See enum.   
  
**enumerator -**a member of an enumeration.   
  
**exception -**a value of some type that is thrown. See exception handling.   
  
**exception handler -**a piece of code that catches an exception. See catch and try block.   
  
**exception handling -**the process of signalling that an exceptional condition (such as divide by zero) has occurred. An exception is thrown and then caught by an exception handler, after stack unwinding has occurred.   
  
**explicit -**a C++ keyword used in the declaration of constructors to indicate that conversion of an initializer should not take place.   
  
**expression -**a combination of constants, variables, and operators used to produce a value of some type.   
  
**expression statement -**a statement that is an expression, such as a function call or assignment.   
  
**extern -**a C++ keyword used to declare an external name.   
  
**external name -**a name available to other translation units in a program. See linker and global variable.   
  
**false -**C++ keyword used to specify a value for the bool type.   
  
**finalization -**to declare that an object or resource is no longer needed, and initiate cleanup of that object. See initialization.   
  
**float -**a C++ keyword used to declare a floating point type.   
  
**floating point -**non-integral arithmetic. A floating-point number is typically represented as a base-two fraction part and an exponent.   
  
**for -**a C++ keyword used to specify an iteration or looping statement.   
  
**forward class -**a class for which only the tag has been declared. Such a class can be used where the size of the class is not needed, for example in pointer declarations.   
  
**free store -**see dynamic storage.   
  
**friend -**a type of declaration used within a class to grant other classes or functions access to that class. See access control.   
  
**front end -**often refers to the early stages of C++ compilation, such as parsing and semantic analysis.   
  
**function -**a C++ entity that is a sequence of statements. It has its own scope, accepts a set of argument values, and returns a value on completion.   
  
**function template -**a template used for generating function types.   
  
**fundamental type -**a type built in to the C++ language. Examples would be integral types like int and pointer types such as void\*.   
  
**garbage collection -**a way of automatically managing dynamic storage such that explicit cleanup of storage is not required. C++ does not have garbage collection. See new operator and delete operator.   
  
**generic programming -**see template.   
  
**global name -**a name declared at global scope.   
  
**global namespace -**the implicit namespace where global variables reside.   
  
**global scope -**see global namespace.   
  
**global variable -**a variable that is accessible throughout the whole program, whose lifetime is that of the program.   
  
**goto -**C++ keyword, used to transfer control within a C++ function. See label.   
  
**grammar -**a way of expressing the syntax of a programming language, to describe exactly what usage is valid and invalid.   
  
**header -**see header file.   
  
**header file -**a file containing class declarations, preprocessor directives, and so on, and included in a translation unit. It is expanded by the preprocessor.   
  
**heap storage -**see dynamic storage.   
  
**helper class -**a class defined as part of implementing the details of another class.   
  
**hiding -**see encapsulation.   
  
**if -**C++ keyword used in conditional statements.   
  
**implementation-dependent behavior -**not every aspect of a programming language like C++ is specified in a language standard. This term refers to behavior that may vary from implementation to implementation.   
  
**implicit conversion -**a conversion done as part of another operation, for example converting a pointer type to bool in an if statement.   
  
**inheritance -**the process whereby a derived class inherits members from a base class. A derived class will also add its own members to those of the base class.   
  
**initialization -**to give an initial value to an object. See constructor and assignment.   
  
**initialize -**the process of initialization.   
  
**initializer -**a value or expression used to initialize an object during initialization.   
  
**inline -**C++ keyword used to declare an inline function.   
  
**inline function -**a function that can be expanded by a compiler at the point of call, thereby saving the overhead time required to call the function.   
  
**instantiation -**see template instantiation.   
  
**int -**a C++ keyword and fundamental type, used to declare an integral type.   
  
**integral conversion -**the process by which an integer is converted to signed or unsigned.   
  
**integral promotion -**the process by which a bool, char, short, enumerator, or bit field are converted to int for use in expressions, argument passing, and so on.   
  
**keyword -**a reserved identifier in C++, used to denote data types, statements of the language, and so on.   
  
**label -**a name that is the target of a goto statement.   
  
**layout -**refers to the way that objects are arranged in memory.   
  
**library -**a set of object files grouped together. A linker will search them repeatedly and use whatever object files are needed. See class library.   
  
**lifetime -**refers to the duration of the existence of an object. Some objects last for the whole execution of a program, while other objects have a shorter lifetime.   
  
**linkage -**refers to whether a name is visible only inside or also outside its translation unit.   
  
**linker -**a program that combines object files and library code to produce an executable program.   
  
**literal -**a constant like 1234.   
  
**local -**typically refers to the scope and lifetime of names used in a function.   
  
**local class -**a class declared local to a function.   
  
**local variable -**a variable declared local to a function.   
  
**long -**C++ keyword used to declare a long integer data type.   
  
**long double -**a floating point type in C++.   
  
**lvalue -**an expression referring to an object. See rvalue.   
  
**macro -**a preprocessor feature that supports parameter substitution and expansion of commonly-used code sequences. See inline function.   
  
**mangling -**see name mangling.   
  
**member -**see class member and namespace member.   
  
**member function -**a function that is an element of a class and that operates on objects of that class via the this pointer to the object.   
  
**memberwise copy -**to copy an object a member at a time, taking into account a copy constructor for the member. See bitwise copy.   
  
**method -**see member function.   
  
**mixed-mode arithmetic -**mixing of integral and floating point arithmetic.   
  
**module -**see translation unit.   
  
**multiple inheritance -**a derived class with multiple base classes. See inheritance.   
  
**mutable -**C++ keyword declaring a member non-constant even if it is a member of a const object.   
  
**name -**an identifier that denotes an object, function, a set of overloaded functions, a type, an enumerator, a member, a template, a namespace, or a label.   
  
**name lookup -**refers to taking a name and determining what it refers to, or its value, based on the scope and other rules of C++.   
  
**name mangling -**a way of encoding an external name representing a function so as to be able to distinguish the types of its parameters. See overload.   
  
**name space -**a grouping of names.   
  
**namespace -**a C++ keyword used to declare a namespace, which is a collection of names such as function declarations, classes, and so on.   
  
**namespace alias -**an alias for a namespace, that can be used to refer to the namespace.   
  
**namespace member -**an element of a namespace, such as a function, typedef, or class declaration.   
  
**nested class -**a class declaration nested within another class.   
  
**new handler -**a function established by calling set\_new\_handler. It is called when the new operator cannot obtain dynamic storage.   
  
**new operator -**C++ keyword and operator used to allocate dynamic storage.   
  
**new-style cast -**a cast written in functional notation.   
  
**new[] operator -**see new operator. Used to allocate dynamic storage for array objects.   
  
**NULL -**a special constant value that represents a null pointer.   
  
**null pointer -**a pointer value that evaluates to zero.   
  
**object -**has several meanings. In C++, often refers to an instance of a class. Also more loosely refers to any named declaration of a variable or other entity that involves storage.   
  
**object file -**in C or C++, typically the output of a compiler. An object file consists of machine language plus an external name list that is resolved by a linker.   
  
**object layout -**refers to the ordering of data members within a class.   
  
**object-oriented -**this term has various definitions, usually including the notions of derived classes and virtual functions. See data abstraction.   
  
**old-style cast -**a cast written in C style, with the type in parentheses before the value being casted.   
  
**OOA / OOD -**acronym for object-oriented analysis and object-oriented design, processes of analyzing and designing object-oriented software.   
  
**OOP -**acronym for object-oriented programming.   
  
**operator -**a builtin operation of the C++ language, like addition, or an overloaded operator corresponding to a member function of a class. See function and operator overloading.   
  
**operator overloading -**to treat a C++ operator like << as a function and overload it for particular parameter types.   
  
**overload -**to specify more than one function of the same name, but with varying numbers and types of parameters. See argument matching.   
  
**overload resolution -**see argument matching.   
  
**parameter -**refers to the variables passed into a function. See also argument.   
  
**parameterized type -**see template.   
  
**parser -**see parsing.   
  
**parsing -**the process by which a program written in some programming language is broken down into its syntactic elements.   
  
**placement -**the ability to define a variant of the new operator to take an additional argument that specifies what storage is to be used.   
  
**pointer -**an address of an object.   
  
**pointer to data member -**a pointer that points at a data member of a class.   
  
**pointer to function -**an address of a function or a member function.   
  
**pointer to member -**see pointer to data member, pointer to function.   
  
**polymorphism -**the ability to call a variety of member functions for a given class object using an identical interface in each case. See virtual function.   
  
**postfix -**refers to operators that appear after their operand. See prefix.   
  
**pragma -**a preprocessor directive used to affect compiler behavior in an implementation-defined way.   
  
**prefix -**refers to operators that appear before their operand. See postfix.   
  
**preprocessing -**a stage of compilation processing that occurs before the compiler proper is invoked. Preprocessing handles macro expansion among other things. In C++ use of const and inline functions makes preprocessing   
less important.   
  
**preprocessor -**see preprocessing.   
  
**private -**a C++ keyword used to specify that a class member can only be accessed from member functions and friends of the class. See access control, protected, and public.   
  
**programming environment -**a set of integrated tools used in developing software, including a compiler, linker, debugger, and browser.   
  
**promotion -**see integral promotion.   
  
**protected -**a C++ keyword used to specify that a class member can only be accessed by member functions and friends of its own class and by member functions and friends of classes derived from this class. See private, public, and access control.   
  
**PT -**see parameterized type.   
  
**public -**a C++ keyword used to specify that class members are accessible from any (non-member) function. See access control, protected, and private.   
  
**pure virtual function -**a virtual function with a "= 0" initializer. See abstract class.   
  
**qualification -**to prefix a name with the name of a class or namespace.   
  
**recursive descent parser -**see parsing. This is a type of parsing used in C++ compilers. It is more flexible than the older Yacc approach often used in C compilers.   
  
**reference -**another name for an object. Access to an object via a reference is like manipulating the object itself. References are typically implemented as pointers in the underlying generated code.   
  
**register -**C++ keyword used as a hint to the compiler that a particular local variable should be placed in a machine register.   
  
**reinterpret\_cast -**a C++ keyword used as a style of cast for performing unsafe and implementation dependent casts.   
  
**repository -**a location where an instantiated template class can be stored. See template instantiation.   
  
**resolution -**see overload resolution.   
  
**resumption -**a style of exception handling where program execution continues from the point where an exception is thrown. C++ uses the termination style.   
  
**return -**C++ keyword used for returning values from a function.   
  
**return value -**the value returned from a function.   
  
**RTTI -**acronym for run-time type information.   
  
**run-time -**refers to actions that occur during program execution.   
  
**run-time efficiency -**refers to the issue of whether basic C++ operations will cause a performance penalty when the program is run.   
  
**run-time type information -**a system for determining at run-time what the type of an object is.   
  
**rvalue -**a value that may appear on the right-hand side of an assignment.   
  
**scope -**the region of a program where a name has visibility.   
  
**semantic analysis -**a stage that a compiler goes through after parsing. In this stage the meaning of the program is analyzed.   
  
**semantics -**the meaning of a program, as opposed to its syntax.   
  
**separate compilation -**refers to the process by which each translation unit of a program is compiled separately to produce an object file. The object files are then combined by a linker.   
  
**set\_new\_handler -**a function used to establish a new handler.   
  
**short -**a C++ fundamental type used to declare small integers.   
  
**signed -**C++ keyword used to indicate a signed data type.   
  
**sizeof -**C++ keyword for taking the size of an object or type.   
  
**smart pointer -**an object that acts like a pointer but also does some processing whenever an object is accessed through them. The C++ operator -> can be overloaded to achieve this effect.   
  
**specialization -**a special case of a template defined for particular template argument types.   
  
**stack frame -**refers to a region of storage on the hardware stack, used to store information such as local variables for each invocation of a function.   
  
**stack unwinding -**see exception handling. When an exception is thrown, each active stack frame must be removed from the stack until an exception handler is found. This process involves calling a destructor as appropriate for   
each local object in the stack frame, and so on.   
  
**standard conversion -**refers to standardized conversions between types, such   
as integral conversion.   
  
**standard library -**see library. The C++ standard library includes much of the C standard library along with new features such as strings and container class support.   
  
**statement -**the parts of a program that actually do the work.   
  
**static -**see static member, static object, and static storage.   
  
**static member -**a class member that is part of a class for purposes of access control but does not operate on particular object instances of the class.   
  
**static object -**an object that is local to a function or to a translation unit   
and whose lifetime is the life of the program.   
  
**static storage -**storage that persists throughout the life of the program.   
See static object and dynamic storage.   
  
**static type checking -**refers to type checking that occurs during compilation   
of a program rather than at run-time.   
  
**static\_cast -**a C++ keyword specifying a style of cast meant to replace old-style   
C casts.   
  
**storage class -**see auto and static.   
  
**stream -**an object used to represent an input or output channel. See stream I/O.   
  
**stream I/O -**a C++ I/O library using overloaded operators << and >>. It   
has more type safety than C-style I/O.   
  
**string -**see C-style string.   
  
**struct -**a C++ class in which all the class members are by default public.   
  
**switch -**C++ keyword denoting a statement type, used to dispatch to one of   
several sequences of statements based on the value of an expression.   
  
**symbol table -**a compiler structure used to record type information about   
program names. The symbol table is used to generate compiler output.   
  
**syntax -**the rules that govern how C++ expressions, statements, declarations,   
and programs are constructed. See grammar and semantics.   
  
**systems programming -**refers to low-level programming, for example writing   
I/O drivers or operating systems. C and C++ are suitable languages for this   
type of programming.   
  
**tag -**a name given to a class, struct, or union.   
  
**template -**a parameterized type. A template can accept type parameters that   
are used to customize the resulting type.   
  
**template argument -**an actual value or type given to a template to form a   
template class. See argument.   
  
**template class -**a combination of a template with a template argument list   
via the process of template instantiation.   
  
**template declaration -**a declaration of a template with its associated template parameter   
list.   
  
**template definition -**an actual definition of a template or one of its members.   
  
**template instantiation -**the process of combining template arguments with   
a template to form a template class.   
  
**template parameter -**a value or type declared to be passed in to a template.   
See parameter.   
  
**temporary -**an unnamed object used during the evaluation of an expression   
to store intermediate values.   
  
**termination -**a style of exception handling where control does not return   
to the point where an exception is thrown. C++ uses this style of exception   
handling.   
  
**this -**C++ keyword used in a member function to point at the object currently   
being operated on.   
  
**throw -**C++ keyword used to throw (initiate) an exception. See exception handling.   
  
**translation limit -**a limit on the size of a source program that a compiler   
will accept.   
  
**translation unit -**a source file presented to a compiler with an object file   
produced as a result.   
  
**trigraph -**a sequence of characters used to represent another character,   
for example to represent a character not normally found in the character   
set.   
  
**true -**C++ keyword used to specify a value for the bool type.   
  
**try -**C++ keyword used to delimit a try block.   
  
**try block -**a statement that sets up a context for exception handling. A   
subsequent throw from a function called from within the try block will be   
caught by the exception handler associated with the try block or by a handler   
further out in the chain of handlers.   
  
**type -**a property of a name that determines how it can be used. For example,   
an object of a class type cannot be assigned to an integer variable.   
  
**type checking -**see type system.   
  
**type conversion -**converting a value from one type to another, for example   
via a constructor.   
  
**type safety -**see type system.   
  
**type system -**a system of types and operations on objects of those types.   
Type checking is done to ensure that the operations for given types are   
appropriate, for example that a function is called with arguments of the   
appropriate types.   
  
**type-safe linkage -**refers to the process of encoding parameter type information   
in external names so that the linker will reject mismatches between the use   
and definition of functions. See name mangling.   
  
**typedef -**a C++ keyword used to declare an alias for a type.   
  
**typeid -**an operator that returns an object describing the type of the operand.   
See run-time type information.   
  
**union -**a structure somewhat like a class or struct, except that individual   
union members share the same memory. See class layout.   
  
**unsigned -**a C++ keyword used to declare an integral unsigned fundamental type.   
  
**unwinding -**see stack unwinding.   
  
**user-defined conversion -**a member function that supports conversion from   
an object of class type to any target type.   
  
**user-defined type -**a class or typedef.   
  
**using declaration -**a declaration making a class or namespace name available   
in another scope.   
  
**using directive -**a way of making available to a program the members of a   
namespace.   
  
**using namespace -**see using directive.   
  
**variable -**an object that can be assigned to.   
  
**vector -**a one-dimensional array.   
  
**virtual base class -**a base class where a single subobject of the base class   
is shared by every derived class that declared the base class as virtual.   
  
**virtual function -**a member function whose interpretation when called depends   
on the type of the object for which it is called; a function for an object   
of a derived class will override a function of its base class.   
  
**virtual table -**a lookup table used for dispatching virtual function calls.   
A class object for a class containing virtual functions will contain a pointer   
to a virtual table.   
  
**visibility -**refers to the processing of doing name lookup without regard   
to whether a name is accessible. Once a name is found, then type checking   
and access control are applied.   
  
**void -**a C++ keyword used to declare no type. It has special uses in C++,   
for example to declare that a function has no parameter list. See also void\*.   
  
**void\* -**a pointer to a void type. Often used as the lowest common denominator   
type of pointer in C and C++.   
  
**volatile -**a type qualifier used to indicate that an object may unpredictably   
change value (for example if it is mapped to a machine register) and thus   
should not have accesses to it optimized.   
  
**wchar\_t -**C++ keyword to declare a fundamental type used for handling wide   
characters.   
  
**while -**C++ keyword used to declare an iteration statement.

**Keywords**:

asm, auto, bool, break, case, catch, char, class, const, const\_cast, continue, default, delete, do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret\_cast, return, short, signed, sizeof, static, static\_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar\_t, while

## Definition of C++ wchar\_t

In C++, wide characters are like character datatype except the fact that char data type takes space of one byte whereas wide-character takes space of two bytes. In some cases, the wide-character takes up four bytes of memory depending on the compiler. This can hold different 64K (65536) characters in those two bytes of space. That is, it can hold characters of UNICODE which an international standard that permits encoding characters any character in any language virtually. Let us see more details on wide characters in the below sections. In this article, we will discuss the functions and examples of C++ wchar\_t.

**Syntax:**

Wide characters are written in the format as mentioned below.

wchar\_t

This will be used in the programs for the implementation of wide characters.

### Functions of Wide Characters

Below are some of the functions that are used in wide characters.

#### Function: wcslen()

**Syntax:** wcslen ( const wchar\_t\* str ) ;

**Description:** Function that helps in getting the wide-character string length.

#### Function: wcsncpy()

**Syntax:** wchar\_t\* wcsncpy( wchar\_t\* dst, const wchar\_t\* sr, size\_t sn) ;

Popular Course in this category

**Description:** Function that helps in copying the sn characters from the source to destination. If the source end is smaller than the size sn, then the destination will have some null characters.

#### Function: wcscat()

**Syntax:** wchar\_t\* wcscat ( wchar\_t\* dst, const wchar\_t\* sr) ;

**Description:** Function that helps in concatenating the source string to destination string.

#### Function: wcscpy()

**Syntax:** wchar\_t\* wcscpy ( wchar\_t\* dst, const wchar\_t\* sr) ;

**Description:** Function that helps in copying the source string to destination string.

#### Function: wcscmp()

**Syntax:** wcscmp ( const wchar\_t\* str1, const wchar\_t\* str2) ;

**Description:** Function that helps in comparing the first string and second string. This function is similar to normal string comparison.

#### Function: wcsstr()

**Syntax:** const wchar\_t\* wcsstr ( const wchar\_t\* str1, const wchar\_t\* str2) ;

**Description:** Function that helps in finding the first appearance of the second string in the first string. Null will be returned if it is not present.

#### Function: wcstok()

**Syntax:** wchar\_t\* wcstok ( const wchar\_t\* str1, const wchar\_t\* delim , wchar\_t \*\* ptr) ;

**Description:** Function that helps in tokenizing the string that generated with the help of wide characters. A delimiter delim is also used for string tokenization.

### Examples of C++ wchar\_t

Let us see some sample examples on wchar\_t in this section.

#### Example #1 – CPP program to implement wide character and get the size of it

**Code:**

#include <iostream>  
using namespace std;  
int main()  
{  
//declare a wide character  
wchar\_t c = L'S' ;  
//print the character value  
cout << "The wide character value 'S' is: " << c << endl ;  
//print the size of wide character  
cout << "Wide character size is " << sizeof(c) ;  
return 0;  
}

In this program, a wide character is declared first. On executing the code, the value and its size gets printed. Here, it can be seen that L is used as a prefix for wide-character literals as well as wide-character string literals that notifies the compiler that string or character is of type wide-char.

#### Example #2 – CPP program to implement wide character and get the size of it using wcslen()

**Code:**

#include <iostream>  
using namespace std;  
int main()  
{  
//declare a wide character array string  
wchar\_t c[] = L"Hope never dies" ;  
//print the character value  
cout <<"The wide character length of Hope never dies " <<" is : " << wcslen(c) << endl ;  
return 0 ;  
}

In this program, a wide character array string is declared first. On executing the code, the size of the string gets printed.

#### Example #3 – CPP program to copy a wide-character string to another location

**Code:**

#include <iostream>  
#include<cwchar>  
using namespace std;  
int main()  
{  
//declare a wide character array string  
wchar\_t c[] = L"Hope never dies" ;  
wchar\_t d[15] ;  
//copy the string  
wcscpy(d, c);  
wcout << L"Original string is : " << c << L"\n Copied string is :  " << d << endl;  
return 0;  
}

In this program, an additional header file <cwchar> is also used along with other header files. Two string arrays are also used where one array is to store the string and the other one is to copy the string to. On executing the code, it can be seen that string has copied to another location.

#### Example #4 – CPP program to concatenate a wide-character string with another string

**Code:**

#include <iostream>  
#include<cwchar>  
using namespace std;  
int main()  
{  
//declare a wide character array string  
wchar\_t c[] = L"Hope never dies" ;  
wchar\_t d[] = L" and trust yourself" ;  
//concatenate the string  
wcscat(c, d);  
wcout << L"Concatenated string is : " << c << endl;  
return 0;  
}

In this program, two string arrays are declared first. On executing the code, it can be seen that both strings are concatenated using the function wcscat().

#### Example #5 – CPP program to compare a wide-character string with another string

**Code:**

#include <iostream>  
#include<cwchar>  
using namespace std;  
int main()  
{  
//declare a wide character array string  
wchar\_t c[] = L"Hope never dies" ;  
wchar\_t d[] = L" and trust yourself" ;  
//compare the strings  
wcout << L"Comparison of first string with second = " << wcscmp(c, d) << endl;  
wcout << L"Comparison of first string with first string = " << wcscmp(c, c) << endl;  
wcout << L"Comparison of second string with first string = " << wcscmp(d, c) << endl;  
return 0;  
}

In this program also, two string arrays are declared first. Unlike the above program, this program is to compare two strings. On executing the code, it can be seen that 3 values are shown. When the first string is compared with the second string, 1 is returned as the value of a first string is higher than the second. In the second case, 0 is returned because the string is compared with itself. At last, in the third case, -1 is returned as the value of the first string is less than the second.

**asm**:

C++ is a comprehensive and powerful programming language but there are few highly specialized situations that it cannot handle. For those situations, C++ provides an option using which one can drop an assembly code at any time. This option is the use of the ‘asm’ statement. Using asm statement, the assembly language can be embedded directly into the C++ program. The asm keyword takes a single field which must be a string literal.  
The general form of asm keyword is:

asm("op-code");

**op-code**: This is assembly language instruction that will be included in the program.

Some of the compilers uses the following form of asm statement:

asm instruction;

asm instruction newline

asm {instruction sequence }

Ex:

#include<bits/stdc++.h>

using namespace std;

int main() {

int res;

// move value to register %eax

// move value to register %ebx

// subtracting and storing result in res

\_\_asm\_\_ ( "movl $20, %%eax;"

                "movl $10, %%ebx;"

                "subl %%ebx, %%eax ":"=a"(res));

    cout<<res;

   return 0;

}

const\_cast:

dynamic\_cast:

explicit:

export:

**mutable:**

namespace:

reinterpret\_cast:

static\_cast:

typeid:

typename:

wchar\_t:

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances: and, and\_eq, bitand, bitor, compl, not, not\_eq, or, or\_eq, xor, xor\_eq

**Namespaces**Namespaces, like packages in Java, address the problem of naming conflicts between  
different pieces of code. For example, you might be writing some code that has a function  
called foo(). One day, you decide to start using a third-party library, which also has a  
foo() function. The compiler has no way of knowing which version of foo() you are  
referring to within your code.  
Namespaces solve this problem by allowing you to define the context in which names are  
defined. To place code in a namespace, simply enclose it within a namespace block:  
// namespaces.h  
namespace mycode {  
void foo();  
}

2  
// namespaces.cpp  
#include <iostream>  
namespace mycode {  
void foo() {  
std::cout << "foo() in the mycode namespace" << std::endl;  
}  
}  
To call the namespace-enabled version of foo():  
mycode::foo(); // calls the "foo" function in the "mycode" namespace  
To avoid being explicit about the namespace with every call, use using:  
#include "namespaces.h"  
using namespace mycode;  
int main(int argc, char\*\* argv)  
{  
foo(); // implies mycode::foo();  
}  
Namespaces are a great example of the difference between a hacked together C++  
program and a production quality program. The programs you write should be  
namespace-savvy.  
Preprocessor Directives  
Java programmers may not be familiar with the notion of a preprocessor. A preprocessor  
is like a compiler that runs over the code before the real compiler does its work. In C and  
C++, lines that begin with # contain commands for the preprocessor. In C, the  
preprocessor is often used to create faux constants and gross inlined functions known as  
macros. These preprocessor features are present in C++ but are rarely used.  
The main use of the preprocessor in C++ is the #include mechanism. Unlike Java, C and  
C++ functions are declared separately from their definitions. The declaration (which for  
C++ classes is confusingly referred to as a class definition) is usually placed in a file that  
ends in .h. Typically, C++ header files also make use of the preprocessor to make sure

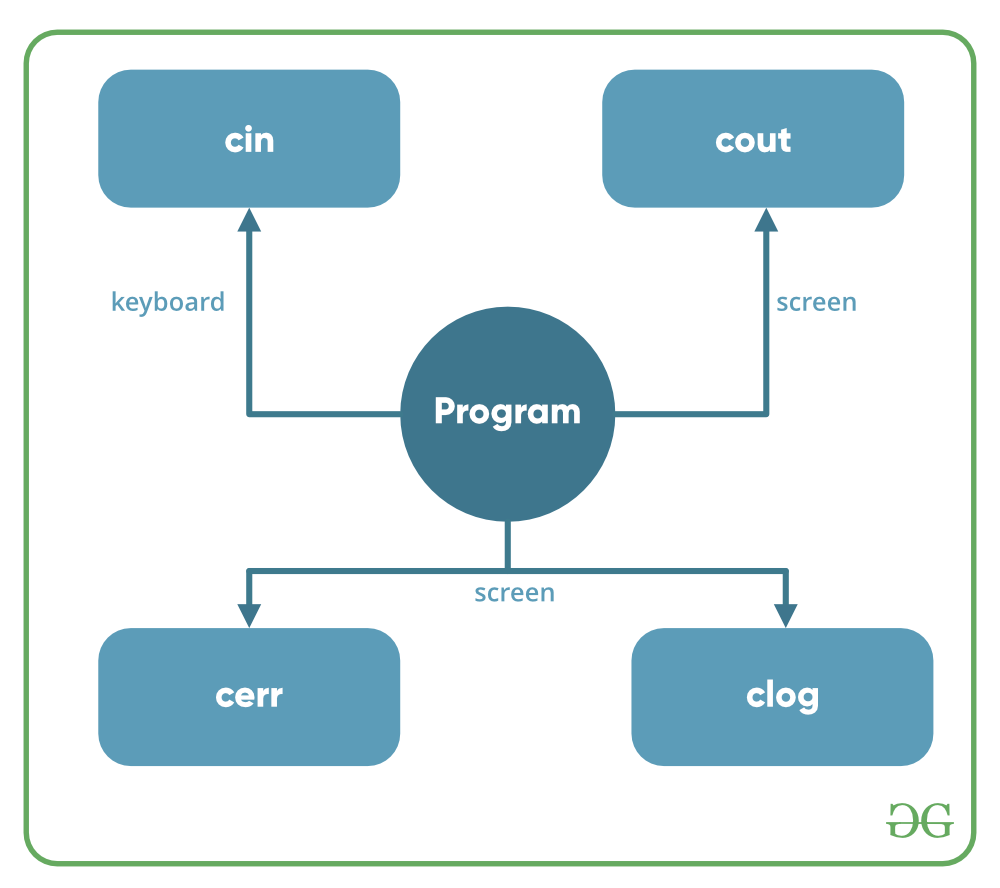
3  
that they are only included a single time. Each header defines a unique symbol and will  
skip its content if the symbol is already defined:  
// myheader.h  
#ifndef \_\_MYHEADER\_H\_\_  
#define \_\_MYHEADER\_H\_\_  
// content of the header goes here  
#endif // \_\_MYHEADER\_H\_\_  
Worth noting: In C++, the standard language headers are included using angle brackets  
(< and >) and do not traditionally use ".h", as shown above with <iostream>. Why not?  
Well, all of the standard C headers are also available within C++. When using the C  
headers, you do include the .h.  
Variables, Types, Conditionals, Loops, etc.  
Most of the basic types and other constructs are similar between C, Java, and C++. For  
example, the following snippet of code works in all three languages:  
int result = inValue;  
for (int i = inValue - 1; i > 1; i--) {  
result \*= i;  
}  
return result;  
Dynamically Created Arrays and Heap Memory  
Recall that standard arrays live on the stack and their size is determined at compile time:  
int\* myArray[30];  
To create an array whose size is determined at runtime, you declare it on the heap by  
allocating new memory:  
int\* myArray = new int[arraySize];  
Heap memory must be released manually by calling delete. When releasing memory that  
was allocated an array, you must use the bracket version of delete:

4  
delete[] myArray;  
Heap memory is also used in C to achieve pass-by-reference. We will get into pointer  
mechanics later in the quarter. For now, we don't need to use pointers for pass-by-  
reference because we have...  
References  
A reference is basically a short-hand for a pointer without the messiness of dereferencing  
or the ambiguity of ownership. Java doesn't have an analogous concept because object  
arguments to Java methods are passed by reference automatically. Since Java has an  
object class for every corresponding basic type (e.g. Integer for int), there is always a  
simple way to achieve pass-by-reference.  
In C++, the & character is used to indicate that a variable is a reference. For now, we'll  
just focus on references as parameters to functions and methods. There are some  
subtleties we'll get into later.  
The first version of addOne() below does not use a reference, so the variable that is  
passed in remains unchanged. The second version uses a reference, so the underlying  
variable is actually modified within the function. References can be used for basic types  
as well as more complex types, like classes.  
void addOne(int i)  
{  
i++; // has no real effect since this is a copy of the original  
}  
void addOne(int& i)  
{  
i++; // actually changes the original variable  
}  
Strings  
C++ programmers make use of both traditional C-style strings (arrays of characters with  
a null terminator) and the C++ string class. The string class works much like the Java  
String class, which is to say that it behaves like you would expect:  
#include <string>  
#include <iostream>  
using namespace std;

5  
int main(int argc, char\*\* argv)  
{  
string str1 = "Hello";  
string str2 = "World";  
string str3 = str1 + " " + str2;  
cout << "str1 is " << str1 << endl;  
cout << "str2 is " << str2 << endl;  
cout << "str3 is " << str3 << endl;  
if (str3 == "Hello World") {  
cout << "str3 is what it should be" << endl;  
} else {  
cout << "hmmm... str3 isn't what it should be" << endl;  
}  
return 0;  
}  
Classes  
For people who don't have much experience with Object-Oriented Programming, we'll be  
talking more about classes in (no pun intended) an upcoming class. Until then, just think  
of classes as similar to C structs with associated functions. Here is the syntax for a class  
definition:  
// Calculator.h  
class Calculator  
{  
public: // external code can call these methods  
Calculator(); // constructor  
~Calculator(); // destructor  
int add(int num1, int num2); // method  
float divide(float numerator, float denominator); // method  
bool getAllowNegatives(); // method  
void setAllowNegatives(bool inValue); // method  
protected: // external code can't access these members  
bool fAllowNegatives; // data member

6  
}; // note the semicolon at the end!  
The functionality, or methods, of the classes are defined as shown:  
// Calculator.cpp  
#include <iostream>  
#include "Calculator.h"  
Calculator::Calculator()  
{  
fAllowNegatives = false; // initialize the data member  
}  
Calculator::~Calculator()  
{  
// nothing much to do in terms of cleanup  
}  
int Calculator::add(int num1, int num2)  
{  
if (!getAllowNegatives() && (num1 < 0 || num2 < 0)) {  
std::cout << "Illegal negative number passed to add()" <<  
std::endl;  
return 0;  
}  
return num1 + num2;  
}  
float Calculator::divide(float numerator, float denominator)  
{  
if (!getAllowNegatives() && (numerator < 0 || denominator < 0)) {  
std::cout << "Illegal negative number passed to divide()" <<  
std::endl;  
return 0;  
}  
return (numerator / denominator);  
}  
bool Calculator::getAllowNegatives()

7  
{  
return fAllowNegatives;  
}  
void Calculator::setAllowNegatives(bool inValue)  
{  
fAllowNegatives = inValue;  
}  
Finally, here's how other parts of the code use and interact with the class:  
// CalculatorTest.cpp  
#include <iostream>  
#include “Calculator.h”  
using namespace std;  
int main(int argc, char\*\* argv)  
{  
Calculator myCalc; // stack-based Calculator  
myCalc.setAllowNegatives(true);  
int result = myCalc.add(2, 2);  
cout << “According to the calculator, 2 + 2 = “ << result << endl;  
Calculator\* myCalc2; // heap-based Calculator  
myCalc2 = new Calculator(); // allocate a new object  
myCalc2->setAllowNegatives(false);  
float result2 = myCalc2->divide(2.5, 0.5);  
cout << "According to the calculator, 2.5 / 0.5 = " << result2 <<  
endl;  
return 0;  
}  
If You Need More Review...  
For most of you, this hand-out was probably just a refresher. However, if you're coming  
from a C background or you need a little more help in any of these areas, check out  
Chapter 1 of the book. It has the same sections with similar examples, but much more  
detail



**Header files available in C++ for Input/Output operations are:**

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip**: iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two keywords **cout in C++** and **cin in C++** are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

insertion operator(**<<**)

extraction operator(**>>**)

* **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream that is used to output the errors. This is also an instance of the ostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
* The main difference between cerr and cout comes when you would like to redirect output using “cout” that gets redirected to file if you use “cerr” the error doesn’t get stored in file.(This is what un-buffered means ..It can’t store the message)

**Type Casting**

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

**Implicit conversion**

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | *short* a=2000;  *int* b;  b=a; |  |

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.  
  
Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | *class* A {};  *class* B { *public*: B (A a) {} };  A a;  B b=a; |  |

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

**Explicit conversion**

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *short* a=2000;  *int* b;  b = (*int*) a; *// c-like cast notation*  b = *int* (a); *// functional notation* |  |

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | *// class type-casting*  *#include <iostream>*  *using* *namespace* std;  *class* CDummy {  *float* i,j;  };  *class* CAddition {  *int* x,y;  *public*:  CAddition (*int* a, *int* b) { x=a; y=b; }  *int* result() { *return* x+y;}  };  *int* main () {  CDummy d;  CAddition \* padd;  padd = (CAddition\*) &d;  cout << padd->result();  *return* 0;  } |  |  |

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

|  |  |  |
| --- | --- | --- |
|  | padd = (CAddition\*) &d; |  |

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result will produce either a run-time error or a unexpected result.  
  
In order to control these types of conversions between classes, we have four specific casting operators: dynamic\_cast, reinterpret\_cast, static\_cast and const\_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.  
  
dynamic\_cast <new\_type> (expression)  
reinterpret\_cast <new\_type> (expression)  
static\_cast <new\_type> (expression)  
const\_cast <new\_type> (expression)  
  
The traditional type-casting equivalents to these expressions would be:  
  
(new\_type) expression  
new\_type (expression)  
  
but each one with its own special characteristics:

**dynamic\_cast**

dynamic\_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.  
  
Therefore, dynamic\_cast is always successful when we cast a class to one of its base classes:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 | *class* CBase { };  *class* CDerived: *public* CBase { };  CBase b; CBase\* pb;  CDerived d; CDerived\* pd;  pb = *dynamic\_cast*<CBase\*>(&d); *// ok: derived-to-base*  pd = *dynamic\_cast*<CDerived\*>(&b); *// wrong: base-to-derived* |  |

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic\_cast unless the base class is polymorphic.  
  
When a class is polymorphic, dynamic\_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | *// dynamic\_cast*  *#include <iostream>*  *#include <exception>*  *using* *namespace* std;  *class* CBase { *virtual* *void* dummy() {} };  *class* CDerived: *public* CBase { *int* a; };  *int* main () {  *try* {  CBase \* pba = *new* CDerived;  CBase \* pbb = *new* CBase;  CDerived \* pd;  pd = *dynamic\_cast*<CDerived\*>(pba);  *if* (pd==0) cout << "Null pointer on first type-cast" << endl;  pd = *dynamic\_cast*<CDerived\*>(pbb);  *if* (pd==0) cout << "Null pointer on second type-cast" << endl;  } *catch* (exception& e) {cout << "Exception: " << e.what();}  *return* 0;  } | Null pointer on second type-cast | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

|  |
| --- |
| **Compatibility note:** dynamic\_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic\_cast to work properly. |

The code tries to perform two dynamic casts from pointer objects of type CBase\* (pba and pbb) to a pointer object of type CDerived\*, but only the first one is successful. Notice their respective initializations:

|  |  |  |
| --- | --- | --- |
| 1 2 | CBase \* pba = *new* CDerived;  CBase \* pbb = *new* CBase; |  |

Even though both are pointers of type CBase\*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic\_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.  
  
When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic\_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.  
  
dynamic\_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void\*).

**static\_cast**

static\_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic\_cast is avoided.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *class* CBase {};  *class* CDerived: *public* CBase {};  CBase \* a = *new* CBase;  CDerived \* b = *static\_cast*<CDerived\*>(a); |  |

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.  
  
static\_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

|  |  |  |
| --- | --- | --- |
| 1 2 | *double* d=3.14159265;  *int* i = *static\_cast*<*int*>(d); |  |

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

**reinterpret\_cast**

reinterpret\_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.  
  
It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.  
  
The conversions that can be performed by reinterpret\_cast but not by static\_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | *class* A {};  *class* B {};  A \* a = *new* A;  B \* b = *reinterpret\_cast*<B\*>(a); |  |

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

**const\_cast**

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | *// const\_cast*  *#include <iostream>*  *using* *namespace* std;  *void* print (*char* \* str)  {  cout << str << endl;  }  *int* main () {  *const* *char* \* c = "sample text";  print ( *const\_cast*<*char* \*> (c) );  *return* 0;  } | sample text | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

**typeid**

typeid allows to check the type of an expression:   
  
typeid (expression)  
  
This operator returns a reference to a constant object of type type\_info that is defined in the standard header file <typeinfo>. This returned value can be compared with another one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | *// typeid*  *#include <iostream>*  *#include <typeinfo>*  *using* *namespace* std;  *int* main () {  *int* \* a,b;  a=0; b=0;  *if* (*typeid*(a) != *typeid*(b))  {  cout << "a and b are of different types:\n";  cout << "a is: " << *typeid*(a).name() << '\n';  cout << "b is: " << *typeid*(b).name() << '\n';  }  *return* 0;  } | a and b are of different types:  a is: int \*  b is: int | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | *// typeid, polymorphic class*  *#include <iostream>*  *#include <typeinfo>*  *#include <exception>*  *using* *namespace* std;  *class* CBase { *virtual* *void* f(){} };  *class* CDerived : *public* CBase {};  *int* main () {  *try* {  CBase\* a = *new* CBase;  CBase\* b = *new* CDerived;  cout << "a is: " << *typeid*(a).name() << '\n';  cout << "b is: " << *typeid*(b).name() << '\n';  cout << "\*a is: " << *typeid*(\*a).name() << '\n';  cout << "\*b is: " << *typeid*(\*b).name() << '\n';  } *catch* (exception& e) { cout << "Exception: " << e.what() << endl; }  *return* 0;  } | a is: class CBase \*  b is: class CBase \*  \*a is: class CBase  \*b is: class CDerived | [Edit & Run](https://www.cplusplus.com/doc/oldtutorial/typecasting/) |

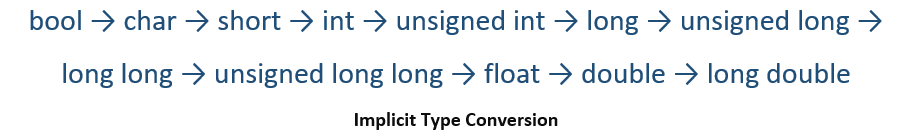
*Note: The string returned by member name of* [*type\_info*](https://www.cplusplus.com/type_info) *depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.*   
Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase \*). However, when typeid is applied to objects (like \*a and \*b) typeid yields their dynamic type (i.e. the type of their most derived complete object).  
  
If the type typeid evaluates is a pointer preceded by the dereference operator (\*), and this pointer has a null value, typeid throws a bad\_typeid exception.  
  
The compiler in the examples above generates names with [type\_info::name](https://www.cplusplus.com/type_info::name) that are easily readable by users, but this is not a requirement: a compiler may just return any string.

The type casting is a method of converting the value of one data type to another data type. It is also known as type conversion. In C++, there are two kinds of conversions, which are given below:

* Implicit Type Casting
* Explicit Type Casting

**Implicit Type Casting**

The implicit type casting happens **automatically** when converting a smaller data types to larger data types. The compiler implicitly typecast the smaller data types to the larger data types. No data will be lost in this process.



**Example**

The example below shows how the implicit type casting is done in C++.

#include <iostream>

using namespace std;

int main (){

char num\_char = 65;

//implicit casting

short num\_short = num\_char;

int num\_int = num\_short;

long num\_long = num\_int;

float num\_float = num\_long;

double num\_double = num\_float;

long double num\_long\_double = num\_double;

//printing variables

cout<<"num\_char = "<<num\_char<<"\n";

cout<<"num\_short = "<<num\_short<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

cout<<"num\_long = "<<num\_long<<"\n";

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_double = "<<num\_double<<"\n";

cout<<"num\_long\_double = "<<num\_long\_double<<"\n";

}

The output of the above code will be:

num\_char = A

num\_short = 65

num\_int = 65

num\_long = 65

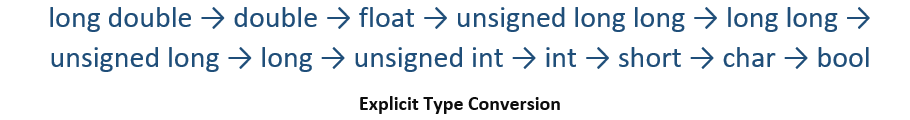
num\_float = 65

num\_double = 65

num\_long\_double = 65

**Explicit Type Casting**

The explicit type casting does not happen automatically. It is performed **manually** by calling the compiler explicitly to typecast the larger data types into the smaller data types. There might be a data loss in this process.



There are three major ways in which explicit conversion in C++ which are mentioned below:

1. C-style type casting
2. Function style type casting
3. Type conversion operators

**1. C-style type casting**

This is also known as **cast notation**. The syntax of this method is given below:

(new\_type)expression;

**2. Function style type casting**

The function style notation can also be used for type casting. The syntax of this style is given below:

new\_type(expression);

**Example**

The example below shows how the perform explicit type casting using **C-style** and **Function style** type casting.

#include <iostream>

using namespace std;

int main (){

long double num\_long\_double = 68.75;

//explicit casting - C style

double num\_double = (double) num\_long\_double;

float num\_float = (float) num\_double;

long num\_long = (long) num\_float;

//explicit casting - Function style

int num\_int = int(num\_long);

short num\_short = short(num\_int);

char num\_char = char(num\_short);

bool num\_bool = bool(num\_char);

//printing variables

cout<<boolalpha;

cout<<"num\_long\_double = "<<num\_long\_double<<"\n";

cout<<"num\_double = "<<num\_double<<"\n";

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_long = "<<num\_long<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

cout<<"num\_short = "<<num\_short<<"\n";

cout<<"num\_char = "<<num\_char<<"\n";

cout<<"num\_bool = "<<num\_bool<<"\n";

}

The output of the above code will be:

num\_long\_double = 68.75

num\_double = 68.75

num\_float = 68.75

num\_long = 68

num\_int = 68

num\_short = 68

num\_char = D

num\_bool = true

**3. Type conversion operators**

A Cast operator is an unary operator which forces one data type to be converted into another data type. C++ supports four types of casting:

1. static\_cast
2. dynamic\_cast
3. const\_cast
4. reinterpret\_cast

**Example**

The example below shows how to use type conversion operators.

#include <iostream>

using namespace std;

int main (){

float num\_float = 100.55;

//using static\_cast operator

int num\_int = static\_cast<int> (num\_float);

//printing variables

cout<<"num\_float = "<<num\_float<<"\n";

cout<<"num\_int = "<<num\_int<<"\n";

}

The output of the above code will be:

num\_float = 100.55

num\_int = 100

STL:

Vector:

C++ vector member functions can be put under the following categories:

construction, capacity, iterators, element access, and modifiers. Each of these categories has many functions. All these functions are not used in many applications. This article explains the most useful of these functions. With the exception of the modifiers category, not more than three functions for each category are explained in this article. The modifiers category can be broken down into more sub categories. In each of these sub categories, not more than three functions will be explained. If more than three functions are to be explained for a given category, then they will be taught by way of illustration.

## Construction/Destruction

The following code segments show different ways of creating the same vector:

vector <float> vtr;  
  
        vtr.push\_back(5.5);  
  
        vtr.push\_back(6.6);  
  
        vtr.push\_back(7.7);  
  
        vtr.push\_back(8.8);  
  
        vtr.push\_back(9.9);  
  
    vector <float> vtr(3);    //with initial number of elements  
  
        vtr[0] = 5.5;  
  
        vtr[1] = 6.6;  
  
        vtr[2] = 7.7;  
  
    vector <float> vtr(5, 0.0);    //No. Elements:5; each value:0.0  
  
    vector <float> vtr{5.5, 6.6, 7.7, 8.8, 9.9};    //initializing  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};    //constructing and copying  
  
    vector <float> vtr;  
  
    vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector <float> vtr1{5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector <float> vtr2(vtr1);  
  
    const vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};

A **const vector** is a vector whose elements cannot be changed. The values are read-only.

**Capacity**

**size() const noexcept**

The number of elements in a vector is returned by this member function. With the following code segment, the output is 5:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
  
    float sz = vtr.size();  
  
  
    cout << sz << '\n';  
  
  
empty() const noexcept

This method returns true (1) if the vector has no element and false (0) if the vector has at least one element. With the following code, the output is 1 (for true):

    vector <float> vtr = {};  
  
    bool bl = vtr.empty();  
  
    cout << bl << '\n';

**Iterator and Vector Access**

An iterator is an elaborated pointer. When the vector, **vtr** has been created, **vtr.begin()** would return an iterator, pointing to the first element of the list. It can then be incremented to access the elements after the first, accordingly.

When the vector, **vtr** has been created, **vtr.end()** would return an iterator, pointing just after the last element of the list. It can then be decremented to access the last element and elements before the last, accordingly. The following program illustrates this:

#include <iostream>  
  
#include <vector>  
  
using namespace std;  
  
int main()  
  
{  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    vector<float>::iterator iterB = vtr.begin();  
  
    iterB++;  
  
    vector<float>::iterator iterE = vtr.end();  
  
    iterE--;  
  
    cout << \*iterB << ", " << \*iterE << ' ' << endl;  
  
    \*iterB = 66.66; \*iterE = 99.99;  
  
    cout << \*iterB << ", " << \*iterE << ' ' << endl;  
  
    return 0;  
  
}

The output is:

    6.6, 9.9  
  
    66.66, 99.99

The values of two elements were accessed, read and changed by two iterators.

**Element Access**

**at(i)**

This is similar to vtr[i], and it is better. It can be used to read or change the value of an element. Index counting begins from zero. The reader can test the following program:

#include <iostream>  
  
#include <vector>  
  
using namespace std;  
  
int main()  
  
{  
  
    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    cout << vtr[1] << ", " << vtr[4] << ' ' << endl;  
  
    vtr[1] = 66.66; vtr[4] = 99.99;  
  
    cout << vtr[1] << ", " << vtr[4] << ' ' << endl;  
  
    return 0;  
  
}

The output is:

   6.6, 9.9  
  
    66.66, 99.99

The values of two elements were accessed, read and changed through referencing.

**Returning the First Value**

The following code returns (copies out) the first element:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    float val = vtr.front();  
  
    cout << val << endl;

The output is, 5.5. The member function used here is: front().

**Returning the Last Value**

The following code returns (copies out) the last element:

    vector <float> vtr = {5.5, 6.6, 7.7, 8.8, 9.9};  
  
    float val = vtr.back();  
  
    cout << val << endl;

The output is, 9.9. The member function used here is: back()

# C++ Class Templates

Templates are powerful features of C++ which allows us to write generic programs. There are two ways we can implement templates:

* Function Templates
* Class Templates

# C++ Function Template

Templates are powerful features of C++ which allows us to write generic programs.

We can create a single function to work with different data types by using a template.

### Defining a Function Template

A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

template <typename T>

T functionName(T parameter1, T parameter2, ...) {

// code

}

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword.

When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

### Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

functionName<dataType>(parameter1, parameter2,...);

For example, let us consider a template that adds two numbers:

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

We can then call it in the main() function to add int and double numbers.

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << result1 << endl;

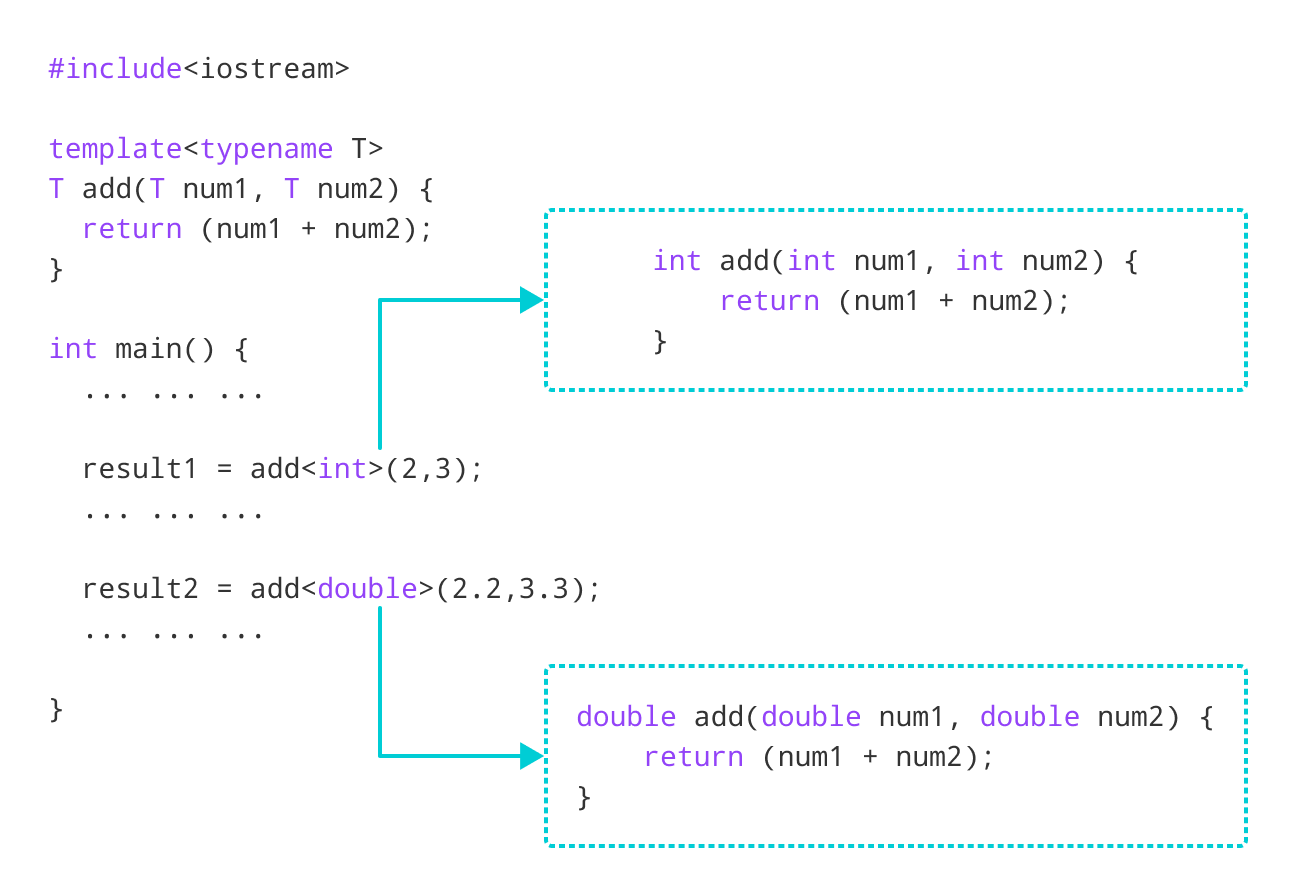
// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << result2 << endl;

return 0;

}

Function Call based on data types

### Example: Adding Two Numbers Using Function Templates

#include <iostream>

using namespace std;

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << "2 + 3 = " << result1 << endl;

// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << "2.2 + 3.3 = " << result2 << endl;

return 0;

}

**Output**

2 + 3 = 5

2.2 + 3.3 = 5.5

Similar to function templates, we can use class templates to create a single class to work with different data types.

Class templates come in handy as they can make our code shorter and more manageable.

## Class Template Declaration

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

template <class T>

class className {

private:

T var;

... .. ...

public:

T functionName(T arg);

... .. ...

};

In the above declaration, T is the template argument which is a placeholder for the data type used, and class is a keyword.

Inside the class body, a member variable var and a member function functionName() are both of type T.

## Creating a Class Template Object

Once we've declared and defined a class template, we can create its objects in other classes or functions (such as the main() function) with the following syntax

className<dataType> classObject;

For example,

className<int> classObject;

className<float> classObject;

className<string> classObject;

## Example 1: C++ Class Templates

// C++ program to demonstrate the use of class templates

#include <iostream>

using namespace std;

// Class template

template <class T>

class Number {

private:

// Variable of type T

T num;

public:

Number(T n) : num(n) {} // constructor

T getNum() {

return num;

}

};

int main() {

// create object with int type

Number<int> numberInt(7);

// create object with double type

Number<double> numberDouble(7.7);

cout << "int Number = " << numberInt.getNum() << endl;

cout << "double Number = " << numberDouble.getNum() << endl;

return 0;

}

**Output**

int Number = 7

double Number = 7.7

In this program. we have created a class template Number with the code

template <class T>

class Number {

private:

T num;

public:

Number(T n) : num(n) {}

T getNum() { return num; }

};

Notice that the variable num, the constructor argument n, and the function getNum() are of type T, or have a return type T. That means that they can be of any type.

In main(), we have implemented the class template by creating its objects

Number<int> numberInt(7);

Number<double> numberDouble(7.7);

Notice the codes Number<int> and Number<double> in the code above.

This creates a class definition each for int and float, which are then used accordingly.

It is compulsory to specify the type when declaring objects of class templates. Otherwise, the compiler will produce an error.

//Error

Number numberInt(7);

Number numberDouble(7.7);

## Defining a Class Member Outside the Class Template

Suppose we need to define a function outside of the class template. We can do this with the following code:

template <class T>

class ClassName {

... .. ...

// Function prototype

returnType functionName();

};

// Function definition

template <class T>

returnType ClassName<T>::functionName() {

// code

}

Notice that the code template <class T> is repeated while defining the function outside of the class. This is necessary and is part of the syntax.

If we look at the code in **Example 1**, we have a function getNum() that is defined inside the class template Number.

We can define getNum() outside of Number with the following code:

template <class T>

class Number {

... .. ...

// Function prototype

T getnum();

};

// Function definition

template <class T>

T Number<T>::getNum() {

return num;

}

## Example 2: Simple Calculator Using Class Templates

This program uses a class template to perform addition, subtraction, multiplication and division of two variables num1 and num2.

The variables can be of any type, though we have only used int and float types in this example.

#include <iostream>

using namespace std;

template <class T>

class Calculator {

private:

T num1, num2;

public:

Calculator(T n1, T n2) {

num1 = n1;

num2 = n2;

}

void displayResult() {

cout << "Numbers: " << num1 << " and " << num2 << "." << endl;

cout << num1 << " + " << num2 << " = " << add() << endl;

cout << num1 << " - " << num2 << " = " << subtract() << endl;

cout << num1 << " \* " << num2 << " = " << multiply() << endl;

cout << num1 << " / " << num2 << " = " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main() {

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl

<< "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

**Output**

Int results:

Numbers: 2 and 1.

2 + 1 = 3

2 - 1 = 1

2 \* 1 = 2

2 / 1 = 2

Float results:

Numbers: 2.4 and 1.2.

2.4 + 1.2 = 3.6

2.4 - 1.2 = 1.2

2.4 \* 1.2 = 2.88

2.4 / 1.2 = 2

In the above program, we have declared a class template Calculator.

The class contains two private members of type T: num1 & num2, and a constructor to initialize the members.

We also have add(), subtract(), multiply(), and divide() functions that have the return type T. We also have a void function displayResult() that prints out the results of the other functions.

In main(), we have created two objects of Calculator: one for int data type and another for float data type.

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

This prompts the compiler to create two class definitions for the respective data types during compilation.

## C++ Class Templates With Multiple Parameters

In C++, we can use multiple template parameters and even use default arguments for those parameters. For example,

template <class T, class U, class V = int>

class ClassName {

private:

T member1;

U member2;

V member3;

... .. ...

public:

... .. ...

};

### Example 3: C++ Templates With Multiple Parameters

#include <iostream>

using namespace std;

// Class template with multiple and default parameters

template <class T, class U, class V = char>

class ClassTemplate {

private:

T var1;

U var2;

V var3;

public:

ClassTemplate(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {} // constructor

void printVar() {

cout << "var1 = " << var1 << endl;

cout << "var2 = " << var2 << endl;

cout << "var3 = " << var3 << endl;

}

};

int main() {

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

cout << "obj1 values: " << endl;

obj1.printVar();

// create object with int, double and bool types

ClassTemplate<double, char, bool> obj2(8.8, 'a', false);

cout << "\nobj2 values: " << endl;

obj2.printVar();

return 0;

}

**Output**

obj1 values:

var1 = 7

var2 = 7.7

var3 = c

obj2 values:

var1 = 8.8

var2 = a

var3 = 0

In this program, we have created a class template, named ClassTemplate, with three parameters, with one of them being a default parameter.

template <class T, class U, class V = char>

class ClassTemplate {

// code

};

Notice the code class V = char. This means that V is a default parameter whose default type is char.

Inside ClassTemplate, we declare 3 variables var1, var2 and var3, each corresponding to one of the template parameters.

class ClassTemplate {

private:

T var1;

U var2;

V var3;

... .. ...

... .. ...

};

In main(), we create two objects of ClassTemplate with the code

// create object with int, double and char types

ClassTemplate<int, double> obj1(7, 7.7, 'c');

// create object with double, char and bool types

ClassTemplate<double, char, bool> obj2(8, 8.8, false);

Here,

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **T** | **U** | **V** |
| obj1 | int | double | char |
| obj2 | double | char | bool |

For obj1, T = int, U = double and V = char.

For obj2, T = double, U = char and V = bool.

**Standard Containers**

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.  
  
The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).  
  
Containers replicate structures very commonly used in programming: dynamic arrays ([vector](https://www.cplusplus.com/vector)), queues ([queue](https://www.cplusplus.com/queue)), stacks ([stack](https://www.cplusplus.com/stack)), heaps ([priority\_queue](https://www.cplusplus.com/priority_queue)), linked lists ([list](https://www.cplusplus.com/list)), trees ([set](https://www.cplusplus.com/set)), associative arrays ([map](https://www.cplusplus.com/map))...  
  
Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.  
  
[stack](https://www.cplusplus.com/stack), [queue](https://www.cplusplus.com/queue) and [priority\_queue](https://www.cplusplus.com/priority_queue) are implemented as *container adaptors*. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as [deque](https://www.cplusplus.com/deque) or [list](https://www.cplusplus.com/list)) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the *container adaptor* independently of the underlying *container* class used.

**Container class templates**

**Sequence containers**:

[**array**](https://www.cplusplus.com/reference/array/array/)

Array class (class template )

[**vector**](https://www.cplusplus.com/reference/vector/vector/)

Vector (class template )

[**deque**](https://www.cplusplus.com/reference/deque/deque/)

Double ended queue (class template )

[**forward\_list**](https://www.cplusplus.com/reference/forward_list/forward_list/)

Forward list (class template )

[**list**](https://www.cplusplus.com/reference/list/list/)

List (class template )

**Container adaptors**:

[**stack**](https://www.cplusplus.com/reference/stack/stack/)

LIFO stack (class template )

[**queue**](https://www.cplusplus.com/reference/queue/queue/)

FIFO queue (class template )

[**priority\_queue**](https://www.cplusplus.com/reference/queue/priority_queue/)

Priority queue (class template )

**Associative containers**:

[**set**](https://www.cplusplus.com/reference/set/set/)

Set (class template )

[**multiset**](https://www.cplusplus.com/reference/set/multiset/)

Multiple-key set (class template )

[**map**](https://www.cplusplus.com/reference/map/map/)

Map (class template )

[**multimap**](https://www.cplusplus.com/reference/map/multimap/)

Multiple-key map (class template )

**Unordered associative containers**:

[**unordered\_set**](https://www.cplusplus.com/reference/unordered_set/unordered_set/)

Unordered Set (class template )

[**unordered\_multiset**](https://www.cplusplus.com/reference/unordered_set/unordered_multiset/)

Unordered Multiset (class template )

[**unordered\_map**](https://www.cplusplus.com/reference/unordered_map/unordered_map/)

Unordered Map (class template )

[**unordered\_multimap**](https://www.cplusplus.com/reference/unordered_map/unordered_multimap/)

Unordered Multimap (class template )

**Call by Value**

Here in this example, we can see that the original value does not get modified. The value being passed is locally stored in a memory location. Any changes in the value will only change the local function and not the original value.

// C++ call by Value

#include <iostream>

using namespace std;

void funct(int x)   // Only a copy of the variable is passed

{

   x = 30;

}

int main() {

   int x = 10;

   funct(x);   // Call by value where the variable is passed

   cout << "x = " << x;

   return 0;

}

**Output**

x = 10

**Call by Reference**

While using call by reference, the value does get changed as we are passing an address. The actual and formal parameters both refer to the same memory location and value. So any value that is changed in the function gets actually changed in the original value.

// C++ call by Reference

#include <iostream>

using namespace std;

void funct(int \*p) // The address of the variable is accepted by a pointer

{

   \*p = 40;   // The value at the address p is updated

}

int main() {

   int x = 10;

   funct(&x);  // Call by Reference where the address is passed

   cout << "x = " << x;

   return 0;

}

**Output**

x = 40

**Key Points to Remember**

For C++ programs, you need to remember the following points for Call by Value and Call by Reference:

* Call by values do not change the values that are passed but Call by reference does.
* You have a choice to pass a variable either by value or by reference except for arrays and functions.
* Arrays and functions are always passed and returned as Call by reference and using pointer operations
* Call by reference is more memory efficient than Call by value.

**Difference Between Call by Value and Call by Reference**

The key differences between them are:

|  |  |
| --- | --- |
| Call by Value | Call by Reference |
| A copy of the value is passed to the function | An address of the value is passed to the function |
| Any change to the value does not reflect back to the original value | All changes get reflected back to the original value |
| Both Formal and Actual Arguments have different memory locations | The same memory location is reserved for the Formal and Actual argument |

# C++ Call by Reference: Using pointers [With Examples]

In this tutorial, we will learn about C++ call by reference to pass pointers as an argument to the function with the help of examples.

In the [C++ Functions](https://www.programiz.com/cpp-programming/function) tutorial, we learned about passing arguments to a function. This method used is called passing by value because the actual value is passed.

However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For example,

// function that takes value as parameter

void func1(int numVal) {

// code

}

// function that takes reference as parameter

// notice the & before the parameter

void func2(int &numRef) {

// code

}

int main() {

int num = 5;

// pass by value

func1(num);

// pass by reference

func2(num);

return 0;

}

Notice the & in void func2(int &numRef). This denotes that we are using the address of the variable as our parameter.

So, when we call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num variable instead of the value **5**.

C++ Pass by Value vs. Pass by Reference

## Example 1: Passing by reference without pointers

#include <iostream>

using namespace std;

// function definition to swap values

void swap(int &n1, int &n2) {

int temp;

temp = n1;

n1 = n2;

n2 = temp;

}

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

// call function to swap numbers

swap(a, b);

cout << "\nAfter swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

In this program, we passed the variables a and b to the swap() function. Notice the function definition,

void swap(int &n1, int &n2)

Here, we are using & to denote that the function will accept addresses as its parameters.

Hence, the compiler can identify that instead of actual values, the reference of the variables is passed to function parameters.

In the swap() function, the function parameters n1 and n2 are pointing to the same value as the variables a and b respectively. Hence the swapping takes place on actual value.

The same task can be done using the pointers. To learn about pointers, visit [C++ Pointers](https://www.programiz.com/cpp-programming/pointers).

## Example 2: Passing by reference using pointers

#include <iostream>

using namespace std;

// function prototype with pointer as parameters

void swap(int\*, int\*);

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

// call function by passing variable addresses

swap(&a, &b);

cout << "\nAfter swapping" << endl;

cout << "a = " << a << endl;

cout << "b = " << b << endl;

return 0;

}

// function definition to swap numbers

void swap(int\* n1, int\* n2) {

int temp;

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

Here, we can see the output is the same as the previous example. Notice the line,

// &a is address of a

// &b is address of b

swap(&a, &b);

Here, the address of the variable is passed during the function call rather than the variable.

Since the address is passed instead of value, a dereference operator \* must be used to access the value stored in that address.

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

\*n1 and \*n2 gives the value stored at address n1 and n2 respectively.

Since n1 and n2 contain the addresses of a and b, anything is done to \*n1 and \*n2 will change the actual values of a and b.

Hence, when we print the values of a and b in the main() function, the values are changed.

# Nested classes (C++ only)

A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

class Car{

//data member 1

int seats;

//inner class 1

class Engine{ };

//inner class 2

class GearBox {

// The compiler cannot allow the following

// declaration because A::B is private:

// Engine b;

//data member 2

int gears;

//member function-1

void f(Car\* p, int i) {

// The compiler cannot allow the following

// statement because A::x is private:

// p->seats = i;

}

};

void g(C\* p) {

// The compiler cannot allow the following

// statement because C::y is private:

// int z = p->y;

}

};

int main() { }

The compiler would not allow the declaration of object b because class A::B is private. The compiler would not allow the statement p->x = i because A::x is private. The compiler would not allow the statement int z = p->y because C::y is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members x and y and member functions f() and g() of the nested class nested by using a qualified type name. Qualified type names allow you to define a typedef to represent a qualified class name. You can then use the typedef with the :: (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

class World

{

public:

class Car

{

public:

//data members

static int wheels;

static int steering;

int seats;

int gears;

//member functions

int start();

int stop();

};

};

//static member initialization

int World::Car::x = 4;

//defining member function

int World::Car:start() { return 0; };

typedef World::Car globalcar; // define a typedef

int globalcar::steering = 1;

// use typedef with ::

int globalcar::stop() { return 0; };

However, using a typedef to represent a nested class name hides information and may make the code harder to understand.

You cannot use a typedef name in an elaborated type specifier. To illustrate, you cannot use the following declaration in the above example:

class globalcar car;

A nested class may inherit from private members of its enclosing class. The following example demonstrates this:

class A {

private:

class B { };

B \*z;

class C : private B {

private:

B y;

// A::B y2;

C \*x;

// A::C \*x2;

};

};

The nested class A::C inherits from A::B. The compiler does not allow the declarations A::B y2 and A::C \*x2 because both A::B and A::C are private.

**Difference between inspector and mutator functions:**

An object state is returned without modifying the object’s ***abstract state*** by using function called inspector. Invoking an inspector does not cause any noticeable change in the object’s behaviour of any of the functions of that object.

A mutator, on the other hand, changes the ***state*** of an object which is noticeable by outsiders. It means, it changes the abstract state of the object.

**Smart pointer**:

A smart pointer is an abstract data type that simulates a pointer while providing added features, such as automatic memory management or bounds checking. Such features are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency. Smart pointers typically keep track of the memory they point to, and may also be used to manage other resources, such as network connections and file handles. Smart pointers were first popularized in the programming language C++ during the first half of the 1990s as rebuttal to criticisms of C++'s lack of automatic garbage collection.

Smart pointer is an abstract data type by using which we can make a normal pointer in such way that it can be used as memory management like file handling, network sockets etc., also it can do many things like automatic destruction, reference counting etc.

Smart pointer in C++, can be implemented as template class, which is overloaded with \* and -> operator. auto\_ptr, shared\_ptr, unique\_ptr and weak\_ptr are the forms of smart pointer can be implemented by C++ libraries.

## Example

[Live Demo](http://tpcg.io/Ekj2aD)

#include<iostream>

using namespace std;

// A generic smart pointer class

template <class T>

class Smartpointer {

   T \*p; // Actual pointer

   public:

      // Constructor

      Smartpointer(T \*ptr = NULL) {

         p = ptr;

      }

   // Destructor

   ~Smartpointer() {

      delete(p);

   }

   // Overloading dereferencing operator

   T & operator \* () {

      return \*p;

   }

   // Overloding arrow operator so that members of T can be accessed

   // like a pointer

   T \* operator -> () {

      return p;

   }

};

int main() {

   Smartpointer<int> p(new int());

   \*p = 26;

   cout << "Value is: "<<\*p;

   return 0;

}

## Shared Pointers in C++

shared\_ptr is one of the form of smart pointer can be implemented by C++ libraries. It is a container of raw pointer and a reference counting (a technique of storing the number of references, pointers or handles to a resource such as an object, block of memory, disk space or other resources) ownership structure of its contained pointer in cooperation with all copies of the shared\_ptr.

An object which is referenced by the contained raw pointer will be destroyed only when the all copy is detroyed of shared\_ptr.

## Example

[Live Demo](http://tpcg.io/L7u29b)

#include<iostream>

#include<memory>

using namespace std;

int main() {

   shared\_ptr<int> ptr(new int(7));

   shared\_ptr<int> ptr1(new int(6));

   cout << ptr << endl;

   cout << ptr1 << endl;

   // Returns the number of shared\_ptr objects

   // referring to the same managed object.

   cout << ptr.use\_count() << endl;

   cout << ptr1.use\_count() << endl;

   // Relinquishes ownership of ptr on the object

   // and pointer becomes NULL

   ptr.reset();

   cout << ptr.get() << endl;

   cout << ptr1.use\_count() << endl;

   cout << ptr1.get() << endl;

   return 0;

}

Weak pointer: A weak pointer is a smart pointer that does not take ownership of an object but act as an observer. In other words, it does not participate in reference counting to delete an object or extend its lifetime. Weak pointers are mainly used to break the circular dependency that shared pointers create.

<https://iamsorush.com/posts/weak-pointer-cpp/>

**Lambda Expressions**

The term “lambda” is short for *lambda expression*, and a lambda is just that:

1. **Lamdba expression** — An *expression* that specifies an anonymous function object, just syntax for creating an unnamed function.

2. **Lamdba function —** This term is used interchangeably with the term “lambda expression.”

As such, it exists only in a program’s source code.

A lambda does not exist at runtime.

A lambda expression **specifies an object**, **not just a function without a name,** capable of capturing variables in scope.

* Lambdas can frequently be passed around as objects.
* A lambda is essentially a function object that is specified inline.
* In addition to its own function parameters, a lambda expression can refer to local variables in the scope of its definition.

**What are closures?**

Closures are special functions that can capture the environment, i.e. variables within a **lexical scope**.

*A* ***closure*** *is any function that* ***closes over*** *the* ***environment*** *in which it was defined. This means that it can access variables, not in its parameter list.*

**\* What is C++ specific part here**

*A closure is a general concept in programming that originated from functional programming. When we talk about the closures in C++, they always come with lambda expressions (some scholars prefer the inclusion of function object in this)*

1. In c++ a lambda expression is the syntax used to create a special temporary object that behaves similarly to how function objects behave.
2. The C++ standard specifically refers to this type of object as a *closure object*. This is a little bit at odds with the broader definition of a closure, which refers to any function, anonymous or not, that captures variables from the environment they are defined in.
3. As far as the standard is concerned, all instantiations of lambda expressions are closure objects, even if they don’t have any captures in their capture group.

*In any other language like Python, closure is unrelated to Lambdas*

**What is a Closure anyway in C++?**

A closure is a general concept in programming that originated from functional programming. When we talk about the closures in C++, they always come with lambda expressions. A closure is any function that closes over the environment in which it was defined. This means that it can access variables not in its parameter list.

A value\* defined (rather encapsulated) by lambda expression that consists of both the code as well as the values of the variables referred to in the code.  
(We will discuss this asterisk \* further in this article)

So closure is an anonymous function object that is created automatically by the compiler as the result of a lambda expression. A closure stores those variables from the scope of the definition of the lambda expression that is used in the lambda expression.

The runtime effect of a lambda expression is the generation of an object. Such objects are known as *closures*.

A closure is a function that encloses its surrounding state by referencing fields external to its body. The enclosed state remains across invocations of the closure

**Lambdas vs. Closures for C++**

S**cott Meyers** puts it beautifully — “**The distinction between a lambda and the corresponding closure is precisely equivalent to the distinction between a class and an instance of the class**”.

Closures are to lambdas as objects are to classes.

As we know, a class exists only in source code; it doesn’t exist at runtime. What exists at runtime are objects of the class type. Similarily -

Each lambda expression causes a unique class to be generated (during compilation) and also causes an object of that class type–a closure–to be created (at runtime).

1. Lambdas occupy no data memory at runtime, for example, though they may occupy code memory.
2. Closures occupy data memory, but not code memory.

**Distinction by Examples**

Let’s take an example —

**auto f =** [&](int x, int y) { return fudgeFactor \* (x + y); };// the expression to the right of the "=" is the lambda expression (i.e., "the lambda"),**// The runtime object created by that expression is the closure.**

f itself is not a closure, it is a ***copy of the closure* 🥴**

**Ok, what do you mean?**

The process of copying the closure into f may be optimized into a move but that doesn't change the fact that f itself is not the closure.

The actual closure object is a temporary that’s typically destroyed at the end of the statement unless you bind it to a [forwarding reference(a.ka. Universal reference)](https://medium.com/pranayaggarwal25/universal-reference-perfect-forwarding-5664514cacf9) or lvalue-reference-to-const.

//===============================================================//auto&& **rrefToClosure** = [&](int x, int y) { return fudgeFactor \* (x + y); };const auto& **lrefToConstToClosure** = [&](int x, int y) { return fudgeFactor \* (x + y); };//===============================================================//

Let’s take another example —

//===============================================================//std**::**function**<void**(**void**)**>** closureWrapper1()  
{  
 **int** x **=** 10;  
 **return** [&x](){ std**::**cout **<<** "Value in the closure: " **<<** x++ **<<** std**::**endl; };  
}**int** **main**()  
{  
 **int** x **=** 10;  
 **auto** func0 **=** [**&**x](){x **+=** 1; std**::**cout **<<** "Value in the closure: " **<<** x **<<** std**::**endl;};  
   
 func0(); // Prints 11std**::**function**<void**(**void**)**>** func1 **=** closureWrapper1();func1(); // Prints garbage value + 1 =~ garbage value}//===============================================================//

func1 is not closure Instead, it’s astd::function wrapper object that wrapped a closure.

func0 is a copy of a closure created by the lambda expression written after it.

**Close analog to a closure**

**Function Object (Functor)** — Function object overload the operator(). It could capture the values by making a copy of the variables to its member variables. The shortcoming is that for each different function call, regardless of how simple it is, we would have to implement a new class, whereas implementing a lambda expression is faster.