C++ programming

NOMENCLATURE

**argument -** Value passed to a function.

**assignment -** Obliterates an object’s current value and replaces that value by a new one.

**block -** Sequence of zero or more statements enclosed in curly braces.

**buffer -** A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently from actions in the program. Output buffers can be explicitly flushed to force the buffer to be written. By default, reading **cin** flushes **cout**; **cout** is also flushed when the program ends normally.

**built-in type -** Type, such as int, defined by the language.

**cerr -** ostream object tied to the standard error, which often writes to the same device as the standard output. By default, writes to cerr are not buffered. Usually used for error messages or other output that is not part of the normal logic of the program.

**character string literal -** Another term for string literal.

**cin -** istream object used to read from the standard input.

**class -** Facility for defining our own data structures together with associated operations. The class is one of the most fundamental features in C++. Library types, such as **istream** and **ostream**, are classes.

**class type -** A type defined by a class. The name of the type is the class name.

**clog -** ostream object tied to the standard error. By default, writes to clog are buffered. Usually used to report information about program execution to a log file.

**comments -** Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a //. Everything from the // to the end of the line is a comment.

Paired comments begin with a /\* and include all text up to the next \*/.

**condition -** An expression that is evaluated as true or false. A value of zero is false, any other value yields true.

**cout -** ostream object used to write to the standard output. Ordinarily used to write the output of a program.

**curly brace -** Curly braces delimit blocks. An open curly ({) starts a block; a close curly (}) ends one.

**data structure -** A logical grouping of data and operations on that data.

**edit-compile-debug -** The process of getting a program to execute properly.

**end-of-file -** System-specific marker that indicates that there is no more input in a file.

**expression** The smallest unit of computation. An expression consists of one or more operands and usually one or more operators. Expressions are evaluated to produce a result. For example, assuming i and j are ints, then i + j is an expression and yields the sum of the two int values.

**for statement** Iteration statement that provides iterative execution. Often used to repeat a calculation a fixed number of times.

**function** Named unit of computation.

**function body** Block that defines the actions performed by a function.

**function name** Name by which a function is known and can be called.

**header** Mechanism whereby the definitions of a class or other names are made available to multiple programs. A program uses a header through a #include directive.

**if statement** Conditional execution based on the value of a specified condition. If the condition is true, the if body is executed. If not, the else body is executed if there is one.

**initialize** Give an object a value at the same time that it is created.

**iostream** Header that provides the library types for stream-oriented input and output.

**istream** Library type providing stream oriented input.

**library type** Type, such as istream, defined by the standard library.

**main** Function called by the operating system to execute a C++ program. Each program must have one and only one function named main.

**manipulator** Object, such as std::endl, that when read or written “manipulates” the stream itself.

**member function** Operation defined by a class. Member functions ordinarily are called to operate on a specific object.

**method** Synonym for member function.

**namespace** Mechanism for putting names defined by a library into a single place. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace std.

**ostream** Library type providing streamoriented output.

**parameter list** Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

**return type** Type of the value returned by a function.

**source file** Termused to describe a file that contains a C++ program.

**standard error** Output stream used for error reporting. Ordinarily, the standard output and the standard error are tied to the window in which the program is executed.

**standard input -** Input stream usually associated with the window in which the program executes.

**standard library -** Collection of types and functions that every C++ compiler must support. The library provides the types that support IO. C++ programmers tend to talk about “the library,”meaning the entire standard library. They also tend to refer to particular parts of the library by referring to a library type, such as the “iostream library,” meaning the part of the standard library that defines the IO classes.

**standard output -** Output stream usually associated with the window in which the program executes.

**statement -** A part of a program that specifies an action to take place when the program is executed. An expression followed by a semicolon is a statement; other kinds of statements include blocks and if, for, and while statements, all of which contain other statements within themselves.

**std -** Name of the namespace used by the standard library. std::cout indicates that we’re using the name cout defined in the std namespace.

**string literal -** Sequence of zero or more characters enclosed in double quotes ("a string literal").

**uninitialized variable -** Variable that is not given an initial value. Variables of class type forwhich no initial value is specified are initialized as specified by the class definition. Variables of built-in type defined inside a function are uninitialized unless explicitly initialized. It is an error to try to use the value of an uninitialized variable. *Uninitialized variables are a rich source of bugs*.

**variable -** A named object.

**while statement -** Iteration statement that provides iterative execution so long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

**() operator -** Call operator. A pair of parentheses “()” following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

**++ operator** Increment operator. Adds 1 to the operand; ++i is equivalent to i = i + 1.

**+= operator** Compound assignment operator that adds the right-hand operand to the left and stores the result in the left-hand operand; a += b is equivalent to a = a + b.

**. operator** Dot operator. Left-hand operand must be an object of class type and the right-hand operand must be the name of a member of that object. The operator yields the named member of the given object.

**:: operator** Scope operator. Among other uses, the scope operator is used to access names in a namespace. For example, std::cout denotes the name cout from the namespace std.

**= operator** Assigns the value of the righthand operand to the object denoted by the left-hand operand.

**-- operator** Decrement operator. Subtracts 1 from the operand; --i is equivalent to i = i - 1.

**<< operator** Output operator. Writes the right-hand operand to the output stream indicated by the left-hand operand: cout << "hi" writes hi to the standard output. Output operations can be chained together: cout << "hi" << "bye" writes hibye.

**>> operator** Input operator. Reads from the input stream specified by the left-hand operand into the right-hand operand: cin >> i reads the next value on the standard input into i. Input operations can be chained together: cin >> i >> j reads first into i and then into j.

**#include** Directive that makes code in a header available to a program.

**== operator** The equality operator. Tests whether the left-hand operand is equal to the right-hand operand.

**!= operator** The inequality operator. Tests whether the left-hand operand is not equal to the right-hand operand.

**<= operator** The less-than-or-equal operator. Tests whether the left-hand operand is less than or equal to the right-hand operand.

**< operator** The less-than operator. Tests whether the left-hand operand is less than the right-hand operand.

**>= operator** Greater-than-or-equal operator. Tests whether the left-hand operand is greater than or equal to the right-hand operand.

**> operator** Greater-than operator. Tests whether the left-hand operand is greater than the right-hand operand.

LITERALS

**Decimal** – numeric representation of a bit e.g. 01011110 is 94

* By default, this is signed

**Octal** – represents bits in groups of 3 e.g. 01011110 is 0135

**Hex** – represents bits in groups of 4, e.g. 01011110 is 0x5E

TYPES

QUALIFIERS

**Const –** defines a variable whose value cannot be changed upon initialization.

* When defining a pointer to a const we must make the pointer to a const explicit and we do so by using the qualifier. We can then not assign to this pointer. A const pointer can point to nonconst the value of which can change but it cannot be changed through the const pointer. This is considered a low-level const.
* We can also define a const pointer, which is defined with **\*const** qualifier. This pointer can never be changed after definition but the object it points to can. This is considered a top-level const.
* One can assign top-level consts to low-level but not the other way around.
* Consts differ from constant expression, because the latter are these expressions which can be evaluated at compile time, e.g., literals.

**Constexpr** – a constant expression qualifier that must be initialized by a constant expression. i.e. values that can be processed at compile time.

* Unlike **const, constexpr** defines a constant pointer when applied to pointers. In other words, there can be no pointer to a constant experion as there can be consts, only constexp pointers.

**Extern** – defines a variable that is shared across multiple files

**Notes**

* When numerical variables are defined outside of a function they are automatically initialized to zero, inside functions they are uninitialized. Strings are initialized to be empty.
* **A declaration provides basic attributes of a symbol: its type and its name.** A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored.
* **C++ is statically typed** means that types are checked only at compile type.
* [**Namespaces**](https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170)allow us to define functions within a set scope which would limit them from clashing with definitions within other scopes
* When creating a pointer make sure to initialize it every time.
* **Templates** are constructs that generate an ordinary type or a function at compile time based on the arguments the user supplies. [Link](https://learn.microsoft.com/en-us/cpp/cpp/templates-cpp?view=msvc-170)
* **#pragma once** is a way to ensure that the contents of a header file are only called once

CLASS VARIABLES

**Public** – can be accessed anywhere outside of the class, including other classes and instances of the class.

**Protected** – can be accessed within the class and the classes hat inherit them.

**Private** – can only be accessed within a given class.