**C++ Introduction**

**Statement:** Type of instruction that causes the program to perform some action.

* End in ‘;’.
* Types of statement:
  + *Declaration*
  + *Jump*
  + *Expression*
  + *Compound*
  + *Selection (conditionals)*
  + *Iteration (Loops)*
  + *Try Blocks*

**Function:** Collection of statements that get executed sequentially (top to bottom).

* ***Identifier:*** Name of function.
* Note: Every C++ program must have a function named **main.** When the program is run, the **main** function is executed in sequential order.

**Cout: “**Character Output”

**Syntax:** Rules that govern how sentences are constructed in a language.

* **Syntax error:** Violating syntax rules of a language, detected by the compiler.

**Quiz:**

What is a statement?

*A statement is an instruction that performs an action.*

What is a function?

*A function is a collection of statements that gets executed sequentially.*

What is the name of the function that all programs must have?

*The* ***main*** *function.*

When a program is run, where does the execution start?

*The execution starts from top to bottom. X (*Sol. Execution starts with the first statement inside the **main** function.)

What symbols are statements in C++ often ended with?

*The ‘;’ or semicolon symbol is used to end statements.*

What is a syntax error?

*A syntax error is a compile error (*occurs at compilation) *that happens with incorrect grammar of a programming language.*

What is the C++ standard library?

*A library containing basic functions. (*Sol. A library file is a collection of precompiled code that has been “packaged up” for reuse in other programs. The C++ standard library is a library that ships with C++. It contains additional functionality to use in your programs.)

**Comment:** Programmer-readable note that is in the source code of the program. Used to help programmers document the code.

* *Single line comment:* Typed using ‘//’
  + Used to quick comments about **single** lines of code.
  + Example:
    - Std::cout <<”Hello World!\n”; // std:cout lives in the iostream library.
* Note: If lines are long, placing comments to the right can make your lines long, in that case single line comments are often placed ***above*** the line it is commenting.
* *Multi Line comments:* Typed using “/\* -- \*/”
  + **Beautify** example:
    - **/\*** This is a multi-line comment.  
       \* The matching asterisk to the left,  
       \* can make this easier to read.

\*/

**Warning:** Don’t use multi-line comments inside other multi-line comments. Wrapping single-line comments inside a multi-line comment is okay.

**Proper use of comments –**

Comments should be used for three things:

* For a given library, program, or function, comments should be used to describe ***what*** the library, program, or function, does. *Placed* at the top of the file orimmediately preceding the function.

*Example:*

A screenshot of a computer

Description automatically generated

**Second,** within a library, program, or function, comments can be used to describe ***how*** the code is going to accomplish its goal.

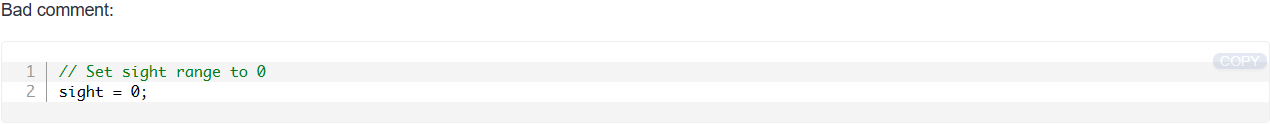
*Example:*

A screenshot of a computer

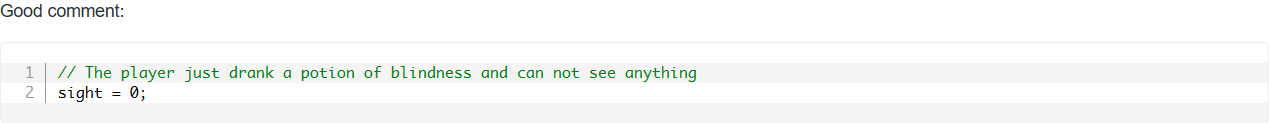
Description automatically generated

**Third,** comments can be used to describe ***why*** the code is doing something.

*Bad comments* explain what the code is doing. If you ever write code that is so complex that needs a comment to explain what a statement is doing, you probably need to *rewrite your statement,* not comment it.



*Reason:* We can see that the sight is being set to 0 by looking at the statement.



*Reason:* Now we know why the player’s sight is being set to 0.

*Comments are a good way* to remind yourself (or somebody else) why you chose to solve a problem one way instead of another.

A screenshot of a computer

Description automatically generated

**Commenting out code**: Converting one or more lines of code into a comment to temporary exclude parts of code from being included in your compiled program.

*Reasons to:*

* **Working on a new piece of code** that won’t compile *yet*, and you need to run the program. *Commenting out the code* *that won’t compile* will allow the program to compile so you can run it. When ready, you can uncomment the code, and continue working on it.
* **You’ve written some code** that *doesn’t work correctly*, and you don’t have time to fix it until later.
* **To find the source of an error**, it can sometimes to be useful to disable parts of your code to isolate the code that is *causing to program to not work* properly.
* **You want to replace one piece of code with another piece of code.** *Instead* of deleting you can comment your code until you’re sure your new code works properly.

For **Visual Studio Code**:

* *Comment a selection via* **Edit menu > Advanced > Comment Selection**

**Data:** Programs produce results by manipulating *data. Data* is any information that can be moved, processed, or stored by a computer.

**Value:** A single piece of data is called a *value,* e.g. a, 5 and text; Hello.

**RAM:** *Random Access Memory.* When a program is run it is loaded into the computer’s RAM. Common uses for this memory are to store values entered by the user, to store data read in from a file or network, or to store values calculated while the program is running (the sum of two values).

**Object:** An *object* is a region of storage (memory) that can store a value as opposed to direct memory access. A compiler would retrieve data stored in an object when a program is executed. An *object* with a name is called a ***variable.***

* **Note:** *Object refers to an unnamed object in memory, a variable, or a function. In C++ object has a narrower definition that* ***excludes functions.***

**Variable *instantiation*:** To create a variable, a declaration statement called a *definition* is used. Instantiation (Also called an *instance)* is creating an object and assigning it a memory address. Whenever the program uses an *instantiated* variable, it accesses the value of the variable in a specific memory location.

*Example:*



**Data Types**

* Integer (int): Any number written without a fraction component, such as 4, 27, 0, -2 are *integer variables.*

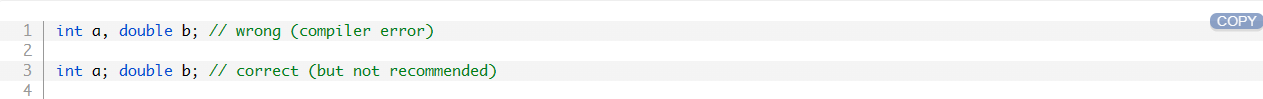
**Defining Multiple Variables:** It is possible to define multiple variables of the *same type* in a single statement separated by a comma.

*Example:*



*Incorrect syntax:*





**Quiz.**

* *What is data?*
* Data is any information that can be moved, processed, or stored by a computer.
* *What is a value?*
* A value is any single piece of data for example a number (5), letter (a) or text (hello).
* *What is a variable?*
* A variable is a named object. (Sol. Named region (*object)* of memory used to store values.
* *What is an identifier?*
* An identifier is the name of the object. X (Sol. An identifier is the name that a variable is assessed by.) In other words, the name of the variable.
* *What is a type?*
* A data type identifies what kind of data a variable is. (Sol. *A type tells the program how to interpret a value in memory.)*
* *What is an integer?*
* An integer is any non-fractional numeric value.

**Variable Assignment and Initialization –**

* The process of giving a variable a value is called assignment.
  + Done using, “**=**”, known as assignmentoperator.
  + Copy assignmentcopies the value on the *right-hand* side of the = operator to the variable on the *left-hand* side of the operator.
* A white background with green text

  Description automatically generatedOne downside of assignment is that it requires at least **two** statements.
* The steps can be combined by using an initialiser, this process is called initialisation.

A close-up of a white background

Description automatically generated

* + Different forms of Initialisation:

A screenshot of a computer code

Description automatically generated

*List initialisation –*

* Also known as “Uniform initialisation”, was introduced to provide a more consistent initialisation syntax and works in most cases as opposed *direct* and *copy* initialisation.

List initialisation has an added benefit; if you try to brace initialise a value that the variable cannot hold, the compiler is *required* to produce a diagnostic (usually an error).

For example:

Int width {4.5}; // error: a number with a fractional value can’t fit into an int.

* + *Copy* and *direct* initialisation would just drop the fractional part, resulting in the initialisation of value 4 into variable *width. List* initialisation would require the compiler to generate a diagnostic.
* **Best practice:** Prefer *direct list initialisation* (or value initialisation) for initialising your variables. *Bjarne Stroustrup* (creator of C++) recommends using list initialisations to initialise your variables.

*Value initialization and zero initialization –*

* When a variable is initialized using *empty braces*, value initialization takes place. Value initialisation will initialise the variable to zero (or empty if that’s more appropriate for a given type). In such cases where zeroing occurs, this is called *zero initialisation*.
* Int width {}; // value initialisation / zero initialisation to value 0

When to use zero initialisation vs value initialisation:

* + Use explicit initialisation (zero initialisation) if you’re using that value.
    - Int x {0}; // explicit initialisation to value 0
    - Std::cout << x; // we’re using that zero value.
  + Use value initialisation if the value is temporary and will be replaced.
    - Int x {}; // value initialisation
    - Std::cin >> x; /// we’re immediately replacing that value
* Best practice is to initialise your variables upon creation. *Always initialise rule.*

*Initialising multiple variables –*

* It’s possible to define multiple variables of the same type in a single statement; int a, b; It’s best practice to avoid this syntax altogether but variables can still be initialised written like this.
  + Int a = 5, b = 6; // copy initialisation
  + Int c(7), d(8); // direct initialisation
  + Int e {9}, f {10}; // direct brace initialisation
  + Int g = { 9 }, h = { 10 } ; // copy brace initialisation
  + Int i {}, j {}; // value initialisation

*Unused initialised variables warnings –*

* Compilers will generate warnings if a variable is initialised but not used, causing the program to potentially fail to compile.
* For example,

Int main () {

Int x { 5 }; // variable defined

// but not used anywhere

Return 0;

}

* When compiling, an error is generated.

A black text on a white background

Description automatically generated

* There are a few ways to fix this.
  + If the variable is unused, the easiest option is to remove *x* (or comment it out).
  + Another option is to simply use the variable somewhere, but that requires some effort to write code that uses it and has the downside of potentially changing the programs behaviour.

***The* [[maybe\_unused]] *attribute –***

In some cases neither of the above options are desirable, consider a case where we have a bunch of math values that we use in many different programs. If we use these a lot, we probably have these saved somewhere and copy/paste/import them all together. However, in any program where we don’t use all of these values, the compiler will complain about each unused variable. To address such cases C++ introduced the [[maybe\_unused]] attribute, which tells the compiler we’re okay with a variable being unused. The compiler will not generate any warnings for such variables.

A screenshot of a computer code

Description automatically generated

**Quiz –**

*What is the difference between initialisation and assignment?*

* Initialisation is the initial activation of a variable; assignment is the value given (or assigned) to the variable. X. (Initialisation gives an *initial value* to a variable **when** it is created. Assignment gives a variable some value at some point **after** the variable is created).

*What form of initialisation should you prefer when you want to initialise a variable with a specific value?*

* Direct List Initialisation (aka. Direct brace initialisation).

*What are default initialisation and value initialisation? What is the behaviour of each? What should you prefer?*

* Default initialisation is the initialisation of a variable with a value that will be used, value initialisation is used when the value of the variable might get changed later. It’s recommended to use value initialisation because the value of the variable might get changed later and the initial value is 0. X. (Default initialisation is when a variable has no initialiser (e.g. int x) The variable is left with an indeterminate value. Value initialisation is when a variable initialisation has an empty brace (e.g. int x {}). In most cases this will perform zero initialisation. It is **preferred** to use value initialisation).

**Introduction to iostream: cout, cin, and endl.**

**The input/output library –**

* The input/output library (io library) is part of the C++ standard library that deals with input and output. To use the functionality defined within the *iostream* library, we need to include the iostream header at the top of any code file that uses the content defined in iostream, like so:

#include <iostream>

//rest of the code that uses iostream functionality here.

*Std::cout* –

* The iostream library contains a few predefined variables for us to use. One of the most useful is *std::cout,* which allows us to send data to the console to be printed as text. Cout stands for “*character output”.*

Hello World –

A screenshot of a computer code

Description automatically generated

* In this program, iostream is included to have access to *std::cout.* Inside the main function, std::cout is used, along with the *insertion operator* (<<), to send the text Hello World! To the console to be printed.
* std::cout can also print numbers, it can also print the value of variables:

#include <iostream> // for std::cout

Int main () {

Int x { 5 }; // define integer variable x, initialised with value

Std::cout << x; // print value of x (5) to console

Return 0;

}

To print more than *one* thing on the same line, the insertion operator (<<) can be used multiple times in a *single* statement to link multiple pieces of output. For example:

#include <iostream> // for std::cout

Int main () {

Std::cout<< “Hello” << “ world!”;

Return 0;

}

* This program prints:

A white background with black text

Description automatically generated

*Another example:*

#include <iostream>

Int main () {

Int x { 5 };

Std::cout << “x is equal to: “ <, x;

Return 0;

}

* Produces the result:

A black and white text

Description automatically generated

std::cout is ***buffered.***

* Output requested/produced by the program is not sent to the console immediately.
* Temporary stored in a memory block; **buffer.**
* **Flushed (**flushing the buffer)**;** all the data stored in the memory block is transferred to its *destination* (e.g. console.)
* If the program is paused, aborted before the buffer is flushed. Output still stored in the buffer is not displayed.
* *Opposite* of buffered output is unbuffered output. Unbuffered output directly sends the output to the output device (i.e. console). Writing to a buffer is comparatively faster compared to transferring data to the output device.

*Std::endl –*

The following program:

#include <iostream>

Int main () {

Std::cout << “Hi!”;

Std::cout << “My name is Alex. “;

Return 0;

}

* Prints:  
  A white background with black text

  Description automatically generated
* Separate output statements don’t result in separate lines of output on the console. If you want to print separate lines of output to the console, you need to tell the console when to move the cursor to the next line.

One way to do that is to use *std::endl.* Std::endl prints a newline to the console, *endl* stands for “end line”.

* *For example:*

#include <iostream> // for std::cout and std::endl

Int main () {

Std::cout << “Hi!” << std::endl; // std::endl will cause the cursor to move to the next line of the console

Std::cout << “My name is Alex.” << std::endl;

Return 0;

}

* Printing:

A white background with black text

Description automatically generated

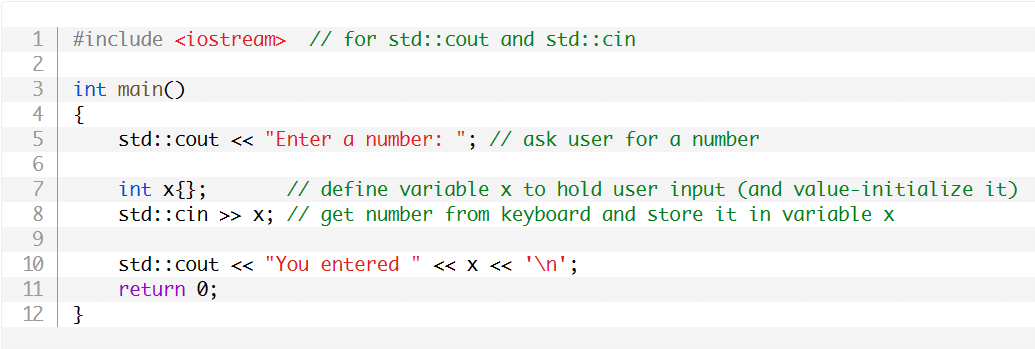
* Best practice: Output a newline whenever a line of output is complete.

**std:endl vs ‘\n’**

* Endl performs two functions; it moves the cursor to the next line and flushes the buffer which slows down performance.
* That’s why it’s recommended to use \n to move to newline as it doesn’t flush the buffer and results in improved performance.
  + Can be **embedded** into text.
* **Best Practice:** Use \n to when outputting text to console over std::endl.

**Std::cin**– *Character Input*

* Used to store input from user using ‘>>’ (**extraction** operator).
* Input **must be** **stored** in a *variable.*



* Not necessary to use ‘\n’ when taking user input because it moves to next line automatically when user presses enter.

Multiple variables –

A screenshot of a computer code

Description automatically generated

Produces the output:

A close up of a text

Description automatically generated

* **Best Practice:** Initialise the variables first when taking user input.

**Quiz.**

Run the program:

A computer code with text

Description automatically generated

1. Enter a letter like a.

The program returns 0.

1. Enter a fractional number like 3.5.

The program gets rid of the .5 and just prints 3.

1. A word, such as *hello.*

The program prints 0.

1. A small negative integer like -3.

The program prints the number.

1. A big number like 3 billion.

The program prints 2147483647 because, x can only hold up to a certain limit.

1. A small number followed by some letters, such as 123abc.

The program prints the numbers, gets rid of the letters.