**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.

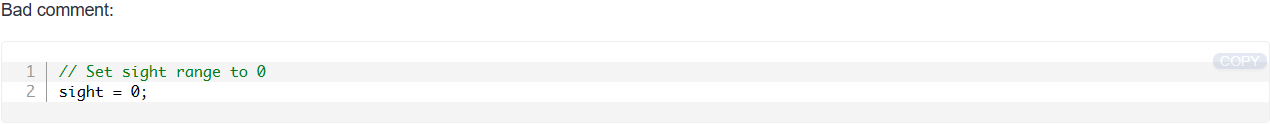
*Example:*

A screenshot of a computer

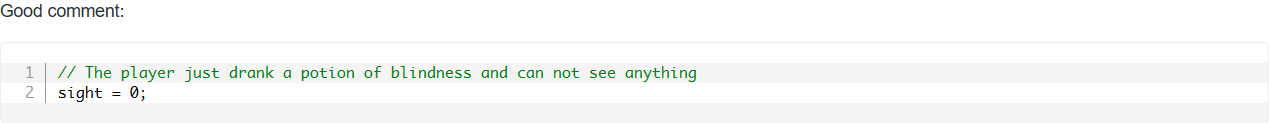
Description automatically generated

**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.*

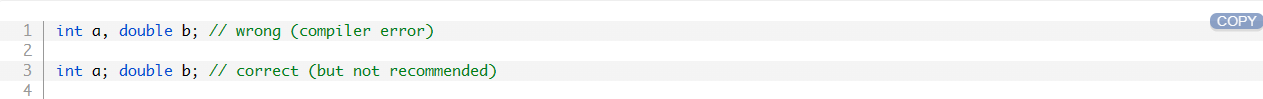
**Defining Multiple Variables:** It is possible to define multiple variables of the *same type* in a single statement separated by a comma.

*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

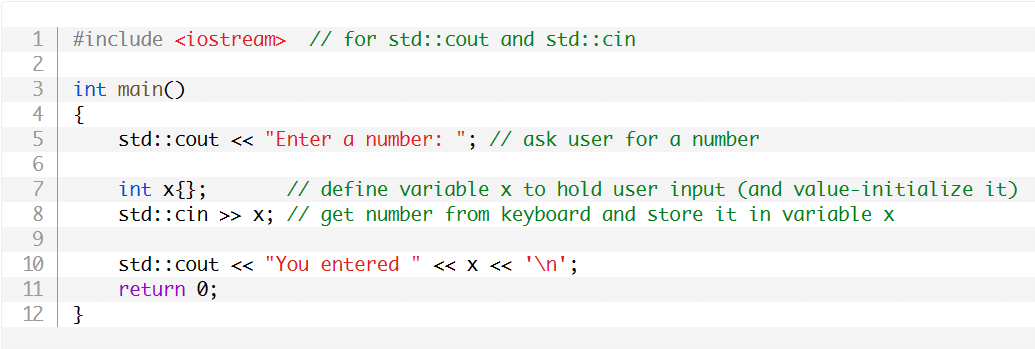
* Best practice: Output a newline whenever a line of output is complete.

**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.

**Uninitialized variable –**

When a variable that is not initialised is given a memory address to use to store data, the value of that variable is whatever value happens to already be in that memory address. A variable that has not been given a known value (through initialisation or assignment) is called an **uninitialized variable.**

A screenshot of a computer

Description automatically generated

*Recap:*

* Initialised = the object is given a known value at the point of definition.
* Assignment = The object is given a known value beyond the point of definition.
* Uninitialized = The object has not been given a known value yet.
  + Int x: Uninitialized.

**Undefined Behaviour –**

Undefined behaviour is the result of executing code whose behaviour is not well-defined by the C++ language. Undefined behaviour may exhibit any of the following symptoms:

* Your program produces different results every time it’s run.
* Your program consistently produces the same incorrect result.
* Your program behaves inconsistently (sometimes the correct result, sometimes not.)
* Your program seems like it’s working but produces incorrect results later in the program.
* Your program crashes, either immediately or later.
* Your program works on some compilers but not others.
* Your program works until you change some other seemingly unrelated code.

Or the program may produce the correct behaviour anyway.

**Implementation-defined behaviour** and **unspecified behaviour –**

**Implementation-defined** behaviour means the behaviour of some syntax is left up to the compiler to define. Such behaviours must be consistent and documented, but different compilers may produce different results.

**Unspecified behaviour** is almost identical to implementation-defined behaviour in that the behaviour is left up to the implementation, but the implementation is not required to document the behaviour.

* We want to *avoid* implementation-defined and unspecified behaviour, as it means our program may not work as expected if compiled on a different compiler (or even on the same compiler if the project settings are changed).

**Best practice:** Avoid implementation-defined and unspecified behaviour as they may cause your program to malfunction on other implementations.

**Quiz –**

What is an uninitialized variable? Why should you avoid using them?

* An uninitialized variable is a variable that is not given a value either through initialisation or assignment upon creation. It’s best practice to avoid using them because the variable will use whatever value is in the memory slot assigned to the variable upon creation resulting in undefined behaviour.

What is undefined behaviour, and what can happen if you do something that exhibits undefined behaviour?

* Undefined behaviour is the result of executing code whose behaviour is not well-defined by the C++ language. Undefined behaviour will result in unexpected results when the program is run.

**Keywords** and *naming identifiers –*

C++ reserves a set of 92 words (as of C++23) for its own use. These words are called keywords (or *reserved words*) and each keyword has a special meaning in the C++ language.

List of keywords –

A screenshot of a computer

Description automatically generated

**Identifier naming rules –**

**Identifier:** Name of a variable (or function, type, or other kind of item).

C++ gives a lot of flexibility to name identifiers but there are a few rules that must be followed when naming identifiers:

* The identifier cannot be a *keyword*.
* The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. This means the name cannot contain any symbols (except the underscore) nor whitespace (spaces or tabs).
* The identifier must *begin* with a letter or an underscore. It cannot start with a *number*.
* C++ is case sensitive. Nvalue is different from nValue and that is different than NVALUE.

**Identifier naming** best practices –

A screenshot of a computer

Description automatically generated

If a variable or function is multi-word, there are two common conventions: words separated by underscores or intercapped (for e.g. camelCase).

A screenshot of a computer program

Description automatically generated

**Best practice:** When working in an existing program, use the conventions of that program. Use modern best practices when you’re writing new programs.

* Avoid naming identifiers starting with an underscore as these are typically reserved for *OS, Library,* and/or *compiler use.*

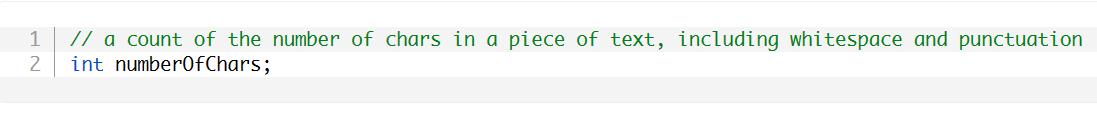
An identifier with a trivial use can have a short name (e.g. such as i.) An identifier that is used more broadly (e.g. a function that is called from many different places in a program) should have a longer and more descriptive name (e.g. instead of open, try openFileOnDisk).

A screenshot of a computer

Description automatically generated

Avoid abbreviations (unless they are common/unambiguous). They may save the time to write the code but make it harder for the reader to read.

For variable declarations, it is useful to use a comment to describe what a variable is going to be used for, or to explain anything else that might not be obvious. For example, say we’ve declared a variable named numberofChars that is supposed to store the number of characters in a piece of text. Does the text “hello world!” have 10, 11, or 12 characters? It depends whether we’re including whitespace or punctuation.   
Rather than naming the variable numberOfCharsIncludingWhitespaceAndPunctuation, which is rather lengthy, a well-placed comment on or above the declaration line should help the user figure it out:



**Quiz –**

Based on how you *should* name a variable, indicate whether each variable name is conventional (follows best practices), unconventional (compiler will accept but does not follow best practices) or invalid (will not compile), and why.

Int sum {}; (Assume it’s obvious what we’re summing)

* Conventional

Int \_apples {}.

* Unconventional, starts with an underscore.

Int VALUE {}.

* Unconventional, it should be all lower case. (Single word variables only lowercase).

Int my variable name {}.

* Invalid, won’t function because of whitespaces.

Int TotalCustomers {}.

* Conventional, if it’s obvious what it’s referring to. X (Unconventional, variable names should start with a lower-case letter).

Int void {}.

* Invalid, it’s a reserved keyword.

Int numFruit {}.

* Conventional.

Int 3some {}.

* Invalid, it cannot start with a number.

Int meters\_of\_pipe {}.

* Conventional.

**Whitespace and basic formatting –**

**Whitespace** is a term that refers to characters that are used for formatting purposes. In C++, this refers to spaces, tabs, and newlines. Whitespace in C++ is used for 3 things: separating certain language elements, inside text, and for formatting code.

Some language elements must be whitespace-separated –

The language requires that some elements are separated by whitespace. This mostly occurs when two keywords or identifiers must be placed consecutively, so the compiler can tell them apart.

For example, a variable declaration must be whitespace separated:

*Int x*; // int and x must be whitespace separated.

* If we typed intx instead, the compiler would interpret this as an identifier, and then complain it doesn’t know what identifier intx is.

Another example: a function’s return type and name must be whitespace separated:

*Int main()*; // int and main must be whitespace separated.

When whitespace is required as a separator, the compiler doesn’t care how much whitespace is used, if some exists.

The following variable definitions are all valid:

A white background with black text

Description automatically generated

An example:

Std::cout << “Hello World!”; // this is part of the comment and

*This is not part of the comment* – Because newline terminates single-line comments.

In certain cases, newlines are used as a separator –

A screenshot of a computer code

Description automatically generated

**Quoted text –**

Inside quoted text, the amount of whitespace is taken literally.

Std::cout << “Hello World!”;

Is different than:

Std::cout << “Hello World!”;

Newlines are **not allowed** in quoted text:

Std::cout << “Hello

World!”; // **not allowed!**

Quoted text separated by nothing, but whitespace (spaces, tabs, or newlines) will be concatenated:

A white background with red and black text

Description automatically generated

* **Concatenated:** *Link together.*

**Use whitespace to format code –**

Whitespace is generally ignored. This means we can use whitespace wherever we like to format our code to make it easier to read.

For example, the following is hard to read:

A white background with black text

Description automatically generated

The following is better (but still dense):

A screen shot of a computer code

Description automatically generated

And the following is even better:

A screenshot of a computer code

Description automatically generated

Statements may be split over multiple lines if desired:

A screenshot of a computer code

Description automatically generated

* This can be useful for long statements.

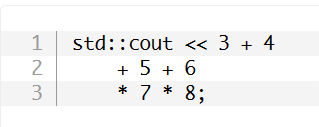
**Basic formatting –**

Unlike other languages, C++ does not enforce any kind of formatting restrictions on the programmer. For this reason, we say that C++ is a whitespace-independent language.

This is a mixed blessing. On one hand, it’s nice to have the freedom to do whatever you like. On the other hand, many different methods of formatting C++ programs have been developed throughout the years, and you will find (sometimes significant and distracting) disagreement on which ones are best. Our basic rule of thumb is that the best styles are the ones that produce the most readable code and provide the most consistency.

**Best practice:** Consider keeping lines to 80 chars or less in length.

If a long line is split with an operator (e.g. << or +), the operator should be placed at the beginning of the next line, not the end of the current line.



This makes it clearer that the subsequent lines are continuations of the previous lines and allows you to align then operators on the left, which makes for easier reading.

Use whitespace to make your code easier to read by aligning values or comments or adding spacing between blocks of code.

*Harder to read:*

Cost = 57;

pricePeritem = 24;

value = 5;

numberOfItems = 17;

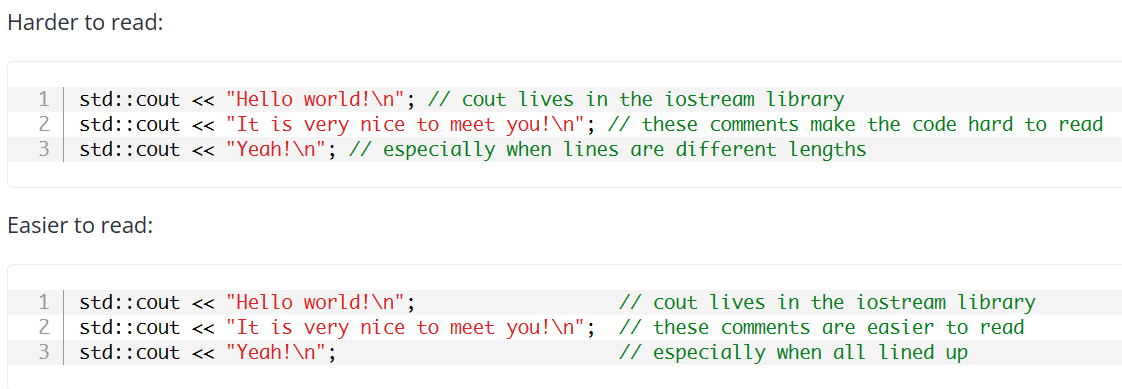
*Easier to read:*

Cost = 57;

pricePerItem = 24;

Value = 5;

numberOfItems = 17;



**Best Practice:** When working in an existing project, be consistent with whatever style has already been adopted.

**Best Practice:** Using the automatic formatting feature for your IDE is highly recommended to keep your code’s formatting style consistent.

**Style guides –**

A style guide is a concise, opinionated document containing (sometimes arbitrary) programming conventions, formatting guidelines, and best practices. The goal of a style guide is to ensure that all developers on a project are programming in a concise manner.

Examples:

* C++ Core Guidelines, maintained by Bjarne Stroustrup and Herb Sutter.
* Google.
* LLVM.
* GCC/GNU.

C++ Code guidelines is favoured.

**Literals –**

A **Literal** (also known as a literal constant) is a fixed value that has been inserted directly into the source code. Literals and variables both have a value (and a type). Unlike a variable (whose value can be set and changed through initialisation and assignment) the value of a literal is fixed (e.g. 5 is *always*). This is why literals are called **constants.**

An example:

A screenshot of a computer code

Description automatically generated

In the case of the literal, the value 5 is printed directly. It is compiled into the executable.   
In the case of the variable, the value 5 must be fetched from the memory the variable represents. A variable’s value is placed in memory, and the value of memory can be changed while the executable is running.

**Operators –**

An *operation* is a process involving zero or more input values (called *operands*) that produces an output value. The specific operation to be performed is denoted by a symbol called an **operator**.

Operator example:

A screenshot of a computer code

Description automatically generated

In this program, the literals 1 and 2 are operands to the plus (+) **operator.** The output value is then printed to the console, the output value of an operation is often called a **return value**.

The number of operands that an operator takes as input is called the operator’s **arity.**

There are 4 different arities:

1. **Urary** operators act on one operand. For example, the ‘- ‘operator, given -5, operator- takes literal operand 5 and flips its sign to produce new output value -5.
2. **Binary** operators act on two operands. For example, the ‘+’ operator, given 3 + 4, operator+ takes the left operand 3 and the right operand 4 and applies mathematical addition to produce new output value 7. The insertion (<<) and extraction (>>) operators are binary operators, taking std::cout or std::cin on the left side, and the value to output or variable to input to on the right side.
3. **Ternary** operators act on three operands.
4. **Nullary** operators act on zero operands.

**Chaining operators –**

Operators can be chained together so that the output one operator can be used as the input for another operator. For example: 2 \* 3 + 4. The multiplication happens first and produces the output value 6 which becomes the left operand for the plus operator and is added onto the 4 which is the right operand (or second operand for the + **operation**).

**Note:** PEMDAS is used (also known as PEDMAS, BEDMAS, BODMAS, or BIDMAS).

**Return values and side effects –**

Most operators in C++ use their operands to calculate a return value. For example, -5 produces return value -5, and 2+3 produces return value 5. There are some operators that do not produce return values (such as delete and throw, discussed later).

**Side effects** are any observable effect beyond producing a return value by an operator. For example, when x = 5 is evaluated, the assignment operator has the side effect of assigning the value 5 to the variable x. The changed value (**the side effect)** is observable by printing the value of x *after* the operator has finished executing. Std::cout << 5 has the *side effect* of printing 5 to the console. We can observe the fact that 5 has been printed to the console even after std::cout << 5 has finished executing.

* Operators with *side effects* are usually called for the behaviour of the side effect *rather* than for the return value those operators produce.

A **side effect** is an observable effect of an operator or function beyond producing a return value.

**Quiz.**

For each of the following, indicate what output they produce:

1. Std::cout << 3 + 4 <, ‘n’;

* Output = 7.

1. Std::cout << 3 + 4 – 5 << ‘\n’;

* 2.

1. Std::cout << 2 + 3 \* 4 << ‘\n’;

* 14.

1. Int x { 2 };
2. Std::cout << (x = 5) << ‘\n’;

* 2 = 5. X (5. ‘X = 5’ assigns the value 5 to x, and then returns x. The value of x (now 5) is then printed to the console.)

**Expressions –**

An ***expression*** is a sequence of literals, variables, operators, and function calls that calculates a single value. The process of executing an expression is called ***evaluation***, and the single value produced is called the ***result*** of the expression.

A computer screen with text on it

Description automatically generated

Each of these statements defines a new variable and initializes it with a value. C++ converts all these different literals, variables, operators, etc. into a single value that can then be used as the initialisation for the ***variable.***

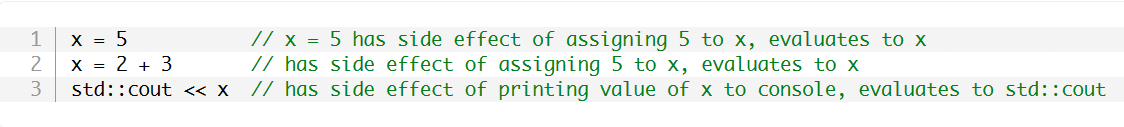
When an expression is evaluated, each of the terms inside the expression are evaluated, until a single value remains. Here are some examples of different kinds of expressions, with comments indicating how they evaluate:

A close-up of a screen

Description automatically generated

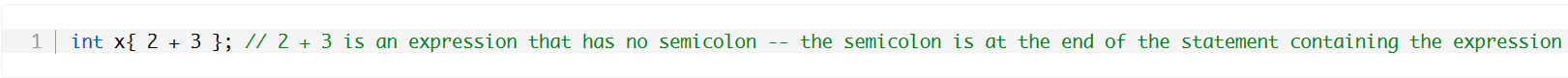
***Literals*** evaluate to their own values (as observed), variables evaluate to the value of the variable. Operators (such as operator+) use their operands to evaluate to some other value.

***Expressions*** involving *operators* *with* *side effects* are trickier:



*Expressions* do not end in a semicolon and cannot be compiled by themselves. For example, your compiler would complain (missing semicolon) about the expression x = 5. *Rather* expressions are always evaluated as part of statements.

For example:



If we were to break this statement down to its syntax, it would look like this:

* **Type identifier { expression };** *type* could be any valid type (like ‘int’). *identifier* could be any valid name (like ‘x’). And *expression* could be any valid expression (like ‘2 + 3’), which uses two literals and an operator.

**Expression statements –**

Certain expressions like x = 5 are useful for their side effects (in this case, to assign the value 5 to the variable x). However, expressions cannot be executed by themselves, they must exist part of a statement. So how can we use such expressions?

You must *convert* the expression into an **equivalent**, **expression statement.** An *expression statement* is a statement that consists of an expression followed by a semicolon. When the expression statement is executed, the expression will be evaluated. Thus, you can take any expression (such as x = 5) and turn it into an *expression statement* (x = 5;) and it will *compile*.

Note: *When an expression is used in an expression statement, any return value generated by the expression is discarded (*because it’s not used).

Useless *Expression Statements –*

Expression statements that compile but have no effect can also be made for example, the expression statement ( 2 \* 3; ) is an expression statement whose expression evaluates to the result value 6 but is discarded because it’s a *useless* expression statement.

**Subexpressions, full expressions, and compound expressions –**

There are specific kinds of expressions, consider the following expressions:

2 // 2 is a literal that evaluates to value 2.

2 + 3 // 2 + 3 uses operator + to evaluate to value 5.

x = 4 + 5 // 4 + 5 evaluates to value 9, which is then assigned to variable x.

Simplifying, a **subexpression** is an expression used as an operand. For example, the subexpressions of x = 4 + 5 are x and 4 + 5. The subexpressions of 4 + 5 are 4 and 5.

A **full expression** is an expression that is not a subexpression. All three expressions above (2, 2 + 3, and x = 4 + 5) are all full expressions.

A **compound expression** is an expression that contains two or more uses of operators. x = 4 + 5 is a compound expression because it contains two uses of operators (operator= and operator+). 2 and 2 + 3 are *not* compound expressions.

**Quiz.**

What is the difference between a statement and an expression?

* An expression is a sequence of literals, variables, operators, and functions that calculate a single value. A statement is used to make expressions work with the compiler to *perform an action*. (~ *Statements are used when we want the program to* ***perform an action****. Expressions are used when we want the program to* ***calculate a value.)***

Indicate whether each of the following lines are statements that do not contain expressions, statements that contain expressions, or are expression statements.

1. Int x;

* Statement that contains an expression. (X. Statement does not contain an expression (this is a variable definition).

1. Int x = 5;

* Statement that contains an expression (x = 5, x is a variable, the ‘=’ is used for copy assignment and the value/initializer to the right of the operator= is the expression (operand)).

1. X = 5;

* This is an expression statement. (x = 5 is a call to operator= with two operands: x and 5. The semicolon makes it an *expression statement*.)

**Extra credit –**

1. Foo(); // foo is a function.

* This is a function, it can contain expressions, statements that contain expressions and expression statements. (Function **calls** are ***part*** of an expression, so this is an expression statement)

1. Std::cout << x; // hint: operator << is a binary operator.

* This is a statement that does not contain expressions. (operator << is a binary operator so std::cout must be the *left hand operand*, and x must be the *right hand operand. Since that’s the entire statement, this is an* ***expression statement****).*

3. Determine what values the following program outputs. Work through it in your head.

A screenshot of a computer program

Description automatically generated

* 5, first value.
* y = 4.
* 36, third value.