**C++ is an extension of C, supporting both procedural programming (like C) and OOP.**

1. **Inputs and Outputs**

// **cout** is object that used to display data on the screen and insertion operator (<<) put data into cout:

Syntax: cout << object;

* **Note:** The endl keyword is equivalent to the escape character “\n”.

// **cin** is object that used to read data from buffer input:

Syntax: cin >> object;

* **Note**: cin will only read up to the first space, tab (\t), or newline (\n).

// **getline**  read a line from input buffer (include whitespace, newline, tab) until encounter newline:

Syntax: getline(cin, object);

* **Note**: ignore is used to ignore certain number of characters from the input buffer:

Syntax:cin.ignore(<amount>, <delimiter>);

1. **Functions**

// C++ functions allow the use of default parameters:

Syntax: return function(parameters = values) {…}

// Inline functions in C++ help optimize performance by reducing the cost of calling functions by replacing function calls with the function definition itself at the call location after compilation:

Syntax: inline return function(parameters) {…}

1. **References**

// When you assign a value to a regular variable, the value is copied into the new variable instead acutal value.

// Reference is an alias of a variable, meaning it points to the actual value instead of a copy.

Syntax: <type>& <alias> = <value>;

* **Note**: References have the same address the original variable, can’t be re-initialized.

1. **Dynamic Allocated Memories**

// **new** keyword is used to allocate a dynamic memory or automatically call constructor:

Syntax: new <type>;

// **delete** keyword is used to free a dynamic memory or automatically call destructor”

Syntax: delete <allocated memory>;

* **Note**: C++11 and later use **nullptr** instead of **NULL** to avoid ambiguous errors.

1. **Strings**

// C++ provides a class that allows using the string data type: #include<string>;

// Strings C++ can apply operator:

* Concatenating with + operator
* Comparing with logical operator

// C++ provides built-in methods to manipulate strings:

|  |  |
| --- | --- |
| length() | returns the number of characters of a string |
| **toupper() / tolower()** | converts all characters to uppercase/lowercase |
| replace(start, end, substring) | replaces all occurrences of a specified substring with another |
| find(substring) | finds the first occurrence of the specified value in range |
| substr(start, end) | returns a substring |
| insert(index, substring) | inserts a subtring into specific position in a string |

1. **Files**

// The **<fstream>** library allows us to work with files:

+ Create a object to open and read a file:

Sytnax: ifstream object(<file name>);

+ Create a object to open and write a file:

Syntax: ofstream object(<file name>);

+ Read a line:

Syntax: getline(file, object);

+ Write data to a file:

Syntax: <file object> << <data>;

+ Close a file:

Syntax: <file object>.close();

1. **Classes and Objects**

// Class is a user-defined type, which holds its own data members and member functions that can be accessed and used by creating an object of that class.

Syntax:class <class name> {

<access specifiers>:

// Define attributes and methods };

// When we use a class to create a specific entity, that entity is called an object or instance:

Syntax: <class name> object(arguments);

* Access members:
* Access attribute of object:

Syntax: object.attribute;

* Access method of object:

Syntax: object.method();

// Static members are members of a class, not a particular object:

* Static attribute has one copy, shared among all objects of a class (declare inside but defined outside class).
* Static methods can only access other static members and no this (it’s not associated with a specific object).
* **Note**: **this** is a special pointer in every method of the class that points to the current object.

// Constructor is a method that is automatically called when an object is initialized:

* Default Constructor is a constructor that doesn’t take any parameters.

Syntax: <class name> () {…}

* Parameterized constructor is a constructor that takes arguments:

Syntax: <class name>(parameters) {…}

* Copy Constructor is a constructor that creates a new object from another object of the same class:

Syntax: <class name>(object) {…}

* Shallow copying (default) is a copy of an object by copying the value of all members (if the member is a pointer, only the pointer address is copied) **➔ Two objects will share the same memory area**
* Deep copying is a copy of an object by copying the data of all member variables (if the member is a pointer, then allocate new memory)
* **Note**: If doesn’t define constructor, the compiler automatically creates the default, copy constructor.

// Destructor is a method that is called automatically when an object is going to be destroyed, to free up resources.

Syntax: ~<class name> () {…}

* **Note**: Destructor is called in reverse order of object creation.
* **Note**: If a class has any virtual functions, always declare the destructor virtual to avoid potential errors when using polymorphism.

1. **Access specifiers**

// Access specifiers are keywords that control how the members of a class can be accessed:

* public keyword which means that they can be accessed outside or inside that class.
* private keyword which means that they can be only accessed inside that class or friend class.
* protected keyword which means that they can be only accessed inside that class or child class.
* **Note**: If we do not specify the access specifier, the private specifier is default.

// The **friend** keyword allows a function or class to access the private/protected members of another class:

* Friend class declared inside the class but defined outside the class:

Syntax: friend <class declaration>;

* Friend function declared inside the class but defined outside the class:

Syntax: friend <function declaration>;

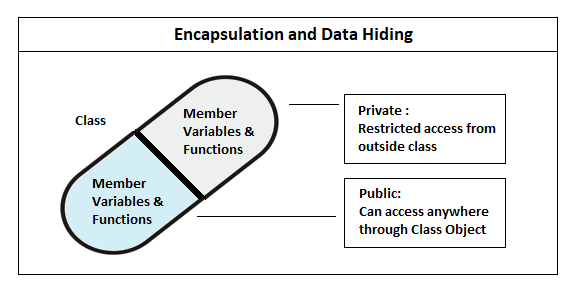
* **Note**: Using friends breaks encapsulation, so should only be used when absolutely necessary.

1. **Encapsulations**

// Encapsulation is the bundling of attribute (variables) and methods (function) that operate on that data into a single unit (a class), while data hiding by restricting access to some of the object's components from outside class through access specifiers (private, protected) and only provide access through methods (getter, setter):

Getter

get data of member private



Setter

set data for member private

Setter

set the value for a private member

Getter

get the value of a private member

1. **Abstractions**

// Abstraction is hiding complex implementation details while showing only the necessary features:

* Abstract class is a template class for subclasses, defining at least one method without implementations (pure virtual function) that subclasses will implement (override).
* Pure virtual function is a method without implementation:

Syntax: virtual <type> <function>() = 0;

* Interface is a class that has only pure virtual functions.

1. **Inheritances**

// Inheritance allows a class (Child - Dervied) that inherits members from another class (Parent - Base).

Syntax:class <Dervied Class>: <inheritance mode> <Base Class>{…}

* Modes of inheritance:

|  |  |
| --- | --- |
| **public** | preserve |
| **protected** | public, protected → protected |
| **private** | public, protected → private |

* Types of inheritance:

|  |  |
| --- | --- |
| **Hierarchical** | Son, Daughter ⊂ Parent |
| **Multilevel** | Child ⊂ Parent ⊂ GrandParent |
| **Multiple** | Child ⊂ Dad, Mom |
| **Hybird** | Hierarchical + Multilevel + Multiple |

* **Problem**: The Diamond problem occurs Child ⊂ Dad, Mom ⊂ GrandParent. If the child class use grandparent a member, the compiler doesn't know which function to use.
* **Solution**: Virtual inheritance is a technique in C++ OOP that guarantees receiving a single copy:

Syntax: class <parent class>: <inheritance mode> <grandparent class> {…}

// Constructors in inheritance are not inherited, but when a derived class is instantiated, the base class constructor is always called first to initialize the inherited data:

* Default Constructor: C++ will automatically call in subclass.
* Parameterized Constructor: Manually call with constructor initializer list.

Syntax: <child constructor>(<parameters>) : <parent constructor>(<arguments>) {…};

// Function Overriding in inheritance:

* Function Hiding occurs when a method of child class have same name with a method not virtual of parent class inside child class, it’s will hide the function of the base class:

**➔** Call function based on pointer type, regardless of what the actual object is

* Function Overriding occurs when a method of child class have same name, same paramters with a virtual method of parent class inside child class, it’s will override the function of the base class:

**➔** Call function based on the actual object **➔** Polymorphism

VTable Base Vtable Dervied

vptr

Function override A

Child Object

Virtual function A

Base

Virtual function B

Virtual function B

* A class have at least one virtual functions own a VTable — a table stores address of virtual methods.
* Each object of that class has a vptr pointer that points to the class's VTable.
* **Note**: When override a virtual function, it remains virtual, even if you don't rewrite the virtual keyword.

1. **Polymorphisms**

// Polymorphism allows an object or a function to have different forms depending on the context:

1. **Compile-time Polymorphism**

// Static Polymorphism occurs when determining about which function, operator to call depend on parameters:

* Function Overloading occurs when defining functions in same scope with the same name but ≠ parameters.
* Operator Overloading occurs when redefining of operators for objects.

Syntax: <return type> operator<operator symbol>(&object) {…}

1. **Runtime Polymorphism**

// Dynamic Polymorphism occurs when determining about which function to call depend on actual object, achieved through overriding function.

Polymorphism

Function Overriding

Call Child A

Child A

Polymorphism

Call Child B

Function Overriding

Child B

If you pass the inherited object with passing by value to the base class, the extension will be slicing, losing polymorphism.

Base

1. **Containers**

// Containers in STL are data structures as objects that stores a set of data (items):

#include <data structure>

// Range-based for loop is a loop used exclusively to loop through container:

Syntax: for (<variable>: <container>) {…};

1. **Vectors**

// Vector is a dynamic array that allow random access fast but add a item in specific position slow:

Syntax: vector<elements type> <vector name> = {items};

* Accessing and Changing items:

Syntax: vector.at(index) = value;

* Adding a item (into the end):

Syntax: vector.push\_back(value);

* Deleting a item:

|  |  |
| --- | --- |
| pop\_back() | removes an item at the end |
| clear() | removes all the items |

* Other operations:

|  |  |
| --- | --- |
| size() | returns the number of items |
| empty() | checks if the vector is empty and return 1 (true) |

1. **Linked Lists**

// Lists stores elements randomly in unrelated locations as doubly-linked list.

Syntax: list<elements type> <list name> = {items};

* Accessing items:

Syntax: list.front/back();

* Adding items:

Syntax: list.push\_front/back();

* Deleting items:

|  |  |
| --- | --- |
| pop\_front/back() | removes an item at the beginning/end |
| clear() | removes all the items |

* Other operations:

|  |  |
| --- | --- |
| size() | returns the number of items |
| empty() | checks if the list is empty and return 1 (true) |
| reverse() | reverses the order of the elements. |
| sort() | sorts the list elements in a particular order |
| unique() | removes consecutive duplicate elements |
| merge(list) | merges two sorted lists |

// Forward Lists stores elements randomly in unrelated locations as singly-linked list.

Syntax: forward\_list<elements type> <list name> = {items};

* Accessing items:

|  |  |
| --- | --- |
| front() | adds a element at the begining |
| insert\_after(<iterator position>, values) | adds elements after a specified position |

* Adding items:

Syntax: foward\_list.push\_front();

* Deleting items:

|  |  |
| --- | --- |
| pop\_front(); | removes an element at the begining |
| remove(values) | removes all elements with specific values. |
| clear() | removes all the items |

* Other operations:

|  |  |
| --- | --- |
| assign(values) | assigns new contents to the list by replacing the current contents. |

1. **Queues**

// Queues provide a different interface for other containers.

Syntax: queue<elements type> <queue name> ;

* Accessing items :

Syntax: queue.front/back();

* Adding items:

Syntax: queue.push();

* Deleting items:

Syntax: queue.pop();

// Deques are like queues but allow adding and removing items at both ends (addition):

* Accessing items :

Syntax: deque.front/at/back();

* Adding items:

Syntax: deque.push\_front/back();

* Deleting items:

Syntax: deque.pop\_front/back();

1. **Stacks**

// Stacks provide a different interface for other containers.

Syntax: stack<elements type> <stack name>;

* Accessing items :

Syntax: stack.top();

* Adding items:

Syntax: stack.push();

* Deleting items:

Syntax: stack.pop();

* Other operations:

|  |  |
| --- | --- |
| size() | returns the number of items |
| empty() | checks if the list is empty and return 1 (true) |

* **Note**: For deque, queue cannot use for loop to iterate, must use while.

1. **Maps**

// Maps stores elements as key:value pairs as dictionary in Python but the keys are sorted as binary search tree.

Syntax: map<key type, value type> <map name> = { {key:value}, …};

* Accessing and changing items:

Syntax: map[key] = value;

* **Note**: Each element in the map is actually a pair<const Key, Value> object.
* Access key: pair.first;
* Access value: pair.second;
* Deleting items:

|  |  |
| --- | --- |
| erase(key) | removes specified item |
| clear() | removes all the items |

* Other operations:

|  |  |
| --- | --- |
| size() | returns the number of items |
| empty() | checks if the list is empty and return 1 (true) |
| find(key) | finds value of specified key and return an iterator |

// Unordered Map is actually like a python dictionary, not ordering elements by key as hashtable, but current Python dicts retain the insertion order, but are still hash tables, so lookups are very fast (O(1)).

Syntax: unordered\_map<key type, value type> <map name> = { {key:value}, …};

* Accessing and changing items:

Syntax: map[key] = value;

* Adding items:

Syntax: map[<new key>] = <new value>;

* Deleting items:

|  |  |
| --- | --- |
| erase(key) | removes specified item |
| clear() | removes all the items |

* Other operations:

|  |  |
| --- | --- |
| size() | returns the number of items |
| empty() | checks if the list is empty and return 1 (true) |
| find(key) | finds value of specified key and return an iterator |

* **Note**: Starting from C++17, you can use structure bindings to simplify looping:

Syntax: for ([<key>,<value>]: <map>) {…};

1. **Algorithms**

// Algorithm in STL is set of functions that help manipulate containers:

#include <algorithm>

* Non-mutating algorithms perform operations without modifying containers:

|  |  |
| --- | --- |
| count(<iterator start>, <iterator end>, value) | count the number of occurrences of a value in a range |
| binary\_search(<start>, <end>, value) | checks if a value exists in a sorted range |
| upper/lower\_bound(start, end, value) | finds the first element > (>=) a value in a sorted range |

* Mutating algorithms perform operations with modifying containers:

|  |  |
| --- | --- |
| sort(<start>, <end>, <compare function>) | sorts items in ascending order |
| reverse(<iterator start>, <iterator end>) | reverses items |
| replace(<start>, <end>, old, new) | replaces all specified values = a new value |
| copy(<start>, <end>, <iterator container>) | copies a range of elements from one container to another. |
| move(<start>, <end>, <iterator container>) | moves a range of elements from one container to another. |
| remove(<start>, <end>, value) | removes the first occurrence of a value from a given range |

1. **Iterators**

// Iterator is an object (like a pointer) that points to an element inside a container used to iterate through container:

* Declaring iterator:

Syntax: <container>::iterator <iterator name>;

* **Note**: The auto keyword automatically determine the data type of a variable based on the value assigned:

Syntax: auto variable = value**;**

// STL provides some member functions in STL container that return the iterators to the first and the last element:

|  |  |
| --- | --- |
| (rbegin) begin() | returns an (reverse) iterator to the beginning of container. |
| (rend) end() | returns an (reverse) iterator to the theoretical element just after the last element |
| (cbegin) cend() | returns an constant iterator (mean only access but can’t modify) |

// Types of iterator:

|  |  |  |
| --- | --- | --- |
| Input Iterator | read and only allows one-way traversal once | istream |
| Output Iterator | write and only allows one-way traversal once | ostream |
| Forward Iterator | Input + Output Iterator but multiple times | forward\_list, set, ... |
| Bidirectional Iterator | Forward Iterator but allow two-way | list, map, set |
| Random Access Iterator | Bidirectional but allow random access | vector, array, deque |

1. **Exceptions Handling**

// Exception is an error during the execution of this program, causing the program to stop:

* The try block executes code that may cause an exception and the catch block catch and handle exception.

Syntax: try {

/\*code can raise an exception\*/

}

catch(parameter) {

/\*handling exception\*/

}

…

catch(...) {

/\*handling any exception type as the default catch block\*/

}

* **Note**: Always catch exceptions by reference to avoid copying.
* The throw keyword throws an exception out of the try block:

Syntax: throw argument;

* Built-in Types are primitive data type like int, …, but it not provide any useful information.
* Standard Exception are classes represent common error types defined in <stdexcept> class:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | | | **Meaning** |
| exception | runtime\_error | | occurs at runtime. |
| logic\_error | out\_of\_range | occurs when accessing element outside the range of the list. |
| invalid\_argument | occurs when the passed argument is invalid. |
| bad\_malloc | | occurs when can’t allocate dynamic memory |

* **Note:** what() method return a string which is present in every stdexcept class to provide information.
* Custom Exception - define your own exception classes:

Syntax: class <my exception>{…};

1. **Templates**

// Templates allows you to write a generic code that can work with any data type.

* Defining generic template:

Syntax:template <typename A, typename B, …, type C, …>

// Defining entity (function, class with generic type)

* Creating generic template entity:

Syntax: entity<type1, type2, …, value, …>

* **Note**: Template Argument Deduction allows you to not specify the type, the compiler automatically infers it from the arguments passed in from C++17.

Syntax: entity

* **Note**: The typename keyword can alternative by class keyword.

// Template specialization allows you to define a specific version of a generic template:

Syntax:template <>

// Defining entity (function, class with specific type)

1. **Namespaces**

// Namespace provides the space where we can define or declare identifier to avoid name conflicts:

Syntax: namespace <name> {

// Define identifiers

}

* **Note**: Unnamed namespace helps avoid name conflicts within the current file scope.
* **Note**: The namespace can be aliased with a new name:

Syntax: namespace <alias> = <name>;

// The **using** keyword is used to specify a specific namespace:

* Directive - declare all identifiers in the namespace

Syntax:using namespace <name>;

* Declaration - declare identifiers specified in the namespace

Syntax:using namespace::<identifier>;