C++ Cheat Sheet

Syntax

Let's kick off our C++ reference sheet with syntax.

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!";
  return 0;
}
```

- **Line 1:** '#include <iostream>' specifies the *header file library*, which helps you deal with input and output objects like "cout." Header files are used to add specific functionality to C++ programs.
- **Line 2:** 'using namespace std' allows you to use names for objects and variables from the standard library.
- **Line 3:** Blank line. C++ ignores the spaces present within the code.
- **Line 4:** 'int main()', which is a function. Any code within the curly brackets {} will be executed.
- **Line 5:** cout is an object used along with the insertion operator (<<) to print the output text.
- **Line 6:** return 0 is used to end the main function.

While writing code in C++, always make sure you end each line with a semicolon to specify the end of the line. You must also add the closing bracket to end the main function; otherwise, you'll get errors while compiling the code.

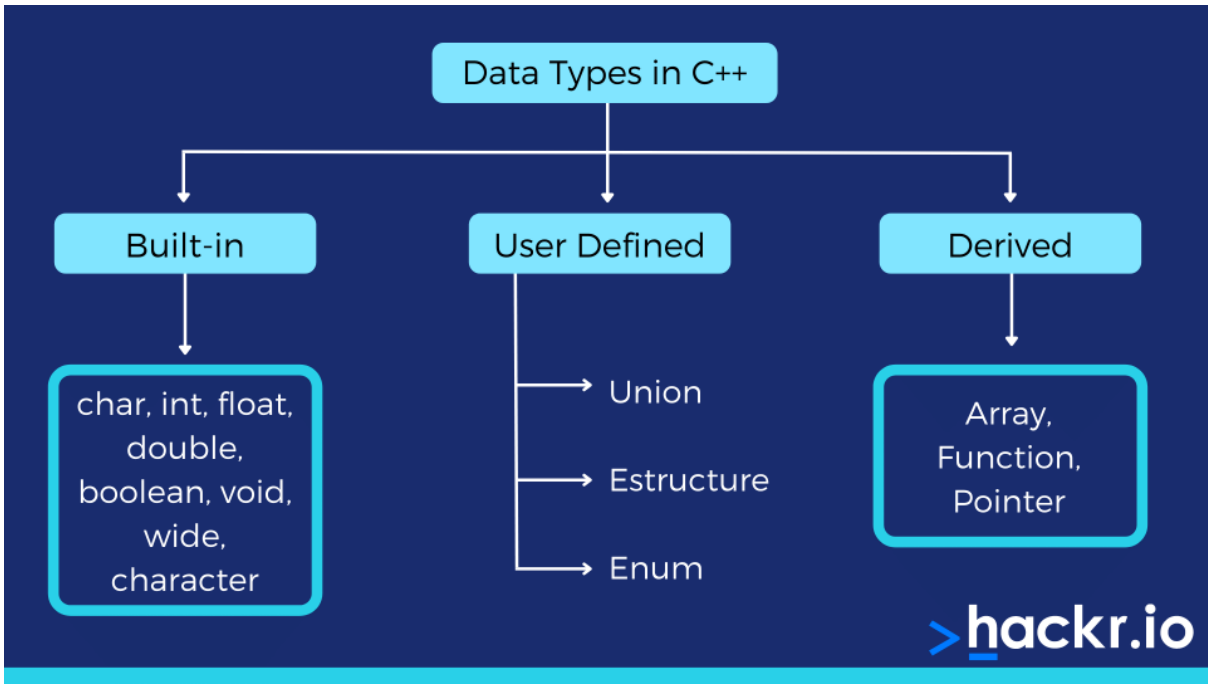
Comments

In C++, the compiler ignores the text followed by the comments. C++ supports two different types of comments:

//: specifies the single-line comment.
/* ….*/: specifies the multi-line comment.

Data Types

Data types specify the type of the data variable. The compiler allocates the memory based on the data types. The following are the C++ data types:



- **Built-in or primitive data types:** Pre-defined data types that can be used directly, including Integer, Character, Boolean, Floating Point, Double Floating Point, Valueless or Void, and Wide Character.
- **Derived data types:** Derived from primitive data types: function, array, pointer, and reference.
- **User-defined data types:** Defined by users: class, structure, union, enumeration, and Typedef.

Variables

Variables store the data values. C++ supports various types of variables, such as int, double, string, char, and float.

For example:

```
int num = 12; // Integer
string name = "Unity Buddy"; // String(text)
char ch = 'U'; //character
float fl = 5.99; // Floating point number
```

You can use alphabets, numbers, and the underscore for a variable name. However, variables cannot start with numbers or the underscore '_' character. Instead, they begin with letters

followed by numbers or the underscore '_' character. Moreover, you cannot use a keyword for the variable name.

Variables Scope

In C++, you can declare your variables within three parts of the program, also known as the scope of the variables:

Local Variables

These variables are declared within a function or block of code. Their scope is only limited to that function or block and cannot be accessed by any other statement outside that block.

For example:

```
#include <iostream>
using namespace std;

int main () {
   // Local variable:
   int a, b;
   int c;

   // initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c;

   return 0;
}
```

Global Variables

Global variables are accessible to any function, method, or block of the program. Usually, it is defined outside all the functions. The value of the global variable is the same throughout the program.

For example:

```
#include <iostream>
```

```cpp
using namespace std;

// Global variable:
int g;

int main () {
   // Local variable:
   int a, b;

   // initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;

   return 0;
}
```

Data Type Modifiers

Data type modifiers are used to modify a data type's maximum length of data. The following table will help you understand the size and range of built-in data types when combined with modifiers. There are four different types of modifiers available in C++, namely signed, unsigned, short, and long.

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615

signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Literals

Literals in C++ are data that you can use to represent the fixed values. You can use them directly within the code.

For example, 1, 2.5, “s”, etc.

There are different types of literal available in C++, as explained below:

Integer literal

An integer literal is numeric and does not have any fractional or exponential part.

For example:

Decimal (base 10): 0, -9, 22, etc.
Octal (base 8) : 021, 077, 033, etc.
Hexadecimal (base 16): 0x7f, 0x2a, 0x521, etc.

Floating-Point Literals

These are numeric literals that have either a fractional part or an exponent part.

For example: (-2.0, 0.8589, -0.26E -5).

Character Literal

These are single characters enclosed within a single quote.

For example: ‘a’, ‘F’, ‘2’, etc.

Escape Sequences

You can use escape sequences in C++ for untypable characters that have special meaning in C++.

For example:

Escape Sequences	**Characters**
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null Character

String Literal

This is a sequence of characters enclosed within double quotes.

For example:

"good"	string constant
""	null string constant
" "	string constant of six white space
"x"	string constant having a single character
"Earth is round\n"	prints string with a newline

Constants

To create a variable for which you do not want to change the values, you can use the “const” keyword.

For example:

```
const int LIGHT_SPEED = 2997928;
LIGHT_SPEED = 2500 // cannot change the value
```

Math Functions

C++ provides several functions that allow you to perform mathematical tasks. The following table highlights all the basic math functions available in C++:

Function	Description
abs(x)	Returns the absolute value of x
acos(x)	Returns the arccosine of x
asin(x)	Returns the arcsine of x
atan(x)	Returns the arctangent of x
cbrt(x)	Returns the cube root of x
ceil(x)	Returns the value of x rounded up to its nearest integer
cos(x)	Returns the cosine of x
cosh(x)	Returns the hyperbolic cosine of x
exp(x)	Returns the value of Ex
expm1(x)	Returns ex -1
fabs(x)	Returns the absolute value of a floating x
fdim(x, y)	Returns the positive difference between x and y
floor(x)	Returns the value of x rounded down to its nearest integer
hypot(x, y)	Returns sqrt(x2 +y2) without intermediate overflow or underflow
fma(x, y, z)	Returns x*y+z without losing precision
fmax(x, y)	Returns the highest value of a floating x and y
fmin(x, y)	Returns the lowest value of a floating x and y
fmod(x, y)	Returns the floating point remainder of x/y

pow(x, y)	Returns the value of x to the power of y
sin(x)	Returns the sine of x (x is in radians)
sinh(x)	Returns the hyperbolic sine of a double value
tan(x)	Returns the tangent of an angle
tanh(x)	Returns the hyperbolic tangent of a double value

User Inputs

C++ supports “cout” and “cin” for displaying outputs and for taking inputs from users, respectively. The cout uses the iteration operator (<<), and cin uses (>>).

For example:

```
int x;  // declaring a variable
cout << "Type a number: "; // Type any number and hit enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the value
```

Strings

A string is a collection or sequence of characters enclosed within double-quotes.

For example:

```
string str= "Hello";
```

To use string within your code, you must include the string library using this code line:

```
#include <string>
```

C++ will then allow you to perform various functions to manipulate strings. The following table describes the function names and their descriptions:

Function	**Description**
int compare(const string& str)	Compare two string objects
int length()	Finds the length of the string
void swap(string& str)	Swaps the values of two string objects

string substr(int pos, int n)	Creates a new string object of n characters
int size()	Return the length of the string in terms of bytes
void resize(int n)	Resizes the length of the string up to n characters
string& replace(int pos, int len, string& str)	Replaces the portion of the string beginning at character position pos and spans len characters
string& append(const string& str)	Adds a new character at the end of another string object
char& at(int pos)	Accesses an individual character at specified position pos
int find(string& str, int pos, int n)	Finds a string specified in the parameter
int find_first_of(string& str, int pos, int n)	Find the first occurrence of the specified sequence
int find_first_not_of(string& str, int pos, int n)	Searches for the string for the first character that does not match with any of the characters specified in the string
int find_last_of(string& str, int pos, int n)	Searches for the string for the last character of a specified sequence
int find_last_not_of(string& str, int pos)	Searches for the last character that does not match with the specified sequence
string& insert()	Inserts a new character before the character indicated by the position pos
int max_size()	Finds the maximum length of the string
void push_back(char ch)	Adds a new character ch at the end of the string
void pop_back()	Removes the last character of the string
string& assign()	Assigns new value to the string
int copy(string& str)	Copies the contents of string into another
void clear()	Removes all the elements from the string
const_reverse_iterator crbegin()	Points to the last character of the string
const_char* data()	Copies the characters of string into an array
bool empty()	Checks whether the string is empty or not

string& erase()	Removes the characters as specified
char& front()	Returns a reference of the first character
string& operator+=()	Appends a new character at the end of the string
string& operator=()	Assigns a new value to the string
char operator[](pos)	Retrieves a character at specified position pos
int rfind()	Searches for the last occurrence of the string
iterator end()	Refers to the last character of the string
reverse_iterator rend()	Points to the first character of the string
void shrink_to_fit()	Reduces the capacity and makes it equal to the size of the string
char* c_str()	Returns pointer to an array containing a null terminated sequence of characters
void reserve(inr len)	Requests a change in capacity
allocator_type get_allocator();	Returns the allocated object associated with the string

Operators

C++ supports different types of operators to add logic to your code and perform operations on variables and their respective values. Here are the C++ operator types:

Arithmetic Operators

You can perform common mathematical operations with arithmetic operators.

Operator	**Name**	**Example**
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y

++	Increment	++x
--	Decrement	--x

Assignment Operators

You can assign values to variables with assignment operators.

Operator	Example	Description	Same As
=	x = 5	For assigning a value to the variable.	x = 5
+=	x += 3	It will add the value 3 to the value of x.	x = x + 3
-=	x -= 3	It will subtract the value 3 from the value of x.	x = x - 3
*=	x *= 3	It will multiply the value 3 with the value of x.	x = x * 3
/=	x /= 3	It will divide the value of x by 3.	x = x / 3
%=	x %= 3	It will return the reminder of dividing the the value x by 3.	x = x % 3
&=	x &= 3		x = x & 3
|=	x |= 3		x = x | 3
^=	x ^= 3		x = x ^ 3
>>=	x >>= 3		x = x >> 3
<<=	x <<= 3		x = x << 3

Comparison Operators

You can use these operators to compare two values to return a true or false value. It will return true if both the values match, and false if they don’t match.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y

>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

These operators determine the logic between variables.

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Decision-Making Statements

Decision-making statements in C++ decide the flow of program execution. Here, programmers specify more than one condition. If a condition holds true the statements in that block are executed. Otherwise, the statements from other blocks are executed instead.

C++ has various decision-making instructions:

- If statement
- if..else statement
- Switch statement
- Nested if statement
- Nested switch statement
- Ternary operator

If Statement

This is the most basic type of decision-making statement. It instructs the compiler to execute the block of code only if the condition holds true.

Syntax:

```
if (expression)
{ //code}
```

Example:

```
#include <iostream>
using namespace std;

int main () {
int b = 10;
if( b < 20 ) {
          cout << "b is less than 20;" << endl;
   }
   cout << "value of a is : " << b << endl;

   return 0;
}
```

If..Else Statement

This is an extension of the ‘if’ statement. It instructs the compiler to execute the ‘if’ block only if the specified condition is true. Otherwise, it executes the ‘else’ block.

Syntax:

```
if (expression)
{//code}
else
{//code}
```

Example:

```
#include <iostream>
using namespace std;

int main () {
int b = 10;
if( b < 20 ) {
          cout << "b is less than 20;" << endl;
```

```
   }
   cout << "value of a is : " << b << endl;

   return 0;
}
```

Switch Statement

When you need to execute conditions against various values, you can use switch statements.

Syntax:

```
switch(expression) {
   case constant-expression  :
      statement(s);
      break; //optional
   case constant-expression  :
      statement(s);
      break; //optional

     default : //Optional
      statement(s);
}
```

Example:

```
#include <iostream>
using namespace std;

int main () {
   // local variable declaration:
   char grade = 'D';

   switch(grade) {
      case 'A' :
         cout << "Outstanding!" << endl;
         break;
      case 'B' :
      case 'C' :
         cout << "Well done" << endl;
         break;
```

```
      case 'D' :
         cout << "Pass" << endl;
         break;
      case 'F' :
         cout << "Try again" << endl;
         break;
      default :
         cout << "Invalid grade" << endl;
   }
   cout << "Your grade is " << grade << endl;

   return 0;
}
```

Nested If Statement

This is an “if” statement inside another “if” statement. You can use this type of statement when you need to base a specific condition on the result of another condition.

Syntax:

```
if( boolean_expression 1) {
   // Executes when the boolean expression 1 is true
   if(boolean_expression 2) {
      // Executes when the boolean expression 2 is true
   }
}
```

Example:

```
#include <iostream>
using namespace std;

int main () {
   // local variable declaration:
   int x = 100;
   int y = 200;

    if( x == 100 ) {
      if( y == 200 ) {
```

```
         cout << "Value of x is 100 and y is 200" << endl;
      }
   }
   cout << "Exact value of x is : " << x << endl;
   cout << "Exact value of y is : " << y << endl;

   return 0;
}
```

Nested Switch Statement

You can include one switch statement within another switch statement.

Syntax:

```
switch(ch1) {
   case 'A':
      cout << "This A is part of outer switch";
      switch(ch2) {
         case 'A':
            cout << "This A is part of inner switch";
            break;
         case 'B': // ...
      }
      break;
   case 'B': // ...
}
```

Example:

```
#include <iostream>
using namespace std;

int main () {
   int x = 100;
   int y = 200;

   switch(x) {
      case 100:
         cout << "This is part of outer switch" << endl;
         switch(y) {
```

```
            case 200:
               cout << "This is part of inner switch" << endl;
         }
   }
   cout << "Exact value of x is : " << x << endl;
   cout << "Exact value of y is : " << y << endl;

   return 0;
}
```

Ternary Operator

Exp1 ? Exp2 : Exp3;

First, expression Exp1 is evaluated. If it's true, then Exp2 is evaluated and becomes the value of the entire '?' expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Loops

Loops are used to execute a particular set of commands for a specific number of time based on the result of the evaluated condition. C++ includes the following loops

- While loop
- Do-while loop
- For loop
- Break statement
- Continue statement

While Loop

The loop will continue till the specified condition is true.

```
while (condition)
{code}
```

Do-While Loop

When the condition becomes false, the do-while loop stops executing. However, the only difference between the while and do-while loop is that the do-while loop tests the condition *after* executing the loop. Therefore, the loop gets executed at least once.

```
do
{
Code
}
while (condition)
```

For Loop

You can use the for loop to execute a block of code multiple times. This loop runs the block until the condition specified in it holds false.

```
for (int a=0; i< count; i++)
{
Code
}
```

Break Statement

This is used to break the flow of the code so the remaining code isn't executed. This brings you out of the loop.

For example:

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  cout << i << "\n";
}
```

Continue Statement

This statement will break the flow and take you to the evaluation of the condition. Later, it starts the code execution again.

For example:

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  cout << i << "\n";
}
```

Arrays

Arrays are derived data types that store multiple data items of similar types at contiguous memory locations.

For example:

```
string vehicles [4]; //declaring array to store up to 4 variables.
string vehicles[4]= {"car", "scooter", "cycle", "bike"};  //initializing
the array
```

Accessing Array Values

You need to use the index number to access the elements stored in an array.

```
string vehicles[4]= {"car", "scooter", "cycle", "bike"};
cout << vehicles [0];
```

Changing Array Elements

You can change the elements stored in an array using the index number.

```
string vehicles[4]= {"car", "scooter", "cycle", "bike"};
vehicles [0]= " "airplane";
cout << vehicles[0];
```

Functions

A function is a group of instructions to carry out a specific task. The common function in every C++ program is the main() function. You can even break down your complex code into multiple small functions and execute them separately.

For this, you need to declare, define, and call that function. C++ has several built-in functions that you can call directly within any program.

Defining a Function

The following is the syntax for defining a function in C++:

```
return_type function_name( parameter list ) {
   body of the function
}
```

Where:

- **return_type** specifies the type of value being returned by that function.
- **function_name** specifies the name of the function and needs to be unique.
- **parameter list** allows you to pass more than one value to your function, along with their data types.
- **body of the function** specifies the set of instructions to accomplish a task.

For example:

```
int max(int num1, int num2) {   // declaring the function max
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Function

You must call a function wherever you need it in your program.

For example:

```
#include <iostream>
using namespace std;

// function declaration
int max(int num1, int num2);

int main () {
    int a = 100;
   int b = 200;
   int ret;

    ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}
```

Function Arguments

You can pass arguments in three ways:

- **Call by value:** Passes the actual value of an argument into the formal parameter of the function. It will not make any change to the parameter inside the function and does not effect on the argument.
- **Call by pointer:** You can copy an argument address into the formal parameter. Here, the address accesses the actual argument used in the call. This means that changes made to the parameter affect the argument.
- **Call by reference:** You can copy an argument reference into the formal parameter. The reference accesses the actual argument used in the call. This means that changes made to the parameter affect the argument.

Storage Classes

Storage classes define the visibility of the variables and functions. C++ supports various storage classes, like auto, register, extern, static, and mutable.

Auto Storage Class

By default, C++ uses this storage class for all variables.

For example:

```
{
   int var;
   auto int var1;
}
```

You can only use the “auto” within functions for defining the local variables.

Register Storage Class

This storage class defines the local variables to be stored within the register rather than in RAM. It’s useful when you want to access the variable frequently, such as counters. The size of the variable will have a maximum size equal to the register size.

For example:

```
{
   register int  miles;
}
```

Static Storage Class

The static storage class tells the compiler to maintain local variables throughout the program without needing to create and destroy them when it comes into and goes out of scope. Defining a variable as static means it will maintain its values between function calls.

Global variables are static, which means their scope will be restricted to their declared file. If you specify a class data member as static, it creates only one copy of that member that all objects of its class will share.

For example:

```
#include <iostream>

// Function declaration
void func1(void);
```

```
static int count = 10; /* Global variable */

main() {
   while(count--) {
      func();
   }

   return 0;
}

// Function definition
void func1( void ) {
   static int i = 5; // local static variable
   i++;
   std::cout << "i is " << i ;
   std::cout << " and count is " << count << std::endl;
}
```

Extern Storage Class

The extern storage class provides a reference of a global variable and makes it visible to ALL the program files. When you specify a variable as ‘extern', the variable cannot be initialized because it points the variable name at a storage location that has been previously defined.

In case of multiple files where you define a global variable or function, also to be used in other files, extern will provide a reference in another file of defined variable or function. You must use the extern modifier when you have to share the same global variables or functions between two or more files.

For example:

Program 1

```
#include <iostream>
int count ;
extern void write_extern();

main() {
   count = 5;
   write_extern();
}
```

Program 2

```
#include <iostream>

extern int count;

void write_extern(void) {
   std::cout << "Count is " << count << std::endl;
}
```

Mutable Storage Class

You can use this storage class if you want an object member to override the member function. That is, a mutable member that can be modified by a const member function.

Structure

Structure allows you to define the data items of the non-similar data types. To use a structure, you must define it and access its structure members.

The following is the syntax for creating a structure:

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

For example, we want to create a structure of books consisting of title, author, subject, and book_id, as follows:

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

You must use the member access operator (.) to access structure members. This is a period between the structure variable name and the structure member that we wish to access.

```
#include <iostream>
#include <cstring>

using namespace std;

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
```

```
    cout << "Book 2 id : " << Book2.book_id <<endl;

    return 0;
}
```

References

When you declare a variable as a reference, it acts as an alternative to the existing one. You need to specify the reference variable with “&”, as shown below:

```
string food = "Pizza";
string &meal = food;    // reference to food
```

Pointer

A pointer in C++ is a variable that stores the memory address of another variable. Similar to regular variables, pointers also have data types. We use ‘*’ to declare pointers in C++.

For example:

```
string food = "Pizza"; // string variable

cout << food;  // Outputs the value of food (Pizza)
cout << &food; // Outputs the memory address of food (0x6dfed4)
```

Classes and Objects

C++ is an object-oriented programming language with classes and objects. Class is a user-defined data type you can use to bind data members and member functions together. You can access them by creating an instance of that class.

Creating a Class

Here’s how to create a class in C++:

```
class MyClass {       // The class
  public:             // Access specifier- accessible to everyone
    int myNum;        // Attribute (int variable)
      string myString;  // Attribute (string variable)
};
```

Creating an Object

Objects work as an instance of the class, allowing you to access its members, functions, and variables. You must use the dot (.) operator, as shown below:

```
class MyClass {
  public:
    int myNum;
      string myString;
};

int main() {
  MyClass myObj;  // Creating an object of MyClass

   myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

Creating Multiple Objects

Here's an example of how to create multiple objects of the same class:

```
class Car {
  public:
      string brand;
      };

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  // Print attribute values
  cout << carObj1.brand "\n";
```

```
  cout << carObj2.brand "\n";
  return 0;
}
```

Class Methods

Methods are like functions that are defined within a class. C++ has two types of methods: inside class and outside class.

Inside Class Method

```
class MyClass {
  public:
    void myMethod() {  // Method/function inside the class
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Outside Class Method

```
class MyClass {
  public:
    void myMethod();   // Method declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // object creation
  myObj.myMethod();  // Call the method
```

```
  return 0;
}
```

Constructors

A constructor is a method automatically called upon object creation. It has the same name as the class name, and no data type.

For example:

```
class Fir_Class {
  public:
      Fir_Class() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  Fir_Class myObj;    // call the constructor
  return 0;
}
```

Access Specifiers

Access specifiers define the access of the class members and variables. C++ supports three types of access specifiers:

- **Public:** Class members and variables are accessible from outside the class.
- **Private:** Class members and variables are accessible only within the class and not outside the class.
- **Protected:** Class members and variables are accessible only in their subclasses.

Encapsulation

Encapsulation helps you hide sensitive data from the users. Here, we use the private access specifier for declaring the variables and methods. If you want to allow others to read or modify those variables and methods, you must use the public get and set methods.

For example:

```
#include <iostream>
```

```
using namespace std;

class Employee {
  private:
        int name;

  public:
    // Setter
    void setName(int n) {
      name= s;
      }
    // Getter
    int getName() {
      return name;
      }
};

int main() {
  Employee myObj;
  myObj.setName("Bob");
  cout << myObj.getName();
  return 0;
}
```

Inheritance

C++ supports inheritance, allowing you to inherit the members and variables of one class to another. The inheriting class is the child class and the other is the parent class. You must use (:) symbol to inherit:

```
// Parent class
class Vehicle {
  public:
    string brand = "Ford";
    void sound() {
      cout << "honk \n" ;
      }
};

// Child class
class Car: public Vehicle {
```

```
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.sound();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

Polymorphism

Polymorphism specifies the “many forms.” It is the ability of a single message to be displayed in multiple forms and takes place when you have multiple child classes and one base class.

For example:

```
// Parent class
class Animal {
  public:
    void sound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Child class
class Pig : public Animal {
  public:
    void sound() {
      cout << "The pig says: wee wee \n" ;
  }
};

// Derived class
class Dog : public Animal {
  public:
    void sound() {
      cout << "The dog says: bow wow \n" ;
  }
};
int main() {
```

```
  Animal ani;
  Pig myPig;
  Dog myDog;

  ani.sound();
  myPig.sound();
  myDog.sound();
  return 0;
}
```

File Handling

You can use an fstream library to handle files. The fstream library consists of <iostream> and <fstream> header file.

#include <iostream>
#include <fstream>

ofstream: create and write to the files.
ifstream: read from the specified file.
fstream: combination of above both.

Creating and Writing

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  // Create and open a text file
  ofstream MyFile("filename.txt");

  // Write to the file
  MyFile << "content";

  // Close the file
  MyFile.close();
}
```

Reading

```
// text string to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// for reading the file line by line
while (getline (MyReadFile, myText)) {
  // Output the text from the file
  cout << myText;
}

// Close the file
MyReadFile.close();
```

Exceptions

While compiling and running, you might run into errors. C++ allows you to handle and catch these errors using exception handling. The following is the syntax for exception handling that includes a try-catch block:

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

For example:

```
try {
  int age = 10;
  if (age >= 20) {
    cout << "you are old enough.";
  } else {
    throw 505;
  }
}
```

```
catch (int num) {
  cout << "Access denied \n";
  cout << "Error number: " << num;
}
```

Preprocessor

The following are some in-built preprocessors available in C++ for various functionalities.

```
#include  <stdio.h>       // Insert standard header file
#include "myfile.h"       // Insert file in current directory
#define X some text       // Replace X with some text
#define F(a,b) a+b        // Replace F(1,2) with 1+2
#define X \
 some text                // Multiline definition
#undef X                  // Remove definition
#if defined(X)            // Conditional compilation (#ifdef X)
#else                     // Optional (#ifndef X or #if !defined(X))
#endif                    // Required after #if, #ifdef
```

Dynamic Memory Management

```
#include <memory>           // Include memory (std namespace)
shared_ptr<int> x;          // Empty shared_ptr to a integer on heap. Uses
reference counting for cleaning up objects.
x = make_shared<int>(12);   // Allocate value 12 on heap
shared_ptr<int> y = x;      // Copy shared_ptr, implicit changes reference count to
2.
cout << *y;                 // Dereference y to print '12'
if (y.get() == x.get()) {   // Raw pointers (here x == y)
    cout << "Same";
}
y.reset();                  // Eliminate one owner of object
if (y.get() != x.get()) {
    cout << "Different";
}
if (y == nullptr) {         // Can compare against nullptr (here returns true)
    cout << "Empty";
}
y = make_shared<int>(15);   // Assign new value
cout << *y;                 // Dereference x to print '15'
cout << *x;                 // Dereference x to print '12'
weak_ptr<int> w;            // Create empty weak pointer
w = y;                      // w has weak reference to y.
```

```
if (shared_ptr<int> s = w.lock()) { // Has to be copied into a shared_ptr before
usage
    cout << *s;
}
unique_ptr<int> z;          // Create empty unique pointers
unique_ptr<int> q;
z = make_unique<int>(16);   // Allocate int (16) on heap. Only one reference
allowed.
q = move(z);                // Move reference from z to q.
if (z == nullptr){
    cout << "Z null";
}
cout << *q;
shared_ptr<B> r;
r = dynamic_pointer_cast<B>(t); // Converts t to a shared_ptr<B>
```

Floating Point Math

You must include the “cmath” library to perform tasks on floating-point numbers.

```
#include <cmath>            // Include cmath (std namespace)
sin(x); cos(x); tan(x);     // you can perform Trig functions, x (double) is in radians
asin(x); acos(x); atan(x);  // Inverses
atan2(y, x);                // atan(y/x)
sinh(x); cosh(x); tanh(x);  // Hyperbolic sin, cos, tan functions
exp(x); log(x); log10(x);   // e to the x, log base e, log base 10
pow(x, y); sqrt(x);         // x to the y, square root
ceil(x); floor(x);          // Round up or down (as a double)
fabs(x); fmod(x, y);        // Absolute value, x mod y
```

iostream.h and iostream

```
#include <iostream>         // Include iostream (std namespace)
cin >> x >> y;              // Read words x and y (any type) from standard input
cout << "x=" << 3 << endl;  // Write line to stdout
cerr << x << y << flush;    // Write to stderr and flush
c = cin.get();              // c = getchar();
cin.get(c);                 // Read char
cin.getline(s, n, '\n');    // Read line into char s[n] to '\n' (default)
if (cin)                    // Good state (not EOF)?
                            // To read/write any type T:
istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;}
```