# SimpleCPP / C++ Reference Sheet

- **Helloworld program**

```
#include <simplecpp>
int main() {
  cout << "Hello World!" << endl;
}
```

- **SimpleCPP Drawing**
  - Initialize: **`turtleSim();`**
  - Move turtle forward by 100: `forward(100);`
  - Turn the turtle left by 90 degrees: `left(90);`
  - Turn the turtle right by 90 degrees: `right(90);`
  - Raise the pen: `penUp();`
  - Lower the pen: `penDown();`
  - `Wait for 1 millisecond: wait(1);`
  - Repeat certain statements/steps N times

```
repeat (N) {
   // Statements
}
```

- **Datatypes**
  - Integer: `int num = 123;`
  - Character: `char ch = 'a';`
  - `String: string str = "My computer course";`
  - Decimal numbers: `float num = 123.45;`
  - Decimal numbers (higher range): `double num = 123.45;`
  - Boolean (true or false): `boolean b = false;`
  - Constant: `const int maxItems = 10;`
- **ASCII range**
  - `A to Z:` 65 to 90
  - `a to z:` 97 to 122
  - `0 to 9:` 48 to 57
- **Input / Output**
  - `cin >> num;  cout << num;`
- **Arithmetic Operators**
  - Add, subtract, multiply, divide, get remainder: **`+  -  *  /  %`**
  - Increment num by 1, Decrement num by 1: **`num++  num--`**
  - Increment num by X, Decrement num by X: **`num += X  num -= X`**
- **Relational Operators:**
  - Less than, greater than: **`<  >`**
  - less than equal to, greater than equal to: **`<=  >=`**
  - Check if equal to: **`==`**
  - Check if not equal: **`!=`**
- **Logical Operators**
  - And, Or, Not: **`&&  ||  !`**
- **Bitwise Operators**
  - AND, OR, XOR: `&  |  ^`
- **Ternary Conditional Operator**
  - E.g. x > y ? x : y  (if x is greater than y then value of x will be considered, else y)

- **Example Conditions**
  - Num less than 10: **`num < 10`**
  - Num in between 1 and 10, both inclusive: **`num >= 1 && num <= 10`**
  - Number less than 0 or num greater than 10: **`num < 0 || num > 10`**
  - Number is equal to 6: **`num == 6`**
  - Number is not equal to 6: **`num != 6`**
  - The actual number (digits) can be replaced with variable names as well
- **Conditional statements**
  - **If Statement**
    ```
    if (condition) { // Code to be executed if the condition is true  }
    ```
  - **If-Else Statement**
    ```
    if (condition) {
       // Code to be executed if the condition is true
    }
    else {
       // Code to be executed if the condition in the if is false
    }
    ```
  - **If-Elseif Statement**
    ```
    if (condition1) {
        // Code to be executed if condition1 is true
    }
    else if (condition2) {
        // Code to be executed if condition1 is false
        // and condition2 is true
    }
    else {
        // Code to be executed if all conditions are false
    }
    ```
- **Loops**
  - **While loop (Syntax)**
    ```
    while (condition) {
       // statements
       ...
       update_condition;
    }
    ```
  - **While loop (Example)**
    ```
    int num = 0;
    while (num <= 10) {
       cout << num;
       num++;
    }
    ```
  - **For loop (Syntax)**
    ```
    for (initialization; condition; increment/decrement) {
        // statements
    }
    ```
  - **For loop (Example)**
    ```
    for (int num = 1; num <= 10; num++) {
        cout << num;
    }
    ```
- Ignore the remaining statements and jump outside the current loop: **`break;`**
- Ignore the remaining statements and go the next iteration in the current loop: **`continue;`**

- **Functions (Pass by value/reference, Returning reference, Overloading)**
  - **Call by Value (Lecture 07)**

**Syntax**

```
returnType functionName(datatype variable, …) {
    … // Some operations
}
int main() {
    functionName (variable, …)
}
```

**Example 1 - Add two numbers (No return type)**

```
void add(int a, int b) {
    cout << (a+b);
}
int main() {
    int x, y; cin >> x >> y;
    add(x,y);
}
```

**Example 2 - Add two numbers (Return type is integer)**

```
int add(int a, int b) {
    return (a+b);
}
int main() {
    int x, y; cin >> x >> y;
    int result = add(x,y);
    cout << result;
}
```

  - **Pass by reference (Returning many values) (Lecture 8)**

**Example - Add and multiply two numbers (Print in main)**

```
void operations(int a, int b, int &c, int &d) {
    c = a + b;  d = a * b;
}
int main() {
    int x, y, add, mul; cin >> x >> y;
    operations(x, y, add, mul);
    cout << add << " " << mul;
}
```

  - **Returning a reference (Lecture 8)**

**Example**

```
int& maximum (int &c, int &d) {
    if (x>=y) return x; else return y;
}
int main() {
    int x, y; cin >> x >> y;
    maximum(x, y) = 0;
}
```

- **Function Overloading (Lecture 9 - Slides 1 to 11)**

  Same function name, different types/number of parameters

  **Syntax**

```
returnType functionName(parameter_list1) {
    ... // Some operations
}

returnType functionName(parameter_list2) {
    ... // Some operations
}
```

  **Example 1 - Add numbers int and double**
  **(Same function name but different datatypes)**

```
int add(int a, int b) {
    return a + b;
}
double add(double a, double b) {
    return a + b;
}
int main() {
    cout << add(5, 10);
    cout << add(5.5, 10.3);
}
```

  **Example 2 - Greet person**
  **(Default / Optional arguments - )**

```
void greet(string name, string title = "Mr./Ms.") {
    cout << "Hello, " << title << " " << name;
}
int main() {
    greet("Vijay"); // default title
    greet("Manoj", "Prof.");  // overrides default title
}
```

- **Templates**

  Allows writing functions/classes that work with any data type

  Syntax: **`template <typename T>`**. **`T`** is a placeholder for int, float, string, bool, etc.

  Example - Add two integers, float, and string using the same function add

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
int main() {
    int num1 = 5, num2 = 10;
    float num3 = 2.4, num4 = 3.5;
    string s1 = "hello", s2 = "world";
    cout << add(num1, num2); // Output: 15
    cout << add(num3, num4); // Output: 5.9
    cout << add(s1, s2);     // Output: helloworld
}
```

- **Structures (Lecture 9 - Slides 12 onwards)**

```
// Syntax
struct NameOfStructure {
   datatype variableName; // Member variables
   ...
};
```

  - Creating object in the main function
    - Syntax: NameOfStructure objectName
  - Accessing member variables in the main function
    - Syntax: objectName.memberVariable

```
// Example
struct myVector {
   double x, y, z;
};
int main() {
    myVector a;
    a.x = 12.34;
}
```

- **Pass structure to a function (Lecture 9 - Slides 12 onwards)**

**Syntax**

```
ReturnType functionName(NameOfStructure object, ...) {
   ...
   // Operation on/using object.memberVariable
   ...
}
int main() {
   NameOfStructure objectName
   functionName(objectName, ...);
}
```

**Example**

```
void add(myVector a, myVector b) {
   cout << a.x + b.x << " ";
   cout << a.y + b.y << " ";
   cout << a.z + b.z << " ";
}
int main() {
   myVector v1, v2;
   // ... Some operations
   add(v1, v2);
}
```

- **Pass structure to a function by reference (Lecture 9 - Slide 14)**

  **Syntax**

```
ReturnType functionName(const NameOfStructure& object, ...) {
   ...
   // Operation on/using object.memberVariable
   ...
}
// Example - Area of rectangle
struct Rectangle {
    int length;
    int width;
};
int area(const Rectangle& rect) {
    return rect.length * rect.width;
}
int main() {
    Rectangle rect;
    rect.length = 5;
    rect.width = 3;
    cout << area(rect);
}
```

- **Functions returning a structure**

```
// Syntax
NameOfStructure functionName(NameOfStructure object, ...) {
   ...
   // Operation on/using object.memberVariable
   ...
}
int main() {
   NameOfStructure objectName1, objectName2;
   objectName2 = functionName(objectName1, ...);
}

// Example
myVector add(myVector a, myVector b) {
   myVector ans;
   ans.x = a.x + b.x;
   ans.y = a.y + b.y;
   ans.z = a.z + b.z;
   return ans;
}
int main() {
   myVector v1, v2;
   // ... Some operations
   myVector ans1 = add(v1, v2);
   // ... Some operations
}
```

- **Functions returning a structure as reference**

```
// Syntax
NameOfStructure& functionName(NameOfStructure object, ...) {
   ...
}
```

- **Arrays (Lectures 12 and 13)**

Index starts from 0

**datatype arrayName[No. of elements]**

```
// Replace int with any other datatype
int A[10];
int B[] = {1, 2, 3, 4};
int C[10] = {0};
int D[5] = {1, 2};
```

**Accessing a particular element**

**Syntax:** arrayName[index]

```
A[5], A[0], etc.
Index can be replaced with a variable
cin >> A[i]; cout << A[i];
cin.getline(A, 1000); // Here A is a char array
```

**Accept 10 elements and print them**

```
int arr[10];
for(int i = 0; i < 10; i++) {
    cin >> arr[i];
}
for(int i = 0; i < 10; i++) {
    cout << arr[i] << " ";
}
```

- **Passing Arrays to Functions**

Array elements are always passed by reference

**Syntax**

```
returnType functionName(datatype arrayName[], …) {
    … // Some operations
}
int main() {
    functionName (arrayName, …)
}
```

**Example 1 - Double each element of the array (No return type)**

```
void compute(int arr[], int N) {
    for(int i = 0; i < N; i++) {
        arr[i] = arr[i] * 2;
    }
}
int main() {
    int arr[100], N = 10;
    compute(arr,N);
}
```

- **Creating Array of type Struct**

  **Syntax**

```
struct structName {
    // member variables
}
int main() {
    structName obj[N];
}
```

  **Example**

```
struct student {
    int roll;
    int marks;
};
int main(){
    student st[100];
    int N; cin >> N;
    for(int i = 1; i <= N; i++) {
        cin >> st[i].roll >> st[i].marks;
    }
    for(int i = 1; i <= 2; i++) {
        cout << st[i].roll << " " << st[i].marks << endl;
    }
}
```

- **Recursion Basics (Lectures 15)**
  `A function that calls itself`

- **Recursion cntd (Lectures 16)**
  - Binary search (Slide 4)
  - Sqrt (Slide 6)
  - Tail recursion (Slide 8)
  - Merge sort (Slide 13)
  - Memo-ization (Slide 16)

- **Pointers (Lectures 17 and 18)**
  - **Basics**
    Declaration: `int* ptr1;`
    Initialization: `int a = 5; int* ptr = &a;`
    Dereferencing: `int value = *ptr;`
    Modifying value: `*ptr = 10; OR (*p)++;`
  - **Pointer arithmetic**

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;
for (int i = 0; i < 5; i++) {
    cout << *ptr << " " << *(p+i);
    ptr++; // Move to the next element
}
```

- **Passing pointers to function**

```
void fun(int* num) {
    *num = 10;
}
int main() {
    int a = 5;
    int *p = &a;
    fun(p);
    cout << a; // 10
    int b = 20;
    fun(&b);
    cout << b;  // 10
}
```

- Pointer to pointer: `int *p; int **p = &p;`
- Dynamic Memory Allocation: `int *p = new int; int *arr = new int[10];`
- Deallocating memory: `delete p; delete[] arr;`
- Null pointer: `int *ptr = nullptr`

- **Pointers and Structures**

```
struct Student {
    int id;
    float marks;
};
int main() {
    // Dynamically allocate memory for Student structure
    Student* st = new Student();
    cin >> st->id;  // Access member variable id using ->
    cin >> st->marks;
    cout << st->id;
    cout << st->marks;
    delete st; // free the memory
}
```

- **Classes (Lecture 19)**

```
// Syntax
class NameOfClass {
    datatype variableName; // Private Member variables
public:
    returnType functionName(...); // member functions/methods
};
```

- Creating object in the main function
  - Syntax: NameOfClass objectName
- Accessing member variables in the main function
  - Syntax: objectName.variableName

```
// Example - Compute the area of a rectangle. Print in the main
// All in one single file (xyz.cpp)
class Rectangle {
private:
    int length, width;
public:
    Rectangle() : length(1), width(1) {}
    Rectangle(int l, int w) : length(l), width(w) {}
    int computeArea() {
        int area = length * width;
        return area;
    }
};
int main() {
    Rectangle rect1;
    cout << rect1.computeArea();
    Rectangle rect2(5, 3);
    cout << rect2.computeArea();
}
```

```
// Example - Compute the area of a rectangle. Print in the main
// Multiple files

// main.cpp
#include <iostream>
#include "rect.h"
using std::cout; using std::cin; using std::endl;
int main() {
    Rectangle rect1;
    cout << rect1.computeArea() << endl;
    Rectangle rect2(5, 3);
    cout << rect2.computeArea() << endl;
}
```

```
// rect.h
class Rectangle {
private:
    int length, width;
public:
    Rectangle();
    Rectangle(int l, int w);
    int computeArea();
};
// rect.cpp
#include "rect.h"
Rectangle::Rectangle() : length(1), width(1) {}
Rectangle::Rectangle(int l, int w) : length(l), width(w) {}
int Rectangle::computeArea() {
    return length * width;
}
```

- **Pointers and Classes**

```
struct Student {
    int id;
    float marks;
};
int main() {
    // Dynamically allocate memory for Student structure
    Student* st = new Student();
    cin >> st->id;  // Access member variable id using ->
    cin >> st->marks;
    cout << st->id;
    cout << st->marks;
    delete st; // free the memory
}
```

- **Operator Overloading**

```
class Complex {
public:
    int real, imag;
    Complex(int r, int i) : real(r), imag(i) {}

    // Overload + operator
    Complex operator+(const Complex& rhs) {
        return Complex(real + rhs.real, imag + rhs.imag);
    }
};
int main() {
    Complex c1(3, 4);   Complex c2(1, 2);
    Complex c3 = c1 + c2;
    cout << c3.real << " + " << c3.imag << "i" << endl;
}
```

- **C++ Standard Library - Vectors**

| **Purpose** | **Usage/Syntax** |
|---|---|
| Empty int vector | `vector<int> v1;` |
| Integer vector container of 10 integers with all set to value 3. | `vector<int> v2 (10,3);` |
| Find the number of elements in the vector<br>Currently, v1 is 0 | `v1.size();` |
| Test if the vector is empty or not | `v1.empty()`<br>`if (v1.empty()) ...` |
| Adds an item at the end of the vector | `v1.push_back(12);`<br>`v1.push_back(52);`<br>`v1.push_back(13);`<br>`v1.push_back(-4);`<br>`// Vector: 12 52 13 -4` |
| Access the 0th element of the vector | `v1.front();  // 12` |
| Access the last element of the vector | `v1.back();  // -4` |
| Access the element at the 2nd position | `v1.at(2);  // 0 based indexing` |
| Remove the last element from the vector | `v1.pop_back();  // 12 52 13` |
| Print the elements of the vector<br>Method 1 | `for(int i = 0; i < v1.size(); i++)`<br>`  cout << v1[i] << " ";` |
| Print the elements of the vector<br>Method 2 | `typedef vector<int>::iterator itr;`<br>`for(itr i = v1.begin(); i!=v1.end(); i++)`<br>`  cout << *i << " ";` |
| Print the elements of the vector<br>Method 3 | `for(int x: v1)`<br>`  cout << x << " ";` |
| Insert element 25 at the 2nd position in the vector (0 based indexing) | `v1.insert(v1.begin()+2,25);`<br>`// 12 52 25 13` |
| Delete the element from the 2nd position (0 based indexing) | `v1.erase(v1.begin()+2);`<br>`// 12 52 13` |
| Sort the vector in ascending order | `sort(v1.begin(), v1.end());`<br>`// 12 13 52` |
| Sort the vector in descending order | `sort(v1.begin(),v1.end(),std::greater<int>());`<br>`// 52 13 12` |