**American International University Bangladesh (AIUB)**

****

**Faculty of science & Technology**

**Department of Computer Science**

**LAB MANUAL 01**  
CSC 2207 Programming Language 2 [EEE]

|  |
| --- |
| **TITLE** |

**Array, Function and Pointer in C++**

|  |
| --- |
| **PREREQUISITE** |

* To know about variables, input and output
* To know about arrays
* To know about functions
* To know about control structure
* To know about class and Objects
* Pointers and pointer to objects

|  |
| --- |
| **OBJECTIVE** |

* To get an idea about pointers and object
* To get knowledge about stack memory and heap memory
* To understand about Dynamic memory allocations
* To know about STL(Standard Template Library)

|  |
| --- |
| **THEORY** |

**Stack memory**

The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated.

🡪Array and Pointer

🡪 Dynamic Memory allocation

🡪Call by reference and call by value (swap)

Function

============

input:

--> no arguments

--> a number

--> List of number/ Pointer(1D-array/2D-Matrix)

output:

--> No return(void in C++/Java)

--> return a number

--> return list of number/ Pointer(1D-array/2D-Matrix)

**Examples**

Here is a short program that creates its variables on the stack. It looks like the other programs we have seen so far.

#include <stdio.h>

double multiplyByTwo (double input) {

double twice = input \* 2.0;

return twice;

}

int main (int argc, char \*argv[])

{

int age = 30;

double salary = 12345.67;

double myList[3] = {1.2, 2.3, 3.4};

printf("double your salary is %.3f\n", multiplyByTwo(salary));

return 0;

}

double your salary is 24691.340

we declare variables**: an int, a double, and an array of three doubles**. These three variables are pushed onto the stack as soon as the main() function allocates them. When the **main()** function exits (and the program stops) these variables are popped off of the stack. Similarly, in the function **multiplyByTwo(),** the **twice** variable, which is a double, is pushed onto the stack as soon as the **multiplyByTwo()** function allocates it. As soon as the **multiplyByTwo()** function exits, the twice variable is popped off of the stack, and is gone forever.

**Heap memory**

The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate.

#include <stdio.h>

#include <stdlib.h>

double \*multiplyByTwo (double \*input) {

double \*twice = malloc(sizeof(double));

\*twice = \*input \* 2.0;

return twice;

}

int main (int argc, char \*argv[])

{

int \*age = malloc(sizeof(int));

//int \*age = new int; // you can use new keywords.

\*age = 30;

double \*salary = malloc(sizeof(double));

\*salary = 12345.67;

double \*myList = malloc(3 \* sizeof(double));

//double \*myList = new double[3]; // alternative

myList[0] = 1.2;

myList[1] = 2.3;

myList[2] = 3.4;

double \*twiceSalary = multiplyByTwo(salary);

printf("double your salary is %.3f\n", \*twiceSalary);

free(age);

free(salary);

free(myList);

free(twiceSalary);

return 0;

}

**New keywords**

#include<iostream>

using namespace std;

int main(){

int n;

cout<<"Enter the size of array: ";

cin>>n;

//int \*ar = (int\*)malloc(n\*sizeof(int)); // dynamic memory allocation

int \*ar = new int[n]; // use new keyword instead of malloc

for(int i=0;i<n;i++){

cin>>ar[i];

}

for(int i=0;i<n;i++){

cout<<ar[i]<<" ";

}

free(ar);

return 0;

}

**Objects can be created with commands of the following form:**

* **On the stack**

My\_class my\_object(arglist);

1. **On the heap:**

My\_class\* my\_objptr = new My\_class(arglist);

1. C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory.
2. C++ supports these functions and also has two operators **new and delete** that perform the task of allocating and freeing the memory in a better and easier way.

**More Details:**

1. <https://courses.engr.illinois.edu/cs225/fa2020/resources/stack-heap/>
2. <https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html>
3. <https://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Companion/first_steps/stack_and_heap.html\>
4. <https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/>
5. <https://www.programiz.com/cpp-programming/memory-management>

**STL (Standard Template Library) in C++**

**Vector**

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

#include<iostream>

#include<vector>

using namespace std;

void printVector(vector<int> vec){

cout<<"Size of vector: "<<vec.size()<<endl;

for(int i=0;i<vec.size();i++){

cout<<vec[i]<<endl;

}

}

int main(){

// initialising the vector

//vector<int> v = { 10, 20, 30, 40 };

vector<int> v;

printVector(v);

v.push\_back(10);

v.push\_back(20);

v.push\_back(30);

printVector(v);

return 0;

}

**Vector printing using iterator and insert value in any position**

#include<iostream>

#include<vector>

#include<iterator>

using namespace std;

int main(){

vector<int> vec={10,20,30,40,50};

vector<int>::iterator it;

//insert 500 in the front

vec.insert(vec.begin(),500);

//insert 100 in 2 index position

vec.insert(vec.begin()+2,100);

//printing vector using iterator

for(it=vec.begin();it!=vec.end();it++){

cout<<\*it<<" ";

}

return 0;

}

**Insert and delete from vector**

#include<iostream>

#include<vector>

#include<iterator>

using namespace std;

void printVector(vector<string> vec){

cout<<"Size of vector: "<<vec.size()<<endl;

for(int i=0;i<vec.size();i++){

cout<<vec[i]<<" ";

}

cout<<endl;

}

int main(){

vector<string> v={"Richard","Aaron","Florence"};

//insert beginning

v.insert(v.begin(),"Supta");

//insert position 2

v.insert(v.begin()+2,"Philip");

//insert end

v.insert(v.end(),"Mr. Hi");

printVector(v);

//delete from end

// v.erase(v.end());

//delete from begin

// v.erase(v.begin());

//delete elements from 3 to end

v.erase(v.begin()+3,v.end());

printVector(v);

//delete all elements

v.clear();

printVector(v);

return 0;

}

**Vector of objects(class or struct)**

#include<iostream>

#include<vector>

using namespace std;

class student{

private:

int id;

string name;

public:

student(int id, string name){

this->id=id;

this->name=name;

}

int getId(){

return id;

}

string getName(){

return name;

}

};

void printVector(vector<student> vec){

cout<<"Size of vector: "<<vec.size()<<endl;

for(int i=0;i<vec.size();i++){

cout<<vec[i].getId()<<" "<<vec[i].getName()<<endl;

}

}

int main(){

vector<student> v;

printVector(v);

student s1(100,"Richard");

student s2(200,"Philip");

v.push\_back(s1);

v.push\_back(s2);

printVector(v);

return 0;

}

**List**

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a **List**, we talk about **doubly linked list**. For implementing a **singly linked list**, we use **forward list**.

**List Example**

#include <iostream>

#include <list>

#include <iterator>

using namespace std;

//function for printing the elements in a list

void printList(list <int> g)

{

list <int> :: iterator it;

for(it = g.begin(); it != g.end(); ++it)

cout << " " << \*it;

cout << "\n";

}

int main()

{

list <int> l{20,100,30,10,12,1,150};

printList(l);

l.push\_back(123);

printList(l);

l.push\_front(900);

printList(l);

cout << "\nl.front() : " << l.front();

cout << "\nl.back() : " << l.back();

cout << "\nl.pop\_front() : ";

l.pop\_front();

printList(l);

cout << "\nl.pop\_back() : ";

l.pop\_back();

printList(l);

cout << "\nl.reverse() : ";

l.reverse();

printList(l);

cout << "\nl.sort(): ";

l.sort();

printList(l);

return 0;

}

**Forward list Example**

#include<iostream>

#include<forward\_list>

#include <iterator>

using namespace std;

void printList(forward\_list <int> g)

{

forward\_list <int> :: iterator it;

for(it = g.begin(); it != g.end(); ++it)

cout << " " << \*it;

cout << "\n";

}

int main()

{

// Declaring forward list

forward\_list<int> fl{10,20,50,20,40};

printList(fl);

fl.push\_front(20);

printList(fl);

fl.pop\_front();

printList(fl);

return 0;

}

[**https://www.studytonight.com/cpp/stl/stl-container-list#**](https://www.studytonight.com/cpp/stl/stl-container-list)

**Stack**

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only**.**

#include <bits/stdc++.h>

using namespace std;

void printStack(stack <int> s)

{

while (!s.empty())

{

cout <<" "<< s.top();

s.pop();

}

cout << '\n';

}

int main ()

{

stack <int> s;

s.push(10);

s.push(30);

s.push(20);

s.push(5);

s.push(1);

cout << "The stack is : ";

printStack(s);

cout << "\ns.size() : " << s.size();

cout << "\ns.top() : " << s.top();

cout << "\ns.pop() : ";

s.pop();

printStack(s);

return 0;

}

**Queue**

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

#include<iostream>

#include<queue>

using namespace std;

int main(){

queue<int> q;

q.push(10);

q.push(20);

q.push(30);

while(!q.empty()){

int x=q.front();

cout<<x<<" ";

q.pop();

}

return 0;

}

**Map and Multimap**

* Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.
* Multimap is similar to map with an addition that multiple elements can have same keys.

#include <iostream>

#include <iterator>

#include <map>

using namespace std;

int main()

{

map<string, int> studentmap;

// insert elements in random order

studentmap.insert(pair<string, int>("Richard", 80));

studentmap.insert(pair<string, int>("Aaron", 90));

studentmap.insert(make\_pair("Florence", 100));

// studentmap.insert({key, element})

// printing map

map<string, int>::iterator itr;

for (itr = studentmap.begin(); itr != studentmap.end(); ++itr) {

cout <<"Name: " << itr->first

<< " Id: "<< itr->second <<"\n";

}

return 0;

}

**Total unique words count in sentence**

#include<iostream>

#include<vector>

#include<iterator>

#include<map>

#include<bits/stdc++.h>

using namespace std;

int main(){

//1. read string from user.

string str;

getline(cin,str);

//2. split the word from sentence

// and push them into vector.

istringstream ss(str);

vector<string> w;

string word;

while (ss >> word)

{

w.push\_back(word);

}

//3. Read word one by one from vector until empty

//and check the word in the map<string,int>

// a. if the word found in the map, increment value part of the map

// b. else insert the new entry to the map.

map<string, int> wordMap;

map<string, int> ::iterator it;

while(!w.empty()){

string b = w.back();

w.pop\_back();

it = wordMap.find(b);

if(it!=wordMap.end()){

int v = it->second;

it->second=v+1;

}

else{

wordMap.insert(pair<string,int>(b, 1));

}

}

//4. print the map

for(map<string, int> ::iterator itr = wordMap.begin();

itr!=wordMap.end(); ++itr){

cout<<itr->first<<" "<<itr->second<<endl;

}

//5. Total unique words in the map.

cout<<"Total unique words: "<<wordMap.size()<<endl;

return 0;

}

<https://www.cplusplus.com/reference/stl/>

<https://www.cplusplus.com/reference/algorithm/>

|  |
| --- |
| **ASSIGNMENT** |

1. Write a point class with x and y coordinates; declare 3 point objects and push them to the vector and print the vector.
2. Write a point class with x and y coordinates; declare 3 point objects and push them into the list and print the list.
3. Declare a map <int,Student> to store serial number and student object. Student class contain id, name and cgpa. Print all map data.
4. Write a Employee class with id, name and salary; declare 3 dynamic objects (pointer and new keywords) and push them to the vector and print the vector.