Functions

Functional Pointers

Keywords in C++

Cstring vs String in C++

Constructor, Initialization, Copy Constructor, Assignment

Pointer and Reference

Arrays

Vector, List and Deque

Priority Queue

Map and Unordered Map

Frequently Used Functions

C++ 11

1. Functions

1.1 Function Parameters

a. Passed by Reference

1) is needed when we need to modify the argument value
2) is needed when we pass a big object and copy is not worthy
3) is needed when there is no way to do the copy.

b. Default Function Parameters

1) default function parameters are set by giving the default value of the function parameter in the function definition
2) if one parameter has default parameter, then all other following parameter has to have default parameter.

1.2 Function Return

1) We can not return reference or pointer of the local variable
2) Return by reference
   Operator return by reference allows chaining, ex a=b=c.
3) Return by const reference
   Operator return by constant reference prohibits chaining.

1.3 Inline Function

1) inline function call is not a real function call, the compiler expands the function body in the call place
2) inline function has to be defined in the header file
3) for short function, inline function call saves more time, however it is not true for large and complex functions

1.4 Function Overloading vs. Overriding

a. Overloading: adding a different definition of the function with the same name but different parameters. It can happen within the same class
b. Overriding: adding a different definition of the function with same name and same parameters. It is used in inheritance. A function we would like to override is virtual in the base class, and its overriding definition is defined in the derived class. Static function can not be virtual because virtual functions have to be called by object pointer but static function does not belong to any objects.

2. Functional Pointers

a. Def

```
void fun(int a)
{
   // main body
}
 void (*fun_ptr)(int) = &fun;
```

like a normal function, it needs to have return type, parameter list, the only difference the function name is replaced by a pointer definition name

b. Usage and fact

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code
2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers
3) A function's name can also be used to get functions' address

```
void (*fun_ptr)(int) = fun
```

4) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

5) Can be used to create array of functions

Example:

```
void (*fun_ptr)(int) = fun
void add(int a, int b)
{
}
void subtract(int a, int b)
{
}
void multiply(int a, int b)
{
}
void (*fun_ptr_arr[])(int, int) = {add, subtract,
multiply};
```

3. Keyword in c++

3.1 static

a. Static variables in a function

Variable is allocated for the life time of the program and maintains its value between invocations of the function.

b. Static member variables in a class

Static variables of a class belong to the class and are shared by all the object instances.

c. Static objects inside a local scope

Static objects are allocated till the end of the program

d. Static member function

Static function belongs to the class itself and it can only access static member variables. Use ClassName::staticFunctionName to call static member function. Static member function can not be virtual.

e. Static global variable

Static global variable can be accessed only within the file

3.2 Const

1) const pointer vs pointer pointing to const data
2) const pointer is equivalent to const iterartor
3) const member function
   It can not change the member value, const key word is put after the function name

Const object can only call const member function

3.3 Explicit

See constructor section

3.4 Volatile

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.
Example:
Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

3.5 Extern

a. Use it for non-const global variables

Extern is used for global variables for multiple source files. When we need to use global variables across multiple source files, we use

1) extern in front to declare a variable in the header, say a.h,
2) Define the variable as a global variable in one source file only once in a.cpp
3) All other source files(b.cpp, c.cpp) using this global variables have to include the header

3.6 Static vs Extern and Internal Linkage vs External Linkage

a. External Linkage
   External linkage means the identifier is visible to linker from other files, in other words it is a global variable shared between different translation units.
   extern int x,
b. Internal Linkage
   Internal linkage means, each translation unit has its own copy of the identifier.
   static int x,
c. Default linkage
   Non-const global variables have external linkage by default
   Const global variables have internal linkage by default
   Functions have external linkage by default

4. Char vs String

a. char* c

1) It is basically a pointer to the (const)string literal. In C++, it is a good practice to declare as const char* c as the string literal is a const.
2) When initialize with a string literal, a null terminator is automatically added so it becomes a c string.
   Example:
   const char *c = “aaaa”; // null terminator is added.

Pros:

1) Only one pointer is required to refer to whole string. That shows this is memory efficient.
2) No need to declare the size of string beforehand.

Cons:

We cannot modify the string at later stage in program.

b. char[]

This is an array of char, when initialized by a string literal, a null terminator is added as well.

Pros:

We can modify the string at later stage in program.

Cons:

1) This is statically allocated sized array which consumes space in the stack.
2) We need to take the large size of array if we want to concatenate or manipulate with other strings since the size of string is fixed.

c. String in c++

String type in c++ is always preferred. Use .c_str() to convert string to cstring.

5. Constructor, Initialization, Copy Constructor, Assignment

5.1 Constructor

a. Define:

1) Function name is the same as class name
2) Function name is followed by parameters in ()
3) Then it is the initialization list
   Start with: followed by variable name, parenthesis in which are the initialization values
4) Body {}, usually empty.

Example:

```
ClassName::ClassName(parameterlist): variableA(valueA),
variableB(valueB) {}
```

b. Initialization list

We have to use the initialization list in the following cases

1) members whose class does not have default constructor.

2) const member variable.

3) reference member variable

c. Default constructor

The complier provides a default constructor if we do not define one

But if we define a constructor, the compiler does not provide one anymore

d. Explicit key word in constructor

When an explicit key word is put in front of the constructor function, the compiler does not do any implicit conversion for the arguments using constructor

Example: if A has constructor that has B as the parameter and there is a function func(A a),with no explicit keyword in the constructor, we can write

func(new B), the compiler calls the constructor of A and convert B to A. With explicit keyword, this does not happen.

5.2 Define and Initialization

a. Direct initialization

A a(xxx): direct initialization, the constructor function is called

b. Copy initialization

A a = a1;

1) Call the constructor to create a temporary object
2) Call the copy constructor to copy this temporary object to the new object

c. Copy Initialization

A a(a1);

d. Special variable define and initialization

Non const global variable

In a.h

extern int myVariable

In a.cpp

int myVariable = 5;

Const global variable

In a.h

const int CONSTANT = 255;

comment: this uses internal linkage which means each compilation unit gets its own copy of this variable and has its own address. It should not matter in most cases. But if we really need one copy, we should do similar to non const global variable with const keyword

In a.h

extern int myVariable

In a.cpp

const int myVariable = 5;

Static variable

In the source file

Type ClassName:: variable name = value.

5.3 Copy constructor

a. Def

1) Same name as class name

2) No return value
3) Take a const reference as parameter. The reason why it takes a reference is it needs passing by reference. We can not do pass by value because it ends up a dead loop. Passing by value needs to create a copy which calls the copy constructor, and the copy constructor pass by value which requires copy constructor again.

b. Usage

1) Copy initialization(using =)
2) When passing an object as a function parameter, copy constructor is called to create a temporary argument from the parameter object
3) When an object of the class is returned by value.
4) Initialize container elements
   Vector<string> vec(5)
   First copy constructor is called to create a temp object, then use copy constructor to copy the temp object to vec
5) Curly bracket initialization of array

c. When to define

1) C++ compiler always provides us synthesized copy constructor, however

We need to define our own copy constructor only if an object has pointers. Because when the member variables have pointers pointing to other objects, the compiler does not do allocation of new objects automatically for us. We need to allocate new objects for us manually, this is also called as deep copy.

2) The compiler provides default copy constructor if we define neither constructor nor copy constructor.

5.4 Assignment Operator

a. Assignment operator parameters and return type

Assignment operator takes const reference as a parameters because

1) We do not need to modify the rvalue.
2) It does not need to copy the parameter to a temporary object.

Assignment operator returns a reference

The following example shows returning a reference comparing returning a value.

Example:

(a=b)=c;

This is equivalent to (a.operator=(b)).operator=(c))

When returning by value, a.operator=(b) returns a temporary objects, then we assign to value c to a temporary object, so the result of a would be b

When returning by reference, a.operator=(b) returns a itself, then we assign c to a, so the result of a would be c.

b. Copy constructor vs assignment operator

1) Copy constructor is called when a new object is created from an existing object, as a copy of the existing object (see this G-Fact). And assignment operator is called when an already initialized object is assigned a new value from another existing object.

2) example

```
Test t1, t2;
t2 = t1;    // assignment operator is called
Test t3 = t1;  // copy constructor is called
```

c. When to define

Same answer as copy constructor

6. Pointer and Reference

a. Reference vs Pointer

References are less powerful than pointers

1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.

2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

3) A reference must be initialized when declared. There is no such restriction with pointers

References are safer and easier to use:

1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location.

2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

b. Use Criteria

Use pointer when we have to, otherwise use reference as much as possible

1) Define the class member variables that shares the data with other class. Like class A has data B, class B has data A.
2) If we want to use polymorphism. Like bind a pointer to a derived class in runtime.
3) Pointers if pointer arithmetic or passing NULL-pointer is needed.

c. const reference vs non-const reference

1) A non-const reference must be bound to lvalue

(i.e. its address should be available).

Example:

```
int &p = 10 // fails, as 10 does not have an address
```

3) A const reference could be bound to rvalue

Example

const int &p = 255 and for this case, a temporary int will be created and initialized from `255`. The temporary `int`'s lifetime will be the same as the const reference.

d. Smart Pointer

When a class has pointer member variables, lots of issue could happen during copy, assignment, destructor. Smart Pointer is a way of defining a class that contains a raw pointer, but with uses' own definition of copy constructor, assignment operator and destructor function.

unique_ptr

unique_ptr guarantees there can only be at most one unique_ptr at any one resource and when that unique_ptr is destroyed, the resource is automatically claimed. Any attempt to make a copy of unique_ptr will cause a compile time error.

But, unique_ptr can be moved using the new move semantics i.e. using std::move() function to transfer ownership of the contained pointer to another unique_ptr.

shared_ptr

A shared_ptr is a container for raw pointers. It is **a** reference counting ownership model i.e. it maintains the reference count of its contained pointer in cooperation with all copies of the shared_ptr. So, the counter is incremented each time a new pointer points to the resource and decremented when destructor of object is called.

Examples:

Suppose we have class

```
class A
{
    cont int i = 11;
}
```

If we have two raw pointers

```
A* a1 = new A();
```

A* a2(a1);

We allocated a new object A, and have two raw pointers pointing to that object A at the same time. Now if we delete a1 by calling

delete a1;

The object of class A allocated by pointer a1 is deleted, however, we have pointer a2 pointing a deleted object of class A, which leads to an undefined behavior.

That comes the advantage of using shared_ptr

Define a shared ptr p1 pointing object A

```
shared_ptr<A> p1(new A());
```

```
Define another shared ptr p2 pointing the same
object as p1
```

```
shared_ptr<A> p2(p1);
```

```
When we call reset of p1, the p1 points to NULL, but
since p2  points to the object of class A, the object is
not deleted.
p1.reset();
The object of class A is only deleted until no pointers
refer to it. So when we reset p2, then the object is
deleted and memory space is freed up.
p2.reset();
```

7. Array

a. Array declaration

1) int array[5]

The array of ints is created on the stack as an automatic array

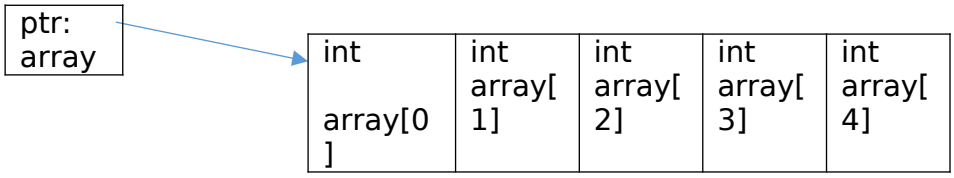
int array[0]	int array[1]	int array[2]	int array[3]	int array[4]

Size of array = 5* size of (int) = 20

2) int* array;

array = new int[5]

The pointer variable array is created on the stack; the objects are created on the heap as a single dynamic array.

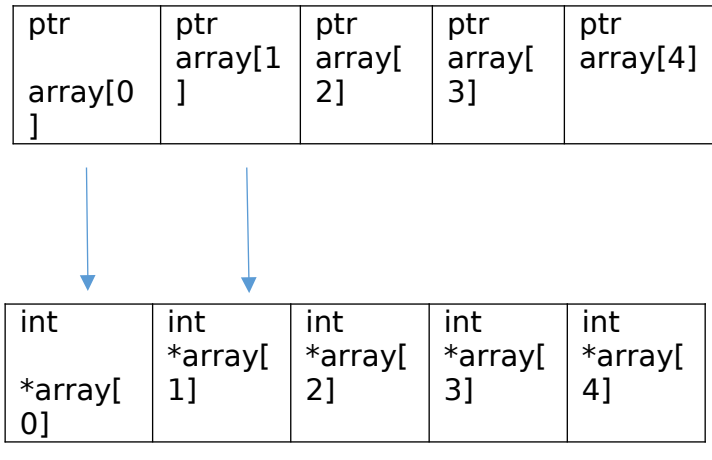


Size of array is 8, which is the size of pointer.

3) int* array[5];

```
for(int i=0; i<5; i++)
    array[i] = new int(0);
```

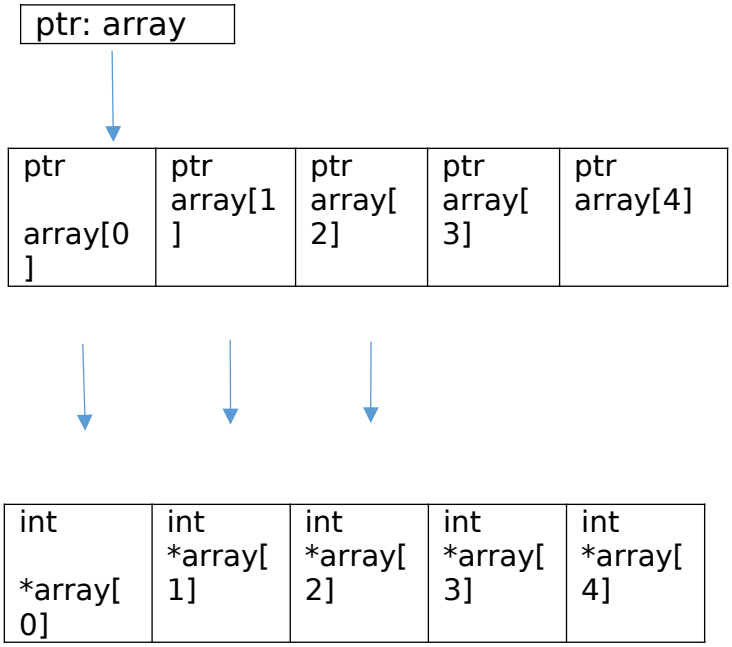
The array of pointers is created on the stack as an automatic array; the individual objects are each created on the heap as dynamic objects



Size of array is 5* size of pointer = 40

Size of array[0] is the size of pointer, which is 8

4) int** array;
   array = new int*[5];
   for(int i=0; i<5; i++)
     int[i] = new int(0);
   The pointer variable emp is created on the stack as an automatic variable; the array of pointers is created on the heap as a dynamic array; the individual objects are each created on the heap as dynamic objects



Size of array is 8, which is the size of pointer.

b. Dynamic allocating 2d Array

Allocating array Example

```
int **array = new int*[3];
for(int i =0; i < 3; i++)
{
    array[i] = new int[3];
     for(int j=0; j<3; j++)
     {
           array[i][j] = 1;
     }
}
```

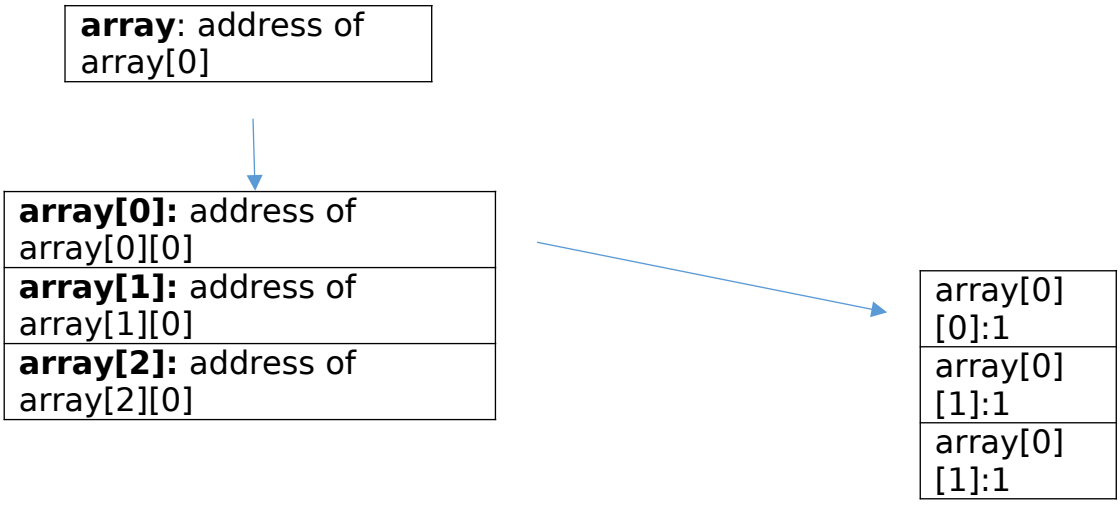
Explanation:

int **array = new int*[3];

1) Creating a pointer array, which is pointer to pointer,
2) Creating an array of pointers: array[0], array[1], array[2]
3) The value of array is the address of array[0]
4) The value of array[0] is null;

array[i] = new int[3];

5) Creating an array of pointers: array[i][0], array[i][1], array[i][2]
   The value of array[0] is the address of array[0][0];



Variable name	Address	Value
array	0x0858	0x4b60
array[0]	0x4b60	0x4b80
array[0][0]	0x4b80	1
array[0][1]	0x4b84	1
array[1]	0x4b68	0x4b90
array[1][0]	0x4b90	1

6) array[1] = array[0] + 1x size of pointer = array[0]+8;
   array[0][1] = array[0][0] + 1x size of int = array[0]+4

c. Deallocating array

Example:

```
for(int I =0; i<2; i++)
{
    delete[] array[i];
      array[i] = NULL;
}
delete[] array;
array = NULL;
```

Explanation:

Delete free the space the pointer points to. However, you may still access it, it may get the previous object before deleting, or it may be something undefined, and furthermore, it could be segmentation fault. So in order to make sure one is not able to access the deleting object, we set the pointer to NULL.

d. Relationship between array and pointer

1) Array name gives address of first element of array.
   Example:
   ```
   int array[5];
   int *ptr = arr;
   ```
2) Array members are accessed using pointer arithmetic.

```
int array[] = {10, 20, 30, 40, 50};
int *ptr = arr;
```

Those four following syntax are the same and all can access the 2nd element.

```
arr[2]
*(arr + 2)
ptr[2];
*(ptr + 2)
```

3) Array parameters are always passed as pointers, even when we use square brackets.

8. Vector vs List vs Deque

8.1 Vector

a. Initialization

1) One by one by push_back
2) Specify size and values

```
vector<int> vect(n, 10);
```

3) From another vector

```
vector<int> vect1{ 10, 20, 30 };
vector<int> vect2(vect1.begin(), vect.end());
```

4) Like an array

```
vector<int> vect{ 10, 20, 30 };
```

b. Operation

1) begin() – Returns an iterator pointing to the first element in the vector
2) end() -Returns an iterator pointing to the theoretical element that follows the last element in the vector
3) size() – Returns the number of elements in the vector.
4) capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
5) resize() – Resizes the container so that it contains ‘g’ elements.
6) empty() – Returns whether the container is empty.
7) reference operator [g] – Returns a reference to the element at position ‘g’ in the vector

8) at(g) – Returns a reference to the element at position ‘g’ in the vector
9) front() – Returns a reference to the first element in the vector
10) back() – Returns a reference to the last element in the vector
11) assign() – It assigns new value to the vector elements by replacing old ones
12) push_back() – It push the elements into a vector from the back
13) pop_back() – It is used to pop or remove elements from a vector from the back.
14) insert() – It inserts new elements before the element at the specified position
15) erase() – It is used to remove elements from a container from the specified position or range.
16) swap() – It is used to swap the contents of one vector with another vector of same type and size.
17) clear() – It is used to remove all the elements of the vector container

8.2 List

a. Initialization

1) One by one by push_back
2) Specify size and values

```
list<int> myList(n, 10);
```

3) From another list

```
list<int> myList2(myList.begin(), myList.end());
```

4) Like an array

```
int myints[] = {16,2,77,29};
std::list<int> fifth (myints, myints + sizeof(myints) /
sizeof(int) );
```

b. Operation

1) begin() – Returns an iterator pointing to the first element in the list
2) end() -Returns an iterator pointing to the theoretical element that follows the last element in the list
3) size() – Returns the number of elements in the list.
4) front() – Returns a reference to the first element in the list.

5) back() – Returns a reference to the last element in the list.
6) push_front(g) – Adds a new element 'g' at the beginning of the list
7) push_back(g) – Adds a new element 'g' at the end of the list
8) pop_front() – Removes the first element of the list, and reduces size of the list by 1
9) pop_back() – Removes the last element of the list, and reduces size of the list by 1
10) empty() – Returns whether the list is empty(1) or not(0)
11) insert() – Inserts new elements in the list before the element at a specified position.
12) erase() – Removes a single element or a range of elements from the list
iterator erase (iterator position);
iterator erase (iterator first, iterator last);
13) assign() – Assigns new elements to list by replacing current elements and resizes the list
14) remove() – Removes all the elements from the list, which are equal to given element
15) reverse() – Reverses the list

8.3 Deque

a. Initialization

1) One by one by push_back
2) Specify size and values

```
std::deque<int> second (4,100);        // four ints with value 100
std::deque<int> third (second.begin(),second.end());  // iterating through second
```

3) From another deque

```
std::deque<int> third (second.begin(),second.end());  // iterating through second
```

4) Copy
   std::deque<int> fourth (third);

b. Operation

begin() – Returns an iterator pointing to the first element in the deque
end() -Returns an iterator pointing to the theoretical element that follows the last element in the deque
size() – Returns the number of elements in the deque.
resize() – Resizes the container so that it contains 'g' elements.
empty() – Returns whether the container is empty.
reference operator [g] – Returns a reference to the element at position 'g' in the vector
at(g) – Returns a reference to the element at position 'g' in the deque
front() – Returns a reference to the first element in the deque
back() – Returns a reference to the last element in the deque
assign() – It assigns new value to the deque elements by replacing old ones
push_back() – It push the elements into a deque from the back
pop_back() – It is used to pop or remove elements from a deque from the back.
insert() – It inserts new elements before the element at the specified position
erase() – It is used to remove elements from a container from the specified position or range.
swap() – It is used to swap the contents of one vector with another deque of same type and size.
clear() – It is used to remove all the elements of the deque container

8.4 Comparison

a. How is data store in terms of memory allocation?

1) Vector: Contiguous location in memory, can be viewed as resizable array.

Element [0]	Element [1]	Element [2]	Element [3]	Element [4]

2) List: Discontinuous location in memory, can be viewed as double linked list

3) Deque: is a linked list of vectors, in other words, is discrete blocks of continuous locations in memory

Element [0]	Element [1]	Element [2]

→

Element [3]	Element [4]	Element [5]

b.

b. Criteria to choose which container

1) Fast random access: Vector.
Since it guarantees the data are stored in contiguous location in memory.
Deque can do random access but since the data are not stored in completely contiguous location in memory, it is not as fast as vector.

2) Insert or delete element in the middle: List
Since inserting an element in the list need only access and move the pointers.

3) Insert or delete element in the beginning and end: Deque

When we insert an element in end it stores that in allocated memory block untill it gets filled and when this memory block gets filled with elements then it allocates a new memory block and links it with the end of previous memory block. Now further inserted elements in the back are stored in this new memory block.

When we insert an element in front it allocates a new memory block and links it with the front of previous memory block. Now further inserted elements in the front are stored in this new memory block unless it gets filled.

c. Function Summary

	vector	list	deque
begin(),end()	Y	Y	Y
size(),resize(),clear()	Y	Y	Y
Capacity()	Y	N	N
front(), back()	Y	Y	Y
[], at	Y	N	Y
push_back(),pop_back()	Y	Y	Y
push_front(), pop_front()	N	Y	Y
insert(),erase(),swap(),	Y	Y	Y

9. Stack, Queue, Priority Queue

9.1 Stack Queue

a. Implementation

Deque

9.2 Priority Queue

a. Implementation: A heap using vectors. The elements of the heap is stored in the vector by level order traversal from top to bottom.

b. Operation

empty()
size()
top()
push()
pop()

c. Example with comparison class

```
mycomparison(const bool& revparam=false)
    {reverse=revparam;}
  bool operator() (const int& lhs, const int&rhs) const
  {
    if (reverse) return (lhs>rhs);
    else return (lhs<rhs);
  }
};

  // using mycomparison:
  typedef
std::priority_queue<int,std::vector<int>,mycomparison>
mypq_type;
```

9.3 Function Summary

	stack	queue	Priority queue
empty(), size()	Y	Y	Y
pop(), push()	Y	Y	Y
front(), back()	N	Y	N
top()	Y	N	Y

10. Map vs Unordered_map

In c++ map is implemented as a self-balanced tree, while unordered_map in c++ 11 is implemented as a has table

Difference

	map	unordered_map
Ordering	increasing order (by default) Need to define < operator	no ordering
Implementation	Self balancing BST like Red-Black Tree	Hash Table
Search time	log(n)	O(1) -> Average O(n) -> Worst Case
Insertion time	log(n) + Rebalance	Same as search
Deletion time	log(n) + Rebalance	Same as search

11. Frequently Used Functions

11.1 String(Different base) to Number

```
int convertToBaseTen(string s, const in base)
{
   int result = 0;
    for(string::size_type j=0; j<s.size(); j++)
    {
          int digit = s[j] –‘0’;
          result = result*base + digit;
    }
   return result;
}
```

11.2 Number to String(Different Base)

```
string convertFromBaseTen(int number, int base, int length)
{
   stringstream ss;
    for(string::size_type j=0; j<length; j++)
    {
          ss<< number/(int) (pow(base, length – j -1));
          number = number % (int) (pow(base, length –j -1));
    }
    Return ss.str();
}
```

12. C++ 11

a. Brace-Initialization

Examples

```
int a{0};
string s{"hello"};
string s2{s}; //copy construction
vector <string> vs{"alpha", "beta", "gamma"};
map<string, string> stars
{ {"Superman", "+1 (212) 545-7890"},
{"Batman", "+1 (212) 545-0987"}};
double *pd= new double [3] {0.5, 1.2, 12.99};
class C
{
        int x[4];
        public:
        C(): x{0,1,2,3} {}
};
```

b. auto type deduction

With the auto type deduction feature enabled, you no longer need to specify a type while declaring a variable. Instead, the compiler deduces the type of an auto variable from the type of its initializer expression. For example:

```
auto i = 1.1;    // i : double
double* pd;
auto x = pd;     // x : double*
auto* y = pd;    // y : double*
 int g();
auto x = g();          // x : int
const auto& y = g();   // y : const int&
```

c. rvalue reference and move constructor

d.