**PPL Midterm –II Solution for set-A**

**Q.1 a.** 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. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.Removing the last element takes only constant time, because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

**Iterators**  
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 last element in the vector  
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element  
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

**Capacity**  
1. size() – Returns the number of elements in the vector  
2. max\_size() – Returns the maximum number of elements that the vector can hold  
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements  
4. resize(size\_type g) – Resizes the container so that it contains ‘g’ elements  
5. empty() – Returns whether the container is empty

**1.The array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

**Declaring Arrays**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

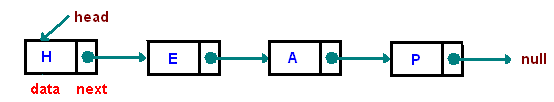
## Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

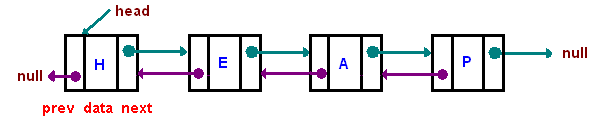
**Q.1B.** A linked list is a linear data structure where each element is a separate object.



### Types of Linked Lists

A **singly linked list** is described above

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



## Iterator

The whole idea of the iterator is to provide an access to a private aggregated data and at the same moment hiding the underlying representation. An iterator is Java is an object, and therefore it's implementation requires creating a class that implements the *Iterator* interface. Usually such class is implemented as a private inner class. The *Iterator* interface contains the following methods:

* AnyType next() - returns the next element in the container
* boolean hasNext() - checks if there is a next element
* void remove() - (optional operation).removes the element returned by next()

Sets:- a **set** is an [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type) that can store certain values, without any particular [order](https://en.wikipedia.org/wiki/Sequence), and no repeated values. It is a computer implementation of the [mathematical](https://en.wikipedia.org/wiki/Mathematics) concept of a [finite set](https://en.wikipedia.org/wiki/Finite_set). Unlike most other [collection](https://en.wikipedia.org/wiki/Collection_(abstract_data_type)) types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and deletion of elements from the set.

### Static sets

Typical operations that may be provided by a static set structure *S* are:

* is\_element\_of(*x*,*S*): checks whether the value *x* is in the set *S*.
* is\_empty(*S*): checks whether the set *S* is empty.
* size(*S*) or [cardinality](https://en.wikipedia.org/wiki/Cardinality)(*S*): returns the number of elements in *S*.
* [iterate](https://en.wikipedia.org/wiki/Iterator)(*S*): returns a function that returns one more value of *S* at each call, in some arbitrary order.
* enumerate(*S*): returns a list containing the elements of *S* in some arbitrary order.
* build(*x*1,*x*2,…,*xn*,): creates a set structure with values *x*1,*x*2,…,*xn*.
* create\_from(*collection*): creates a new set structure containing all the elements of the given [collection](https://en.wikipedia.org/wiki/Collection_(computing)) or all the elements returned by the given [iterator](https://en.wikipedia.org/wiki/Iterator).

### Dynamic sets

Dynamic set structures typically add:

* create(): creates a new, initially empty set structure.
  + create\_with\_capacity(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.
* add(*S*,*x*): adds the element *x* to *S*, if it is not present already.
* remove(*S*, *x*): removes the element *x* from *S*, if it is present.
* capacity(*S*): returns the maximum number of values that *S* can hold.

**Q.2.a The activation records:-**

A **block** is a region of program text, delimited by begin and end markers.

The **scope** of a variable is the region of text in which the binding of that variable is visible.

The **lifetime** of a variable is the amount of time that the variable occupies memory during program execution.

**Activation records** keep track of values as a program executes. More specificly, an activation record has a set of names that are bound to values. We link activation records together in two ways:

* with a control link
* with an access link

A **control link** from record A points to the previous record on the stack. The chain of control links traces the dynamic execution of the program.

An **access link** from record A points to the record of the closest enclosing block in the program. The chain of access links traces the static structure (think: scopes) of the program.

# Creating Activation Records

This section contains a few simple guidelines for working with activation records. Follow these guidelines as you work on examples, and you should begin to understand this topic pretty quickly.

Essentially, three actions that occur in a program can cause us to create an activation record:

* Function definition
* Variable definition
* Function application (call)

Every time we create an activation record, we must set the control and access links, and populate the bindings with values. Let's see how to do that for each of the above actions.

Function definition:

Control Link: The previous record on the stack   
Access Link: The previous record on the stack   
Bindings:

* Function name -> Function closure

Variable definition:

Control Link: The previous record on the stack   
Access Link: The previous record on the stack   
Bindings:

* Variable name -> Function closure

Function application:

Control Link: The record that called the function   
Access Link: The record pointed to by the function's closure   
Bindings:

* Formal parameter names -> Actual parameter values
* Local variable names -> Local variable values *(e.g., from lets)*
* (1) Line number 1
* [1] Activation record 1
* [A] Closure A
* {expr} A bit of code
* The Stack
* Used to hold local variables.
* Large array which typically grows downwards in memory toward lower addresses,
* shrinks upwards.
* Push(r1):
* stack\_pointer--;
* M[stack\_pointer] = r1;
* r1 = Pop():
* r1 = M[stack\_pointer];
* stack\_pointer++;
* Previous activation records need to be accessed, so push/pop not sufficient.
* –
* Treat stack as array with index off of
* stack
* pointer
* .
* –
* Push and pop entire activation records.
* Computer Science 320
* Prof. David Walker
* -2-
* Example
* Consider:
* let
* function g(x:int) =
* let
* vary:=10
* in
* x+y
* end
* function h(y:int):int =
* y + g(y)
* in
* h(4)
* end
* Example
* Step 1:
* h(4)
* called
* Chunk of memory allocated on the stack in order to hold local variables of h. The
* activation record (or stack frame) of h is pushed onto the stack.
* Stack
* Frame
* for h
* y=4
* Step 2:
* g(4)
* called
* Activation record for g allocated (pushed) on stack.
* Stack
* Frame
* for h
* y=4
* Stack
* Frame
* for g
* x=4
* y=10
* Computer Science 320
* Prof. David Walker
* -4-
* Example
* Step 3:
* g(4)
* returns with value 14
* Activation record for g deallocated (popped) from stack.
* Stack
* Frame
* for h rv = 14
* y=4
* Step 4:
* h(4)
* returns with value 18
* Activation record for h deallocated (popped) from stack. Stack now empty.

**Q.2Actual parameters and formal parameters**

There are two other categories that you should know about that are also referred to as "parameters". They are called "parameters" because they define information that is passed to a function.

* Actual parameters are parameters as they appear in function calls.
* Formal parameters are parameters as they appear in function declarations.

A parameter cannot be both a formal and an actual parameter, but both formal parameters and actual parameters can be either value parameters or variable parameters.

Let's look at calculate\_bill again:

1. #include <stdio.h>
3. int main (void);
4. int calculate\_bill (int, int, int);
6. int main()
7. {
8. int bill;
9. int fred = 25;
10. int frank = 32;
11. int franny = 27;
13. bill = calculate\_bill (fred, frank, franny);
14. printf("The total bill comes to $%d.00.\n", bill);
16. exit (0);
17. }
19. int calculate\_bill (int diner1, int diner2, int diner3)
20. {
21. int total;
23. total = diner1 + diner2 + diner3;
24. return total;
25. }

In the function main in the example above, fred, frank, and franny are all actual parameters when used to call calculate\_bill. On the other hand, the corresponding variables in calculate\_bill (namely diner1, diner2 and diner3, respectively) are all formal parameters because they appear in a function definition.

Although formal parameters are always variables (which does not mean that they are always variable parameters), actual parameters do not have to be variables. You can use numbers, expressions, or even function calls as actual parameters. Here are some examples of valid actual parameters in the function call to calculate\_bill:

1. bill = calculate\_bill (25, 32, 27);
3. bill = calculate\_bill (50+60, 25\*2, 100-75);
5. bill = calculate\_bill (fred, franny, (int) sqrt(25));

Q.2 Type checking and type conversion

A **programming language** is called a **language** with static **type checking** or strongly typed **language**, if the **type** of each expression can be determined at compile-time, thereby guaranteeing that the **type**-related errors cannot occur in object program. Pascal is an example of a strongly typed **language**

Type checking means checking that each operation should receive proper number of arguments and of proper data type.

Like

A=B\*j+d;

\* and - are basically int and float data types based operations and if any variable in this A=B\*j+d;Is of other than int and float then compiler will generate type error.

Two ways of Type Checking:

1) Dynamic Type Checking:

• It is done at runtime.

• It uses concept of type tag which is stored in each data objects that indicates the data type of the object.

Example:

**An integer data object contains its'type' and 'values' attribute.**

**so Operation only be performed after type checking sequence in which type tag of each argument is checked. If the types are not correct then error will be generated.**

• Perl and Prolog follow basically dynamically type checking because data type of variables A+B in this case may be changed during program execution.

• so that type checking must be done at runtime.

## Advantages of Dynamic Type:

• It is much flexible in designing programs or we ca say that the flexibility in program design.

• In this no declarations are required.

• In this type may be changed during execution.

• In this programmerare free from most concern about data type.

## Disadvantage of Dynamic Type:

• 1) difficult to debug: We need to check program execution paths for testing and in dynamic type checking, program execution path for an operation is never checked.

• 2) extra storage: Dynamic type checking need extra storage to keep type information during execution.

• 3) Seldom hardware support : As hardware seldom support the dynamic type checking so we have to implement in software which reduces execution speed.

# Type checking: Static Type Checking:

Static Type Checking is done at complete time.

**Information needed at compile time is provided**- by declaration- by language structures.

The information required includes:

1) for each operation: The number, order, and data type, of its arguments.

2) For each variables: Name and data type of data object.

Example-

A+B

**in this type of A and B variables must not be changed.**

3) for each constant: Name and data type and value

const int x=28;

const float x=2.087;

**In this data type, the value and name is specified and in further if checked value assigned should match its data type.**

## Advantages of Static Type Checking:

1) compiler saves information:- if that type of data is according to the operation then compiler saves that information for checking later operations which further no need of compilation.

2) checked execution paths: As static type checking includes all operations that appear in any program statement, all possible execution paths are checked, and further testing for type error is not needed. So no type tag on data objects at run-time are not required, and no dynamic checking is needed.

## Disadvantages of Static Type Checking

: It affects many aspects of languages

1) declarations

2) data control structures

3) provision of compiling separately some subprograms.

# Strong Typing:

**If we change detect all types of errors statically in a program, we can say that language is' strongly typed'.**

It provides a level of security to our program.

Example

f:s-> R

**In this function f mail signature s generate output R and R is not outside the range of R data type.**

IF every operation is type safe then automatically language is strongly typed.

Example of strongly typed languages are:

C,Java, C++, RubyRail, smalltalk, python.

Type infer:- In this, like in ML, the language implementation will infer any missing type information from other declared type.

Example:

funarea(length:int, width:int):int= length \*width;

This is the standard declaration which tells length and width of int data type and its return type is int and function name area. But leaving any two of these declarations still leaves the function will only one interpretation. Knowing that \* can multiply together either two reals or two integers. ML interprets the following as equivalent to the previous example.

Funarea(length,width)int= length\*weight;

Funarea(length:int,width)= length\*weight;

Funarea(length,width:int)= length\*weight;

However:

Funarea(length,width)= length\*weight;

**Is invalid as it is now ambiguous as to that type of arguments. They could all be int or they could be real.**

Q.3 A **recursive subprogram** is one that calls itself. Think of a **recursive** call as a call to some other **subprogram** that does the same task as your **subprogram**. Each **recursive** call creates a new instance of any objects declared in the **subprogram**, including parameters, variables, cursors, and exceptions.

A separate activation record (frame) is created for each invocation of the **recursive subprogram**. ... For avoiding the performance penalty and the run-time stack overflow due to function calls, Scheme—a functional **programming language**—dictates that tail-**recursive** calls be transformed to goto statements by the compiler.

void fun1(float r) {

int s, t;

...

fun2(s);

...

}

void fun2(int x) {

int y;

...

fun3(y);

...

}

void fun3(int q) {

...

}

void main() {

float p;

...

fun1(p);

...

}

**main → fun1 →fun2 → fun3**

he activation record used in the previous example supports recursion

int factorial (int n) {

if (n <= 1) return 1;

else return (n \* factorial(n - 1));

}

void main() {

int value;

value = factorial(3);

}

Q.3 **Block structure**

A **block** or code **block** is a lexical **structure** of source code which is grouped together. ... A programming language that permits the creation of **blocks**, including **blocks** nested within other **blocks**, is called a **block**-structured programming language. Process Control Block (PCB, also called Task Controlling Block, Entry of the Process Table, **Task Struct**, or Switchframe) is a data structure in the **operating system kernel** containing the information needed to manage the scheduling of a particular process.

if (. . . ) // -- outer if scope

{

int counter; // defines counter

if (. . .) // -- intermediate scope

{

int x; // defines x

int y; // defines y

if (. . .) // -- inner most scope

{

counter = x + y; // uses counter, x, and y

. . .

}

}

}

Local scope

Also known as function scope, are variables that are defined inside of a function. Local scope variables obviously include the variables defined in the function body (i.e., the outermost block of the function). But local scope also includes the variables defined in the function's argument list. Functions are covered in greater detail in chapter 6.

Class scope

These are the member variables specified in a class (introduced in chapter 9).

Global scope

These are variables that are not defined inside a class or in a function. Java, being a pure object-oriented language, does not permit global variables.

A *block* is one or more [statements](http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/definitions.html#statement) enclosed between braces: { and }. A block can appear at arbitrary locations within a program:

#include <iostream>

using namespace std;

int main()

{

cout << "Hello, World!" << endl;

{

cout << "This is a block" << endl;

}

return 0;

}

Although the example above is correct and it compiles, it's very uncommon to create a block without a purpose. However, recognizing the behavior of blocks will help us to understand some common programming errors when we see them later.

Blocks can be used to extend control statements by replacing a single statement with a block statement, also called a compound statement. Notice that the example in the first column consists of a single statement (there is only one terminating semicolon). By itself, cout << counter << endl; is also a statement, which seems to imply that complex statements can be built from simple statements. Specifically, control statements are created by adding a some kind of control to a simple statement (any legal statement: I/O, assignment, etc.). But what happens if we want the control to apply to more than just a single, simple statement? That is where a block statement is required.

|  |  |
| --- | --- |
| **"{" and "}" Optional** | **"{" and "}" Required** |
| if (. . .)  cout << counter << endl;  Or  if (. . .)  {  cout << counter << endl;  } | if (. . .)  {  cout << "Hello, World!" << endl;  cout << counter << endl;  } |

**Braces and control statements**. Braces are not required when a single statement is nested inside a control statement, but some programmers believe that that braces make the code more clear and use them even when they are optional. Braces are required when two or more sequential statements are nested inside a single control statement.

A block statement can appear wherever a single statement is legal. Furthermore, each block represents a distinct scope.

Q.4 finally keyword

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

## Syntax of Finally block

try {

//Statements that may cause an exception

}

catch {

//Handling exception

}

finally {

//Statements to be executed

}

class Example

{

public static void main(String args[]) {

try{

int num=121/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("Number should not be divided by zero");

}

/\* Finally block will always execute

\* even if there is no exception in try block

\*/

finally{

System.out.println("This is finally block");

}

System.out.println("Out of try-catch-finally");

}

}

**Output:**

Number should not be divided by zero

This is finally block

Out of try-catch-finally

## Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for [exception handling](https://beginnersbook.com/2013/04/java-exception-handling/), however if you place a finally block then it will always run after the execution of try block.

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.  
Lets see an example to see how finally works when return statement is present in try block:

**Q.4 Control statements** enable us to specify the flow of program control; ie, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

There are four types of control statements in C:

1. Decision making statements
2. Selection statements
3. Iteration statements
4. Jump statements

### How break statement works?

### Example #1: break statement

// Program to calculate the sum of maximum of 10 numbers

// Calculates sum until user enters positive number

# include <stdio.h>

int main()

{

int i;

double number, sum = 0.0;

for(i=1; i <= 10; ++i)

{

printf("Enter a n%d: ",i);

scanf("%lf",&number);

// If user enters negative number, loop is terminated

if(number < 0.0)

{

break;

}

sum += number; // sum = sum + number;

}

printf("Sum = %.2lf",sum);

return 0;

}

**Output**

Enter a n1: 2.4

Enter a n2: 4.5

Enter a n3: 3.4

Enter a n4: -3

Sum = 10.30

## continue Statement

The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

### Syntax of continue Statement

continue;

### How continue statement works?

### Example #2: continue statement

// Program to calculate sum of maximum of 10 numbers

// Negative numbers are skipped from calculation

# include <stdio.h>

int main()

{

int i;

double number, sum = 0.0;

for(i=1; i <= 10; ++i)

{

printf("Enter a n%d: ",i);

scanf("%lf",&number);

// If user enters negative number, loop is terminated

if(number < 0.0)

{

continue;

}

sum += number; // sum = sum + number;

}

printf("Sum = %.2lf",sum);

return 0;

}

**Output**

Enter a n1: 1.1

Enter a n2: 2.2

Enter a n3: 5.5

Enter a n4: 4.4

Enter a n5: -3.4

Enter a n6: -45.5

Enter a n7: 34.5

Enter a n8: -4.2

Enter a n9: -1000

Enter a n10: 12

Sum = 59.70