

3.2 STACKS

In our everyday life, we come across many examples of stacks, for example, a stack of books, a stack of dishes, or a stack of chairs. The data structure *stack* is very similar to these practical examples (Fig. 3.1).



Fig. 3.1 Sample real world stacks

Consider a stack of books on a table. We can easily put a new book on the top of the stack, and similarly, we can easily remove the topmost book as compared to the books lying in-between or at the bottom positions. In the same way, only the topmost element of a stack can be accessed while direct access of other intermediate positions is not feasible. Elements may be added to or removed from only one end, called the *top* of a stack.

The linear data structures such as arrays and linked lists allow users to insert or delete an element at any position in the list, that is, we can insert or delete an element at the beginning, at the end, or at any intermediate position.

A *stack* is defined as a restricted list where all insertions and deletions are made only at one end, the *top*. Each stack abstract data type (ADT) has a data member, commonly named as *top*, which points to the topmost element in the stack. There are two basic operations *push* and *pop* that can be performed on a stack; insertion of an element in the stack is called *push* and deletion of an element from the stack is called *pop*. In stacks, we cannot access data elements from any intermediate positions other than the top position.

Given a stack $S = (a_1, a_2, \dots, a_n)$. We say that a_1 is the bottommost element, a_n is on top of the stack, and the element a_{i+1} is said to be on the top of a_i , $1 < i \leq n$.

In Fig. 3.2, $S = (A, B, C)$, where A is the bottommost element and C is the topmost element.

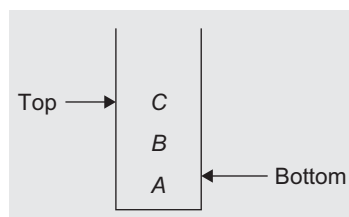


Fig. 3.2 A stack of three letters A , B , and C

3.2.1 Primitive Operations

The three basic stack operations are `push`, `pop`, and `getTop`. Besides these, there are some more operations that can be implemented on a stack such as `stack_initialization`, `stack_empty`, and `stack_full`. The `stack_initialization` operation prepares the stack for use and sets it to a vacant state. The `stack_empty` operation simply tests whether the stack is empty. The `stack_empty` operation is useful as a safeguard against an attempt to `pop` an element from an empty stack. Popping an empty stack is an error condition. The `stack_empty` condition is also termed *stack underflow*. In ideal conditions, stacks should possess infinite capacity so that the subsequent elements can always be pushed, regardless of the number of elements already present on the stack. However, computers always have finite memory capacity, and we do need to check the `stack_full` condition before doing `push` because pushing an element in a full stack is also an error condition. Such a stack full condition is called *stack overflow*.

Another stack operation is `GetTop`. This returns the top element of the stack without actually popping it. A few more stack operations include traversing the stack, counting the total number of elements in the stack, and copying the stack.

Let us quickly recall all the stack operations:

1. `Push`—inserts an element on the top of the stack
2. `Pop`—deletes an element from the top of the stack
3. `GetTop`—reads (only reading, not deleting) an element from the top of the stack
4. `Stack_initialization`—sets up the stack in an empty condition
5. `Empty`—checks whether the stack is empty
6. `Full`—checks whether the stack is full

Push

The `push` operation inserts an element on the top of the stack. The recently added element is always at the top of the stack. Before every `push`, we must ensure whether there is a room for a new element (Fig. 3.3).

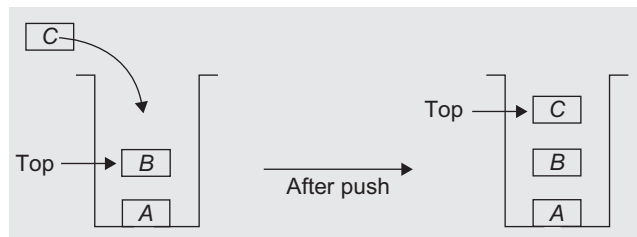


Fig. 3.3 The push operation

When there is no space to accommodate the new element on the stack, the stack is said to be *full* (Fig. 3.4). If the operation `push` is performed when the stack is full, it is said to be in *overflow* state, that is, no element can be added when the stack is full. The `push` operation modifies the *top* since the newly inserted element becomes the topmost element (Fig. 3.3).

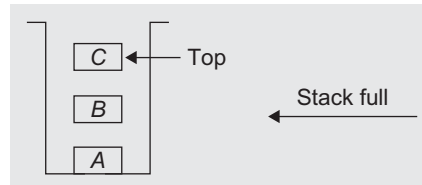


Fig. 3.4 The stack full condition (stack capacity = 3)

Pop

The `pop` operation deletes an element from the top of the stack and returns the same to the user. It modifies the stack so that the next element becomes the top element (Fig. 3.5).

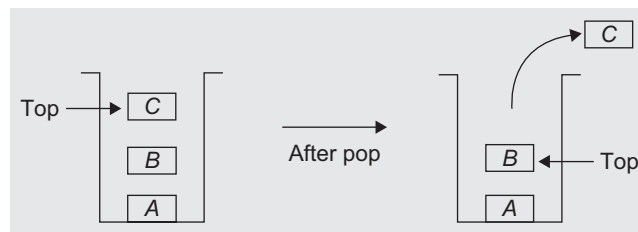


Fig. 3.5 The pop operation

When there is no element available on the stack, the stack is said to be *empty*. If `pop` is performed when the stack is empty, then the stack is said to be in an *underflow state* (Fig. 3.6).

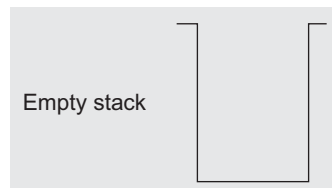


Fig. 3.6 The empty stack

The `pop` operation should not be performed when the stack is empty, and hence before every `pop`, we must ensure that the stack is not empty. After deleting the last element from the stack, the stack should be set to an empty state.

GetTop

The `getTop` operation gives information about the topmost element and returns the element on the top of the stack. In this operation, only a copy of the element, which is at the top of the stack, is returned. Hence, the `top` is still set to the same element (Fig. 3.7).

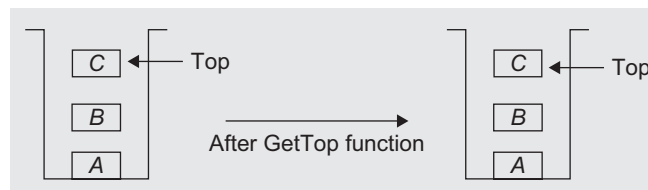


Fig. 3.7 The `getTop` operation

This is the key difference between the `pop` and `getTop` operations. The `getTop` operation does not modify the variable `top`. It signals the stack underflow error if the stack is empty.

As both insert and delete operations are allowed only at one end of the stack, it retrieves data in the reverse order in which the data is stored. In Fig. 3.8, let $S = \{A, B, C\}$.

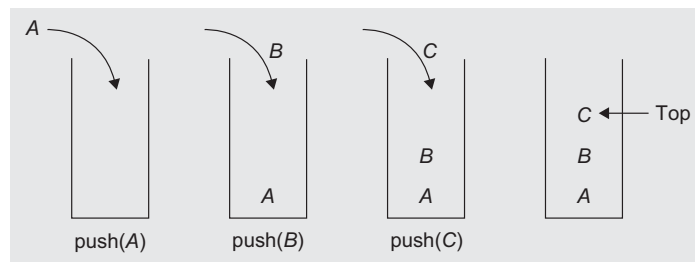


Fig. 3.8 Stack and push operations

Suppose that the order of the operations is `push(A)`, `push(B)`, and then `push(C)`. When we remove these elements out of the stack, they will be removed in the order C, B, and then A. This is shown in Fig. 3.9.

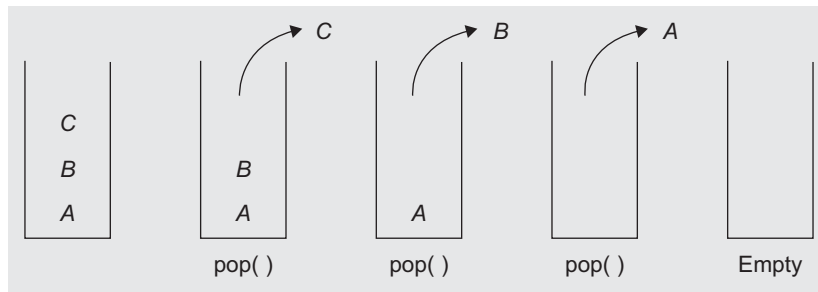


Fig. 3.9 Stack and pop operations

Elements are taken out in the reverse order of the insertion sequence. So a stack is often called *last in first out (LIFO)* or *first in last out (FILO)* data structure.

3.3 STACK ABSTRACT DATA TYPE

Let us now see the data object, operations, and axioms associated with the stack. Any sets of elements that are of the same data type can be used as a data object for stacks. The meaning of ‘same data type’ is that all the elements in the stack should be of the same nature, having common representational logical properties. For example, stack of integers, stack of names of students, stack of employee records, and stack of records of processes of the operating system.

The following five functions comprise a functional definition of a stack:

1. `Create(S)`—creates an empty stack
2. `Push(i, S)`—inserts the element i on the stack S and returns the modified stack
3. `Pop(S)`—removes the topmost element from the stack S and returns the modified stack
4. `GetTop(S)`—returns the topmost element of stack S
5. `Is_Empty(S)`—returns true if S is empty, otherwise returns false

However, when we choose to represent a stack, it must be possible to build these operations. Before we do this, let us describe formally the structure `stack`.

```
ADT Stack(element)
1. Declare Create() → stack
2. push(element, stack) → stack
3. pop(stack) → stack
4. getTop(stack) → element
5. Is_Empty(stack) → Boolean;
6. for all  $S \in \text{stack}$ ,  $e \in \text{element}$ , Let
7. Is_Empty(Create) = true
8. Is_Empty(push(e, S)) = false
```

```

9. pop(Create()) = error
10. pop(push(e, S)) = S
11. getTop(Create) = error
12. getTop(push(e, S)) = e
13. end
14. end stack

```

The five functions with their domains and ranges are declared in lines 1 through 5. Lines 6 through 13 are the set of axioms that describe how the functions are related. Lines 10 and 12 are important because they define the LIFO behaviour of the stack. This description shows an infinite stack of no upper bound or roof on the number of elements specified. This will be discussed when we represent this structure using C++.

We studied the concept of ADT in Chapter 1. The ADT stack is defined in Section 3.3. To implement the ADT stack in C++, the operations are often implemented as functions to provide data abstraction. A program that uses stacks would access the stacks only through these functions and would not be concerned about the implementation.

3.4 REPRESENTATION OF STACKS USING SEQUENTIAL ORGANIZATION (ARRAYS)

A stack can be implemented using both a static data structure (array) and a dynamic data structure (linked list). The simplest way to represent a stack is by using a one-dimensional array. A stack implemented using an array is also called a *contiguous stack*.

An array is used to store an ordered list of elements. A stack is an ordered collection of elements. Hence, it would be very simple to manage a stack when represented using an array. The only difficulty with an array is its static memory allocation. Once declared, the size cannot be modified during run-time. We have already read that this leads to either poor utilization of the space or inability to accommodate all possible data elements. This is because we declare an array to be of arbitrarily maximum size before compilation.

Figure 3.10 shows the realization of a stack using arrays.

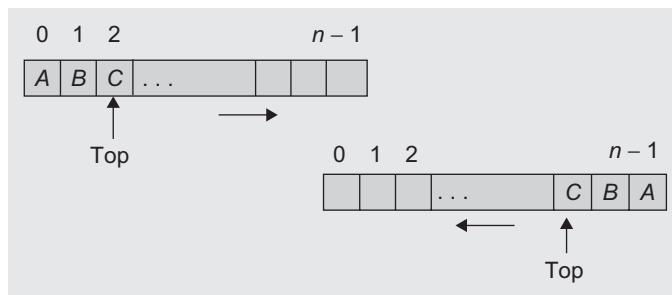


Fig. 3.10 Stack using array

Let $\text{Stack}[n]$ be a one-dimensional array. When the stack is implemented using arrays, one of the two sides of the array can be considered as the top (upper) side and the other as the bottom (lower) side as in Fig. 3.10.

Let us discuss the top side, which is most commonly used. The elements are stored in the stack from the first location onwards. The first element is stored at the 0th location of the array Stack , which means at $\text{Stack}[0]$, the second element at $\text{Stack}[1]$, the i^{th} element at $\text{Stack}[i - 1]$, and the n^{th} element at $\text{Stack}[n - 1]$. Associated with the array will be an integer variable, top , which points to the top element in the stack. The initial value of top is -1 when the stack is empty. It can hold the elements from index 0, and can grow to a maximum of $n - 1$ as this is a static stack using arrays.

Program Code 3.1 gives the definition of class `Stack` and lists the function prototypes for a set of basic operations.

PROGRAM CODE 3.1

```
class Stack
{
    private:
        int Stack[50];
        int MaxCapacity;
        int top;
    public:
        Stack()
        {
            MaxCapacity = 50;
            top = -1;
            currentsize = 0;
        }
        int getTop();
        int pop();
        void push(int Element);
        int Empty();
        int CurrSize();
        int IsFull();
};
```

The simplest way to implement an ADT stack is using arrays. We initialize the variable `top` to -1 using a constructor to denote an empty stack. The bottom element is represented using the 0th position, that is, the first element of the array. The next element is stored at the 1st position and so on. The variable `top` indicates the current element at the top of the stack.

3.4.1 Create

The stack when created is initially empty. The implementation of the stack could be using an array or using a linked list implementation. For array implementation, its size should be predefined, and its implementation time should not exceed run-time. However, in case of a linked implementation, this limitation is overcome. Let us first look at a simple stack implementation. At the end of this chapter, we shall study about other better array-based implementations using C++ features such as templates and dynamic arrays.

For each and every stack, there is an operational end operator variable called the *top* which points to the element at the top of the stack. Hence, this integer variable holds the index of the array. It can also be implemented as a pointer variable. Let us currently use it as an integer variable. Even though we call it as a pointer pointing to the top element of the stack, it is an integer index variable.

The constructor must initialize the stack *top*, so as to represent an empty stack, to a value that represents the top of the empty stack. We cannot initialize it to one of the values in the range of 0 to $n - 1$ because these are the indices of the stack array. The indices 0 to $n - 1$ represent one of the locations going to hold the stack elements. However, it can be initialized to any arbitrary integer value other than 0 to $n - 1$. Each `push` operation increments *top* by one. This is to update *top* to point to a newly added element. When the element is added to the empty stack, *top* should be set to 0 as the new element will be stored at `Stack[0]`. Hence, it is suitable to initialize the *top* to -1 . This is the most suitable initialization instead of any other arbitrary value.

```
int Stack[100];
int top = -1;
```

These statements create an empty stack of size 100, which will hold integer values, and the variable *top* is initialized to -1 .

3.4.2 Empty

`Empty` is an operation that takes the stack as an argument, checks whether it is empty or not, and returns the Boolean value *true* or *false*, respectively.

The stack empty state can be checked by comparing the value of *top* with the value -1 , because *top* = -1 represents an empty stack.

```
if(top == -1)
    return 1;
else
    return 0;
```

3.4.3 GetTop

The `getTop` operation checks for the stack empty state. If the stack is empty, it reports the 'stack underflow' error message; else it returns a copy of the element that

is at the top of the stack. Here, *top* is not updated as the element is not deleted from the stack; rather, the element is still at the top location. The element is just read from the stack.

Hence, its behaviour can be described using the following statement:

```
if(top == -1)
    cout << "Stack underflow (empty)" << endl;
else
    return(Stack[top]);
```

3.4.4 Push

The `push` operation inserts an element onto the stack of maximum size `MaxCapacity`. Element insertion is possible only if the stack is not full. We have not discussed the full operation in ADT. The stack is practically full when the array size exceeds (or the memory is full, which can happen when we use the linked list representation of the stack). Hence, the stack full state can be verified by comparing the `top` with `MaxCapacity - 1`. If the stack is not full, the `top` is incremented by 1 and the element is added on the top of the stack. In brief,

```
if(top == MaxCapacity - 1)
    cout << "Stack overflow (full)";
else
{
    top++;           //increment top by one
    Stack[top] = Element; //add the element in new top position
}
```

3.4.5 Pop

The `pop` operation deletes the element at the top of the stack and returns the same. This is done only if the stack is not empty. If the stack is empty, no deletion is possible. This is checked by the `empty()` function. If the stack is not empty, then the element at the top of the stack is returned and the `top` is decreased by one.

This is executed as

```
if(top == -1)
    cout << "Stack underflow\n";
else
    return(Stack[top--]);
```

The stack full condition signals that more storage is needed, and in many applications of stacks, the stack empty state signals the end of processing. Program Code 3.2 illustrates the basic operations on a stack.

PROGRAM CODE 3.2

```

class Stack
{
    private:
        int Stack[50];
        int MaxCapacity;
        int top;
    public:
        Stack()
        {
            MaxCapacity = 50;
            top = -1;
        }
        int getTop();
        int pop();
        void push(int Element);
        int Empty();
        int CurrSize();
        int IsFull();
};

int Stack :: getTop()
{
    if(!Empty())
        return(Stack[top]);
}

int Stack :: pop()
{
    if(!Empty())
        return(Stack[top--]);
}

int Stack :: Empty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}

```

```

int Stack :: IsFull()
{
    if(top == MaxCapacity - 1)
        return 1;
    else
        return 0;
}

int Stack :: CurrSize()
{
    return(top + 1);
}

void Stack :: push(int Element)
{
    if(!IsFull())
        Stack[++top] = Element;
}

void main()
{
    Stack S;
    S.pop();
    S.push(1);
    S.push(2);
    cout << S.getTop() << endl;
    cout << S.pop() << endl;
    cout << S.pop() << endl;
}

```

3.5 STACKS USING TEMPLATE

The stack using an array and its operations in Program Code 3.2 is defined to operate on integer data. To define stack for floating point data, we need to change `int Stack[]` to `float Stack[]` in the declaration of data members of the class. This can be done each time the data type of array elements varies, by editing the code using a text editor and then recompiling it. A *template* is a variable that can be instantiated to any data type. This data type could be of the built-in or user-defined type. Program Code 3.2 is rewritten using templates as Program Code 3.3.

PROGRAM CODE 3.3

```

template <class T>
class Stack
{
    private:
        T * Stack;           // stack using pointer
        int top;
        int Size;
    public:
        Stack(int StackSize = 20 );           // constructor
        T& getTop();
        T& pop();
        void push(const T& Element);
        bool IsEmpty();
        int CurrSize();
};

template <class T>
Stack <T> :: Stack(int StackSize) : Size(StackSize)
{
    Stack = new T[Size];
    top = -1;
}

template <class T>
T& Stack :: getTop()
{
    if !IsEmpty()
        return(Stack[top]);
    else
        cout << "Stack is Empty" << endl;
}

template <class T>
T& Stack :: pop()
{
    if !IsEmpty()
        return(Stack[top--]);
    else

```

```

        cout << "Stack is Empty" << endl;
    }
    Bool Stack :: IsEmpty()
    {
        if(top == -1)
            return 1;
        else
            return 0;
    }

    Bool Stack :: IsFull()
    {
        if(top == MaxCapacity - 1)
            return 1;
        else
            return 0;
    }

    int Stack :: CurrSize()
    {
        return(top + 1);
    }

    void Stack :: push(const T & Element)
    {
        if(!IsFull())
            cout << "Stack is Full" << endl;
        else
            Stack[++top] = Element;
    }

```

3.6 MULTIPLE STACKS

Often, data is represented using several stacks. The *contiguous stack* (stack using an array) uses separate arrays for more than one stack, if needed. The use of a contiguous stack when more than one stack is needed is not a space-efficient approach, because many locations in the stacks are often left unused. An efficient solution to this problem is to use a single array to store more than one stack. Figure 3.11 shows two stacks using one array.

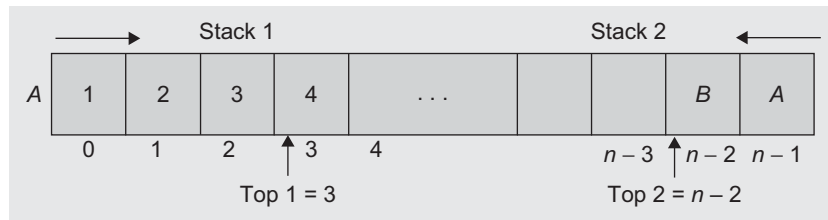


Fig. 3.11 Initial configuration for two stacks in $A[0], \dots, A[n-1]$

Multiple stacks can be implemented by sequentially mapping these stacks into $A[0], \dots, A[n-1]$. The solution is simple if we implement only two stacks. The first stack grows towards $A[n-1]$ from $A[0]$ and the second stack grows towards $A[0]$ from $A[n-1]$.

This way, we can make use of the space most efficiently so that the stack is full only when the top of one stack reaches the top of other stack.

The difficulty arises when we have to represent m stacks in the memory. We can divide $A[0, \dots, n-1]$ into m segments and allocate one of these segments to each of the m stacks. This initial division into segments may be done in proportion to the expected sizes of the various stacks, if the sizes are known. In the absence of such information, $A[0, \dots, n-1]$ may be divided into equal segments. For each stack i , we shall use $s[i]$ to represent a position one less than the position in A for the bottommost element of that stack as shown in Fig. 3.12.

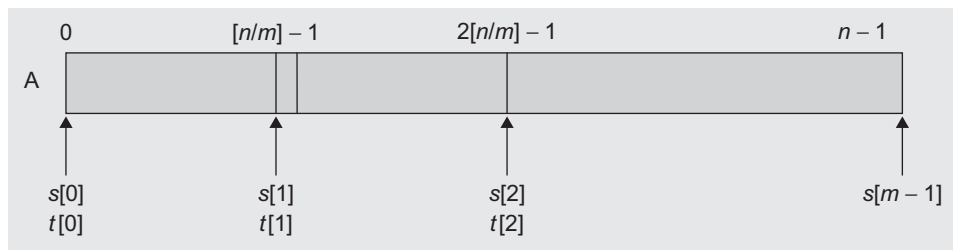


Fig. 3.12 Initial configuration for m stacks in $A[0, \dots, n-1]$

Here, $t[i], 0 \leq i \leq m-1$ will point to the topmost element of the stack i .

We shall use the boundary condition $s[i] = t[i]$ if the i^{th} stack is empty.

Initially, $s[i] = t[i] = [n/m] \times (i-1), 0 \leq i \leq n-1$.

Stack[i] will grow from $s[i] + 1$ to $s[i+1]$ before it catches up with the $(i+1)^{\text{th}}$ stack. Using this scheme, the `m_push` and `m_pop` programs can be written as in Program Code 3.4.

PROGRAM CODE 3.4

```

Stack :: m_push(int i, char x)
{
    // push x to the ith stack
    if(t[i] == s[i + 1])
        Stack_full(i);
    else
    {
        t[i] = t[i] + 1;
        A[t[i]] = x;
    }
}

char Stack::m_pop(int i)
{
    // pop topmost element of stack i
    if(t[i] == s[i])
        Stack_empty(i);
    else
    {
        t[i] = t[i] - 1;
        return(A[t[i] + 1]);
    }
}

```

`Stack_full()` and `Stack_empty()` are the functions to be written depending on the strategy followed in each case. For example, if we permit the addition of elements to stacks as long as there is some free space in array *A*, the following steps may be one of the solutions to this:

1. Determine the last $i < j \leq m$, such that there is a free space between the stacks j and $j + 1$, that is, $t[j] < s[j + 1]$. If there is such an $A[j]$, we can move the stacks $i + 1, i + 2, \dots, j$ one position to the right (treating $A[n]$ as the rightmost) and can create a space between the stacks i and $i + 1$.
2. If there is no j in step 1, then check the left side of stack i . Find the largest j such that $1 \leq j \leq i$ and there is space between the stacks j and $j + 1$, that is, $t[j] < s[j + 1]$. If there is such a j , then move the stacks $j + 1, j + 2, \dots, i$ by one space left, creating a free space between the stacks i and $i + 1$.
3. If there is no such j satisfying the conditions of either steps 1 or 2, then all the n spaces of *A* are utilized, and there is no free space.

3.7 APPLICATIONS OF STACK

The stack data structure is used in a wide range of applications. A few of them are the following:

1. Converting infix expression to postfix and prefix expressions
2. Evaluating the postfix expression
3. Checking well-formed (nested) parenthesis
4. Reversing a string
5. Processing function calls
6. Parsing (analyse the structure) of computer programs
7. Simulating recursion
8. In computations such as decimal to binary conversion
9. In backtracking algorithms (often used in optimizations and in games)

3.8 EXPRESSION EVALUATION AND CONVERSION

The most frequent application of stacks is in the evaluation of arithmetic expressions. An arithmetic expression is made of *operands*, *operators*, and *delimiters*. When high-level programming languages came into existence, one of the major difficulties faced by computer scientists was to generate machine language instructions that could properly evaluate any arithmetic expression.

A complex assignment statement such as

$$X = (A/B + C \times D - F \times G/Q)$$

might have several meanings, and even if the meanings were uniquely defined, it is still difficult to generate a correct and reasonable instruction sequence. Fortunately, the solution we have today is both elegant and simple. Till date, this conversion is considered as one of the major aspects of compiler writing.

Let us see the difficulties in understanding the meaning of expressions. The first problem in understanding the meaning of an expression is to decide the order in which the operations are to be carried out. This demands that every language must uniquely define such an order.

For instance, consider the following expression:

$$X = a/b \times c - d$$

Let $a = 1$, $b = 2$, $c = 3$, and $d = 4$.

One of the meanings that can be drawn from this expression could be

$$X = (1/2) \times (3 - 4) = -1/2$$

Another way to evaluate the same expression could be

$$X = (1/(2 \times 3)) - 4 = -23/6$$

To avoid more than one meaning being drawn out of an expression, we have to specify the order of operation by using parentheses. For instance,

$$X = (a/b) \times (c - d)$$

To fix the order of evaluation, assign each operator a priority. Even though we write the expression in parentheses, we still query whether to evaluate (A/B) first or to evaluate $(C - D)$ first. Once the priorities are assigned, then within any pairs of parentheses the operators with the highest priority are to be evaluated first. While evaluating an expression, the following operation precedence is usually used:

The following operators are written in descending order of their precedence:

1. Exponentiation (^), Unary (+), Unary (−), and not (~)
2. Multiplication (×) and division (/)
3. Addition (+) and subtraction (−)
4. Relational operators <, ≤, =, ≠, ≥, >
5. Logical AND
6. Logical OR

Some integer values can be assigned as priority, as in Table 3.1.

Table 3.1 Operators and their priorities

Arithmetic, boolean, and relational operators	Priority
^, Unary +, Unary −, ~	1
×, /	2
+, −	3
<, ≤, =, ≠, ≥, >	4
AND	5
OR	6

Note that all the relational operators have the same priority. Exponentiation (^) and unary operators (+, −, and ~) have the highest priority. When there are two adjacent operators with the same priority, again the question arises as to which one to evaluate first. For example, the expression, $A + B - C$ can be understood in two ways— $(A + B) - C$ or $A + (B - C)$.

This needs a decision on whether to evaluate the expression from right to left or left to right. Expressions such as $A + B - C$ and $A \times B/C$ are to be evaluated from left to right. However, the expression $A \wedge B \wedge C$ is to be evaluated from right to left as

$A \wedge (B \wedge C)$. For example, to compute $2 \wedge 3 \wedge 2$, we need to represent and evaluate it as $2 \wedge (3 \wedge 2)$. When evaluated from left to right, the expression may be evaluated as $((2 \wedge 3) \wedge 2)$, which is wrong!

Hence, the operators need to decide on a rule for proceeding from left to right for all expressions except the operator exponential. This order of evaluation, from left to right or right to left, is called *associativity*. Exponentiation is right associative and all other operators are left associative. When we write a parenthesized expression, these rules can be overridden. In the parenthesized expressions, the innermost parenthesized expression is evaluated first.

Let us consider the expression

$$X = A/B \wedge C + D \times E - A \times C$$

By using priorities and associativity rules, the expression X is rewritten as

$$X = A/(B \wedge C) + (D \times E) - (A \times C)$$

For example, let X be an infix expression as $((2 + 3) \times 4)/2$

We manually evaluate the innermost expression first as $((5) \times 4)/2$, followed by the next parenthesized inner expression $(20)/2$, which produces the result 10.

Still the question remains as to how a compiler can accept such an expression and produce the correct code. The solution is to rework on the expression to a form called the *postfix notation*.

3.8.1 Polish Notation and Expression Conversion

The Polish Mathematician Han Lukasiewicz suggested a notation called *Polish notation*, which gives two alternatives to represent an arithmetic expression, namely the *postfix* and *prefix* notations. The fundamental property of Polish notation is that the order in which the operations are to be performed is determined by the positions of the operators and operands in the expression. Hence, the advantage is that parentheses is not required while writing expressions in Polish notation. The conventional way of writing the expression is called *infix*, because the binary operators occur between the operands, and unary operators precede their operand. For example, the expression $((A + B) \times C)/D$ is an infix expression. In postfix notation, the operator is written after its operands, whereas in prefix notation, the operator precedes its operands. Table 3.2 shows one sample expression in all three notations.

Table 3.2 Example expression in various forms—infix, prefix, and postfix

Infix	Prefix	Postfix
(operand)(operator)(operand)	(operator)(operand)(operand)	(operand)(operand)(operator)
$(A + B) \times C$	$\times + ABC$	$AB + C \times$

In Example 3.1, the conversion of an expression to its postfix and prefix notations is discussed.

EXAMPLE 3.1 Convert the following expression to its postfix and prefix notations:

$$X = A/B \wedge C + D \times E - A \times C$$

Solution By applying the rules of priority and associativity, this expression can be written in the following form:

$$X = ((A/(B \wedge C)) + (D \times E) - (A \times C))$$

It can be reworked to get its equivalent postfix and prefix expressions.

Postfix: $ABC \wedge / DE \times + AC \times -$

Prefix: $- + / A \wedge BC \times DE \times AC$

3.8.2 Need for Prefix and Postfix Expressions

We just studied that evaluation of an infix expression using a computer needs proper code generation by the compiler without any ambiguity and is difficult because of various aspects such as the operator's priority and associativity. This problem can be overcome by writing or converting the infix expression to an alternate notation such as the prefix or the postfix. The postfix and prefix expressions possess many advantages as follows:

1. The need for parenthesis as in an infix expression is overcome in postfix and prefix notations.
2. The priority of operators is no longer relevant.
3. The order of evaluation depends on the position of the operator but not on priority and associativity.
4. The expression evaluation process is much simpler than attempting a direct evaluation from the infix notation.

Let us see how postfix expressions are evaluated.

3.8.3 Postfix Expression Evaluation

The postfix expression may be evaluated by making a left-to-right scan, stacking operands, and evaluating operators using the correct number from the stack as operands and again placing the result onto the stack. This evaluation process is much simpler than attempting a direct evaluation from the infix notation. This process continues till the stack is not empty or on occurrence of the character #, which denotes the end of the expression.

Algorithm 3.1 lists the steps involved in the evaluation of the postfix expression E .

ALGORITHM 3.1

```

1. Let  $E$  denote the postfix expression
2. Let Stack denote the stack data structure to be used & let Top = -1
3. while(1) do
    begin
        X = get_next_token(E) // Token is an operator, operand, or delimiter
        if(X = #) {end of expression}
            then return
        if(X is an operand)
            then push(X) onto Stack
        else {X is operator}
            begin
                OP1 = pop() from Stack
                OP2 = pop() from Stack
                Tmp = evaluate(OP1, X, OP2)
                push(Tmp) on Stack
            end
        {If X is operator then pop the correct number of operands
        from stack for operator X. Perform the operation and push the
        result, if any, onto the stack}
    end
4. stop

```

It is assumed that the last character in E is '#'. A procedure `get_next_token` is used to extract the next token from E . A token is an operand, an operator, or a '#'. A one-dimensional array `Stack[n]` is used as a stack.

Let us consider an example postfix expression $E = AB + C \times \#$. Now, let us scan this expression from left to right, character by character, as represented in Fig. 3.13.

This evaluation process is much simpler than the evaluation of the infix expression. Let us now devise an algorithm for converting an infix expression to a postfix notation. To see how to devise an algorithm for translating from infix to postfix, note that the operands in both notations appear in the same sequence. Let us also learn how we can manually convert an infix expression into a postfix expression.

The following are the steps for manually converting an expression from one notation to another:

1. Initially, fully parenthesize the given infix expression. Use operator precedence and associativity rules for the same.
2. Now, move all operators so that they replace their corresponding right parenthesis.
3. Finally, delete all parentheses, and we get the postfix expression.

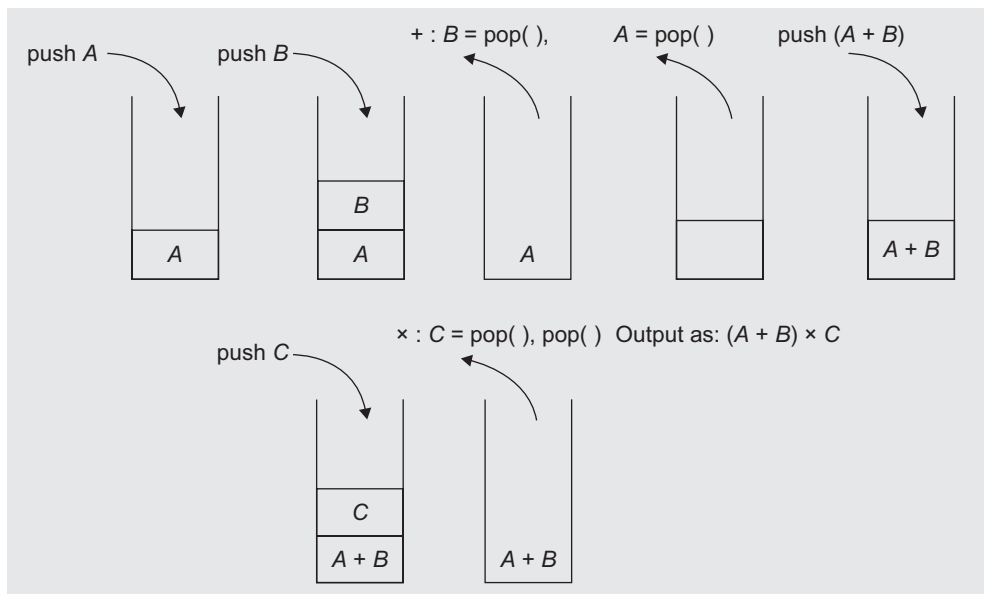


Fig. 3.13 Evaluation of postfix expression $AB + C \times$

The evaluation of a postfix expression is simple, but now we need to convert an infix expression to its postfix form. Let us consider an example $E = A/B \wedge C + D \times E - A \times C$.

Let us fully parenthesize the same as

$$E = (((A/(B \wedge C)) + (D \times E)) - (A \times C))$$

Let us move all operators to the corresponding right parenthesis and replace the same.

$$E = (((A/(B \wedge C)) + (D \times E)) - (A \times C))$$

Now let us eliminate all parentheses. We get the postfix equivalent of the infix expression.

$$E(\text{postfix}) = ABC \wedge / DE \times + AC \times -$$

This method can be used to get an equivalent prefix notation too as follows:

$$(((A/(B \wedge C)) + (D \times E)) - (A \times C))$$

We now get the prefix expression after eliminating the parentheses as

$$E(\text{prefix}) = - + / A \wedge BC \times DE \times AC$$

This procedure is a suitable method to manually convert the expression only. Let us try to work out the algorithm to convert an infix to a postfix (also to prefix).

We have observed that the order of the operand remains the same in the infix and the postfix notations. The output of the conversion should be a postfix notation. This postfix expression has a sequence of operands which is the same as that of the input infix expression. Hence, the operands from the infix expression can be immediately sent to the output as they occur. To handle the operators, the operands are stored in the stack until the right moment and they are unstacked (removed from the stack); they are then passed to the output.

For example, Let E be an infix expression as

$$E = A + B \times C$$

After conversion, the expression should yield $ABC \times +$, that is, the sequence of stacking them should be as given in Table 3.3.

Table 3.3 Infix to postfix conversion of the expression $E = A + B \times C$

Next character	Stack	Output
A	Empty	A
+	+	A
B	+	AB

Now, we have to decide about the operator \times . This is illustrated in Table 3.4(a).

Here, note that the algorithm must decide whether the operator \times gets placed on the top of the stack or the operator $+$ is to be popped off. Since operator \times has the highest priority, we should stack it so as to get the sequence of operations for expression X_2 as shown in Table 3.4(b).

Table 3.4 Handling and stacking of the \times operator in expressions

(a) Handling of the \times operator		
	Infix	Postfix
Examples	$X_1 = (A + B) \times C$	$AB + C \times$
	$X_2 = A + (B \times C)$	$ABC \times +$

(Continued)

Table 3.4 (Continued)

(b) Stacking of the \times operator		
Next character	Stack	Output
<i>A</i>	Empty	<i>A</i>
+	+	<i>A</i>
<i>B</i>	+	<i>AB</i>
\times	$+\times$	<i>AB</i>
<i>C</i>	$+\times$	<i>ABC</i>
# (Pop all)	$+\times$	<i>ABC \times +</i>

In addition, when the input is exhausted, we should output all remaining operators in the stack to get the postfix expression as *ABC \times +*.

Let us consider one more example. The infix expression $A \times (B + C) \times D$, after conversion, should generate the postfix expression *ABC $+\times D\times$* , and hence, the sequence of operations should be as shown in Table 3.5.

Table 3.5 Infix to postfix conversion of the expression $A \times (B + C) \times D$

Next character	Stack	Output
<i>A</i>	Empty	<i>A</i>
\times	\times	<i>A</i>
($\times($	<i>A</i>
<i>B</i>	$\times($	<i>AB</i>
+	$\times(+$	<i>AB</i>
<i>C</i>	$\times(+$	<i>ABC</i>

(Continued)

At this point, unstack the corresponding left parenthesis and then delete the left and right parentheses; this should give the stack contents as follows:

Table 3.5 (Continued)

Next character	Stack	Output
)	\times	<i>ABC+</i>
\times	\times	<i>ABC$+\times$</i>
<i>D</i>	\times	<i>ABC $+\times D$</i>
Done	Empty	<i>ABC $+\times D\times$</i>

From these examples and discussion, we can say that the operators are popped out of the stack if their in-stack priority (ISP) is greater than the priority of the incoming operator that is to be added onto the stack.

Consider the infix expression $E = A \times B + C\#$. The conversion of this expression into its postfix form is shown in Table 3.6.

Table 3.6 Infix to postfix conversion of the expression $E = A \times B + C$

Next character	Stack	Output
A	Empty	A
\times	\times	A
B	\times	AB
+	+	AB \times
C	+	AB \times C
# (Pop all)	+	AB \times C+

Now, let us consider the infix expression $X = A \wedge B \wedge C$

For its equivalent postfix expression, the sequence of `push` and `pop` operations should be as given in Table 3.7.

Table 3.7 Infix to postfix conversion of the expression $X = A \wedge B \wedge C$

Next character	Stack	Output
A	Empty	A
\wedge	\wedge	A
B	\wedge	AB
\wedge		

We have decided the strategy for pushing and popping out the operator from the stack. In this example, the operator at the top of the stack and the operator to be pushed onto the stack are the same. If this rule is applied, then the output is $AB \wedge C \wedge$, which is wrong! Hence, we need to add a few more checks. We must take into account the associativity of operators and prepare a hierarchy scheme for the binary arithmetic operators and delimiters. When an operator is at the top of the stack or in an expression (current token), they are to be treated with different priorities, as shown in Table 3.8.

Table 3.8 The operator and its ISP and ICP

Symbol	In-stack priority (ISP)	Incoming priority (ICP)
)	—	—
\wedge	3	4
$\times/$	2	2
$+ -$	1	1
(0	4

Thus, we can say that when the operators are taken out from the stack, their ISP, is greater than or equal to the ICP, of the new operator.

Hence, each operator is to be assigned two priorities—the incoming priority (ICP) and the in-stack priority (ISP). Incoming priority is considered when the operator is located in the given infix expression, whereas ISP is the priority when the operator is at the top of the stack. In Example 3.1, we observed that the lower priority operators should spend more time in the stack and the higher priority operators should be popped out earlier. To achieve this, we need to assign the appropriate ICPs and ISPs to the operators. Table 3.8 shows these values. If the incoming operator is the same as that of the in-stack operator and if the operator is left associative, then the operator from the stack should be popped and printed.

For example, consider the infix expressions $X = A \times B \times C$ and $Y = A/B \times C$. The expression $X = A \times B \times C$ should yield the postfix expression as $AB \times C \times$, and $Y = A/B \times C$ should generate the postfix expression as $AB/C \times$.

If the priority of the operator on the top of stack (in-stack operator) is greater than the priority of the operator coming from the expression (incoming operator), then the incoming operator is pushed onto the stack.

In short, the following points should be taken into consideration while assigning ICPs and ISPs:

1. Higher priority operators should be assigned higher values of ISP and ICP.
2. For right associative operators, ISP should be lower than ICP. For example, $A \wedge B \wedge C$ should generate $ABC \wedge \wedge$, which means $(A) \wedge (B \wedge C)$.
3. If ICP is higher than ISP, the operator should be stacked.
4. The ISP and ICP should be equal for left associative operators.

Summing up The following are the steps involved in the evaluation of an expression.

1. Assign priorities to all operators and define associativity (left or right).
2. Assign appropriate values of ICPs and ISPs accordingly. For left associative operators, assign equal ISP and ICP. For right associative operators, assign higher ICP than ISP. For example, assign a higher ICP for '^' and for the right parenthesis ')'.
 3. Scan the expression from left to right, character by character, till the end of expression.
 4. If the character is an operand, then display the same.
 5. If the character is an operator and if $ICP > ISP$
 - then push the operator
 - else
 - while($ICP \leq ISP$)
 - pop the operator and display it.
 - end while
 - Stack the incoming operator
 6. Continue till end of expression

The expression could be in one of the three forms—infix, postfix, or prefix.

An expression in one form can be converted to the other two forms. Let us write algorithms for all these conversions.

1. Infix expression to postfix expression
2. Infix expression to prefix expression
3. Prefix expression to infix expression
4. Prefix expression to postfix expression
5. Postfix expression to infix expression
6. Postfix expression to prefix expression

Let E be the expression made of characters. Characters here include *operators*, *operands*, and *delimiters*. In addition, let '#' be the character denoting the end of the expression.

Infix to Postfix Conversion

Algorithm 3.2 illustrates the infix to postfix conversion.

ALGORITHM 3.2

1. Scan expression E from left to right, character by character, till character is '#'
`ch = get_next_token(E)`
 2. while($ch \neq \text{'\#'}$)
`if(ch = \text{'('}) then ch = pop()`
`while(ch != \text{'('})`
`Display ch`
`ch = pop()`
`end while`
`if(ch = operand) display the same`
`if(ch = operator) then`
`if(ICP > ISP) then push(ch)`
`else`
`while(ICP <= ISP)`
`pop the operator and display it`
`end while`
`ch = get_next_token(E)`
`end while`
 3. if($ch = \text{'\#'}$) then while(!emptystack()) pop and display
 4. stop
-

For this algorithm, we refer to the operators and the respective ICPs and ISPs as assigned in Table 3.8. Example 3.2 illustrates the conversion of an infix expression to its postfix form (function in Program Code 3.5).

EXAMPLE 3.2 Convert the following infix expression to its postfix form:

$$A \wedge B \times C - C + D/A/(E + F)$$

Solution Conversion of infix to postfix form can be illustrated as in Table 3.9

Table 3.9 Infix to postfix conversion of the expression $A \wedge B \times C - C + D/A/(E + F)$

Character scanned	Stack contents	Postfix expression
A	Empty	A
^	^	A
B	^	AB
×	×	AB^
C	×	AB^C
−	−	AB^C×
C	−	AB^C×C
+	+	AB^C×C−
D	+	AB^C×C−D
/	+/	AB^C×C−D
A	+/	AB^C×C−DA
/	+/	AB^C×C−DA/
(+/ (AB^C×C−DA/
E	+/ (AB^C×C−DA/E
+	+/ (+	AB^C×C−DA/E
F	+/ (+	AB^C×C−DA/EF
)	+/	AB^C×C−DA/EF+
	Empty	AB^C×C−DA/EF+/+

Infix to Prefix Conversion

For converting the infix expression to a prefix expression, two stacks are needed—the operator Stack and the display Stack. The display Stack stores the prefix expression. This approach is discussed in Algorithm 3.3.

ALGORITHM 3.3

1. Scan expression E, character by character from right to left
`ch = get_next_token(E)`
2. while(`ch != '#'`) do
 - if(`ch = operand`) then push(`ch`) in display Stack
 - if (`ch = ')'`) then
`ch = pop()` from operator Stack
 - while(`ch != '('`)
 push(`ch`) in display Stack
`ch = pop()`
 - end while
 - if(`ch = operator`) then
 if `ICP(op) >= ISP(op)` then
 push `ch` in operator Stack
 else
 `ch = pop()`
 while(`ICP < ISP`)

```

        ch = pop() from operator Stack and push 'ch' in
        display Stack
    end while
    ch = get_next_token(E)
end while
3. if (ch = '#') then
    while(!emptystack(operator))
        ch = pop(operator)
        push ch on display stack
    end while
4. while(!emptystack(display))
    ch = pop(operator)
    display ch
end while
5. stop

```

Example 3.3 illustrates the conversion of an infix expression to its prefix form.

EXAMPLE 3.3 Convert the following infix expression to its corresponding prefix form:

$$A \wedge B \times C - C + D/A/(E + F)$$

Solution The conversion to prefix notation is as given in Table 3.10

Table 3.10 Infix to prefix conversion of the expression $A \wedge B \times C - C + D/A/(E + F)$

Character scanned	Stack	Prefix expression
))	
F)	F
+) +	F
E) +	EF
(Empty	+EF
/	/	+EF
A	/	A + EF
/	//	A + EF
D	//	DA + EF
+	+	//DA + EF
C	+	C//DA + EF
-	+ -	C//DA + EF
C	+ -	CC//DA + EF
×	+ ×	CC//DA + EF
B	+ ×	BCC//DA + EF
^	+ × ^	BCC//DA + EF
A	+ × ^	ABCC//DA + EF
	Empty	+ × ^ ABCC//DA + EF

The corresponding program for infix to prefix conversion is illustrated in Program Code 3.5.

PROGRAM CODE 3.5

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define Max 20
//class Stack
class stack
{
    char stack[Max];    // array of characters
    int top;
public:
    Stack()    // constructor to initialize top
    {
        top = -1;
    }
    int isempty(); // function to check empty condition
    int isfull();  // function to check full condition
    void push(char ch); // to push a character into stack
    char pop(); // function to pop a character from stack
    char getTop(); // function to get the top element of
stack
};

int Stack::isempty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}

int Stack::isfull()
{
    if(top == Max - 1)
        return 1;
    else
        return 0;
}
```

```

void Stack::push(char ch)
{
    if(isfull())
        cout << "\nStack full";
    else
    {
        top++;
        stack[top] = ch;
    }
}

char Stack::pop()
{
    char ch;
    if(isempty())
        cout << "\n stack empty \n";
    else
    {
        ch = stack[top];
        top--;
    }
    return(ch);
}

char Stack::getTop()
{
    char ch;
    if(isempty())
        cout << "\n stack empty \n";
    else
    {
        ch = stack[top];
    }
    return(ch);
}

// Function to get in-stack priority
char isp(char ch)
{
    switch(ch)

```

```

{
    case '+':
    case '-':return 1;
    case '*':
    case '/':return 2;
    case '^':return 3;
    case '(':return 0;
    case '#':return -2;
}
}

// Function to get incoming priority
char icp(char ch)
{
    switch(ch)
    {
        case '+':
        case '-':return 1;
        case '*':
        case '/':return 2;
        case '^':return 3;
        case '(':return 4;
    }
}

void intopost(char infix[20],char postfix[20])
{
    int i = 0;
    char ch, x;
    stack s;
    s.push('#');
    while(infix[i] != '\0')
    // extract character till end of expression
    {
        ch = infix[i];
        i++;
        if(ch >= 'a' && ch <= 'z')    // operand
        {
            cout << ch;
        }
        else    // operator

```

```

        {
            if(ch == '(')
            {
                while(s.getTop() != '(')
                {
                    x = s.pop();
                    cout << x;
                }
                x = s.pop();
            }
            else
            {
                while(isp(s.getTop()) >= icp(ch))
                {
                    x = s.pop();
                    cout << x;
                }
                s.push(ch);
            }
        }
    }
    while(!s.isempty())
    {
        x = s.pop();
        if(x != '#')
            cout << x;
    }
}

void intopre(char infix[20], char prefix[20])
{
    int i, j;
    char ch, x;
    stack s;
    s.push('#');
    i = strlen(infix) - 1;
    j = 0;
    while(i != -1)
    {
        ch = infix[i];

```



```

    i--;
    if(ch >= 'a' && ch <= 'z')
    {
        prefix[j] = ch;
        j++;
    }
    else
    {
        if(ch == '(')
        {
            while(s.getTop() != ')')
            {
                x = s.pop();
                prefix[j] = x;
                j++;
            }
            x = s.pop();
        }
        else
        {
            while(isp(s.getTop()) > icp(ch))
            {
                x = s.pop();
                prefix[j] = x;
                j++;
            }
            s.push(ch);
        }
    }
}
while(!s.isEmpty())
{
    x = s.pop();
    if(x != '#')
        prefix[j] = x;
    j++;
}
prefix[j] = '\0';
strrev(prefix);
}

```

```

void main()
{
    char infix[20], postfix[20], prefix[20];
    int choice;
    do
    {
        cout << "\nMenu.....";
        cout << "\n1.Infix to postfix conversion";
        cout << "\n2.Infix to prefix conversion";
        cout << "\nEnter your choice:";
        cin >> choice;
        switch(choice)
        {
            case 1:
                cout << "\nEnter the infix expression:";
                cin >> infix;
                cout << "\nPostfix expression is:";
                intopost(infix,postfix);
                break;
            case 2:
                cout << "\nEnter the infix expression:";
                cin >> infix;
                intopre(infix,prefix);
                cout << "\nPrefix expression is:" << prefix;
                break;
        }
    }
    while(choice < 3);
}

```

Postfix to Infix Conversion

Algorithm 3.4 illustrates the postfix to infix conversion.

ALGORITHM 3.4

1. Scan expression E from left to right character by character
 ch = get_next_token(E)
2. while(ch != '#') do
 - if(ch = operand) then push(ch)
 - if(ch = operator) then
 - begin
 - t2 = pop() and t1 = pop()
 - push(strcat['(', t1, ch, t2, ''])

```

    end
    ch = get_next_token(E)
  end while
3. if ch = '#', while(!emptystack()) pop and display
4. stop

```

Example 3.4 illustrates the conversion of a postfix expression to its infix form.

EXAMPLE 3.4 Convert the following postfix expression to its infix form:

$$AB^{\wedge}C \times C - DA/EE + +$$

Solution The conversion of the given postfix expression to its infix form is given in Table 3.11.

Table 3.11 Postfix to infix conversion of the expression $AB^{\wedge}C \times C - DA/EE++$

Character scanned	Stack contents
A	A
B	AB
^	$A^{\wedge}B$
C	$A^{\wedge}B, C$
\times	$A^{\wedge}B \times C$
C	$A^{\wedge}B \times C, C$
-	$A^{\wedge}B \times C - C, D$
D	$A^{\wedge}B \times C - C, D$
A	$A^{\wedge}B \times C - C, D, A$
/	$A^{\wedge}B \times C - C, D/A$
E	$A^{\wedge}B \times C - C, D/A, E$
E	$A^{\wedge}B \times C - C, D/A, E, E$
+	$A^{\wedge}B \times C - C, D/A, E + E$
/	$A^{\wedge}B \times C - C, D/A/E + E$
+	$A^{\wedge}B \times C - C + D/A/E + E$

Postfix to Prefix Conversion

Algorithm 3.5 illustrates the postfix to prefix conversion.

ALGORITHM 3.5

1. Scan expression E from left to right character by character
`ch = get_next_token(E)`
2. while(ch != '#') do
 - if(ch = operand) then push(ch)
 - if(ch = operator) then

```

begin
    t2 = pop() and t1 = pop()
    push(strcat(ch, t1, t2])
end
ch = get_next_token(E)
end while
3. if ch = '#', while(!emptystack()) pop and display
4. stop

```

Example 3.5 illustrates the conversion of a postfix expression to its prefix form.

EXAMPLE 3.5 Convert the following postfix expression to its prefix form:

$$AB \wedge C \times C - DA/EE+/+$$

Solution The conversion of the given postfix expression to its infix form is given in Table 3.12.

Table 3.12 Postfix to prefix conversion of the expression $AB \wedge C \times C - DA/EE+/+$

Character scanned	Stack contents
A	A
B	AB
^	^AB
C	^ABC
×	×^ABC
C	×^ABC, C
−	−×^ABCC
D	−×^ABCC, D
A	−×^ABCC, D, A
/	−×^ABCC, /DA
E	−×^ABCC, /DA, E
E	−×^ABCC, /DA, E, E
+	−×^ABCC, /DA, +EE
/	−×^ABCC, //DA + EE
+	+^ABCC//DA + EE

Prefix to Infix Conversion

Algorithm 3.6 illustrates the prefix to infix conversion.

ALGORITHM 3.6

1. Scan expression E from right to left character by character
`ch = get_next_token(E)`
2. while(ch != '#') do

```

        if(ch = operand) then push(ch)
        if(ch = operator) then
        begin
            t2 = pop() and t1 = pop()
            push(strcat['(', t1, ch, t2, ')'])
        end
        ch = get_next_token(E)
    end while
3. if ch = '#', while(!emptystack()) pop and display
4. stop

```

Prefix to Postfix Conversion

Algorithm 3.7 illustrates the prefix to postfix conversion.

ALGORITHM 3.7

```

1. Scan expression E from left to right character by character
   ch = get_next_token(E)
2. while(ch != '#') do
    if(ch = operand) then push(ch)
    if(ch = operator) then
    begin
        t2 = pop() and t1 = pop()
        push(strcat [t1, t2, ch])
    end
    ch = get_next_token(E)
end while
3. if ch = '#', while(!emptystack()) pop and display
4. stop

```

The corresponding program for postfix to infix conversion is illustrated in Program Code 3.6.

PROGRAM CODE 3.6

```

//postfix to infix conversion
#include<conio.h>
#include<iostream.h>
#include<string.h>
#define Max 20
//definition of class stack
class stack
{
    char stack[Max][Max];      //stack of string
    int top;
public:

```

```

    //constructor to initialize top
    stack()
    {
        top = -1;
    }
    //function declaration
    int isempty();
    int isfull();
    void push(char str[max]);
    void pop(char str[max]);
};

//definition of isempty condition
int stack::isempty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}

//definition of isfull condition
int Stack::isfull()
{
    if(top == Max - 1)
        return 1;
    else
        return 0;
}

//definition of push function
void Stack::push(char str[Max])
{
    if(isfull())
        cout << "\nStack full";
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

```

//definition of pop function
void Stack::pop(char str[20])
{
    if(isempty())
        cout << "\nStack empty";
    else
    {
        strcpy(str, stack[top]);
        top--;
    }
}

//definition of postfix to infix conversion
void postfixtoinfix()
{
    char postfix[20], infix[20];
    char s1[10], s2[10], s3[10], ch, temp[10];
    int i;
    Stack s;          //creating of object of class stack
    cout << "\nEnter the postfix expression:";
    cin >> postfix;
    i = 0;
    while(postfix[i] != '\0')
    {
        ch = postfix[i];
        i++;
        s1[0] = ch;
        s1[1] = '\0';
        if(ch >= 'a' && ch <= 'z')
        {
            s.push(s1);
        }
        else
        {
            s.pop(s2);
            s.pop(s3);
            strcpy(temp, "(");
            strcat(temp, s3);
            strcat(temp, s1);
            strcat(temp, s2);
            strcat(temp, ")");
        }
    }
}

```

```

        s.push(temp);
    }
}
cout << "\nInfix expression is:" << temp;
}

//definition of postfix to prefix conversion
void postfixtoprefix()
{
    char postfix[20], prefix[20];
    char s1[10], s2[10], s3[10], ch, temp[10];
    int i;
    Stack s;          //creating of object of class stack
    cout << "\nEnter the postfix expression:";
    cin >> postfix;
    i = 0;
    while(postfix[i] != '\0')
    {
        ch = postfix[i]; i++;
        s1[0] = ch;
        s1[1] = '\0';
        if(ch >= 'a' && ch <= 'z')
        {
            s.push(s1);
        }
        else
        {
            s.pop(s2);
            s.pop(s3);
            strcpy(temp, s1);
            strcat(temp, s3);
            strcat(temp, s2);
            s.push(temp);
        }
    }
    cout << "\nPrefix expression is:" << temp;
}

//definition of prefix to infix conversion
void prefixtoinfix()
{
    char prefix[20], infix[20];
    char s1[10], s2[10], s3[10], ch, temp[10];

```



```

int i;
Stack s;          //creating of object of class stack
cout << "\nEnter the prefix expression:";
cin >> prefix;
for(i = strlen(prefix); i >= 0; i--)
{
    ch = prefix[i];
    s1[0] = ch;
    s1[1] = '\0';
    if(ch >= 'a' && ch <= 'z')
    {
        s.push(s1);
    }
    else
    {
        s.pop(s2);
        s.pop(s3);
        strcpy(temp, "(");
        strcat(temp, s2);
        strcat(temp, s1);
        strcat(temp, s3);
        strcat(temp, ")");
        s.push(temp);
    }
}
cout << "\nInfix expression is:" << temp;
}

//definition of prefix to postfix conversion
void prefixtopostfix()
{
    char prefix[20];
    Stack s;        //creating of object of class stack
    char s1[10], s2[10], s3[10], ch, temp[10];
    int i;
    cout << "\nEnter the prefix expression:";
    cin >> prefix;
    for(i = strlen(prefix); i >= 0; i--)
    {
        ch = prefix[i];
        s1[0] = ch;

```

```

        s1[1] = '\\0';
        if(ch>= 'a' && ch <= 'z')
        {
            s.push(s1);
        }
        else
        {
            s.pop(s2);
            s.pop(s3);
            strcpy(temp, s2);
            strcat(temp, s3);
            strcat(temp, s1);
            s.push(temp);
        }
    }
    cout << "\\nPostfix expression is:" << temp;
}

//definition of main function
void main()
{
    int choice;
    clrscr();
    do
    {
        cout << "\\n.....menu.....";
        cout << "\\n1.postfix to infix.....$";
        cout << "\\n2.postfix to prefix.....$";
        cout << "\\n3.prefix to infix.....$";
        cout << "\\n4.prefix to postfix.....$";
        cout << "\\n5.exit.....$";
        cout << "\\n\\nEnter your choice";
        cin >> choice;
        switch(choice)
        {
            //function call of functions
            case 1:
                postfixtoinfix();
                break;
            case 2:
                postfixtoprefix();
                break;

```

```

        case 3:
            prefixtoinfix();
            break;
        case 4:
            prefixtopostfix();
            break;
        default:
            cout << "\n\nSorry, wrong choice";
    }
}while(choice < 5);
getch();
}

```

3.9 PROCESSING OF FUNCTION CALLS

One natural application of stacks, which arises in computer programming, is the processing of function calls and their terminations. The program must remember the place where the call was made so that it can return there after the function is complete. Suppose we have three functions, say, A, B, and C, and one *main* program. Let the *main* invoke A, A invoke B, and B in turn invoke C. Then, B will not have finished its work until C has finished and returned. Similarly, *main* is the first to start work, but it is the last to be finished, not until sometime after A has finished and returned. Thus, the sequence by which a function actively proceeds is summed up as the LIFO or FILO property, as shown in Fig. 3.14. The output is shown in Fig. 3.15.

From the output in Fig. 3.15, it can be observed that the *main* program is invoked first but finished last, whereas the function C is invoked last but finished first. Hence, to keep track of the return addresses *ra*, *rb*, and *rc* the only data structure required here is the *stack*.

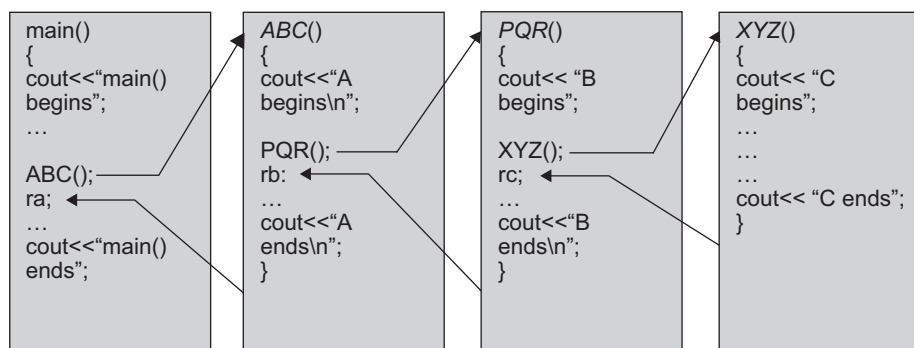


Fig. 3.14 Processing of function calls

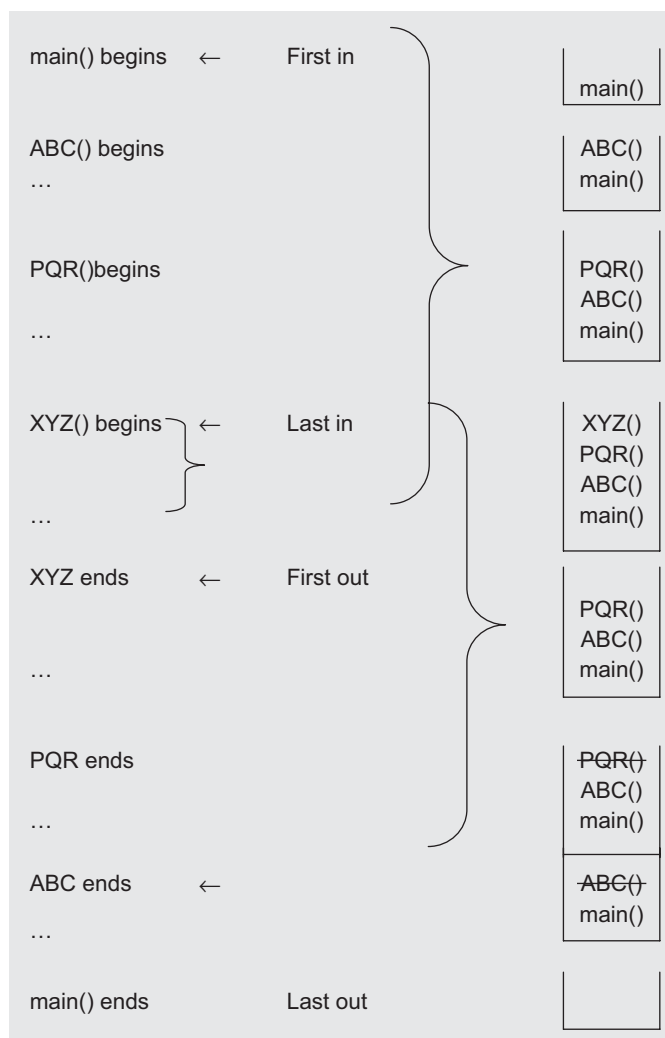


Fig. 3.15 Use of stack for processing of function calls

3.10 REVERSING A STRING WITH A STACK

Suppose a sequence of elements is presented and it is desired to reverse the sequence. Various methods could be used for this, and in the beginning, the programmer will usually suggest a solution using an array. A conceptually simple solution, however, is based on using a stack. The LIFO property of the stack access guarantees the reversal.

Suppose the sequence *ABCDEF* is to be reversed. With a stack, one simply scans the sequence, pushing each element onto the stack as it is encountered, until the end of the sequence is reached. The stack is then popped repeatedly, with each popped element sent to the output, until the stack is empty. Table 3.13 illustrates this algorithm:

Table 3.13 Reversal of a string using a stack

Input	Action	Stack	Display
<i>ABCDEF</i>	Push <i>A</i>	<i>A</i> ← top of stack	–
<i>BCDEF</i>	Push <i>B</i>	<i>AB</i> ← top of stack	–
<i>CDEF</i>	Push <i>C</i>	<i>ABC</i> ← top of stack	–
<i>DEF</i>	Push <i>D</i>	<i>ABCD</i> ← top of stack	–
<i>EF</i>	Push <i>E</i>	<i>ABCDE</i> ← top of stack	–
<i>F</i>	Push <i>F</i>	<i>ABCDEF</i> ← top of stack	–
End	Pop and display	<i>ABCDE</i> ← top of stack	<i>F</i>
	Pop and display	<i>ABCD</i> ← top of stack	<i>FE</i>
	Pop and display	<i>ABC</i> ← top of stack	<i>FED</i>
	Pop and display	<i>AB</i> ← top of stack	<i>FEDC</i>
	Pop and display	<i>A</i> ← top of stack	<i>FEDCB</i>
	Pop and display	Stack empty	<i>FEDCBA</i>
	Stop		

Reading a string character and writing it backward can be accomplished by pushing each character on to a stack as it is read. When the string is finished, pop the characters off the stack, and they will come out in the reverse order. This process is illustrated in Program Code 3.7.

PROGRAM CODE 3.7

```
main()
{
    Stack S;          // here Stack is the character stack
    char str[], ch;
    int i;
    ch = str[0];
    i = 1;
    while(ch != '\0')
    {
        S.push(ch);
        Ch = str[i++];
    }
    while(!S.Isempty())
    {
        cout << S.pop();
    }
}
```

3.11 CHECKING CORRECTNESS OF WELL-FORMED PARENTHESES

Consider a mathematical expression that includes several sets of nested parentheses. For example, $Z - ((X \times ((X + Y/J - 2)) + Y)/3)$.

To ensure that the parentheses are nested correctly, we need to check that

1. there are equal numbers of right and left parentheses
2. every right parenthesis is preceded by a matching left parenthesis

Expressions such as $((X + Y)$ or $(X + Y))$ violate condition 1, and expressions such as $(X + Y) - ($ or $(X + Y))(-A + B)$ violate condition 2.

To solve this problem, let us define the parentheses count at a particular point in an expression as the number of left parenthesis minus the number of right parenthesis that have been encountered in the left-to-right scanning of the expression at that particular point. The two conditions that must hold if the parentheses in an expression form an admissible pattern are as follows:

1. The parenthesis count at each point in the expression is non-negative.
2. The parenthesis count at the end of the expression is 0.

A stack may also be used to keep track of the parentheses count. Whenever a left parenthesis is encountered, it is pushed onto the stack, and whenever a right parenthesis is encountered, the stack is examined. If the stack is empty, then the string is declared to be invalid. In addition, when the end of the string is reached, the stack must be empty; otherwise, the string is declared to be invalid.

3.12 RECURSION

In C/C++, a function can call itself, that is, one of the statements of the function is a call to itself. Such functions are called *recursive functions* and can be used to implement recursive problems in an elegant manner.

To solve a recursive problem using functions, the problem must have an end condition that can be stated in non-recursive terms. For example, in the case of factorials, we know that $1! = 1$. If no such condition exists, then the recursive calls will indefinitely continue until the computer runs or the program is terminated by the operating system.

Consider the recursive implementation of factorial given that

$$1! = 1 \quad \text{and} \quad n! = n \times (n - 1)!$$

The recursive function in C++ is given by the following statement:

```
long int factorial (unsigned int n)
{
    if (n <= 1)
```

```

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

```

As we can see, the C++ function represents the recursive mathematical definition of $n!$. To see how it works, consider the computation of $5!$.

The function calls will proceed as follows:

```

factorial(5) = 5 * factorial(4)
              = 5 * (4 * factorial(3))
              = 5 * (4 * (3 * factorial(2)))
              = 5 * (4 * (3 * (2 * factorial(1))))
              = 5 * (4 * (3 * (2 * 1)))
              = 5 * (4 * (3 * 2))
              = 5 * (4 * 6)
              = 5 * 24
              = 120

```

As the starting number is not 1, the function calls itself with the value $5 - 1$, that is, 4. Therefore, the original function call is kept incomplete and pending, and a second call is made to the factorial with value 4. This process continues until the fifth call is made, with the value 1. In this call, the function terminates without any further recursion and returns the desired value of $1!$, which is 1. Subsequently, each of the pending function calls is completed up to the original factorial (5) function call, which returns the computed value as 120. In the preceding piece of code, parentheses have been used to show how the recursive calls proceed from left to right and the computations are made from right to left.

A program to print the first 15 factorials is given in the following code:

```

#include <iostream>
long int factorial(unsigned int n)
void main(void)
{
    int i;
    for(i = 1; i <= 15; i++)
        cout << "The factorial of" << i << "is =" << factorial(i);
}

```

Recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem. Recursive programs are most inefficient as regards their time and space complexities. Hence, there is a need to convert them into iterative ones. To achieve this conversion stacks need to be used. This is discussed in detail in Chapter 4.

3.13 PARSING COMPUTER PROGRAMS

Parsing is a special phase of compilation. While parsing a semantic expression, we need a parsing stack to hold the operands for expressions. The stack must hold both the value of the expression and its type. The purpose of the expression *value stack* is to turn infix expressions such as $1 + 2$ into postfix expressions where all the required operands are saved on the stack by the parser. The operation is then performed by popping the correct number of arguments off the stack and pushing back the single result value.

3.14 BACKTRACKING ALGORITHMS


A backtracking algorithm systematically considers all possible outcomes for each decision and performs much better than an exhaustive search. To explore a solution space of the problem, depth-first traversal of the solution space can be performed. This traversal uses the stack data structure.

3.15 CONVERTING DECIMAL NUMBERS TO BINARY

To convert a number from decimal to binary, we simply divide the number by 2 until a quotient of 0 is reached. Then, use the successive remainders in reverse order as the binary representation. For example, to convert decimal 35 to binary, we perform the following computation:

Division operation

2	35	1
	17	1
	8	0
	4	0
	2	0
	1	1



If you examine the remainders from the last division to the first one, writing them down as you go, you will get the following sequence: 100011.

$$100011_{\text{base}2} = 35_{\text{base}10}$$

The division generates a one-bit result at every step. These bits are generated in the reverse order, that is, the most significant bit is generated first and the least significant bit is generated at the end. Hence, the result is the reverse of the actual resultant binary number. We need some intermediate storage that will hold the result and finally send the output as the correct result. If we store every bit generated in a stack, we will get the correct result at the end. This is because the working behaviour of the stack is LIFO. Hence, using stack operations, we can write a procedure that accepts a non-negative

base 10 integer as a parameter and then write its binary representation. An example is illustrated in Example 3.6.

EXAMPLE 3.6 Convert the decimal number 254 to its binary equivalent.

Solution Divide the number by 2; then divide what is left by 2, and so on until there is nothing left. Write down the remainder (which is either 0 or 1) at each division stage. Once there are no more divisions, list the remainder values in reverse order. This is the binary equivalent.

254/2 gives 127 with a remainder of 0
 127/2 gives 63 with a remainder of 1
 63/2 gives 31 with a remainder of 1
 31/2 gives 15 with a remainder of 1
 15/2 gives 7 with a remainder of 1
 7/2 gives 3 with a remainder of 1
 3/2 gives 1 with a remainder of 1
 1/2 gives 0 with a remainder of 1

Therefore, the binary equivalent is 11111110. The corresponding program is illustrated in Program Code 3.8.

PROGRAM CODE 3.8

```
void Dec2Bin(int DecNum)
{
    int count = 0, bit;
    Stack S;
    while(DecNum >= 0)
    {
        bit = DecNum % 2;
        S.push(bit);
        DecNum = DecNum/2;
        count++;
    }
    cout << "The binary equivalent of" << DecNum << "is =";
    while(count > 0)
    {
        cout << S.pop();
        count--;
    }
}
```