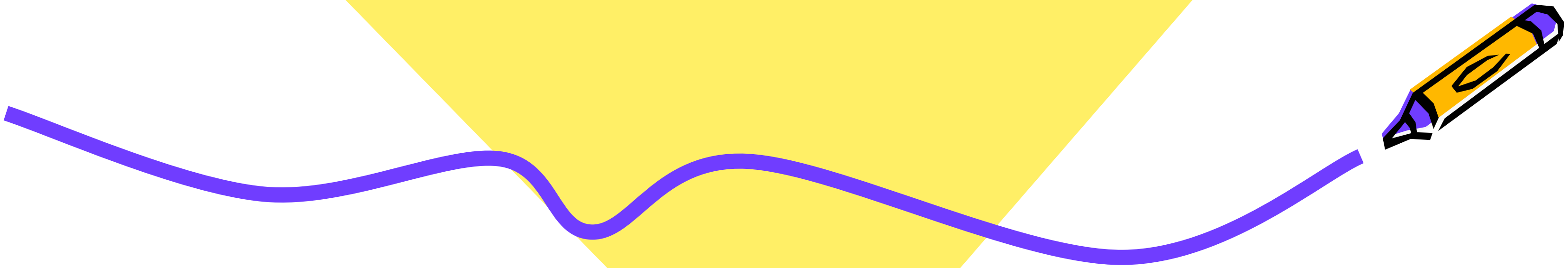
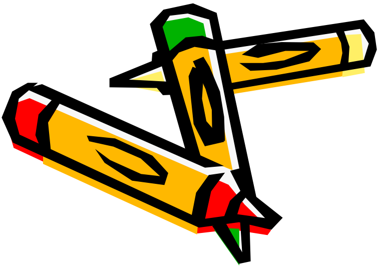
Fundamentals of DS
Stacks and Queues



Templates in C++

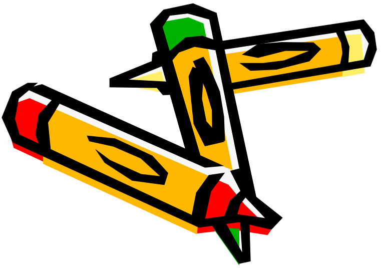
- Template function in C++ makes it easier to reuse classes and functions.
- A template may be viewed as a variable that can be instantiated to any data type, irrespective of whether this data type is a fundamental C++ type or a user-defined type.



Two min() Functions

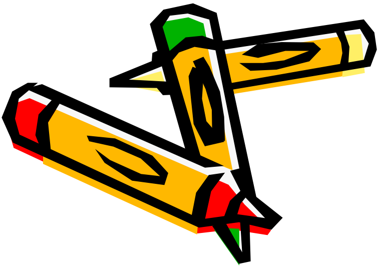
- The following program shows the weakness of strongly-typed languages:

```
int min(int a, int b) {
      return a < b ? a : b;
}

double min(double a, double b) {
      return a < b ? a : b;
}
```



The Template Solution

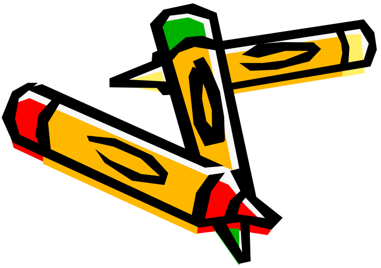
```
template <class Type>
Type min(Type a, Type b) {
    return a < b ? a : b;
}

main() {
    // ok: int min(int, int);
    min(10, 20);

    // ok: double min(double, double);
    min(10.0, 20.0);
}
```



Selection Sort Template

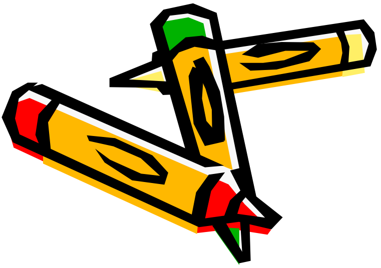
```
template <class KeyType>
void SelectionSort(KeyType *a, int n)
// sort the n KeyType a[0] to a[n-1] into nondecreasing order
{
    for (int i = 0; i < n; i++)
    {
        int j = i;
        // find smallest KeyType in a[i] to a[n-1]
        for (int k = i+1; k < n; k++)
            if (a[k] < a[j]) j = k;
        // interchange
        KeyType temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
}
```

```
float farray[100];
int intarray[200];
  ………..
SelectionSort(farray, 100);
SelectionSort(intarray, 200);
```



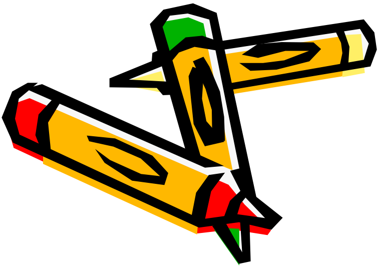
Selection Sort Template (Cont.)

- Can we use the sort template for the Rectangle class?
- Well, not directly. We'll need to use operator overloading to implement "<" for Rectangle class.



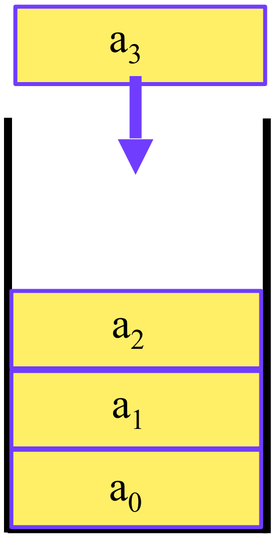
Stack

- What is a stack? A stack is an ordered list in which insertions and deletions are made at one end called the top.
- It is also called a Last-In-First-Out (LIFO) list.

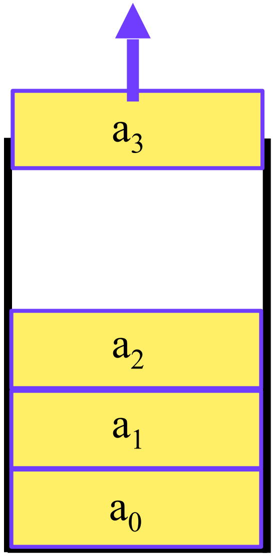


Stack (Cont.)

- Given a stack $S = (a_0, \ldots, a_{n-1})$, a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1}, $0 < i < n$.

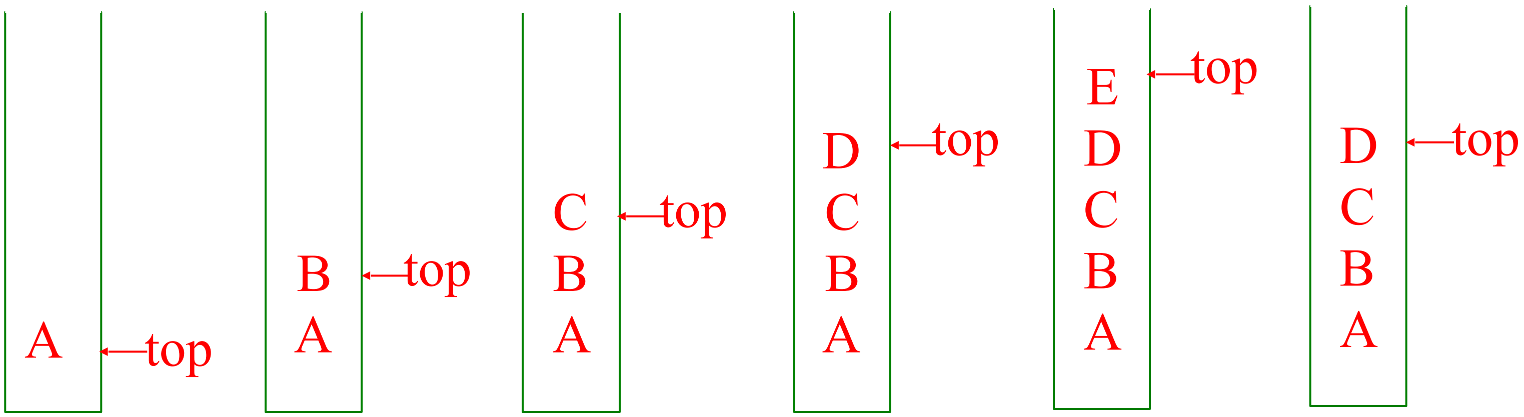


Push (Add)



Pop (Delete)

Inserting and Deleting elements in a stack


A ←top

B ←top
A

C ←top
B
A

D ←top
C
B
A

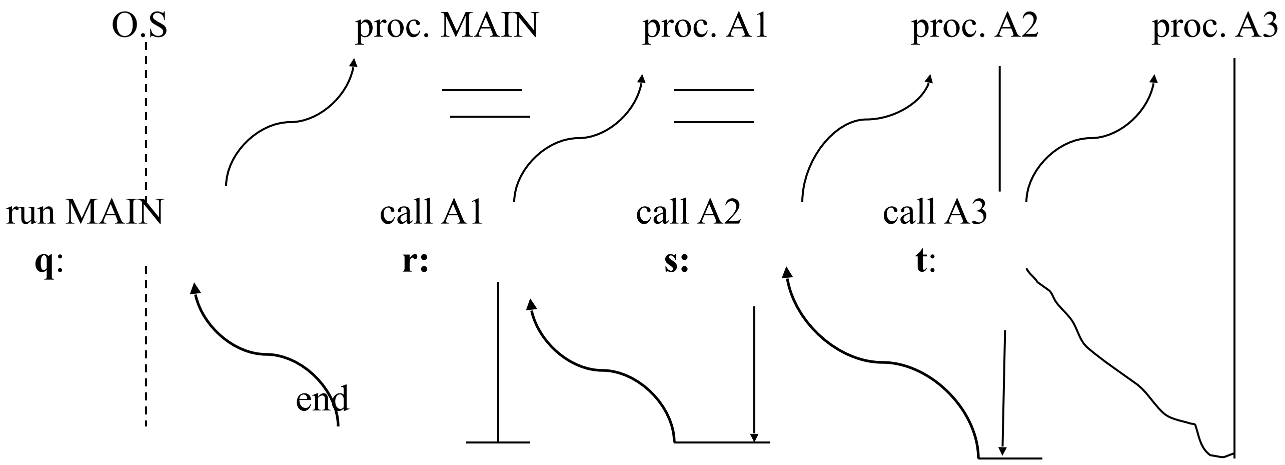
E ←top
D
C
B
A

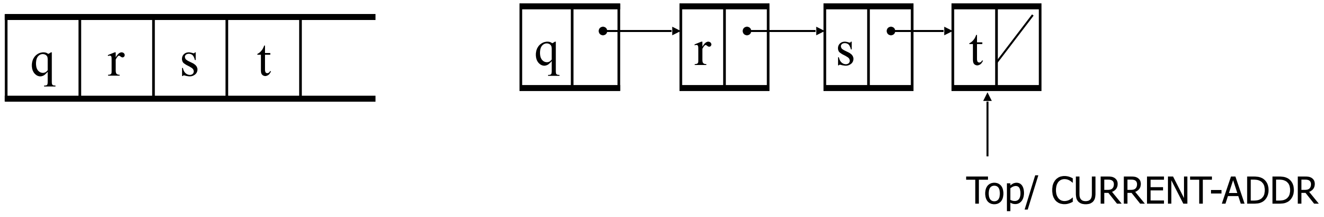
D ←top
C
B
A

Push (A) Push (B) Push (C) Push (D) Push (E) POP( ) = E

System Stack

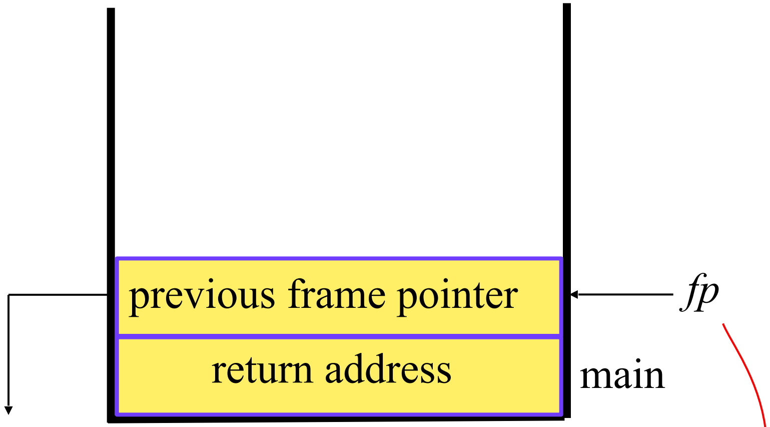
- process of subroutine calls





System Stack (cont.)

- Whenever a function is invoked, the program creates a structure, referred to as an activation record or a stack frame, and places it on top of the system stack.



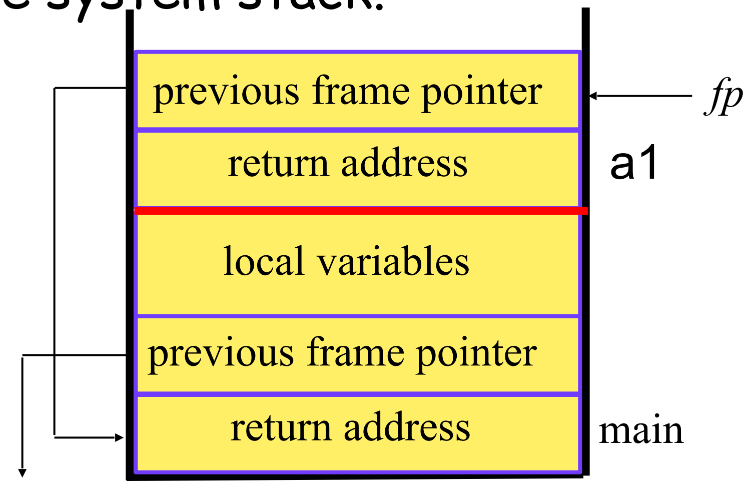
previous frame pointer

return address

main

fp

System Stack before *a*1 is invoked



previous frame pointer

return address

a1

local variables

previous frame pointer

return address

main

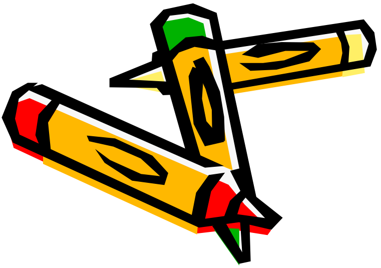
fp

System Stack after *a*1 is invoked

fp: a pointer to current stack frame

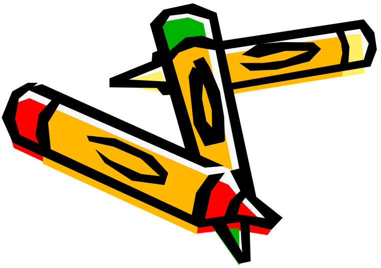
ADT 3.1 Abstract Data Type Stack

```
template <class KeyType>
class Stack
{   // objects: A finite ordered list with zero or more elements
      public:
            Stack (int MaxStackSize = DefaultSize);
            ~Stack();
            // Create an empty stack whose maximum size is MaxStackSize
            bool IsFull();
            // if number of elements in the stack is equal to the maximum size
            // of the stack, return TRUE(1) else return FALSE(0)
            bool IsEmpty();
            // if number of elements in the stack is 0, return TRUE(1) else return FALSE(0)
```



ADT 3.1 Abstract Data Type Stack (cont.)

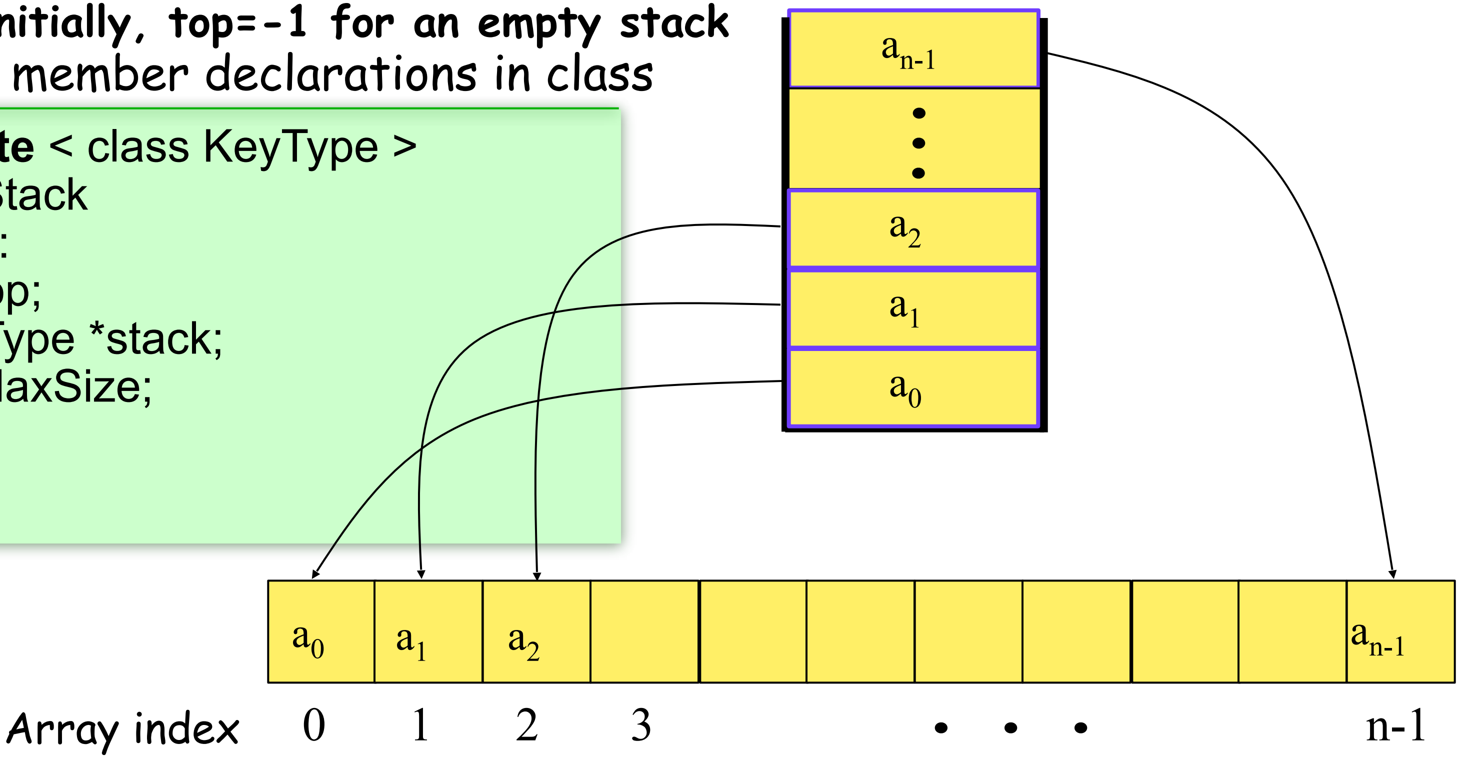
```
        KeyType& Top();
        // Return top element of stack
        void Push(const KeyType& item);
        // if IsFull(), then StackFull(); else insert item into the top of the stack.
        KeyType* Pop(KeyType& );
        // if IsEmpty(), then StackEmpty() and return 0;
        // else remove and return a pointer to the top element of the stack.
};
```

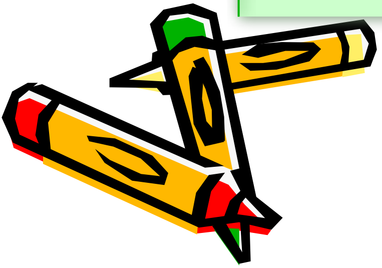


Implementation of Stack by Array

- Implementation of stack ADT
  - use an one-dim array stack[MaxSize]
  - bottom element is stored in stack[0]
  - top points to the top element
    - **initially, top=-1 for an empty stack**
  - data member declarations in class

```
template < class KeyType >
class Stack
private:
    int top;
    KeyType *stack;
    int MaxSize;

public:
   ......
```





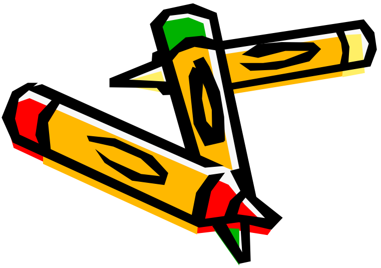
Implementation of Stack by Array (cont.)

- constructor definition

```
template <class KeyType>
Stack<KeyType>::Stack (int MaxStackSize) : MaxSize (MaxStackSize)
{
    stack = new KeyType[MaxSize];
    top = -1;
}
```

- member function IsFull()

```
template <class KeyType>
inline bool Stack<KeyType>::IsFull()
{
   if (top == MaxSize-1) return TRUE;
   else return FALSE;
}
```



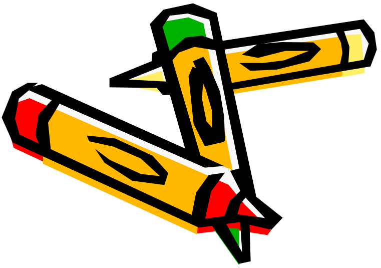
Implementation of Stack by Array (cont.)

- member function IsEmpty()

```
template <class KeyType>
inline bool Stack<KeyType>::IsEmpty() { return top == -1;}
```

- member function Top()

```
template <class KeyType>
inline KeyType& Stack<KeyType>::Top()
{
    if (IsEmpty()) throw “Stack is empty”;
    return stack[top];
}
```



Implementation of Stack by Array (cont.)

- Push operation

```
template <class KeyType>
void Stack<KeyType>::Push(const KeyType& x)
{  // add x to stack
    if (IsFull()) StackFull();
    else stack[++top] = x;
}
```

- Pop operation

```
template <class KeyType>
KeyType  Stack<KeyType>::Pop()
{  // Remove top element from stack.
    if (IsEmpty()) StackEmpty(); return 0;
    KeyType x = stack[top--];
    return x;
}
```

*StackFull() and StackEmpty() depend on the particular application

Parentheses Matching

- scan expression from left to right until eos.
  - when a left parenthesis is encountered, push it on the stack
  - when a right parenthesis is encountered, pop one from the stack and check whether they are matching.
  - If the satck is empty then error.
  - Else if they are Not matching parentheses, then error.
- If the stack is not empty, then error

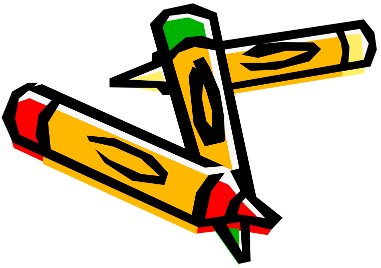
otherwise, expression is correct.

Program

```
int Check(char expr[])
{       int i=0;
        stack<char,100> stk;
        while (expr[i]!=‘\0’)
        {
                char in_symbol = expr[i];
                switch(in_symbol)
                {
                        case ‘(‘:
                        case ‘[‘:
                        case ‘{‘:       stk.Push(in_symbol); break;
                        case ‘)’:
                        case ‘]’:
                        case ‘}’:       if stk.IsEmpty()  return -1;
                                        char st_symbol = stk.Pop();
                                        if (st_symbol == ‘(‘ && in_symbol !=‘)’ ||
                                         (st_symbol == ‘[‘ && in_symbol !=‘]’ ||
                                         (st_symbol == ‘{‘ && in_symbol !=‘}’ )   return -1;
                }
                i++;
        }
        If (!stk.IsEmpty()) return -1;
        return 1;
}
```

Evaluation of Expressions

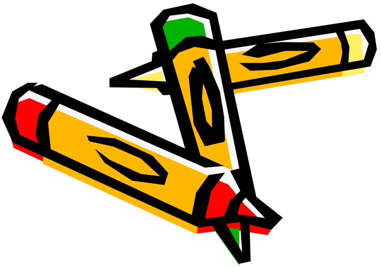
- One of the challenges for higher-level programming languages is to generate machine-language instructions to evaluate an arithmetic expression.
- X = A / B – C + D * E – A * C may have several meanings.
- Still a formidable task to generate a correct instruction sequence.
- Expression = {operands, operators, delimiters}
- Operators = {unary, binary, …}



Evaluation of Expression in C++

- When evaluating operations of the same priorities, it follows the direction from left to right.

Priority	Operator
1	Unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||



Postfix Notation

- Expressions are converted into Postfix notation before compiler can accept and process them.

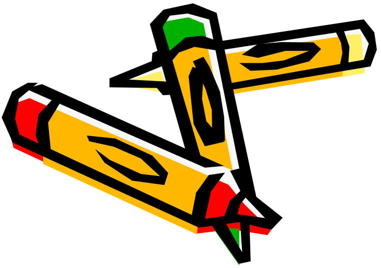
X = A/B – C + D * E – A * C

Infix A/B–C+D*E–A*C

Postfix => AB/C-DE*+AC*-

no need for parentheses and priority of the operators if using postfix notation!

Operation	Postfix
T_1 = A / B	T_1C-DE*+AC*-
T_2 = T_1 - C	T_2 DE*+AC*-
T_3 = D * E	T_2T_3+AC*-
T_4 = T_2 + T_3	T_4AC*-
T_5 = A * C	T_4 T_5 -
T_6 = T_4 - T_5	T_6



Postfix Evaluation

- Read in the expression
- Process each character of the expression until eos:
  - If the character corresponds to a single-digit number (characters ‘**0**’ to ‘**9**’), then push the corresponding number onto the stack.
  - If the character corresponds to one of the arithmetic operators (characters ‘**+**’, ‘**–**’, ‘*****’, and ‘**/**’), then
    - Pop a number off of the stack. Call it operand2.
    - Pop a number off of the stack. Call it operand1.
    - Combine these operands using the arithmetic operator, as follows:

      Result = operand1 operator operand2
    - Push result onto the stack.
- When the end of the expression is reached, pop the remaining number off the stack. This number is the value of the expression.

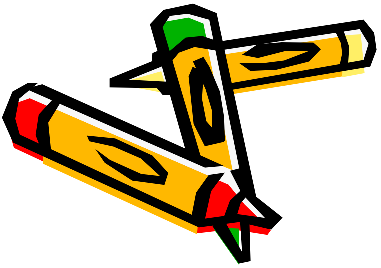
Postfix Expression execution

Input: ABC*D/+EF*-

Stack

		C		D		
	B	B	B*C	B*C	B*C/D	
A	A	A	A	A	A	A+B*C/D

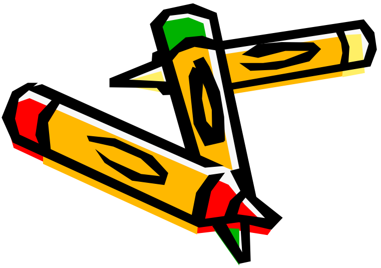
	F		
E	E	E*F	
A+B*C/D	A+B*C/D	A+B*C/D	A+B*C/D-E*F



Infix to Postfix: e.g. 1

- $A + B * C \Rightarrow ABC * +$

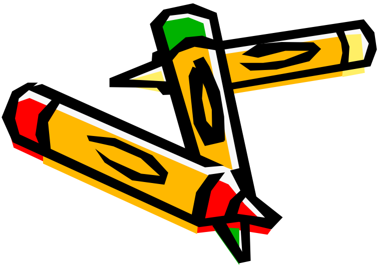
next token	stack	output
none	empty	none
A	empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
done	+	ABC*
done	empty	ABC*+



Infix to Postfix: e.g. 2

- A * (B + C) * D ⇒ ABC+*D*

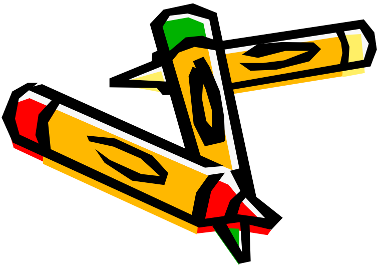
next token	stack	output
none	#	none
A	#	A
*	#*	A
(	#*(	A
B	#*(	AB
+	#*(+	AB
C	#*(+	ABC
)	#*	ABC+
*	#*	ABC+*
D	#*	ABC+*D
done	#	ABC+*D*



Infix to Postfix

Different Cases:

- Operands: Immediately output.
- Close parenthesis: Pop stack symbols until an open parenthesis appears.
- Operator: Pop all stack symbols until a symbol of lower precedence or a right-associative symbol of equal precedence appears. Then push the operator.
- End of input: Pop all remaining stack symbols.

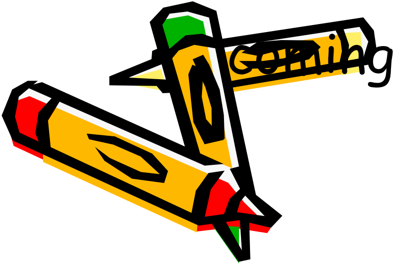


Priority-based Scheme for stacking and unstacking

- Two functions: isp (in-stack priority), icp (in-coming priority)
-

Symbol	In-Stack Priority	In-Coming Priority
)	-	-
**	3	4
*,/	2	2
binary +,-	1	1
(	0	4
#	-1	-

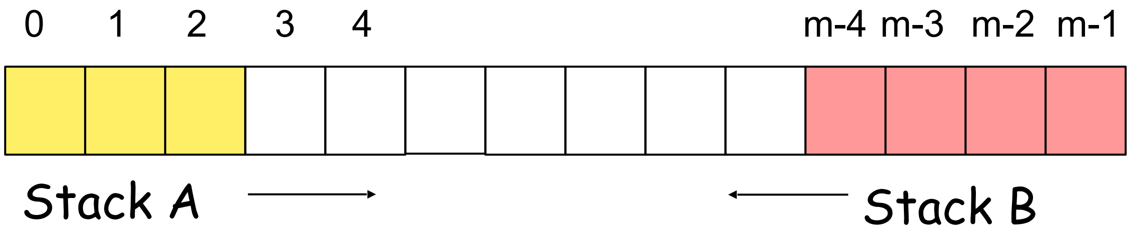
- Golden rule: operators are taken out of the stack as long as their in-stack priority, isp, is greater than or equal to the in-coming priority, icp of the incoming operator.



```
void Infix_Postfix  (char infix[], char postfix[])
// Output the postfix form of the infix expression e. Also, ‘#’ is used at the bottom of the stack
{
    Stack<char> stack; // initialize stack
    char x,y;
    int j=0;
    stack.Push(‘#’);
    for (int i=0, x=infix[i]; x!=‘\0’; i++, x=infix[i])
    {
            switch(x)
            {
            case operand :          postfix[j++]=x; break;
            case  rtpar:            // unstack until ‘(‘
                                    y = stack.Pop();
                                    while (y!=ltpar) {postfix[j++]= y; y = stack.Pop(); }break;
            case operator:          y = stack.Pop();
                                    while(isp(y) >= icp(x))
                                    {postfix[j++]= y; y = stack.Pop(); }
                                    stack.Push(y); // restack the last y that ws unstacked
                                    stack.Push(x);

            }
    }
     // end of expression; empty stack
     y = stack.Pop(); while (y != ‘#’) {postfix[j++]= y; y = stack.Pop(); }
    postfix[j]=‘\0’;
}
```

Multiple Stacks

- Two stacks:


0
1
2
3
4
m-4
m-3
m-2
m-1
Stack A
Stack B