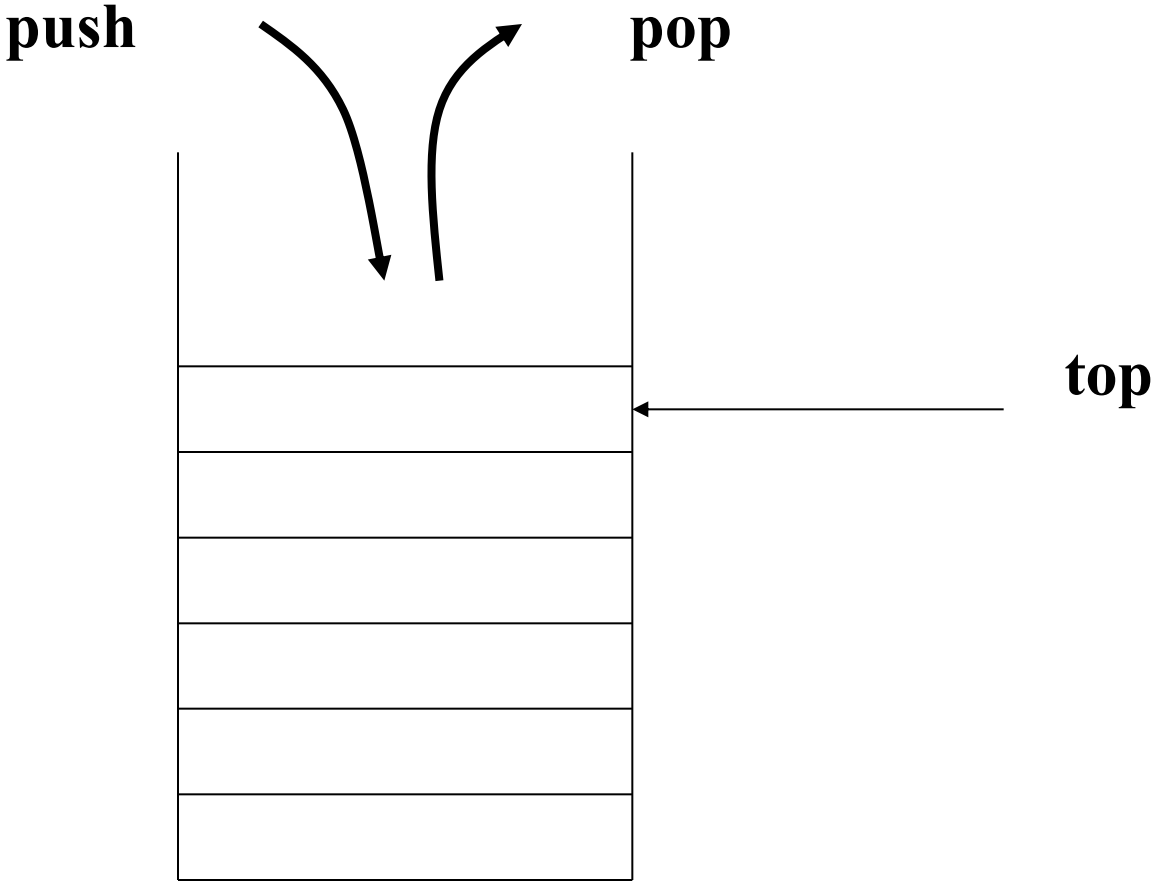
Stacks

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

The Stack ADT

- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the *last-in first-out* (LIFO) scheme.
- It is like a stack of trays:
  - Trays can be added to the top of the stack.
  - Trays can be removed from the top of the stack.
- Main stack operations:
  - **push**(object o): inserts element o on top of the stack
  - **pop**(): removes and returns the last inserted element



Stack

The Stack ADT

- Stack operations:
  - **push(item):** insert an element on top of the stack
  - **pop():** return the element on top of the stack
  - **reset():** empty the stack
- Auxiliary stack operations:
  - **top**(): returns a reference to the last inserted element without removing it
  - **size**(): returns the number of elements stored
  - **isEmpty**(): returns a Boolean value indicating whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty.
- If Stack is full, or we don’t have enough memory, push causes an exception.

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Stacks and Computer Languages

- A stack can be used to check for unbalanced symbols (e.g. matching parentheses)
- Algorithm

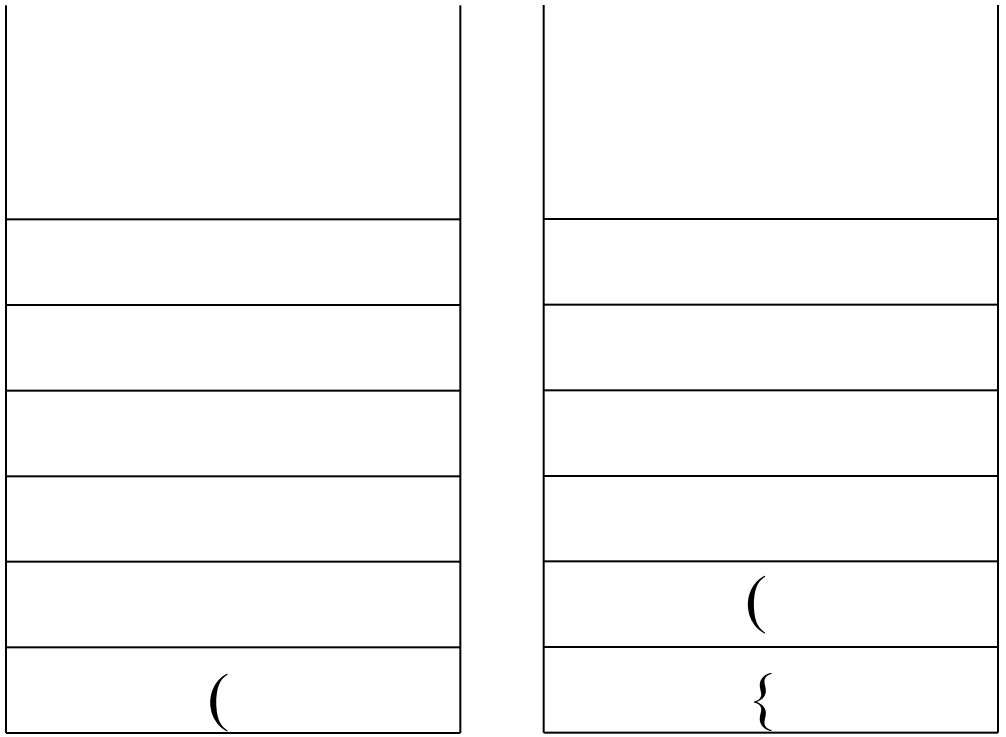
1. **Make an empty stack.**
2. **Read symbols until the end of file.**
   a. **If the token is an opening symbol, push it onto the stack.**
   b. **If it is a closing symbol and the stack is empty, report an error.**
   c. **Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, report an error.**
3. **At the end of the file, if the stack is not empty, report an error.**

Example for matching paranthesses

```
main() {
   int i = 5;
   foo(i);
   }

foo(int j) {
   int k;
   k = j+1;
   bar(k);
   }

bar(int m) {
   …
   }
```



Stack

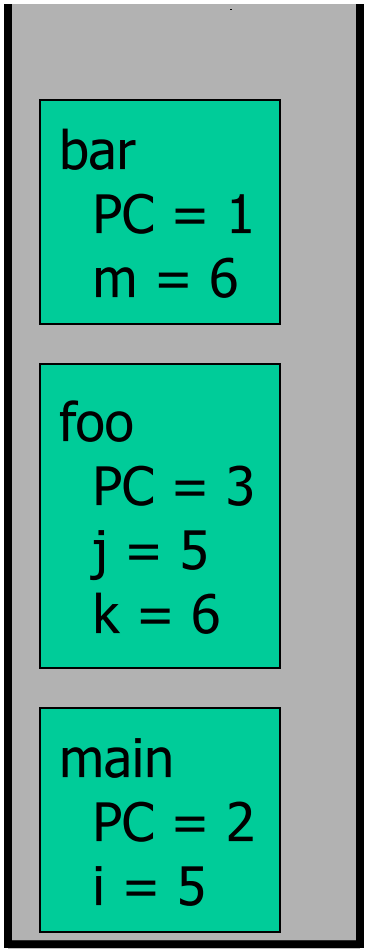
C Run-time Stack

- The C run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



Array-based Stack

- A simple way of implementing the Stack ADT uses an array.
- We add elements from left to right.
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
  else
    t ← t – 1
    return S[t + 1]
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm ***push***(***o***)
  if $t = S.length - 1$ **then**
    throw ***FullStackException***
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori*, and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Stack Implementation Using Arrays

```
#define N 500  // max # of elements that stack can have

int S[N];      // stack of integers
int sp = 0;    // stack pointer, shows the place for insertion

int push(int item){  // insert an element to the stack
   if (sp >= N){
      printf("Stack is full, no insertion is possible\n");
      return -1;
   }
   else {
      S[sp] = item;
      sp++;
      return 0;
   }
}
```

Array-based Stack in C

```
int pop(){  // remove an element from the stack
   if (sp <= 0){
      printf("Stack is empty\n");
      //assume that stack has positive integers
      return -1;
   }
   else {
      sp--;
      return S[sp];
   }
}

void reset(){   // empty the stack
   sp = 0;
}
```

Array-based Stack in C (cont.)

```
int top(){  // returns the reference to the last
            // inserted element without removing it
   return (sp - 1);
}

int size(){   // returns the # of elements stored
   return sp;
}

int isEmpty(){   // returns 0 if no elements are stored
                 // returns 1 otherwise
   if (sp == 0)
      return 0;
   else
      return 1;
}
```

Example

- *Reading a line of text and writing it out backwards.*

```
int main( )
{
  char c;
  while ((c = getchar() )!='\n')
      push(c);

  while( sp > 0 )
     printf("%c", pop( ));

  printf("\n");

  return 0;
}
```

A Simple Calculator

- Calculators can evaluate infix expressions, such as 5 + 2.
- In an infix expression a binary operator has arguments to its left and right.
  - e.g. 1 + 2 * 3
    9 – 5 –3
    2 ^ 3 ^ 2
- When there are several operators, precedence and associativity determine how the operators are processed.

  10 – 3 – 2 ^ 3 * 4 / 5 / 10 ^ 2

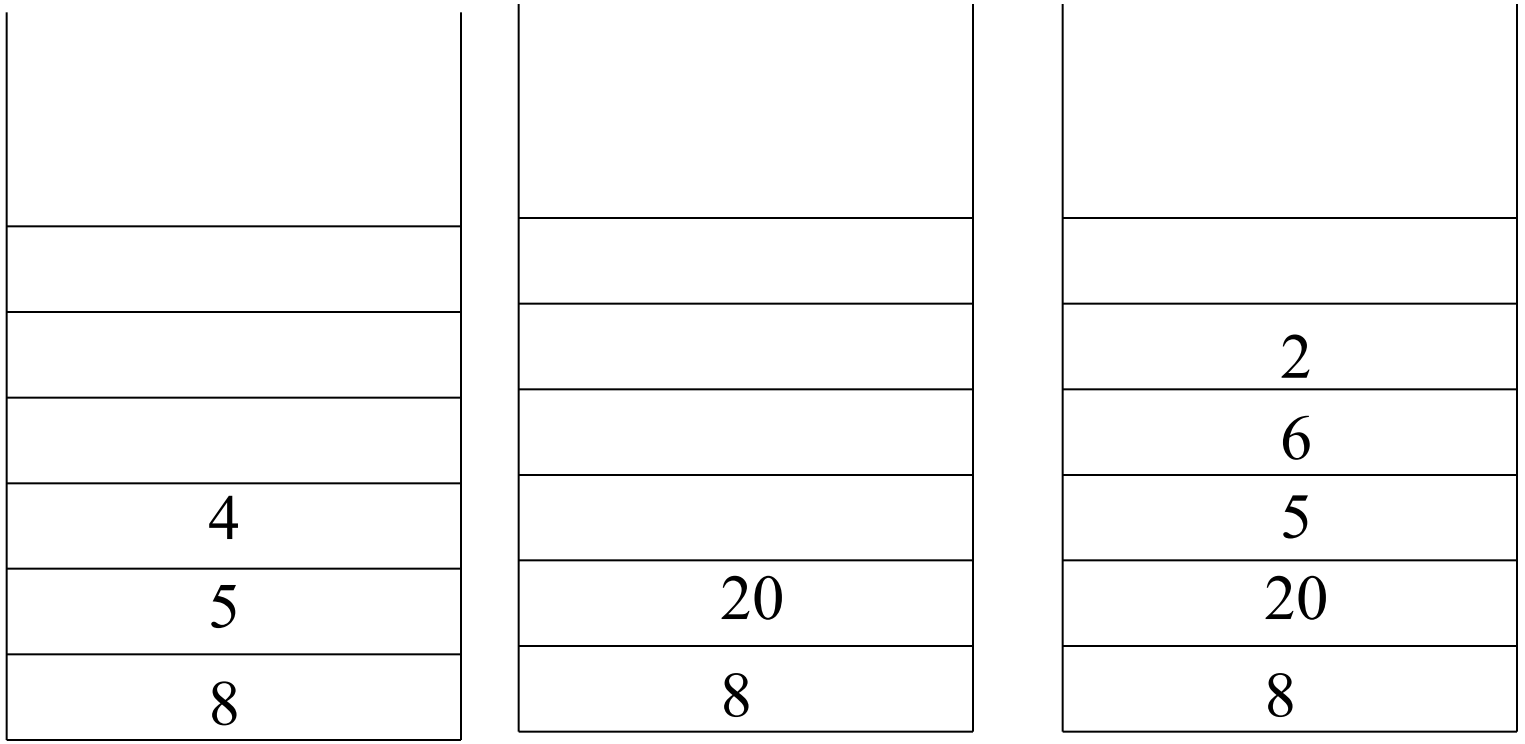
Postfix Machines

- In a postfix expression a binary operator follows its operands.
  - e.g. 5 2 +
    1 2 3 * +
    10 3 – 2 3 ^ 4 * 5 / 10 2 ^ / -
- A postfix expression can be evaluated as follows:
  - Operands are pushed into a single stack.
  - An operator pops its operands and then pushes the result.
  - At the end of the evaluation, the stack should contain only one element, which represents the result.

Example

- Evaluate the following postfix expression.

`8 5 4 * 5 6 2 / + - 2 / +`



Result =14

Infix to Postfix Conversion

- The operator precedence parsing algorithm converts an infix expression to a postfix expression, so we can evaluate the infix expression.
- An ***operator*** stack is used to store operators that have been seen but not yet output.
- When an operator is seen on the input, operators of **higher** priority (or **left associative** operators of equal priority) are removed from the stack, signaling that they should be applied. The input operator is then placed on the stack.
- What about **right associative** operators and **parentheses**?

Example

- Convert the following infix expression to postfix.

```
A + B * C /(D * E - F * G ) + H
```

```
A B C D E * F G * - / * + H +
```

Linked list implementation of Stacks

- In implementing Stack as a linked list, the top of the stack is represented by the first item in the linked list.
- To implement push: create a new node and attach it as the new first node.
- To implement pop: advance the top of stack to the second item in the list (if there is one).
- Each operation is performed in constant time.

Stack Implementation Using Linked Lists

```
typedef struct s{
    int item;
    struct s *next;
}STACK;

STACK *sp = NULL;
```

Stack Implementation Using Linked Lists (cont.)

```
int push(int item){
   STACK *p;
   p = (STACK *)malloc(sizeof(STACK));
   if (p == NULL){
      printf("Error, not enough memory\n");
      return -1;
   }
   else{
      p->item = item;
      p->next = sp;
      sp = p;
      return 0;
   }
}
```

Stack Implementation Using Linked Lists (cont.)

```
int pop(){
   STACK *p;
   int i;
   if (sp == NULL){
      printf("Stack is empty\n");
      return -1;
   }
   else {
      p = sp;
      i = sp->item;
      sp = sp->next;
      free(p);
      return i;
   }
}
```

Stack Implementation Using Linked Lists (cont.)

```
void reset(){
   STACK *p;
   while (sp != NULL){
      p = sp;
      sp = sp->next;
      free(p);
   }
}

STACK *top(){  //returns the address of the top element
   return sp;
}
```

Stack Implementation Using Linked Lists (cont.)

```
int size(){   // returns the # of elements in the stack
   STACK *p;
   int i = 0;
   p = sp;
   while (p){
      i++;
      p = p->next;
   }
   return i;
}

STACK *isEmpty(){ // indicates whether no elements
                  // are stored or not.
   return sp;
}
```