Stack - Data Structures

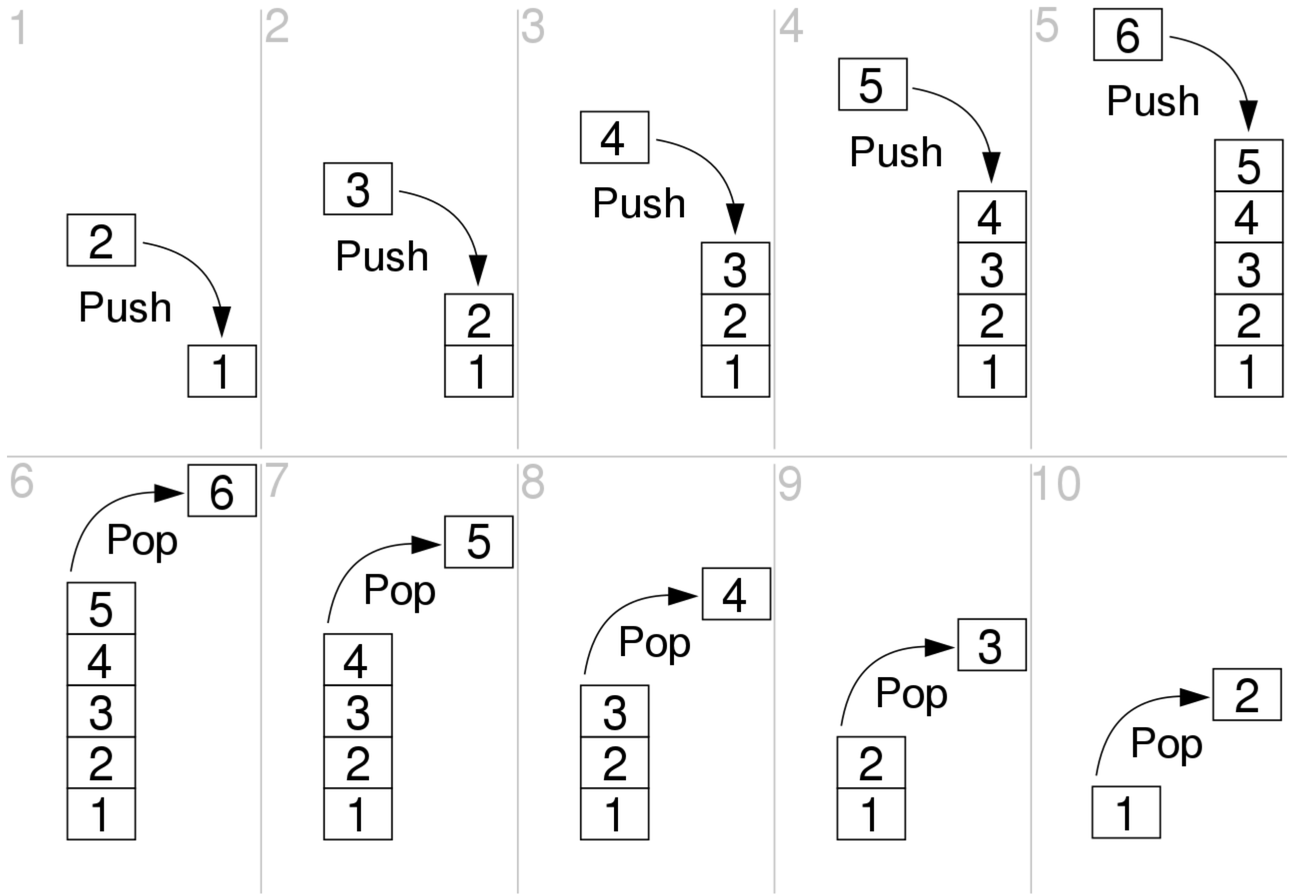
Dr. Animesh Chaturvedi
Assistant Professor: IIIT Dharwad
Young Researcher: Heidelberg Laureate Forum
Postdoc: King's College London & The Alan Turing Institute
PhD: IIT Indore MTech: IIITDM Jabalpur

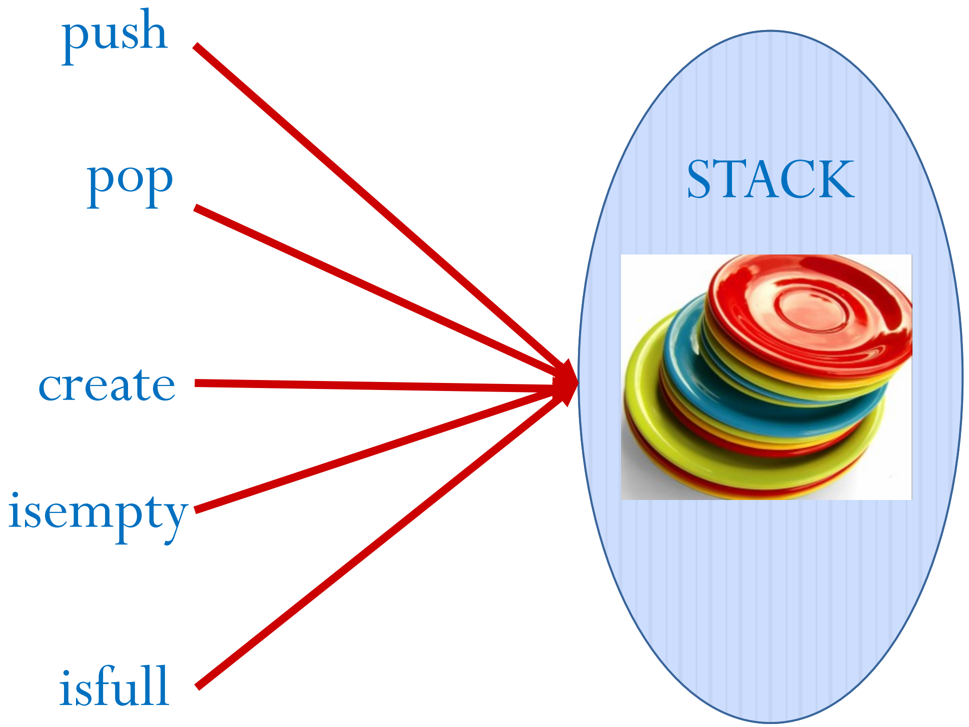
Stack Representation

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack,
- for example – a deck of cards or a pile of plates, etc.

Stack Representation

- Can be implemented by means of Array, Structure, Pointers and Linked List.
- Stack can either be a fixed size or dynamic.



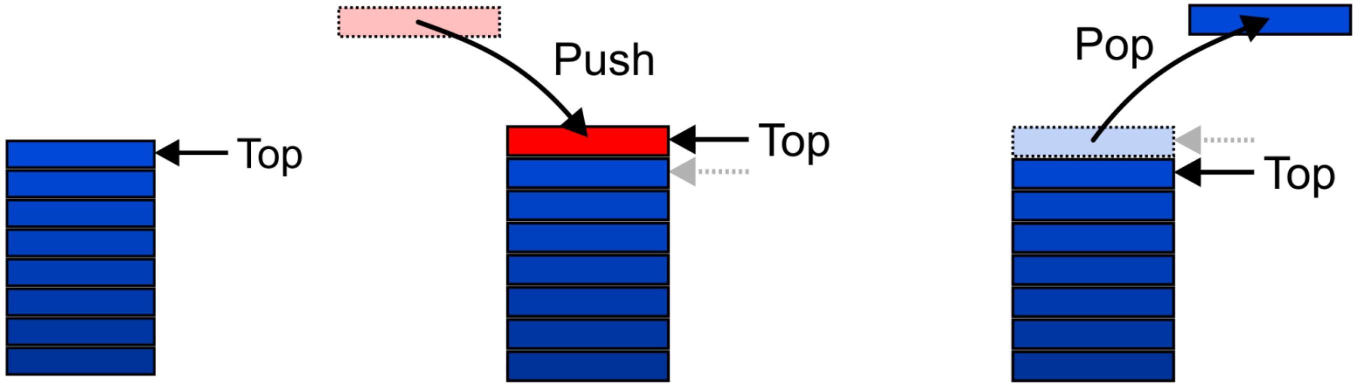


Stack

- Stack is an abstract data type which emphasizes specific operations:
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually
  - Inserted objects are pushed onto the stack
  - The top of the stack is the most recently object pushed onto the stack
  - When an object is popped from the stack, the current top is erased

Stack

- Also called a last-in–first-out (LIFO) behaviour
  - Graphically, we may view these operations as follows:

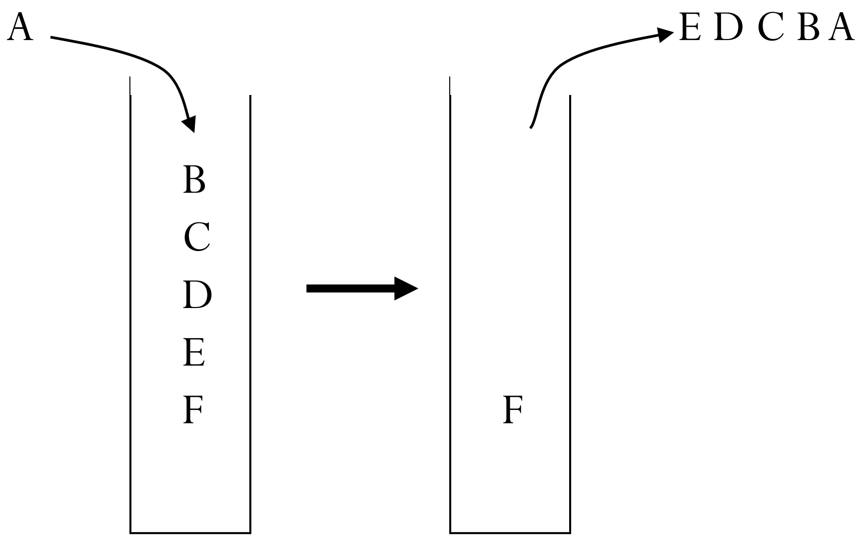


- There are two exceptions associated with abstract stacks:
  - It is an undefined operation to call either pop or top on an empty stack

LIFO Stack ADT

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty
- Stack property: if x is on the stack before y is pushed, then x will be popped after y is popped

  LIFO: Last In First Out



STACK: Last-In-First-Out (LIFO)

- void create (stack *s);

  /* Create a new stack */
- void push (stack *s, int element);

  /* Insert an element in the stack */
- int pop (stack *s);

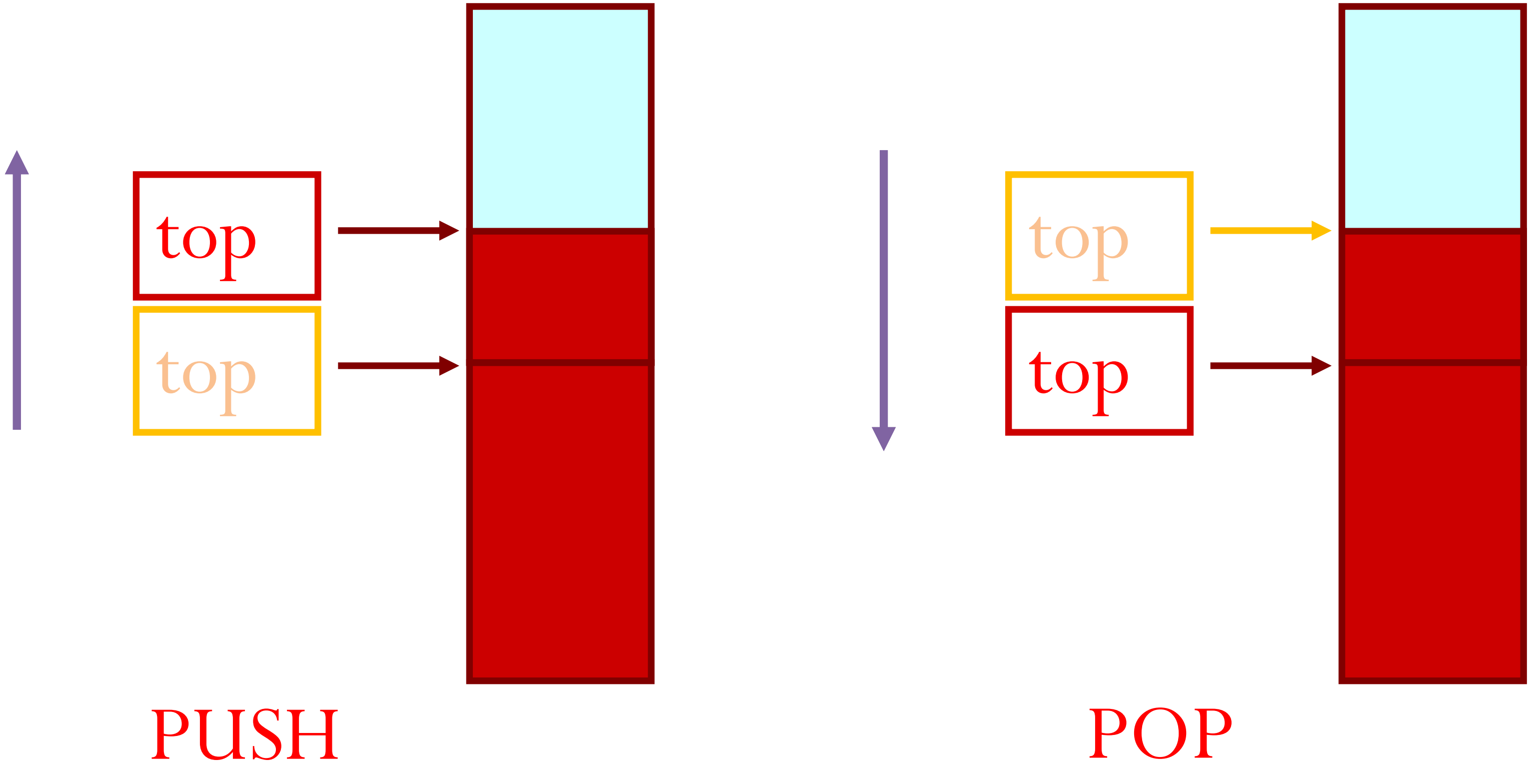
  /* Remove and return the top element */
- int isempty (stack *s);

  /* Check if stack is empty */
- int isfull (stack *s);

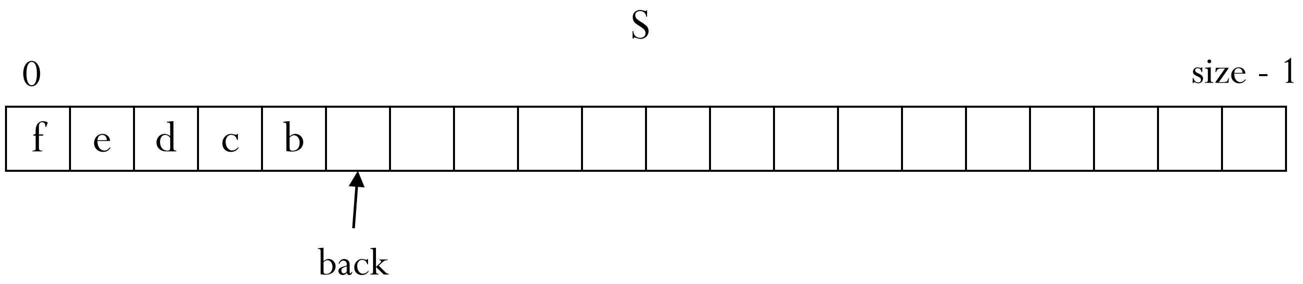
  /* Check if stack is full */

<u>Assumption: stack contains integer elements!</u>

Operations in Stack: Push and Pop



Array Stack Data Structure

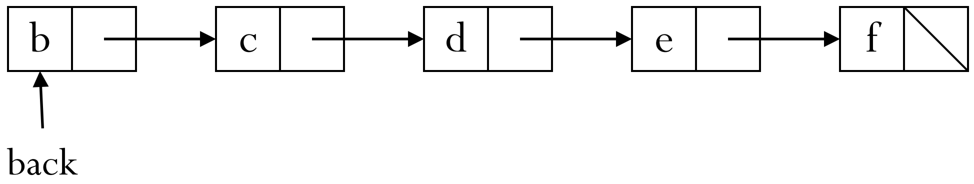



```
void push(Object x) {
  assert(!is_full())
  S[back] = x
  back++
}
Object top() {
  assert(!is_empty())
  return S[back - 1]
}
```

```
Object pop() {
  assert(!is_empty())
  back--
  return S[back]
}

bool is_full() {
  return back == size
}
```

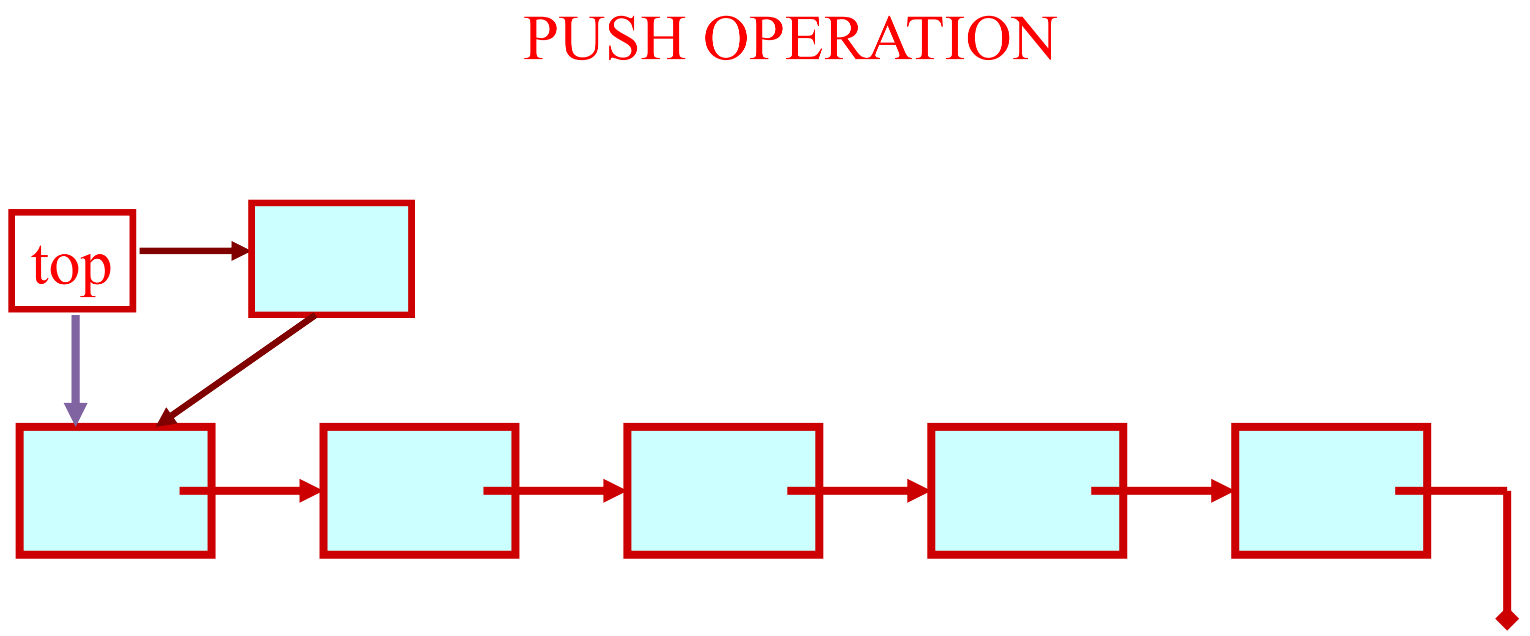
Linked List Stack Data Structure



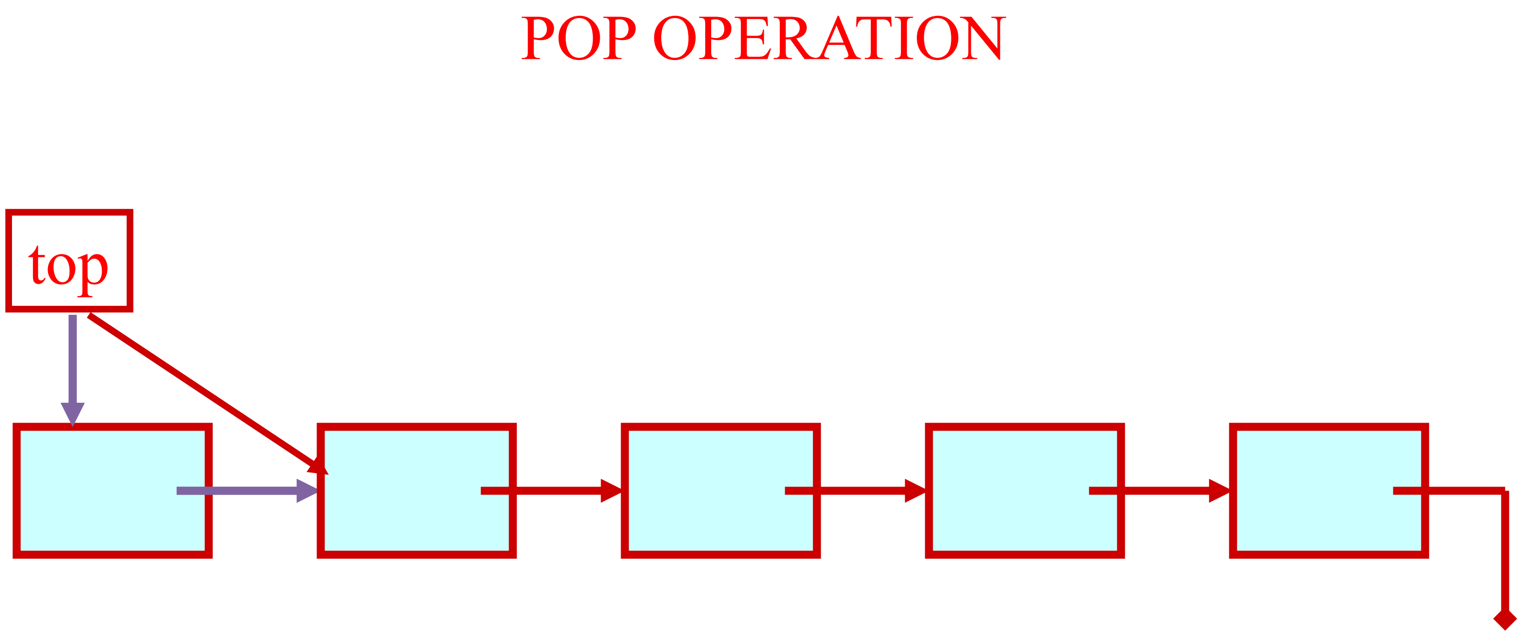

```
void push(Object x) {
   temp = back
   back = new Node(x)
   back->next = temp
}
Object top() {
   assert(!is_empty())
   return back->data
}
```

```
Object pop() {
   assert(!is_empty())
   return_data = back->data
   temp = back
   back = back->next
   free(temp)
   return return_data
}
```

Push using Linked List



Pop using Linked List



Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the "top" of the stack.
  - Contains the array index of the "top" element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100
struct lifo
{
  int st[MAXSIZE];
  int  top;
};
typedef struct lifo
              stack;
stack s;
```

ARRAY

```
struct lifo
{
  int value;
  struct lifo *next;
};
typedef struct lifo
              stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
  s->top = -1;

  /* s->top points to
     last element
     pushed in;
     initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
  *top = NULL;

  /* top points to NULL,
     indicating empty
     stack          */
}
```

LINKED LIST

Pushing an element into stack

```
void push (stack *s, int element)
 {
 if (s->top == (MAXSIZE-1))
  {
   printf ("\n Stack overflow");
      exit(-1);
   }
   else
   {
      s->top++;
      s->st[s->top] = element;
   }
 }
```

ARRAY

```
void push (stack **top, int element)
{
   stack *new;

   new = (stack *)malloc (sizeof(stack));
   if (new == NULL)
   {
     printf ("\n Stack is full");
     exit(-1);
   }

   new->value = element;
   new->next = *top;
   *top = new;
}
```

LINKED LIST

Popping an element from stack

```
int pop (stack *s)
  {
    if (s->top == -1)
    {
      printf ("\n Stack underflow");
      exit(-1);
    }
    else
    {
      return (s->st[s->top--]);
    }
  }
```

ARRAY

```
int pop (stack **top)
{
  int t;
  stack *p;

  if (*top == NULL)
  { printf ("\n Stack is empty");
    exit(-1);
  }
  else
  { t = (*top)->value;
    p = *top;
    *top = (*top)->next;
    free (p);
    return t;
  }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
  if (s->top == -1)
        return 1;
  else
        return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
  if (top == NULL)
        return (1);
  else
        return (0);
}
```

LINKED LIST

Example: Stack in C using Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf (“%d %d”, pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf (“\n B is empty”);
  return;
}
```

Arithmetic Expression Evaluation

Arithmetic Expressions

- Infix Notation: operators placed between operands:

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

- Prefix (Polish) Notation: operands appear before their operators:-

  +AB, -CD, *EF, /GH

- Post fix (Reverse Polish) Notation:

  AB+, CD-, EF*, GH/

Infix and Postfix Notations

- Infix: operators placed between operands:

  A+B*C

- Postfix: operands appear before their operators:-

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

A + B * C → (A + (B * C)) → (A + (B C *)) → A B C * +

A + B * C + D → ((A + (B * C)) + D) → ((A + (B C*))+ D) → ((A B C *+) + D) → A B C * + D +

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

Infix to Postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Infix Notation

- Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

- The operator is placed between to operands

- One weakness: parentheses are required

$$\begin{aligned} (3 + 4) \times 5 - 6 &= 29 \\ 3 + 4 \times 5 - 6 &= 17 \\ 3 + 4 \times (5 - 6) &= -1 \\ (3 + 4) \times (5 - 6) &= -7 \end{aligned}$$

Postfix Notation

- Also known as Reverse-Polish Notation
- Place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3\ 4\ +\ 5\ \times\ 6\ -$$

- Parsing reads left-to-right and performs any operation on the last two operands:

$$3\ 4\ +\ 5\ \times\ 6\ -$$

$$7\ \ 5\ \times\ 6\ -$$

$$35\ \ 6\ -$$

$$29$$

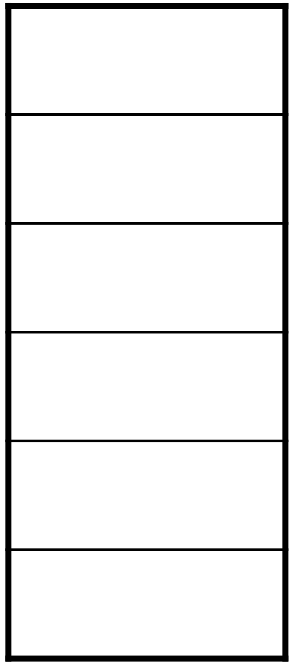
Postfix Notation

- The easiest way to parse Postfix Notation is to use an operand stack:
  - operands are processed by pushing them onto the stack
  - when processing an operator:
    - pop the last two items off the operand stack,
    - perform the operation, and
    - push the result back onto the stack

Postfix Notation

Evaluate the following reverse-Polish expression using a stack:

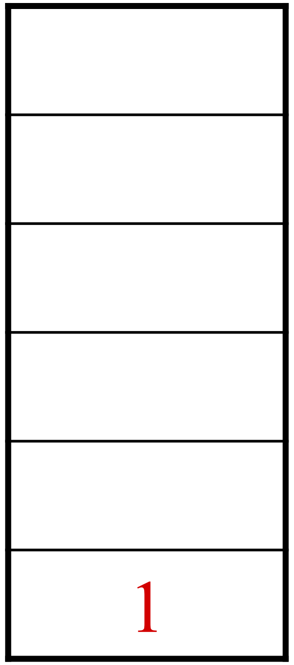
$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$



Postfix Notation

Push 1 onto the stack

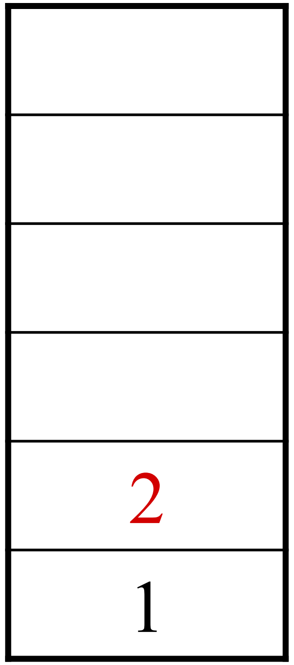
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 1 onto the stack

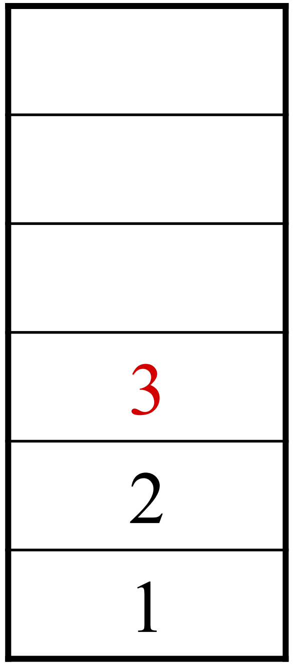
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Postfix Notation

Push 3 onto the stack

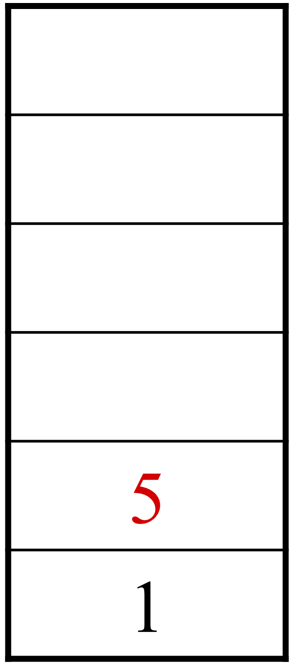
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop 3 and 2 and push $2 + 3 = 5$

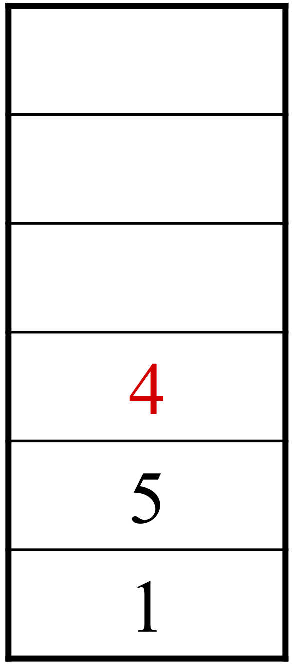
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$



Postfix Notation

Push 4 onto the stack

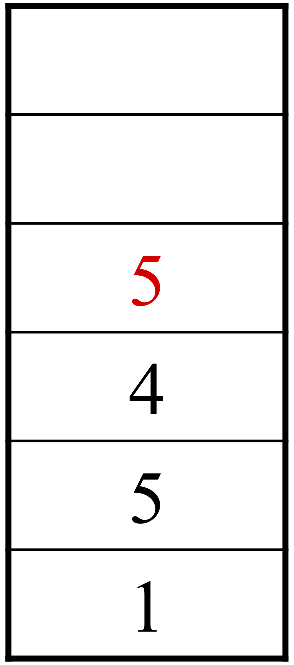
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 5 onto the stack

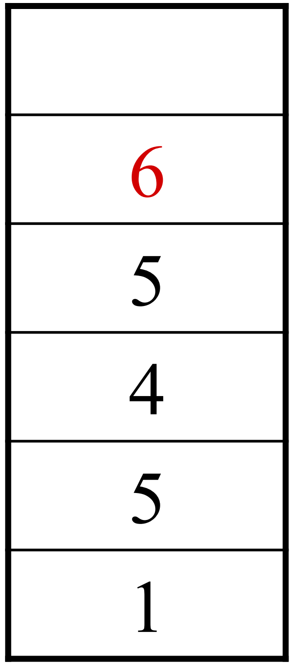
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 6 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop 6 and 5 and push 5 × 6 = 30

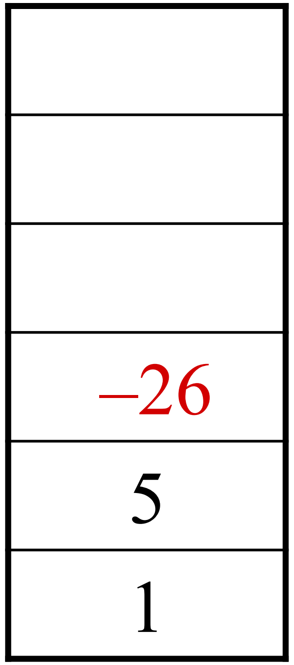
$$1 \;\; 2 \;\; 3 \;\; + \;\; 4 \;\; 5 \;\; 6 \;\; \times \;\; - \;\; 7 \;\; \times \;\; + \;\; - \;\; 8 \;\; 9 \;\; \times \;\; +$$

30
4
5
1

Postfix Notation

Pop 30 and 4 and push 4 – 30 = –26

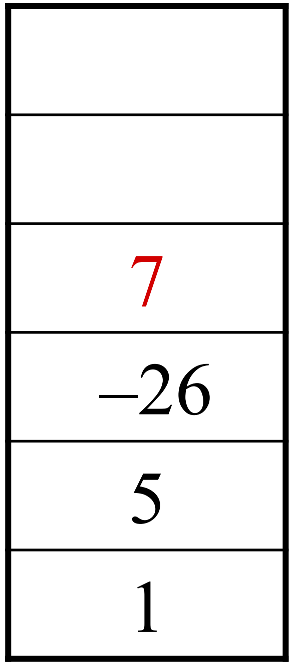
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 7 onto the stack

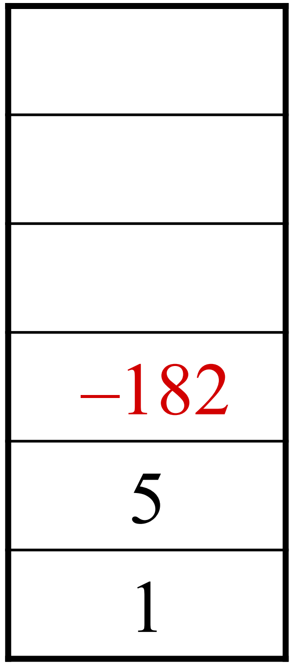
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop 7 and –26 and push –26 × 7 = –182

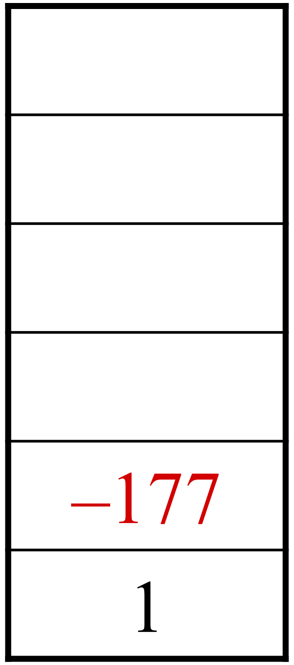
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop –182 and 5 and push –182 + 5 = –177

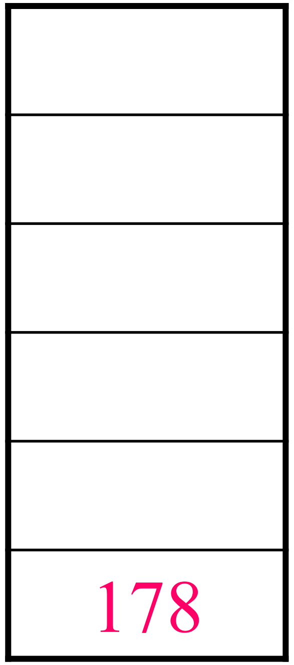
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop –177 and 1 and push 1 – (–177) = 178

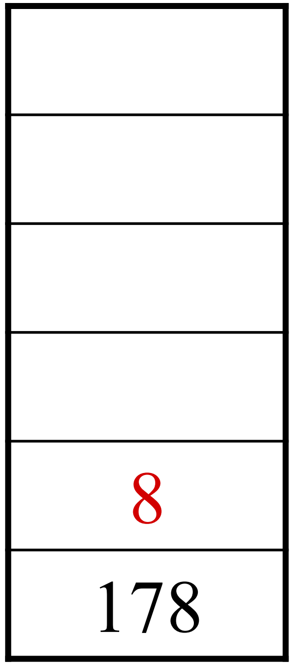
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 8 onto the stack

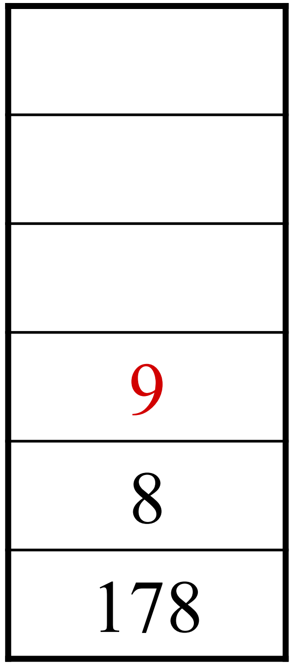
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Push 1 onto the stack

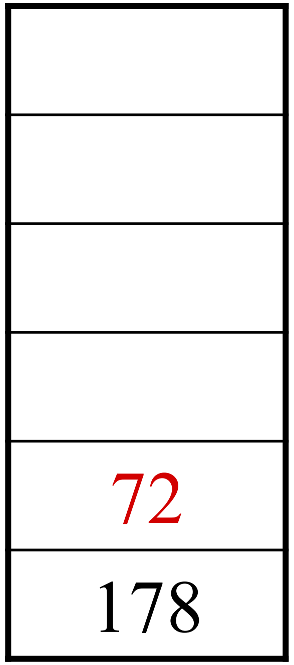
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop 9 and 8 and push 8 × 9 = 72

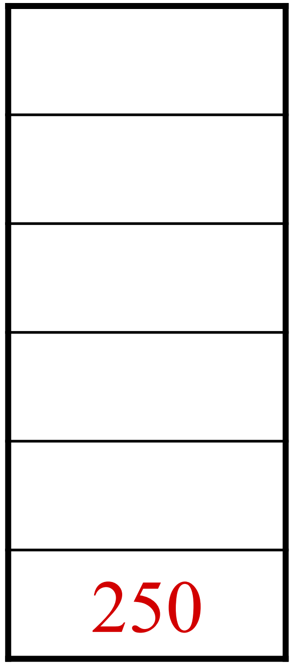
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix Notation

Pop 72 and 178 and push 178 + 72 = 250

$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$



Postfix Notation

- Thus

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

  evaluates to the value on the top: 250
- The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

- Reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Postfix Notation

- Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

  which has the Postfix Notation of

$$1\ \ 2\ -\ 3\ +\ 4\ +\ 5\ \ 6\ \ 7\ \times\ \times\ -\ 8\ \ 9\ \times\ +$$

- For comparison, the calculated expression was

$$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$$

- Benefits:
  - No ambiguity and no brackets are required
  - It is the same process used by a computer to perform computations:
    - operands must be loaded into registers before operations can be performed on them
  - Reverse-Polish can be processed using stacks

Applications of Stack

Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML
- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

Applications of Stacks

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution
- Parsing code: Tracking Function calls using stack
- Assembly language
- Balancing symbols (e.g. {parentheses}):
  - Matching parenthesis
  - XML (e.g., XHTML) tags
- Dealing with undo/redo operations
- Reverse-Polish calculators
- Removing recursion
- Evaluating Reverse Polish Notation
- Depth first search

Function Calls

- Problem solving:
  - Solving one problem may lead to subsequent problems
  - These problems may result in further problems
  - As problems are solved, your focus shifts back to the problem which lead to the solved problem

- Notice that function calls behave similarly:
  - A function is a collection of code which solves a problem

- Donald Knuth

Function Calls

- You will notice that the when a function returns, execution and the return value is passed back to the last function which was called
- This is again, the last-in—first-out property
- Today's CPUs have hardware specifically designed to facilitate function calling

Function Calls

- Function calls are similar to problem solving presented earlier:
  - you write a function to solve a problem
  - the function may require sub-problems to be solved, hence, it may call another function
  - once a function is finished, it returns to the function which called it
- This next example discusses function calls

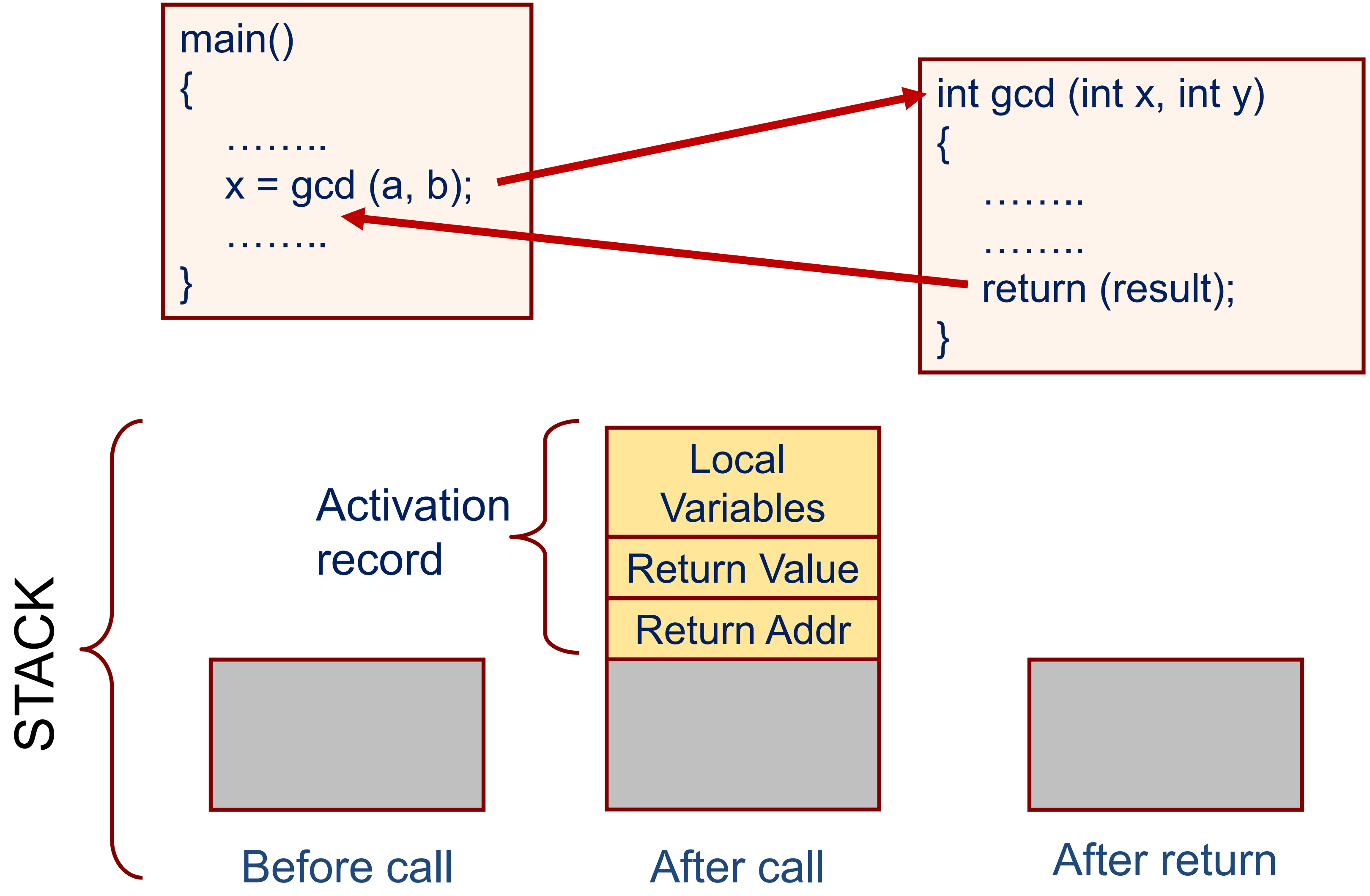
- In Digital Computers, stacks are implemented in hardware on all CPUs to facilitate function calling

- The simple features of a stack indicate why almost all programming languages are based on function calls

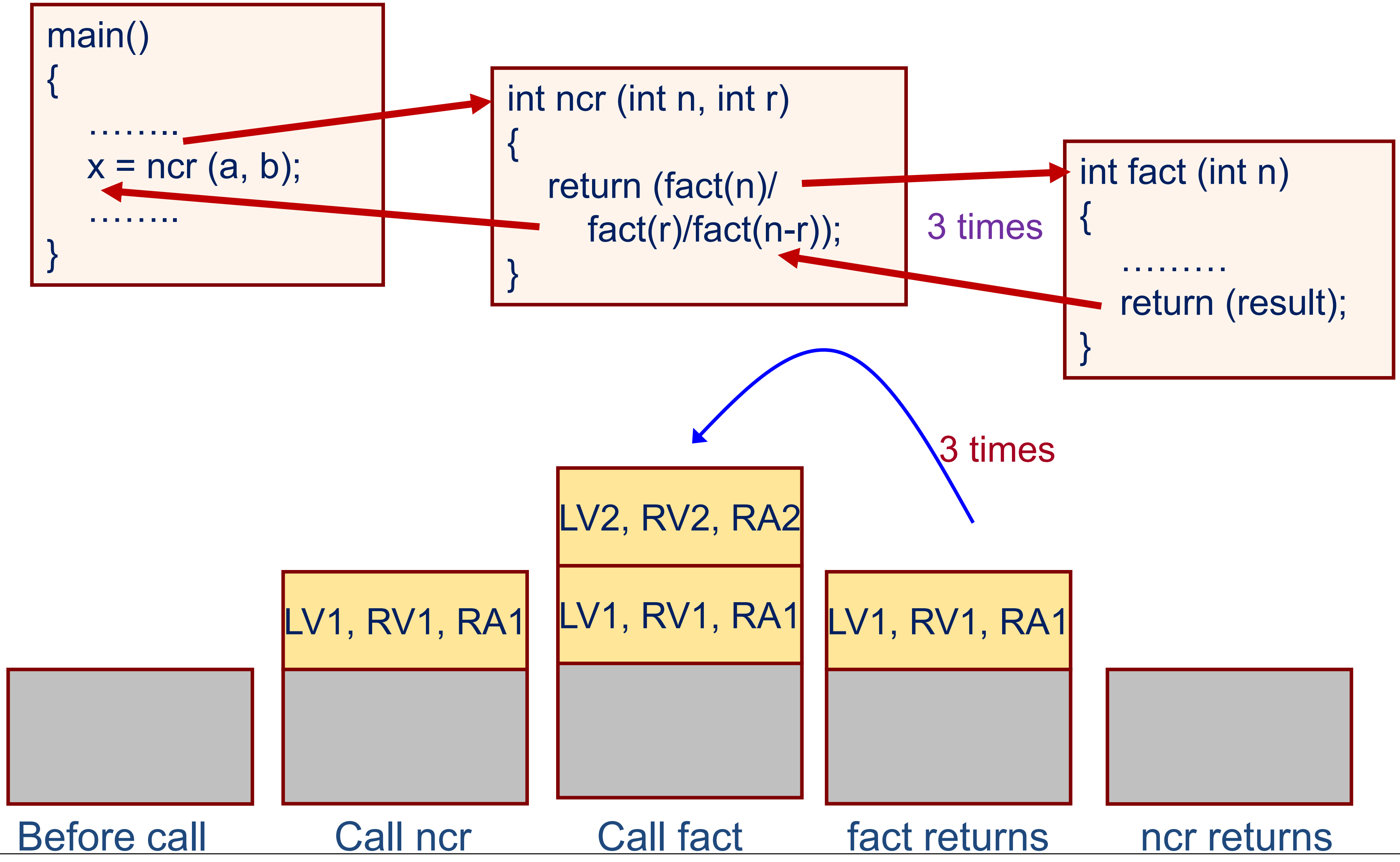
Function calls implementation

- The following applies in general, with minor variations that are implementation dependent.
- The system maintains a stack in memory.
  - Stack is a **last-in first-out** structure.
  - Two operations on stack, push and pop.
- Whenever there is a function call, the activation record gets pushed into the stack.
  - Activation record consists of the return address in the calling program, the return value from the function, and the local variables inside the function.

Function calls implementation



Function calls implementation



Recursive calls

- **What we have seen ….**
  - Activation record gets pushed into the stack when a function call is made.
  - Activation record is popped off the stack when the function returns.

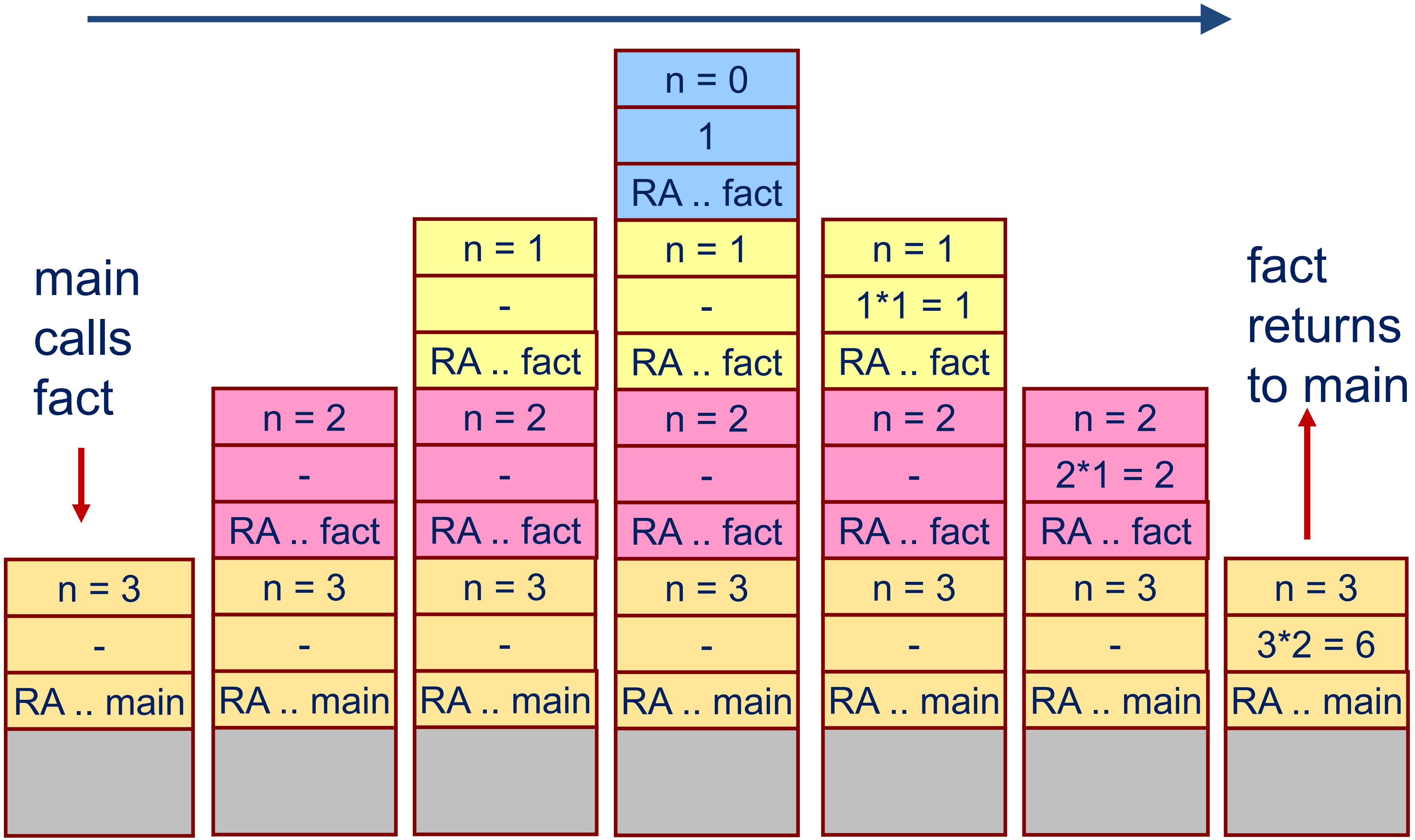
- **In recursion, a function calls itself.**
  - Several function calls going on, with none of the function calls returning back.
    - Activation records are pushed onto the stack continuously.
    - Large stack space required.
  - Activation records keep popping off, when the termination condition of recursion is reached.

Example:: main() calls fact(3)

```
main()
{
    int  n;
    n = 3;
    printf ("%d \n", fact(n) );
}
```

```
int  fact (n)
int  n;
{
     if   (n = = 0)
          return (1);
     else
          return(n * fact(n-1));
}
```

Trace of the Stack During Execution



Trace of the Stack During Execution

```
#include <stdio.h>
int   f (int n)
{
    int a, b;
    if  (n  < 2)   return (n);
    else  {
      a = f(n-1);
      b = f(n-2);
      return (a+b);  }
}

main() {
   printf(“Fib(4) is: %d \n”, f(4));
}
```

X

Y

Local Variables (n, a, b)
Return Value
Return Addr (either main, or X, or Y)

Trace the activation records for the following version of Fibonacci sequence.

Application: Parsing

- Most parsing uses stacks

- Examples includes:
  - Matching tags in XHTML
  - In C++, matching
    - parentheses **(...)**
    - brackets, and **[...]**
    - braces **{ ... }**

Parsing XHTML

- Example will demonstrate parsing XHTML
- Stacks can be used to parse an XHTML document
- Use XHTML (and more generally XML and other markup languages) in the workplace

Parsing XHTML

- A *markup language* is a means of annotating a document to given context to the text
  - The annotations give information about the structure or presentation of the text

- The best known example is HTML, or HyperText Markup Language
  - Look at XHTML

Parsing XHTML

- XHTML is made of nested
  - *opening tags*, e.g., <some_identifier>, and
  - matching *closing tags*, e.g., </some_identifier>
    - <html>
    - <head><title>Hello</title></head>
    - <body><p>This appears in the <i>browser</i>.</p></body>
    - </html>

Parsing XHTML

- *Nesting* indicates that any closing tag must match the most recent opening tag

- Strategy for parsing XHTML:
  - read though the XHTML linearly
  - place the opening tags in a stack
  - when a closing tag is encountered, check that it matches what is on top of the stack and

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

- Finish parsing → the stack is empty
- Possible errors:
  - a closing tag which does not match the opening tag on top of the stack
  - a closing tag when the stack is empty
  - the stack is not empty at the end of the document

HTML

- Old HTML required neither closing tags nor nesting

```
<html>
  <head><title>Hello</title></head>
  <body><p>This is a list of topics:
  <ol>                    <!-- para ends with start of list -->
    <li><i>veni                        <!-- implied </li> -->
    <li>vidi                       <!-- italics continues -->
    <li>vici</i>
  </ol>              <!-- end-of-file implies </body></html> -->
```

- Parsers were therefore specific to HTML
  - Results: ambiguities and inconsistencies

XML

- XHTML is an implementation of XML
- XML defines a class of general-purpose *eXtensible Markup Languages* designed for sharing information between systems
- The same rules apply for any flavour of XML:
  - opening and closing tags must match and be nested

Parsing C++

- Example shows how stacks may be used in parsing C++
- It should help understand, in part:
  - how a compiler works, and
  - why programming languages have the structure they do
- Like opening and closing tags, C++ parentheses, brackets, and braces must be similarly nested:

```
void initialize( int *array, int n ) {
    for ( int i = 0; i < n; ++i ) {
        array[i] = 0;
    }
}
```

Parsing C++

- For C++, the errors are similar to that for XHTML, however:
  - many XHTML parsers usually attempt to “correct” errors (e.g., insert missing tags)
  - C++ compilers will simply issue a parse error:

```
{eceunix:1} cat example1.cpp
#include <vector>
int main() {
        std::vector<int> v(100];
        return 0;
}
```

- {eceunix:2} g++ example1.cpp
- example1.cpp: In function 'int main()':
- example1.cpp:3: error: expected ')' before ']' token

References

- Donald E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.
- Cormen, Leiserson, and Rivest, Introduction to Algorithms, McGraw Hill, 1990, §11.1, p.200.
- Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §3.3.1, p.75.
- Koffman and Wolfgang, "Objects, Abstraction, Data Structures and Design using C++", John Wiley & Sons, Inc..
- Wikipedia, http://en.wikipedia.org/wiki/Double-ended_queue
- CSE326: Data Structure, Department of Computer Science and Engineering, University of Washington https://courses.cs.washington.edu/courses/cse326
- Mike Scott, CS 307 Fundamentals of Computer Science, https://www.cs.utexas.edu/~scottm/cs307/
- Debasis Samanta, Computer Science & Engineering, Indian Institute of Technology Kharagpur, Spring-2017, Programming and Data Structures. https://cse.iitkgp.ac.in/~dsamanta/courses/pds/index.html

תודה רבה
Hebrew

Ευχαριστώ
Greek

Спасибо
Russian

Danke
German

धन्यवादः
Sanskrit

Merci
French

ধন্যবাদ
Bangla

நன்றி
Tamil

شكراً
Arabic

ಧನ್ಯವಾದಗಳು
Kannada

Thank You
English

നന്ദി
Malayalam

Grazie
Italian

多謝
Traditional Chinese

ధన్యవాదాలు
Telugu

આભાર
Gujarati

ਧੰਨਵਾਦ
Punjabi

धन्यवाद
Hindi & Marathi

Gracias
Spanish

多谢
Simplified Chinese

https://sites.google.com/site/animeshchaturvedi07

Obrigado
Portuguese

ありがとうございました
Japanese

ขอบคุณ
Thai

감사합니다
Korean