MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

18 Stacks

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 3, 2022

Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

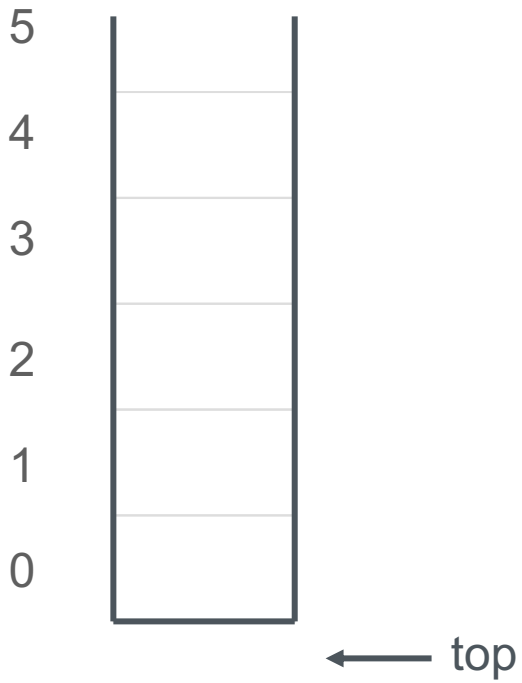
push (B)

push (C)

push (D)

push (E)

pop ()

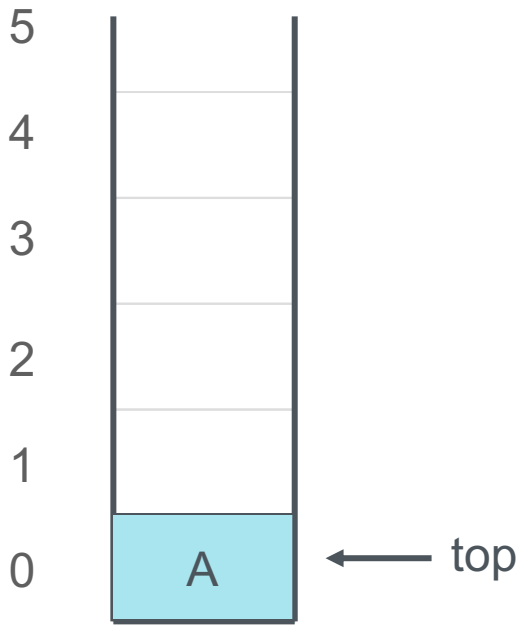


Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

- push (A)
- push (B)
- push (C)
- push (D)
- push (E)
- pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

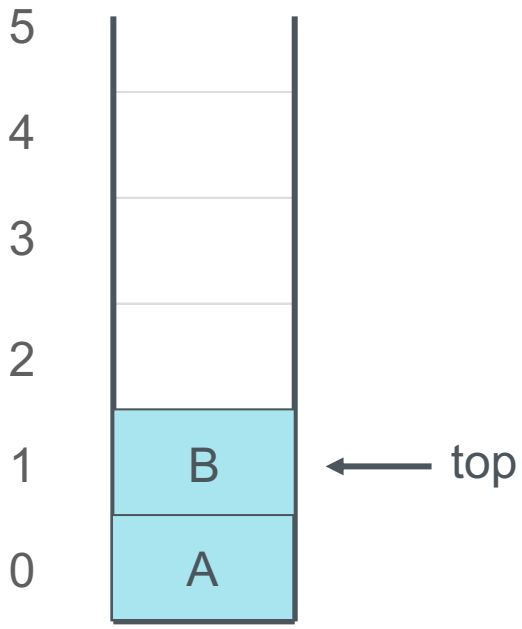
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

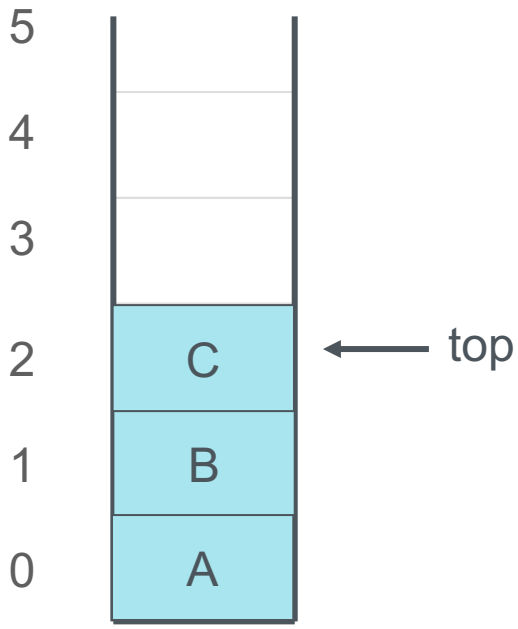
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

push (B)

push (C)

push (D)

push (E)

pop ()

Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

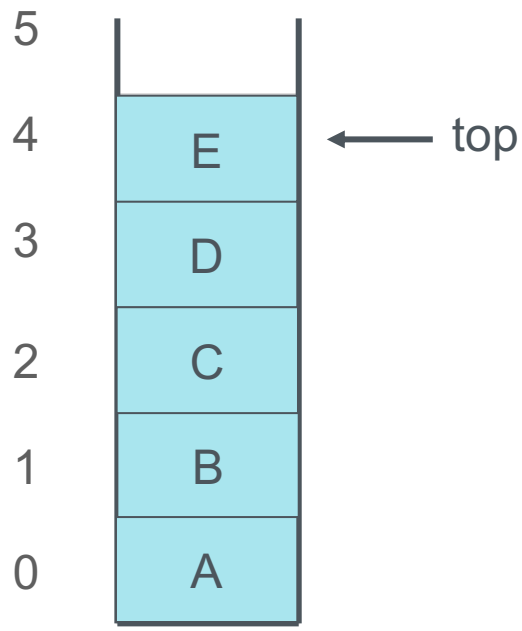
push (B)

push (C)

push (D)

push (E)

pop ()



Stack

- A stack is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

"push" adds a new item on top of the stack.

push (A)

push (B)

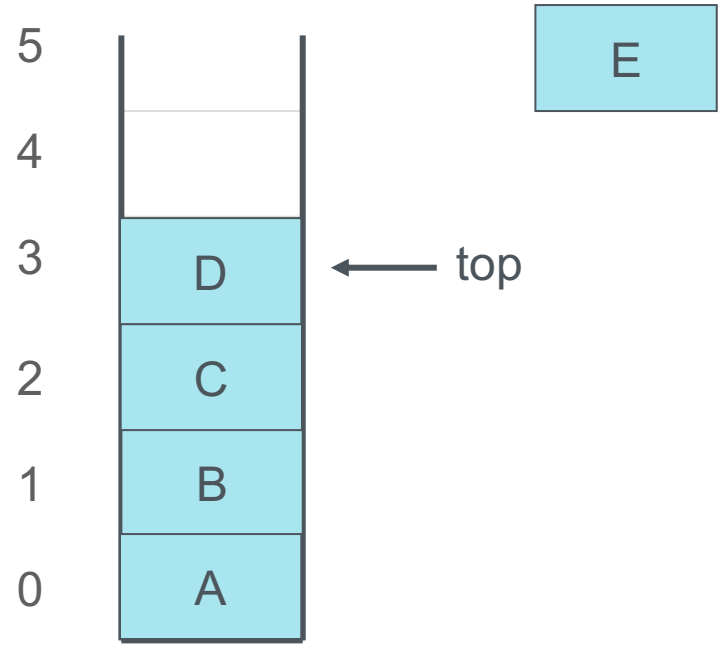
push (C)

push (D)

push (E)

- pop ()

Remove an item from the top of stack.



Stack and its Applications

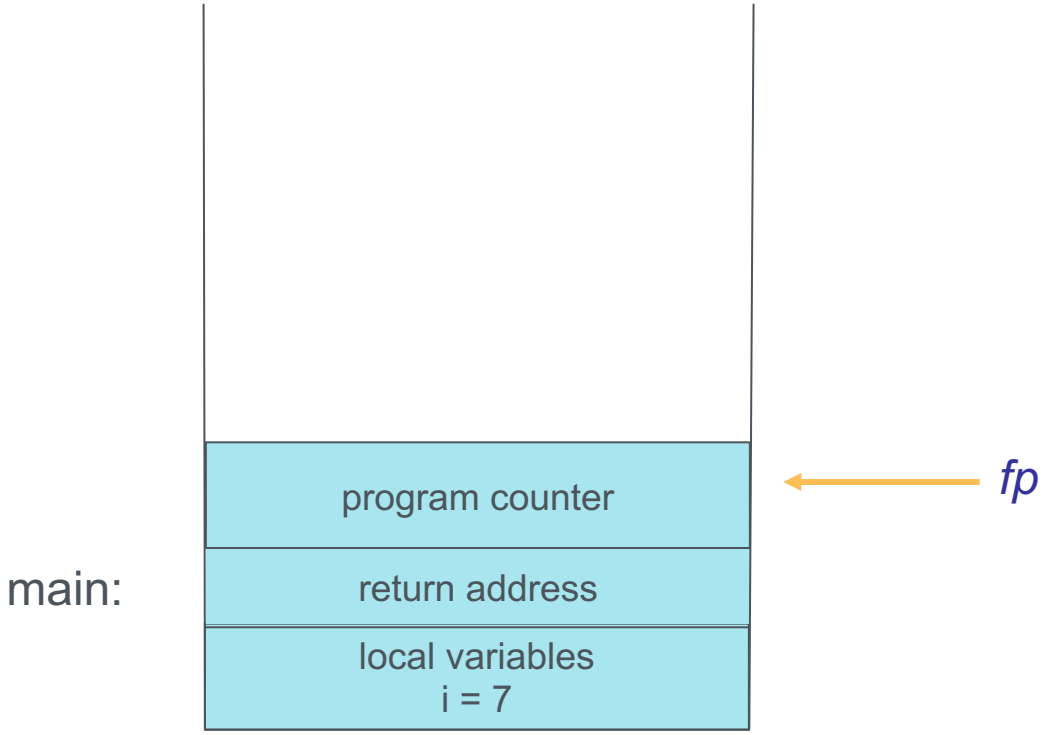
- Applications
  - History of visited pages in a web browser
  - Sequence of Undo operations in a text editor
  - Keep track of function calls in the C++ run-time system (System Stack)
  - As a data structure for algorithms to solve problems
    - Example: Reversing a list
  - Component of other data structures

Stack and its Applications

- System Stack
  - The C++ run-time system keeps track of the chain of active function calls with a stack.
- When a function is called, the system pushes a "**stack frame**" onto the stack which contains:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion

Stack and its Applications

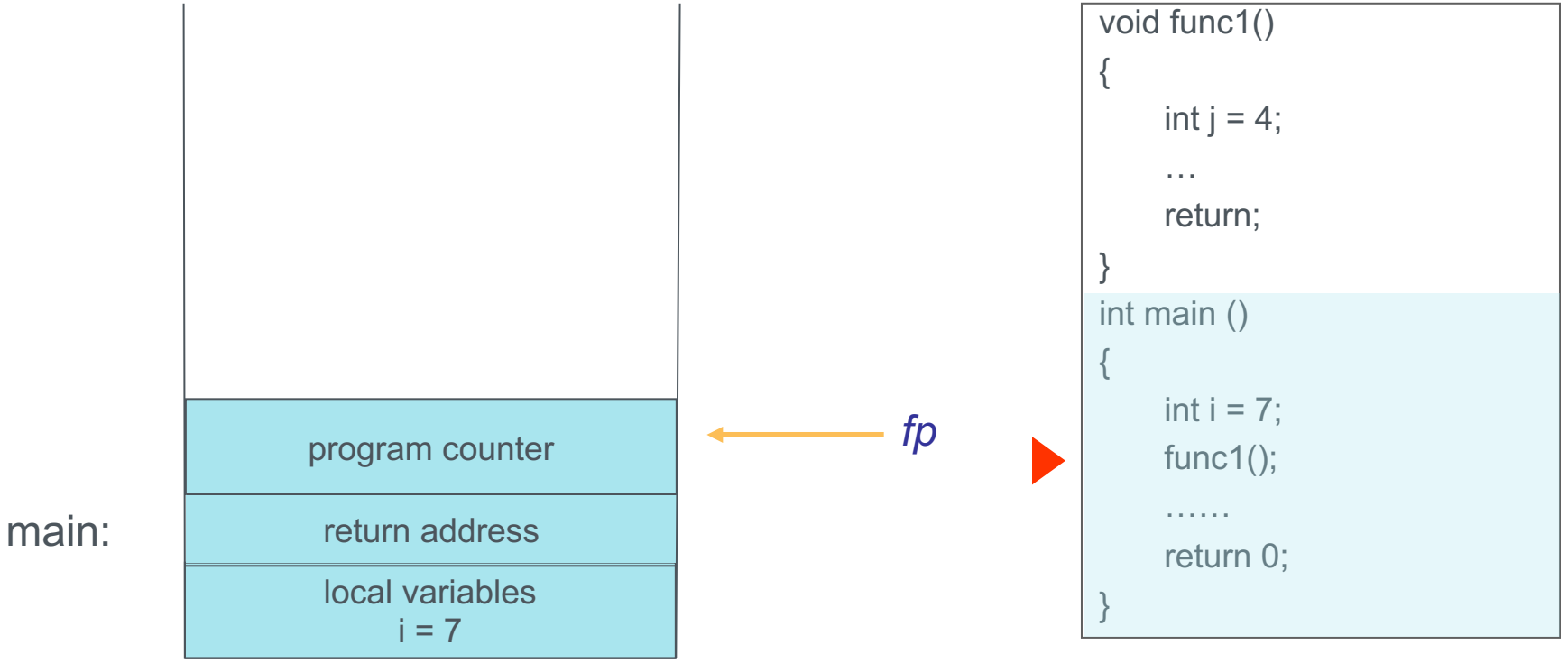
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

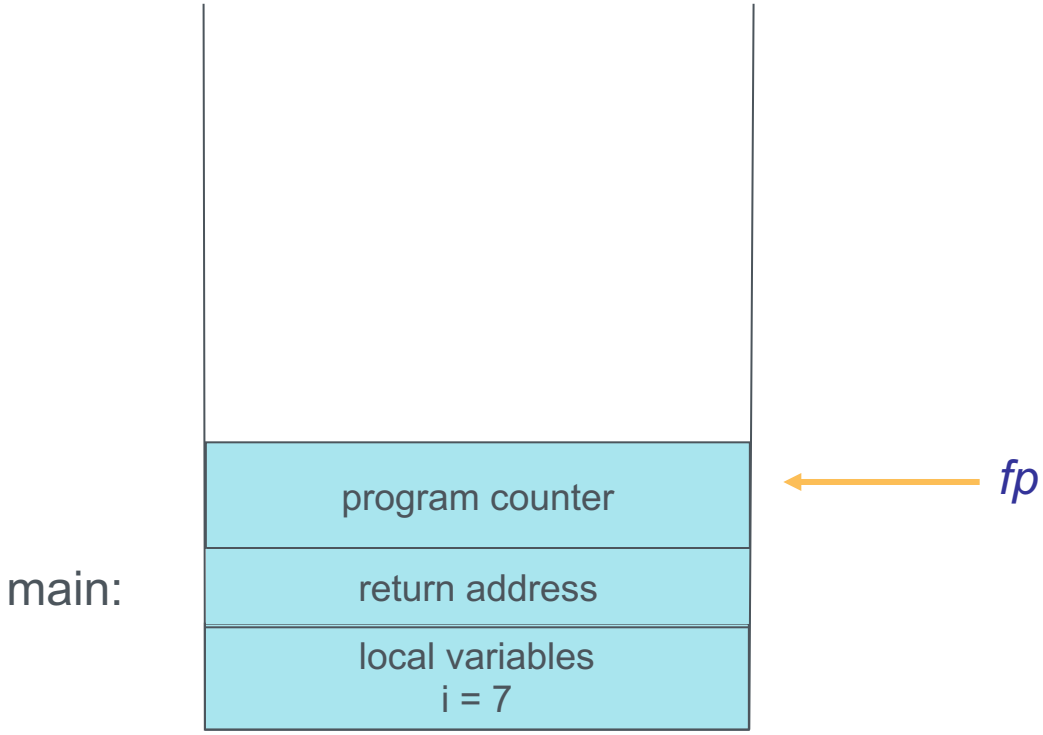
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



Stack and its Applications

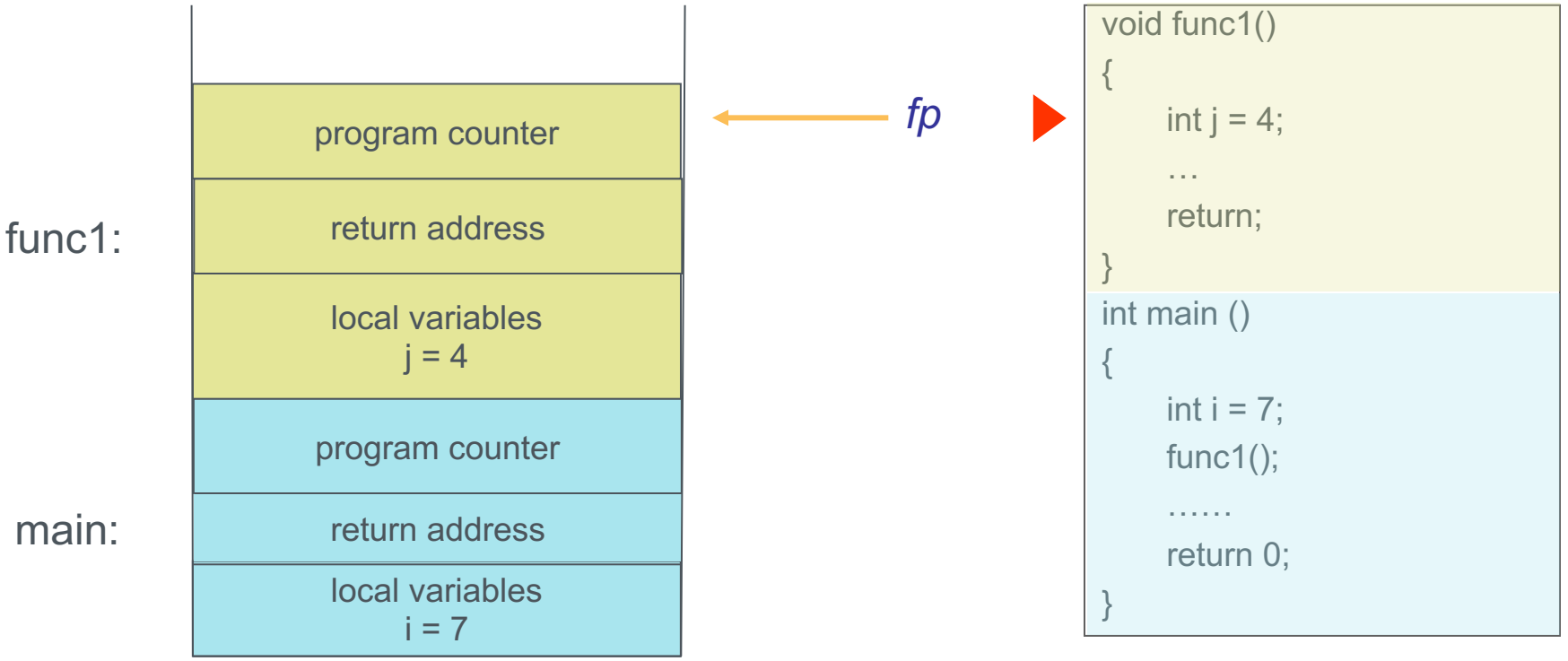
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

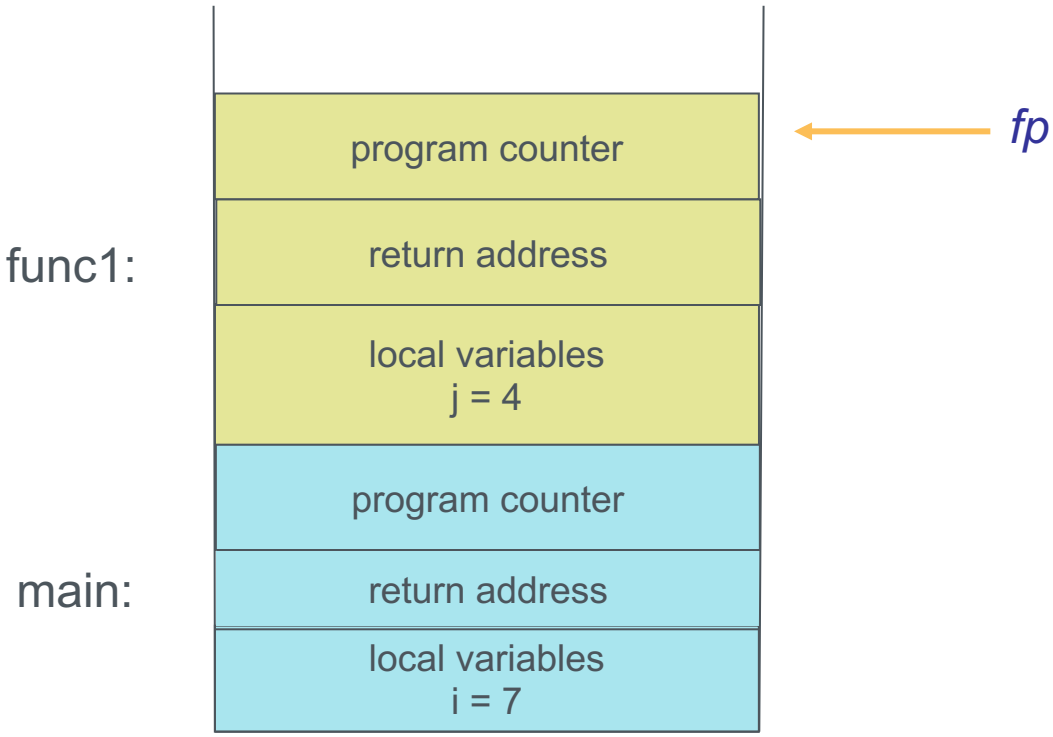
Stack and its Applications

- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



Stack and its Applications

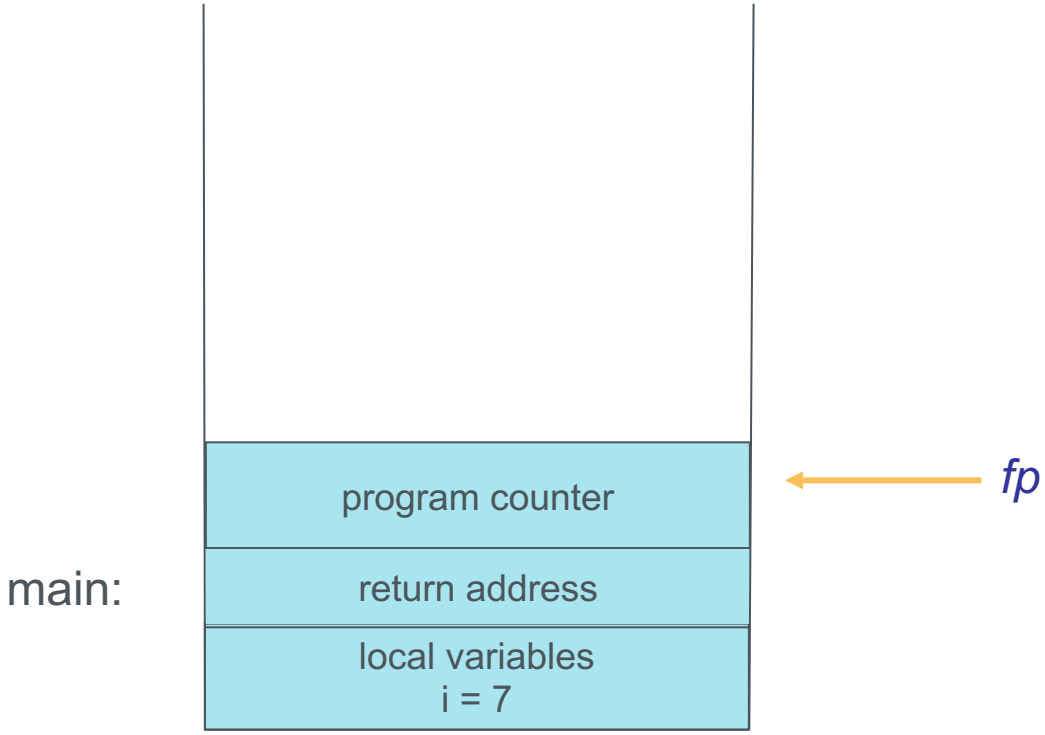
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

Stack and its Applications

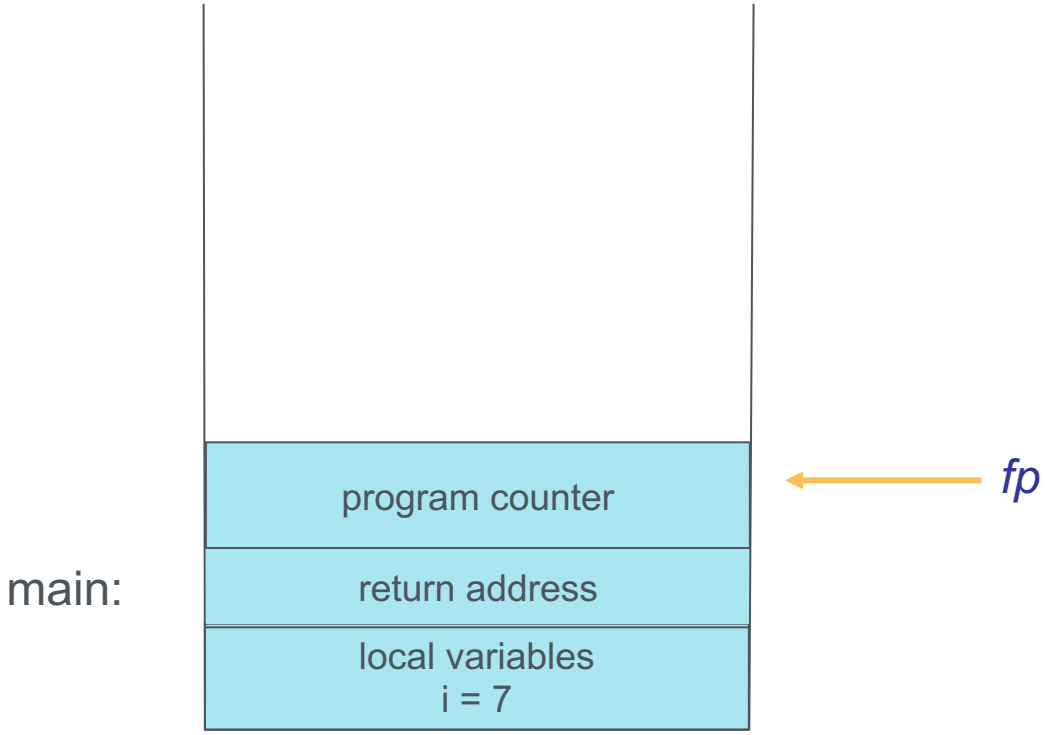
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

Stack and its Applications

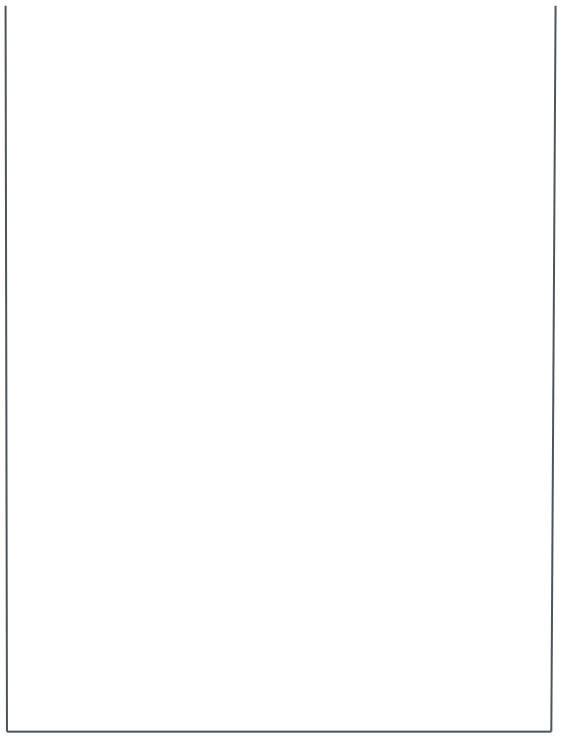
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed




```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

Stack and its Applications

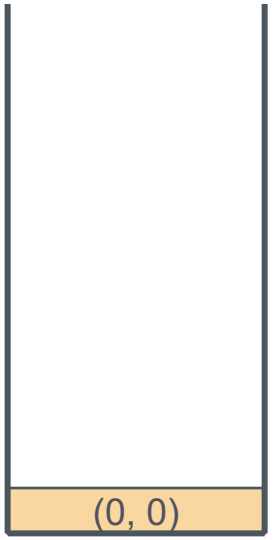
- System Stack
  - "**stack frame**" contains:
    - Local variables and return value
    - Program counter, keeping track of the statement being executed



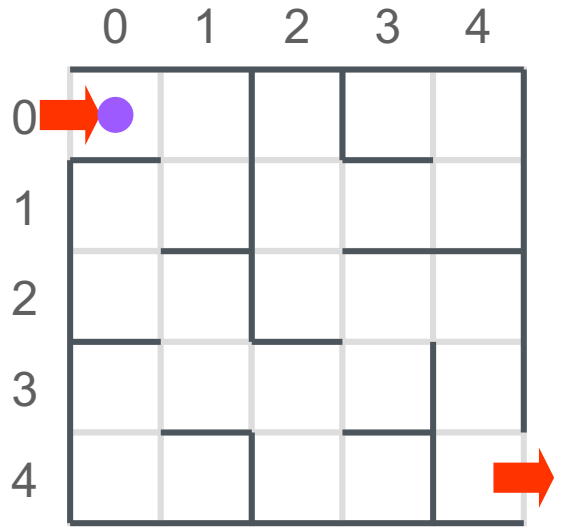

```
void func1()
{
      int j = 4;
      …
      return;
}
int main ()
{
      int i = 7;
      func1();
      ……
      return 0;
}
```

Stack and its Applications

- Path Planning for a Robot

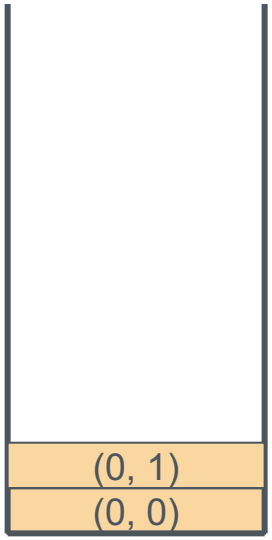


Stack

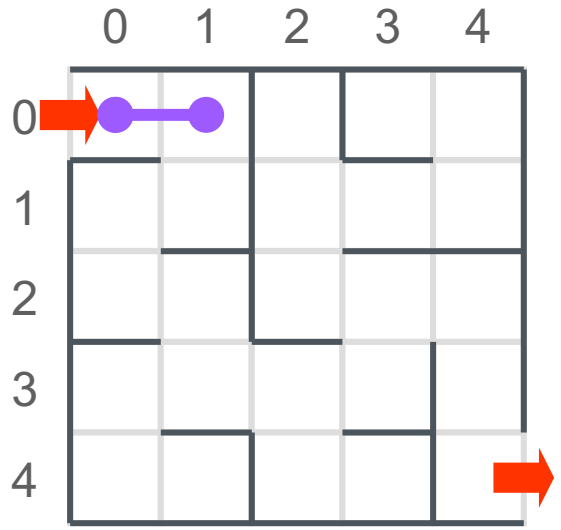


Stack and its Applications

- Path Planning for a Robot

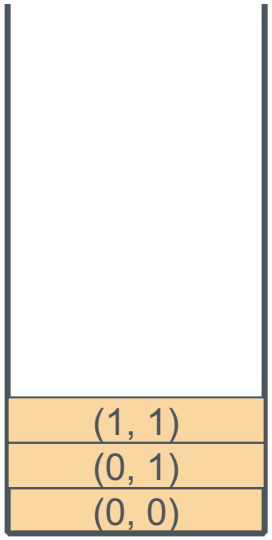


Stack

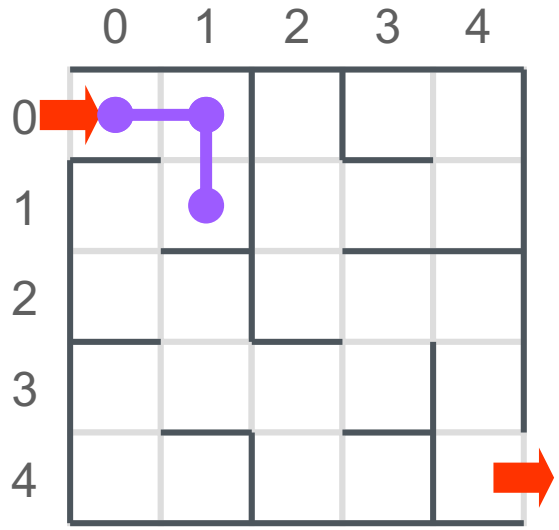


Stack and its Applications

- Path Planning for a Robot

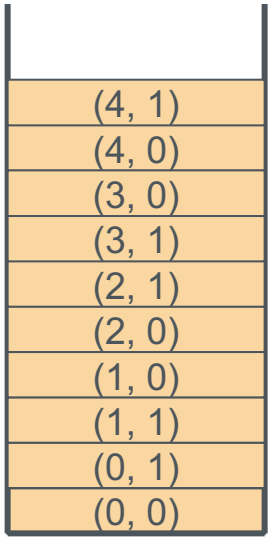


Stack

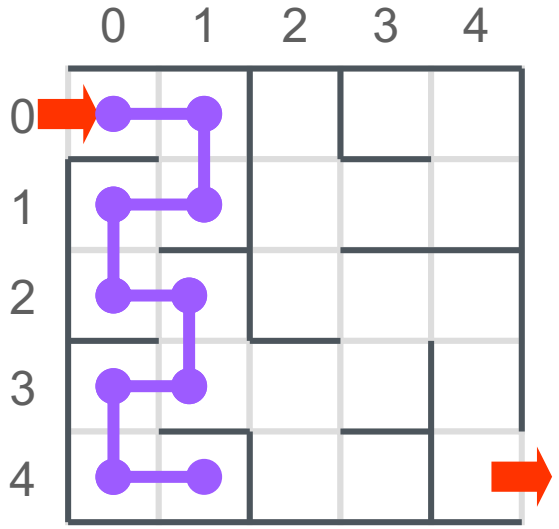


Stack and its Applications

- Path Planning for a Robot

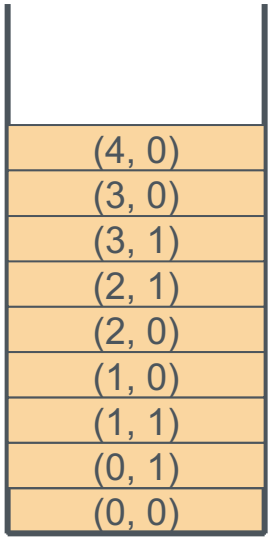


Stack

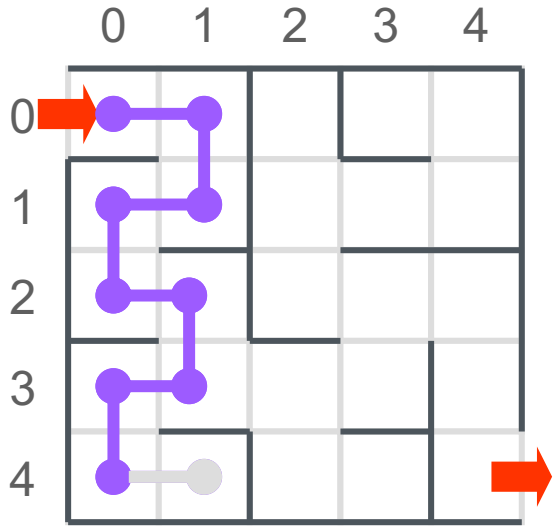


Stack and its Applications

- Path Planning for a Robot

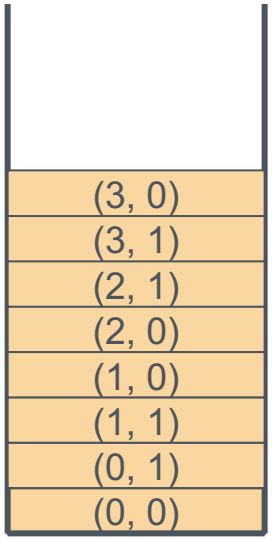


Stack

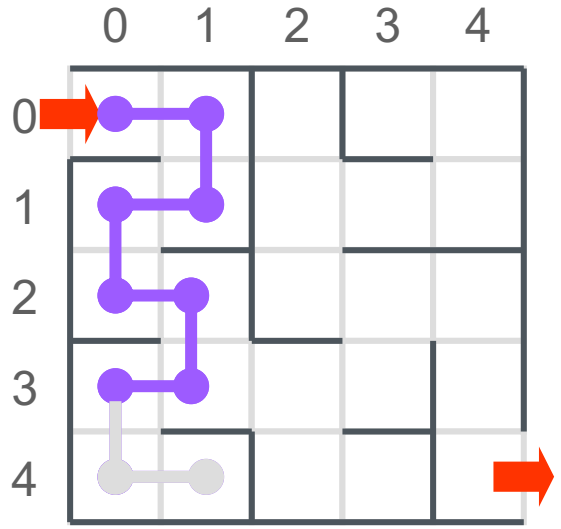


Stack and its Applications

- Path Planning for a Robot

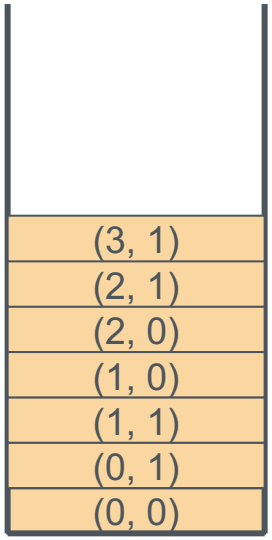


Stack

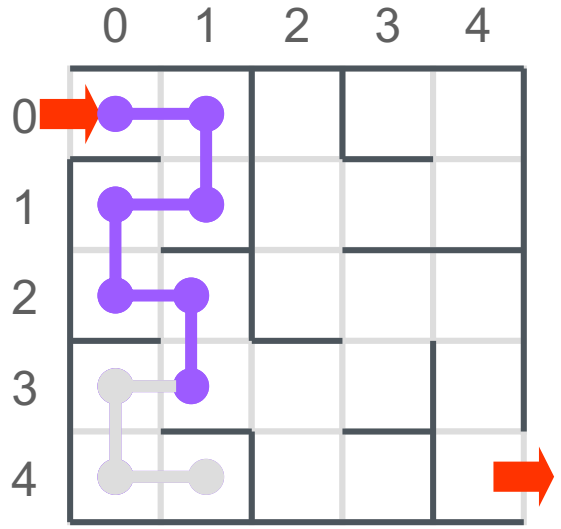


Stack and its Applications

- Path Planning for a Robot

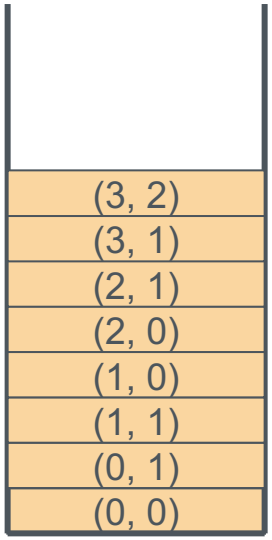


Stack

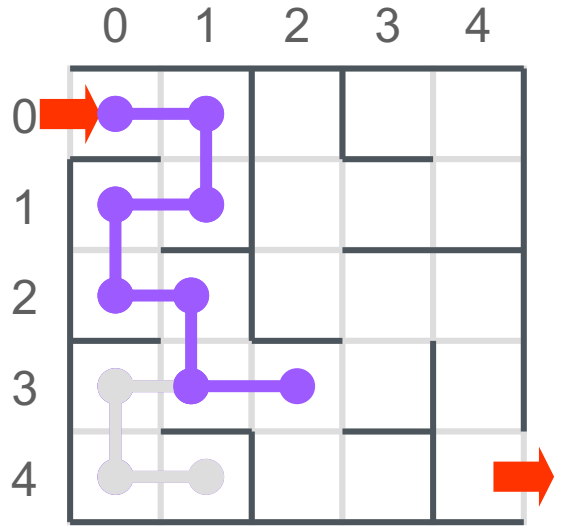


Stack and its Applications

- Path Planning for a Robot



Stack



ADT

- An ADT is a mathematical model (abstraction) of a data structure that specifies
  - the type of the data stored
  - the operations supported on them
  - the types of the parameters of the operations.
  - Error conditions associated with operations
- An **ADT specifies what each operation does**, but **not how** it does it.

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element object
  - pop(): removes the top element
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored in stack
  - Boolean empty(): indicates whether no elements are stored

```
template <typename E>
class Stack {                                   // an interface for a stack
public:
  int size() const;                             // number of items in stack
  bool empty() const;                           // is the stack empty?
  const E& top() const throw(StackEmpty);       // the top element
  void push(const E& e);                        // push x onto the stack
  void pop() throw(StackEmpty);                 // remove the top element
};
```

Exceptions for an ADT

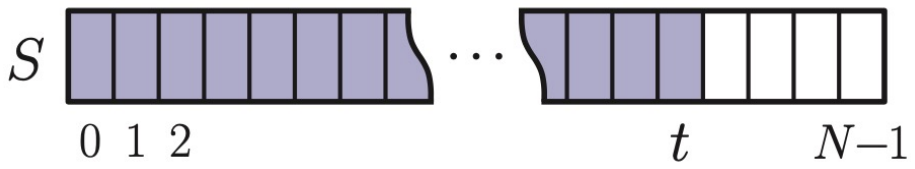
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "**thrown**" by an operation that cannot be executed

- In the Stack ADT, operations **pop** and **top** cannot be performed if the stack is empty
- Attempting **pop** or **top** on an empty stack throws a **StackEmpty** exception

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
  StackEmpty(const string& err) : RuntimeException(err) {}
};
```

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



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

Algorithm size():
    return $t + 1$
Algorithm empty():
    return $(t < 0)$
Algorithm top():
    if empty() **then**
        throw StackEmpty exception
    return $S[t]$
Algorithm push(e):
    if size() $= N$ **then**
        throw StackFull exception
    $t \leftarrow t + 1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() **then**
        throw StackEmpty exception
    $t \leftarrow t - 1$

Array-Based Stack - C++ Implementation

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);         // pop the stack
  // ...housekeeping functions omitted
private:                                // member data
  E* S;                                 // array of stack elements
  int capacity;                         // stack capacity
  int t;                                // index of the top of the stack
};
```

Array-Based Stack - C++ Implementation

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(−1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                      // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                      // is the stack empty?

template <typename E>                      // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                      // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                      // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  −−t;
}
```

Array-Based Stack - C++ Implementation

- Example use:

```
ArrayStack<int> A;                          // A = [], size = 0
A.push(7);                                  // A = [7*], size = 1
A.push(13);                                 // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 13
A.push(9);                                  // A = [7, 9*], size = 2
cout << A.top() << endl;                    // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 9
ArrayStack<string> B(10);                   // B = [], size = 0
B.push("Bob");                              // B = [Bob*], size = 1
B.push("Alice");                            // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();           // B = [Bob*], outputs: Alice
B.push("Eve");                              // B = [Bob, Eve*], size = 2
```

Array-Based Stack - Performance

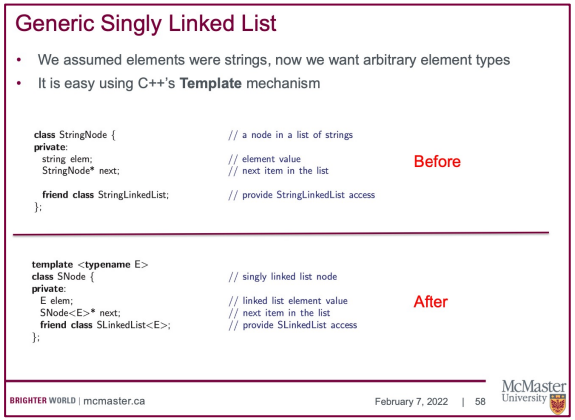
- Performance:
  - Let **n** be the number of elements in the stack
  - The space used is **O(n)**
  - Each operation runs in time **O(1)**

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

- 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

Generic Linked List-Based Implementation

- We have already developed:
  - Generic Singly Linked List (Lecture of Feb 7)
- We use that SLinkedList class
  - **S** stores the stack values
  - **n** stores the number of elements on stack




```
typedef string Elem;                          // stack element type
class LinkedStack {                           // stack as a linked list
public:
  LinkedStack();                              // constructor
  int size() const;                           // number of items in the stack
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:                                      // member data
  SLinkedList<Elem> S;                        // linked list of elements
  int n;                                      // number of elements
};
```

Generic Linked List-Based Implementation

```
LinkedStack::LinkedStack()
  : S(), n(0) { }                          // constructor

int LinkedStack::size() const
  { return n; }                            // number of items in the stack

bool LinkedStack::empty() const
  { return n == 0; }                       // is the stack empty?

                                           // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {    // push element onto stack
  ++n;
  S.addFront(e);
}
                                           // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Algorithm ParenMatch(X, n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.empty() then
            return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
            return false          {wrong type}
        S.pop()
if S.empty() then
    return true                   {every symbol matched}
else
    return false                  {some symbols were never matched}
```

Questions?