Data Structures (in C++)

- Stacks -

Jinsun Park

Visual Intelligence and Perception Lab., CSE, PNU

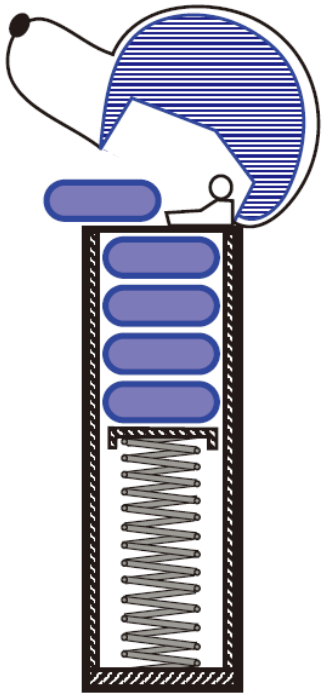
Stacks

Stacks

- **Stack**
  - A container of objects that are inserted and removed according to the **last-in first-out (LIFO)** principle
  - Objects can be inserted into a stack at any time
  - The most recently inserted (*i.e.*, the last) object can be removed from the stack

Example 5.1: *Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 5.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*



A dispenser schematic

Stack ADT

- A stack is an ADT that supports the following operations:

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

- Additional utility functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Stack Example

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

https://stackoverflow.com/a/4560763

```
==============     Highest Address (e.g. 0xFFFF)
|            |
|    STACK   |
|            |
|------------|  <- Stack Pointer   (e.g. 0xEEEE)
|            |
.     ...    .
|            |
|------------|  <- Heap Pointer    (e.g. 0x2222)
|            |
|    HEAP    |
|            |
==============     Lowest Address  (e.g. 0x0000)
```

Low Address

A stack grows from high to low (Platform dependent)

High Address

The STL Stack

- The STL stack implementation is based on the STL deque, vector, or list class

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

Stack's base type

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

push(e): Push e onto the top of the stack.

pop(): Pop the element at the top of the stack.

top(): Return a reference to the element at the top of the stack.

- Applying *top()* or *pop()* to an empty stack is undefined

The STL Stack

- **The STL Stack Reference Manual**

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The std::stack class is a container adapter that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- back()
- push_back()
- pop_back()

The standard containers std::vector, std::deque and std::list satisfy these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container std::deque is used.

Member functions

(constructor)	constructs the stack (public member function)
(destructor)	destructs the stack (public member function)
operator=	assigns values to the container adaptor (public member function)
Element access	
top	accesses the top element (public member function)
Capacity	
empty	checks whether the underlying container is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element at the top (public member function)
emplace (C++11)	constructs element in-place at the top (public member function)
pop	removes the top element (public member function)
swap (C++11)	swaps the contents (public member function)

https://en.cppreference.com/w/cpp/container/stack

The STL Stack

- The container gets one more element appended (*i.e.*, the same result)

- *push()*
  - Takes an existing element and **copy** it to append
  - Takes exactly one argument

- *emplace()*
  - The element to be pushed is constructed **in-place**
  - Takes arguments for the constructor of the element

Think about a class with a costly constructor...

std::stack<T,Container>::push

```
void push( const value_type& value );
void push( value_type&& value );        (since C++11)
```

Pushes the given element value to the top of the stack.

1) Effectively calls `c.push_back(value)`
2) Effectively calls `c.push_back(std::move(value))`

Parameters

value - the value of the element to push

std::stack<T,Container>::emplace

```
template< class... Args >
void emplace( Args&&... args );                    (since C++11)
                                                   (until C++17)

template< class... Args >
decltype(auto) emplace( Args&&... args );          (since C++17)
```

Pushes a new element on top of the stack. The element is constructed in-place, i.e. no copy or move operations are performed. The constructor of the element is called with exactly the same arguments as supplied to the function.

Effectively calls `c.emplace_back(std::forward<Args>(args)...);`

Parameters

args - arguments to forward to the constructor of the element

https://en.cppreference.com/w/cpp/container/stack

The STL Stack

- *swap()*
  - Exchanges the contents of two containers

std::stack<T,Container>::swap

```
void swap( stack& other ) noexcept(/* see below */);    (since C++11)
```

Exchanges the contents of the container adaptor with those of other. Effectively calls `using std::swap; swap(c, other.c);`

Parameters

other - container adaptor to exchange the contents with

Example

Run this code

```
#include <iostream>
#include <stack>
#include <string>
#include <vector>

template <typename Stack>
void print(Stack stack /* pass by value */, int id)
{
    std::cout << "s" << id << " [" << stack.size() << "]: ";
    for (; !stack.empty(); stack.pop())
        std::cout << stack.top() << ' ';
    std::cout << (id > 1 ? "\n\n" : "\n");
}

int main()
{
    std::vector<std::string>
        v1{"1","2","3","4"},
        v2{"∀","B","Ↄ","D","∃"};

    std::stack s1{std::move(v1)};
    std::stack s2{std::move(v2)};

    print(s1, 1);
    print(s2, 2);

    s1.swap(s2);

    print(s1, 1);
    print(s2, 2);
}
```

Output:

```
s1 [4]: 4 3 2 1
s2 [5]: ∃ D Ↄ B ∀

s1 [5]: ∃ D Ↄ B ∀
s2 [4]: 4 3 2 1
```

https://en.cppreference.com/w/cpp/container/stack

C++ Stack Interface

- **An Informal Stack Interface**

```
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
};
```

accessors

no return for pop

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

error message

Stack Implementation: Array-Based

- The stack consists of:
  - an N-element array S
  - an integer variable t indicating the top element in S
  - t is initialized to -1 to denote the empty stack

- Each function executes a constant number of statements
  - Arithmetic operation
  - Comparison
  - Indexing
  - Assignment
- Can be a good option when we have a good estimate on the number of items

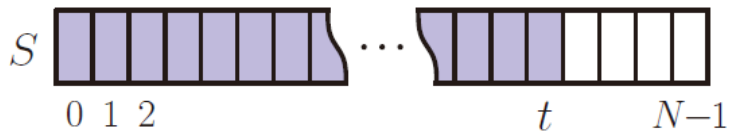


Figure 5.2: Realization of a stack by means of an array S. The top element in the stack is stored in the cell $S[t]$.

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

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
    $t \leftarrow t - 1$

Stack Implementation: Array-Based

- **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
};
```

Stack Implementation: Array-Based

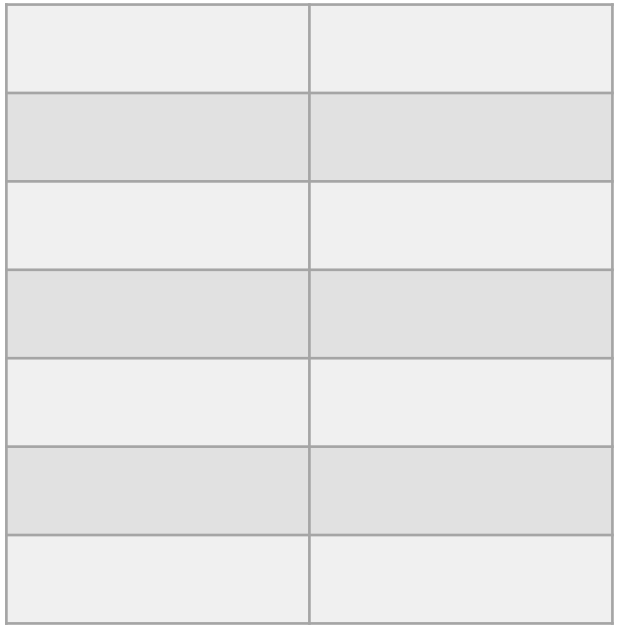
- **C++ Implementation**

```
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;
}
```

Stack Implementation: Array-Based

- **Example Output**

```
ArrayStack<int> A;
A.push(7);
A.push(13);
cout << A.top() << endl; A.pop();
A.push(9);
cout << A.top() << endl;
cout << A.top() << endl; A.pop();
ArrayStack<string> B(10);
B.push("Bob");
B.push("Alice");
cout << B.top() << endl; B.pop();
B.push("Eve");
```



```
// A = [], size = 0
// A = [7*], size = 1
// A = [7, 13*], size = 2
// A = [7*], outputs: 13
// A = [7, 9*], size = 2
// A = [7, 9*], outputs: 9
// A = [7*], outputs: 9
// B = [], size = 0
// B = [Bob*], size = 1
// B = [Bob, Alice*], size = 2
// B = [Bob*], outputs: Alice
// B = [Bob, Eve*], size = 2
```

Stack Implementation: Linked List

- **C++ Implementation (Singly Linked List)**

```
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
};
```

Stack Implementation: Linked List

- **C++ Implementation (Singly Linked List)**

```
class StringLinkedList {
public:
  StringLinkedList();
  ~StringLinkedList();
  bool empty() const;
  const string& front() const;
  void addFront(const string& e);
  void removeFront();
private:
  StringNode* head;
};
```

```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                   // create new node
  v->elem = e;                                      // store data
  v->next = head;                                   // head now follows v
  head = v;                                         // v is now the head
}
void StringLinkedList::removeFront() {              // remove front item
  StringNode* old = head;                           // save current head
  head = old->next;                                 // skip over old head
  delete old;                                       // delete the old head
}
```

Stack Implementation: Linked List

- **C++ Implementation (Singly Linked List)**

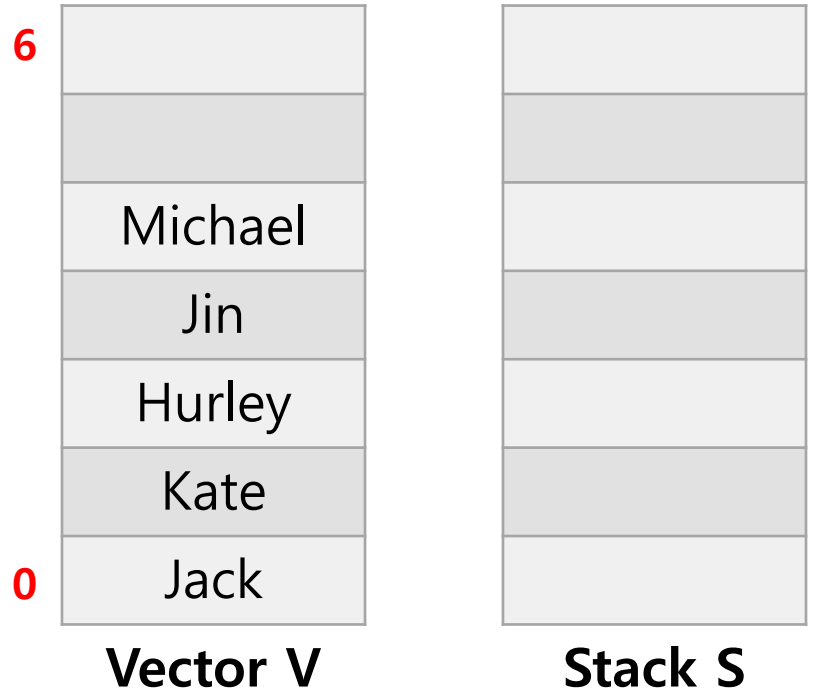
```
LinkedStack::LinkedStack()
  : S(), n(0) { }                    // constructor

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();
}
```

Stack Applications

- **Reversing a Vector Using a Stack**
  - Push all the elements into a stack
  - Fill the vector again by popping the elements off of the stack

```
template <typename E>
void reverse(vector<E>& V) {                    // reverse a vector
  ArrayStack<E> S(V.size());
  for (int i = 0; i < V.size(); i++)            // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) {          // pop them in reverse order
    V[i] = S.top();  S.pop();
  }
}
```



Stack Applications

- **Matching Parentheses**
  - Matching parentheses and grouping symbols
  - Each opening symbol must match with its corresponding closing symbol

- Parentheses: “(” and “)”
- Braces: “{” and “}”
- Brackets: “[” and “]”
- Floor function symbols: “⌊” and “⌋”
- Ceiling function symbols: “⌈” and “⌉,”

- Correct: ()(())\{([()])\}
- Correct: (((())(())\{([()])\}))
- Incorrect:)(())\{([()])\}
- Incorrect: (\{[])\}
- Incorrect: (

Stack Applications

- **An Algorithm for Parentheses Matching**
  - Suppose we are given a sequence $X = x_0 x_1 x_2 \ldots x_{n-1}$
  - Each x_i is a token that can be:
    - A grouping symbol
    - A variable name
    - An arithmetic operator
    - A number
  - Push a token when we encounter an opening symbol
  - Pop the top token when we encounter a closing symbol and check the correctness
  - The symbols in X match if the stack is empty after the whole sequence processing

$O(n)$

Stack Applications

- **An Algorithm for Parentheses Matching**

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}
```

Stack Applications

- **Parentheses Matching Examples**

- Correct: ()(()){([()])}
- Correct: ((()(())){([()])}))
- Incorrect:)(()){([()])}
- Incorrect: ({[])}
- Incorrect: (

Stack

Stack Applications

- **Matching Tags in an HTML Document**
  - HTML: HyperText Markup Language
  - An HTML tag consists of opening and closing tags
    - Opening: <name>
    - Closing: </name>

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even
as a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Stack Applications

- **Matching Tags in an HTML Document**

```
vector<string> getHtmlTags() {              // store tags in a vector
  vector<string> tags;                      // vector of html tags
  while (cin) {                             // read until end of file
    string line;
    getline(cin, line);                     // input a full line of text
    int pos = 0;                            // current scan position
    int ts = line.find("<", pos);           // possible tag start
    while (ts != string::npos) {            // repeat until end of string
      int te = line.find(">", ts+1);        // scan for tag end
      tags.push_back(line.substr(ts, te-ts+1)); // append tag to the vector
      pos = te + 1;                         // advance our position
      ts = line.find("<", pos);
    }
  }
  return tags;
}
```

Vector

std::basic_string<CharT,Traits,Allocator>::npos

```
static const size_type npos = -1;
```

This is a special value equal to the maximum value representable by the type size_type. The exact meaning depends on context, but it is generally used either as end of string indicator by the functions that expect a string index or as the error indicator by the functions that return a string index.

Note

Although the definition uses -1, size_type is an unsigned integer type, and the value of npos is the largest positive value it can hold, due to signed-to-unsigned implicit conversion. This is a portable way to specify the largest value of any unsigned type.

https://en.cppreference.com/w/cpp/string/basic_string/npos

Stack Applications

- **Matching Tags in an HTML Document**

```
                                                    // check for matching tags
bool isHtmlMatched(const vector<string>& tags) {
  LinkedStack S;                                    // stack for opening tags
  typedef vector<string>::const_iterator Iter;      // iterator type
                                                    // iterate through vector
  for (Iter p = tags.begin(); p != tags.end(); ++p) {
    if (p->at(1) != '/')                            // opening tag?
      S.push(*p);                                   // push it on the stack
    else {                                          // else must be closing tag
      if (S.empty()) return false;                  // nothing to match - failure
      string open = S.top().substr(1);              // opening tag excluding '<'
      string close = p->substr(2);                  // closing tag excluding '</'
      if (open.compare(close) != 0) return false;   // fail to match
      else S.pop();                                 // pop matched element
    }
  }
  if (S.empty()) return true;                       // everything matched - good
  else return false;                                // some unmatched - bad
}
```