Lists

A simple programming problem:

> Write a program to read in a number, n, where $0 \leq n \leq 25$, and then read n more integers. After the last integer has been read, output the numbers in reverse order.

Typical problem: "I need to know in advance how big to make my array."

Two concepts:

Array An (fixed size) ordered collection of variables referred to by one name. (A *concrete* data type.)

List An ordered collection of values. (An *abstract* data type.)

Arrays can be used to implement lists.

Stack

One type of list: *Last In, First Out* (LIFO)

Operations:

push(x) – add x to the end of the list.

pop – remove the last element in the list.

top – return the last element in the list.

empty – return True if the list is empty.

size – return the number of elements in the list.

Example: reverse.cpp

Sequences

A list is modeled by a *sequence* — a function from a range of integers $(0, 1, 2 \ldots n)$ to of elements of type $\mathbf{T}$ (the type of elements in the list).

Notation

- $\{a_n\}$ denotes the sequence $a_0, a_1, a_2, \ldots a_n$
- a_i is an *element* of the sequence
- $|\alpha|$ denotes the *length* of the sequence α (note: $|\{a_n\}| = n + 1$).
- αx denotes the *concatenation* of x to α. $|\alpha x| = |\alpha| + 1$.
- $\_$ denotes the empty sequence. $|\_| = 0$.

Stack Specificaiton

Description A LIFO list.
State $\mathbf{s}$: A sequence of type $\mathbf{T}$.
Operations

- $stack()$ — Constructor.
  Post: $\mathbf{s} = \_$, $\mathbf{s}$ is the empty sequence.
- $\tilde{}stack()$ Destructor.
- $push(\mathbf{T}\ x)$ — Mutator. Adds x to the top of the stack.
  Post: $\mathbf{s}' = \mathbf{s}x$, x is appended to $\mathbf{s}$
- $pop()$ — Mutator. Removes the top element.
  Pre: $|\mathbf{s}| > 0$, The stack is not empty.
  Post: $\mathbf{s}' = \mathbf{s}_{\{0,\ldots|s|-2\}}$, The last element of $\mathbf{s}$ is removed.
- $\mathbf{T}\ top()$ — Accessor. Returns the top element of the stack.
  Pre: $|\mathbf{s}| > 0$, The stack is not empty.
  Post: $Result = s_n \wedge \mathbf{s}' = \mathbf{s}$, The last element of $\mathbf{s}$ is returned and $\mathbf{s}$ is unchanged.

– **Bool** $empty()$ — Accessor. Returns True if the stack is empty.
Post: $Result = (|\mathbf{s}| = 0)$. Returns true if the length of s is 0, false otherwise.

Error Handling

What should the program do when something goes wrong?

Three aspects:

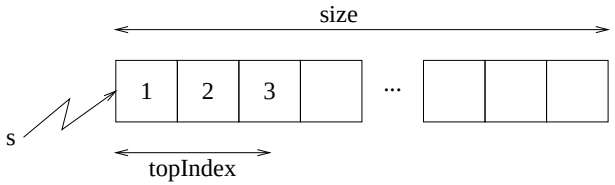
1) Detection
   – Do it where it's easiest (often 'low' level).
2) Reporting — to other parts of the system.
   – Mechanism is part of the interface.
3) Recovery/processing
   – Often best at system level.
   – Don't (in general) assume the presence of a 'user' who can respond.
   – Don't assume standard streams are observed.

- Pre-condition is false
  – Indicates a programmer error on part of **calling** program.
  – Technically any behaviour is correct.
  – Usually best to fail quickly and clearly — make sure the fault is detected and fixed. (`assert()` works well.)
- Resource limitations, failure of another system
  – Unpredictable, but should be expected. (Always check for it.)
  – Recover or fail as gracefully as possible.
- Input error
  – Check for it when reasonable.
  – Processing will depend on UI.

Note: STL pretty much ignores errors (see overflow.cpp and underflow.cpp)

Stack Implementation

Array implementation (fixed or dynamic):



See IntStack.h and IntStack.cpp

Template Implementation

Replacing `int` with any type, say `T`, in a few lines makes a different stack:

```
private:
  T *s;         // Pointer to begining of the stack.
  int size;     // Maximum |S|
  int topIndex; // Index of top item in the stack.

// ...
s = new(std::nothrow) T[_size];

// ...
Stack::push(T x)

// ...
T Stack::top()
```


Define a *class template* to tell the compiler to do this substitution for us.

- `template <class T> class Stack {` — this is a class template with one parameter (`T`).
- `template <class T> void Stack<T>::push(T x)` — a function template.
- `Stack<char> s;` — create a stack of characters.
- `T` is an arbitrary identifier.
- There can be more than one template parameter. E.g.,
  `template <class L, class R> class pair { ...`
- Template parameters don't have to be type (`class`) E.g.,
  `template <class T, int i> class Buffer { ...`

See Stack.h