Lists

vector

C++ equivalent of Java `ArrayList`

```
#include <vector>
...
vector<int> example;
example.push_back(4);
example.push_back(5);
example.push_back(6);
cout << example[0] << "␣"
     << example[2] << "␣"
     << example.size();
// OUTPUT: 4 6 2
```

why not normal arrays?

can't resize arrays

can't assign arrays with $=$

arrays don't track capacity

arrays don't have bounds checking

vector member functions: accessing elements

given `vector<Type>`

```
Type &vector<Type>::operator[](int index)
```

may or may not crash if `index` out of bounds

```
cout << someVector[idx]
someVector[idx] = value;
```

```
Type &vector<Type>::at(int index)
```

throws exception if `index` out of bounds

```
cout << someVector.at(idx)
someVector.at(idx) = value;
```

```
Type &front()
```

```
Type &back()
```

vector member functions: const variants

```
vector<int> example;
...
const vector<int>& ref = example;
cout << ref.at(2);  // OKAY: returns const reference
cout << ref[2];  // OKAY: returns const reference
cout << ref.front();  // OKAY: returns const reference
ref.at(2) = 3;  // ERROR: const reference
ref[2] = 3;     // ERROR: const reference
example.at(2) = 3;  // OKAY
example.front() = 5; // OKAY
```

```
const Type &operator[](int index) const;
const Type &at(int index) const;
const Type &front() const;
const Type &back() const;
```

vector member functions: size and capacity

```
int capacity() const

int size() const

void reserve(int newCapacity)

void resize(int newSize)

void clear()
```

vector member functions: append/prepend

```
void push_back(const T& newElement)
void pop_back()
```

add/remove last element

```
void push_front(const T& newElement)
void pop_front()
```

add/remove first element — O(N)

C++ containers

standard library has collection of ‘container’ classes

- used to be part of a separate “standard template library”

many list-like containers:

- `vector` — dynamic array class
- `string`
- `list` — doubly-linked list
- `map`, `hash_map`
- `stack`
- `deque` — double-ended queue
- …

share common methods, `iterator` interface

standard library in this course

can use any standard library classes

except if it defeats the point of the lab

examples:

- hash lab — don't use `hash_map`
- stack lab — no standard library classes for post-lab

standard library recommendation

use `vector`

use `string`

use `stack`

use what's convenient

- certainly what to do in a job

standard library documentation

my recommendation: `http://en.cppreference.com/` -

NB: we won't be using C++11/14/17/20 features

(this is a reference, *definitely not a tutorial*)

secret templates

`std::string` = `std::basic_string<char>`

`std::ostream` = `std::basic_ostream<char>`

what `cout` is

`std::istream` = `std::basic_istream<char>`

what `cin` is

C++ iterators

nested type representing position

designed to work like a pointer

most methods use operator overloading

example: `vector<T>::iterator`

vector iterator methods

methods within `vector<T>`:

- `iterator begin()`
- `iterator end()` — one past end

methods within `vector<T>::iterator iter`:

- `operator++`: `iter++`, `++iter` (forward)
- `operator--`: `iter--`, `--iter` (backward)
- `operator*`: `*iter` (access at position)
- `operator->`: `iter->member` (access at position)
- `operator==`: `iter1 == iter2` (compare positions)
- `operator<`: `iter1 < iter2` (compare positions)

iterating through a vector

```
vector<int> v;
v.push_back(1); v.push_back(2); v.push_back(3);
...
for (vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {
  cout << *it << "␣";
}
// output: 1 2 3
```

member functions that take iterators

```
iterator vector<T>::insert(iterator pos, const T &x)
```

insert *before* `pos`
return iterator pointing to position of inserted element
O(N) unless `pos` is the end

```
iterator vector<T>::erase(iterator pos)
```

return iterator pointing to position after the end

```
iterator vector<T>::erase(iterator start, iterator end)
```

erase from `start` up to *and not including* `end`

iterator ranges, generally

many standard library functions:
`function(Iterator` *`first`*`, Iterator` *`last`*`, ...)`

always: `first` *up to but not including* `last`

- why `some_vector.end()` is one-past-the-end

iterator ranges, generally

many standard library functions:
`function(Iterator first, Iterator last, ...)`

always: `first` *up to but not including* `last`

 why `some_vector.end()` is one-past-the-end

```
#include <vector>
#include <algorithm>
...
std::vector<int> v = getUnsortedList();
std::sort(v.begin(), v.end());  // sorts the *whole* vector
```

modifying values with iterators

```
vector<int> v;
...
for (vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {
    *it += 1;
}
```

const_iterators (1)

```
void print(const vector<int> &v) {
    for (vector<int>::const_iterator it = v.begin();
         it != v.end();
         ++it) {
        cout << *it << "␣";
    }
}
```

const_iterators (2)

```
void brokenAddOne(const vector<int> &v) {
    for (vector<int>::const_iterator it = v.begin();
         it != v.end();
         ++it) {
         *it += 1;  // ERROR: trying to use modify const(ant)
    }
}

void workingAddOne(vector<int> &v) {
    for (vector<int>::iterator it = v.begin();
         it != v.end();
         ++it) {
         *it += 1;  // OKAY, normal iterator
    }
}
```

templates

templates — C++'s equivalent to *generics*

idea — code with 'fill in the blank'

compiler generates seperate version for each blank

template example: findMax.cpp (1)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<int> v1(37); ... cout << findMax(v1) << endl;
```

template example: findMax.cpp (1)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<int> v1(37); ... cout << findMax(v1) << endl;
```

```
const int& findMax(const vector<int> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

template example: findMax.cpp (1)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<int> v1(37); ... cout << findMax(v1) << endl;
```

```
const int& findMax(const vector<int> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

template example: findMax.cpp (2)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<string> v1(37); ... cout << findMax(v1) << endl;
```

template example: findMax.cpp (2)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<string> v1(37); ... cout << findMax(v1) << endl;
```

```
const string& findMax(const vector<string> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

template example: findMax.cpp (2)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<string> v1(37); ... cout << findMax(v1) << endl;
```

```
const string& findMax(const vector<string> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

template example: findMax.cpp (3)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<IntCell> v4(30);  cout << findMax(v4) << endl;
```

template example: findMax.cpp (3)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

```
vector<IntCell> v4(30);  cout << findMax(v4) << endl;
```

```
const IntCell& findMax(const vector<IntCell> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

template example: findMax.cpp (3)

```
template <typename Comparable>
const Comparable& findMax(const vector<Comparable> &a)
{
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a
}
```

compile error until `IntCell::operator<` created!

```
vector<IntCell> v4(30);  cout << findMax(v4) << endl;
```

```
const IntCell& findMax(const vector<IntCell> &a) {
  int maxIndex = 0;
  for( int i = 1; i < a.size(); i++ )
    if( a[ maxIndex ] < a[ i ] ) maxIndex = i;
  return a[ maxIndex ];
}
```

generating template

exact same effect as replacing the typename everywhere

compiler only creates versions that are used

template classes

```
template <typename Object>
class ObjectCell {
public:
    ObjectCell(const Object & initValue = Object())
            : storedValue(initValue) {}
    const Object & getValue() const {
        return storedValue;
    }
    void setValue(const Object & val) {
        storedValue = val;
    }
private:
    Object storedValue;
};
```

template classes

```
template <typename Object>
class ObjectCell {
public:
    ObjectCell(const Object & initValue = Object())
                : storedValue(initValue) {}
    const Object & getValue() const {
        return storedValue;
    }
    void setValue(const Object & val) {
        storedValue = val;
    }
private:
    Object storedValue;
};
```

`ObjectCell<int>` — replace `Object` with `int`

using template classes

```
int main() {
    ObjectCell<int> m1;
    ObjectCell<double> m2(3.14);
    m1.setValue(37);
    m2.setValue(m2.getValue() * 2);
    // ...
    return 0;
}
```

multiple parameters

```
template <typename Key, typename Value>
class Map {
    ...
};
```

constant value paramters

```
template <typename ValueType, int size>
class Buffer {
    ...
    ValueType data[size];
};
```

default paramters

```
template <typename ValueType=char, int size=4096>
class Buffer {
    ...
    ValueType data[size];
};
...
Buffer<> buf1; // Buffer<char, 4096>
Buffer<int> buf2; // Buffer<int, 4096>
Buffer<string, 2048> buf3;
```

no separate implementations (1)

BROKEN findmax.h

```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);
#endif
```

test.cpp

```
#include "findmax.h"
int main() {
    vector<int> v;
    ...
    int theMax = findMax(v);
}
```

no separate implementations (1)

BROKEN findmax.h

```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);
#endif
```

test.cpp

this is a linker error:

```
$ clang++  test.cpp findmax.cpp
/tmp/test-d6d266.o: In function 'main':
test.cpp:(.text+0xd): undefined reference to 'findMax<int>()'
```

required to have *implementation* included in each .cpp file

no separate implementations (2)

findmax.h

```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);

// implementation in header file directly
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a) {
    ... /* implementation here */
}
#endif
```

no separate implementations (2)

findmax.h

```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);

// implementation in header file directly
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a) {
    ... /* implementation here */
}
#endif
```

no separate implementations (3)

findmax.h

```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);

// implementation file #include'd in header file
#include "findmax_impl.h"
#endif
```

findmax_impl.h

```
const Comparable& findMax(
    const vector<Comparable> &a) {
    ... /* implementation here */
}
```

no separate implementations (3)

findmax.h

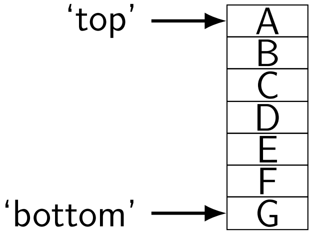
```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);

// implementation file #include'd in header file
#include "findmax_impl.h"
#endif
```

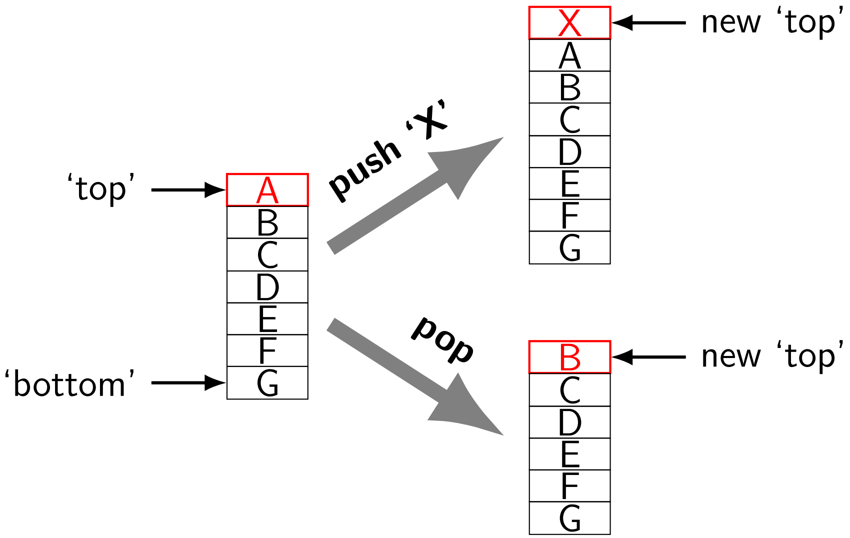
findmax_impl.h

```
const Comparable& findMax(
    const vector<Comparable> &a) {
    ... /* implementation here */
}
```

stacks

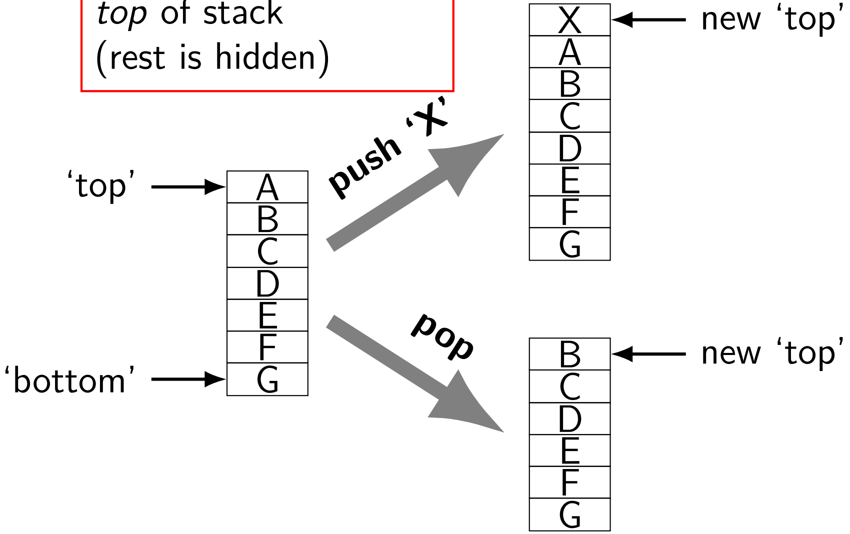


stacks


'top'
A
B
C
D
E
F
G
'bottom'
push 'X'
X
A
B
C
D
E
F
G
new 'top'
pop
B
C
D
E
F
G
new 'top'

stacks

operations only access
top of stack
(rest is hidden)



stack methods

`stack.push(value)` — add at top

`stack.pop()` — remove from top

`value = stack.top()` — return top without removing

`bool wasEmpty = stack.isEmpty()` — check if stack is empty?

last time

vector, iterators, standard library containers

stack applications

- parenthesis matching
- postfix calculator

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls

some stack applications

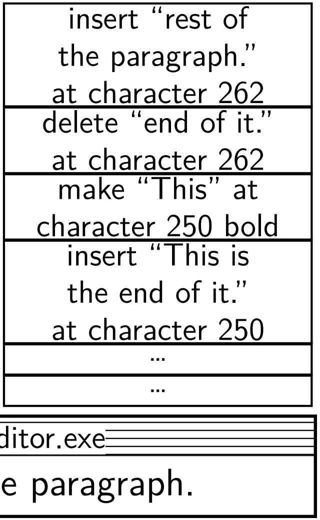
undo

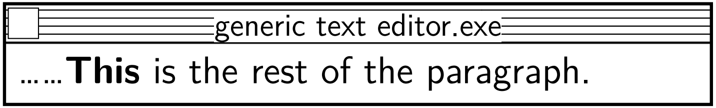
parenthesis matching

postfix calculators

operator precedence

tracking function calls


insert "rest of the paragraph." at character 262
delete "end of it." at character 262
make "This" at character 250 bold
insert "This is the end of it." at character 250
...
...
generic text editor.exe
……**This** is the rest of the paragraph.



some stack applications

undo

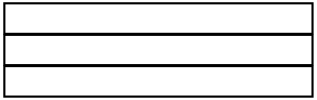
parenthesis matching

postfix calculators

operator precedence

tracking function calls

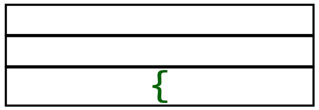
parenthesis matching



```
{ [ ( ) [ ] ( ) }
```

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output **mismatched**
EOF and !empty()? → output **mismatched**
```

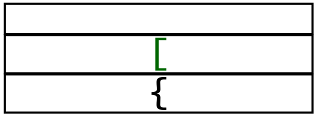
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output **mismatched**
EOF and !empty()? → output **mismatched**
```

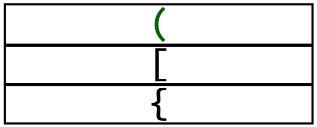
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

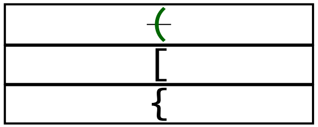
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

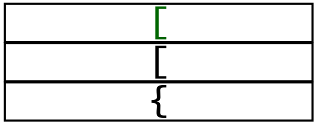
parenthesis matching



`{ [ ( ) [ ] ( ) }`

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

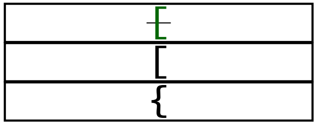
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

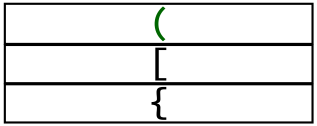
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

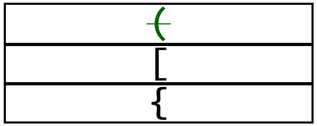
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

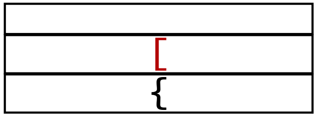
parenthesis matching



{ [( ) [ ] ( ) }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output mismatched
EOF and !empty()? → output mismatched
```

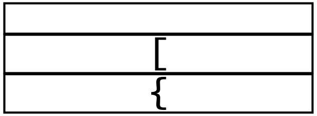
parenthesis matching



```
{ [ ( ) [ ] ( ) }
```

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output **mismatched**
EOF and !empty()? → output **mismatched**
```

parenthesis matching



{ [( ) [ ] ( ) }   mismatched!

{ [( ) [ ] ( )] }

```
for each symbol:
opening symbol? → push(symbol)
closing symbol? →
    !empty() and top() is opposite symbol → pop()
    otherwise → output **mismatched**
EOF and !empty()? → output **mismatched**
```

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls

postfix calculations

```
3 7 * 4 7 8 / * +
```

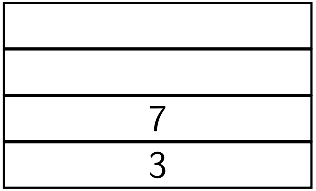
postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

number? → push(number)
operator? →
  a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations

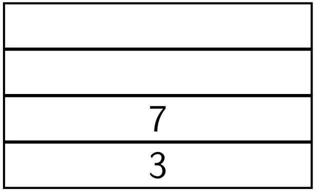


`3 7 * 4 7 8 / * +` postfix expression

`(3 * 7) + (4 * (7 / 8))` equivalent infix expression

number? $\rightarrow$ push(number)
operator? $\rightarrow$
  a $=$ top(), pop(), b $=$ top(), pop(), push(a *OP* b)

postfix calculations

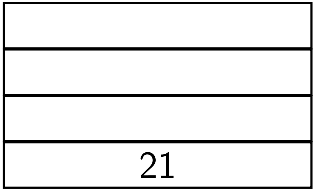


`3 7 * 4 7 8 / * +` postfix expression

`(3 * 7) + (4 * (7 / 8))` equivalent infix expression

number? $\rightarrow$ push(number)
operator? $\rightarrow$
    a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations



3 7 * 4 7 8 / * +

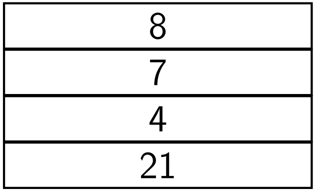
postfix expression

(3 * 7) + (4 * (7 / 8))

equivalent
infix expression

number? → push(number)
operator? →
    a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations

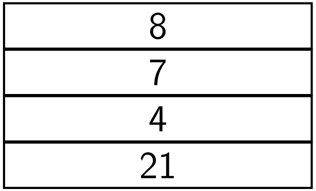


`3 7 * 4 7 8 / * +` postfix expression

`(3 * 7) + (4 * (7 / 8))` equivalent infix expression

number? $\rightarrow$ push(number)
operator? $\rightarrow$
  a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations



```
3 7 * 4 7 8 / * +
```

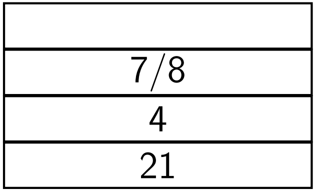
postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

number? → push(number)
operator? →
  a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations

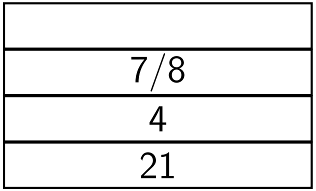


`3 7 * 4 7 8 / * +` postfix expression

`(3 * 7) + (4 * (7 / 8))` equivalent infix expression

number? $\rightarrow$ push(number)
operator? $\rightarrow$
  a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations



```
3 7 * 4 7 8 / * +
```

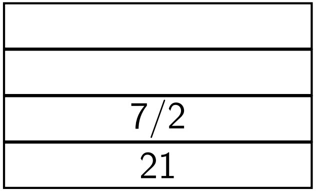
postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

number? → push(number)
operator? →
  a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations



```
3 7 * 4 7 8 / * +
```

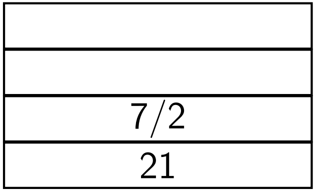
postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

number? → push(number)
operator? →
    a = top(), pop(), b = top(), pop(), push(a *OP* b)

postfix calculations



```
3 7 * 4 7 8 / * +
```

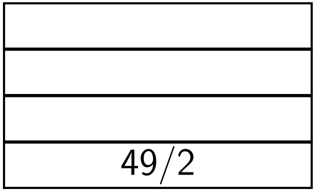
postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

```
number? → push(number)
operator? →
    a = top(), pop(), b = top(), pop(), push(a OP b)
```

postfix calculations



```
3 7 * 4 7 8 / * +
```

postfix expression

```
(3 * 7) + (4 * (7 / 8))
```

equivalent infix expression

```
number? → push(number)
operator? →
    a = top(), pop(), b = top(), pop(), push(a OP b)
```

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls

some stack applications

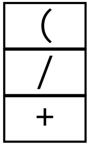
undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls



stack of
unfinished operators

A + B * C / (D + E) + F

A B C * D E + / + F +

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls

some stack applications

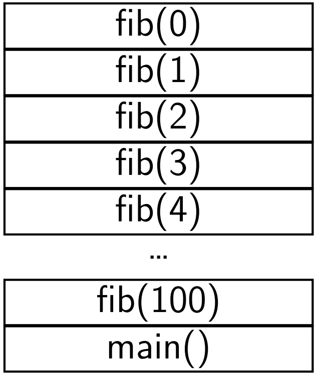
undo

parenthesis matching

postfix calculators

operator precedence

tracking function calls



some stack applications

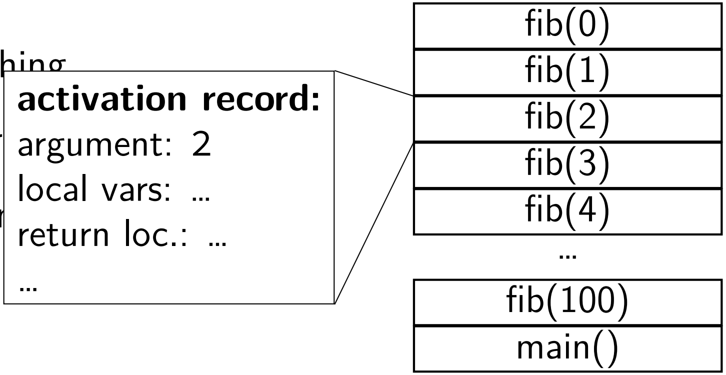
undo

parenthesis matching

postfix calculator

operator preceden

tracking function



activation record:

argument: 2

local vars: ...

return loc.: ...

...

stack implmentation choices

need to keep track of multiple items

several data structures for doing so...

singly linked lists

doubly linked lists

arrays

...

stack implmentation choices

need to keep track of multiple items

several data structures for doing so...

singly linked lists

doubly linked lists

arrays

...

linked list stack of ints

```
class StackNode {
    ...
    int value;
    StackNode *next;
};

class Stack {
  public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    int top() const;
    void push(int value);
    void pop();

  private:
    StackNode *head;
};
```

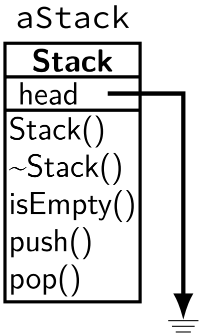
linked list stack of ints

```
class StackNode {
    ...
    int value;
    StackNode *next;
};

class Stack {
  public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    int top() const;
    void push(int value);
    void pop();

  private:
    StackNode *head;
};
```

```
Stack aStack;
```



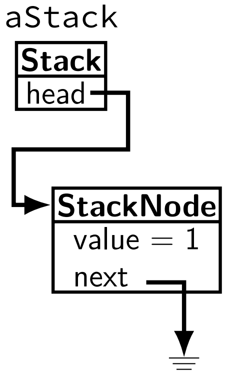
linked list stack of ints

```
class StackNode {
    ...
    int value;
    StackNode *next;
};

class Stack {
  public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    int top() const;
    void push(int value);
    void pop();

  private:
    StackNode *head;
};
```

```
Stack aStack;
aStack.push(1);
```



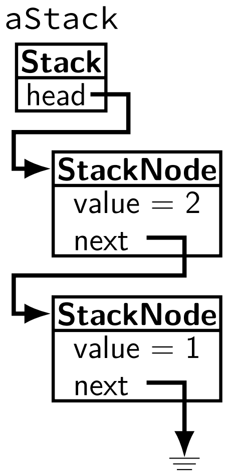
linked list stack of ints

```
class StackNode {
    ...
    int value;
    StackNode *next;
};

class Stack {
  public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    int top() const;
    void push(int value);
    void pop();

  private:
    StackNode *head;
};
```

```
Stack aStack;
aStack.push(1);
aStack.push(2);
```



implementing linked list stack

```
bool Stack::isEmpty() cosnt {
    return head == NULL;
}

int Stack::top() const {
    // FIXME: throw exception if empty?
    return head->value;
}
```

vector stack of ints

```
class Stack {
  public:
    Stack();
    ~Stack();
    bool isEmpty() const;
    int top() const;
    void push(int value);
    void pop();

  private:
    vector<int> data;
};
```

`data` contains elements of stack

last element of `data` is “top”

(lets `push` be fast)

implementing vector stack

```
bool Stack::isEmpty() const {
    return data.size() == 0;
}

void Stack::push(int value) {
    data.push_back(value);
}

void Stack::pop() {
    data.pop_back();
}

// ...
```

implementing top?

```
int Stack::top() {
    return ...
}
```

What could go here?

A. `data.back();`
B. `data.at(data.size());`
C. `data.at(data.size() - 1);`
D. `data[data.capacity() - 1];`
E. `*data.end();`

implementing top?

```
int Stack::top() {
    return ...
}
```

What could go here?

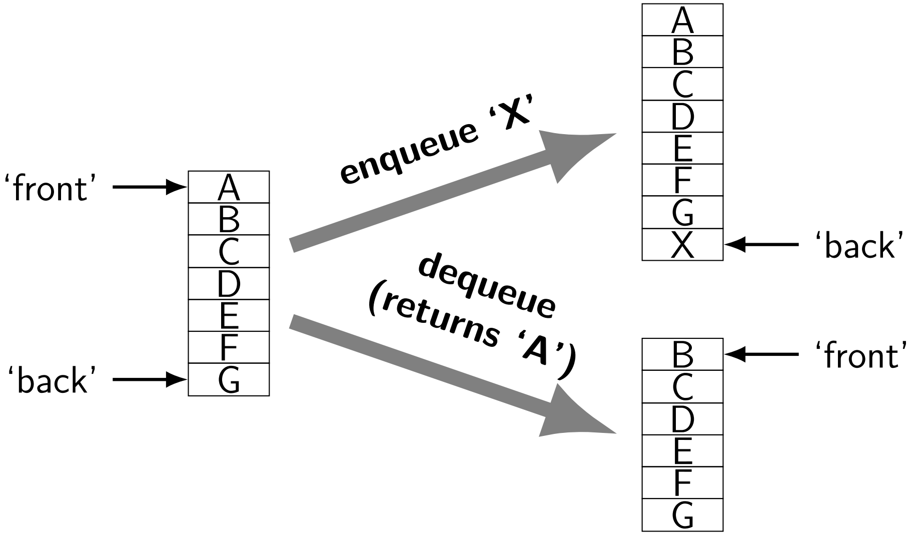
A. `data.back();`
B. `data.at(data.size());`
C. `data.at(data.size() - 1);`
D. `data[data.capacity() - 1];`
E. `*data.end();`

A or C

or `data[data.size() - 1]`

or `*(data.end() - 1);`

queues


‘front’
‘back’
A
B
C
D
E
F
G
enqueue ‘X’
A
B
C
D
E
F
G
X
‘back’
dequeue
(returns ‘A’)
B
C
D
E
F
G
‘front’

queue v stack

queue — first-in, first-out (FIFO)

stack — last-in, first-out (LIFO)

both have linked list and array-based implementations

queue applications

print queue — waiting line of print jobs

web servers — waiting line of web brwoser

…

array-based queue of ints

```
class Queue {
public:
  Queue();
  ~Queue();
  void enqueue(int value);
  int dequeue();
  bool isEmpty() const;
private:
  int *data;
  int dataSize;
  int frontIndex;
  int backIndex;
};
```

array-based queue of ints

```
void Queue::enqueue(int value) {
  backIndex++;
  if (backIndex >= dataSize)
      ...
  data[backIndex] = value;
}
```

```
class Queue {
public:
  Queue();
  ~Queue();
  void enqueue(int value);
  int dequeue();
  bool isEmpty() const;
private:
  int *data;
  int dataSize;
  int frontIndex;
  int backIndex;
};
```

array-based queue of ints

```
class Queue {
public:
  Queue();
  ~Queue();
  void enqueue(int value);
  int dequeue();
  bool isEmpty() const;
private:
  int *data;
  int dataSize;
  int frontIndex;
  int backIndex;
};
```

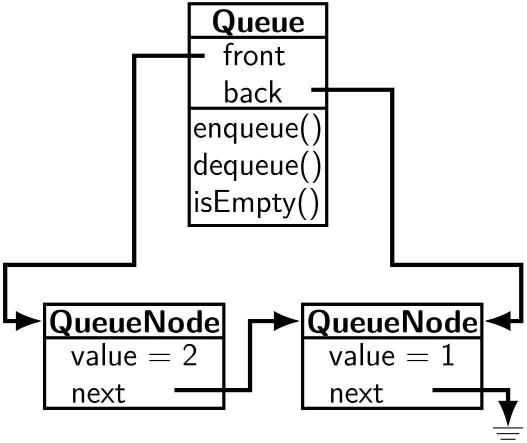
```
void Queue::enqueue(int value) {
  backIndex++;
  if (backIndex >= dataSize)
      ...
  data[backIndex] = value;
}
```

```
int Queue::dequeue() {
  if (frontIndex > backIndex)
      ...
  int value = data[frontIndex];
  frontIndex++;
  return value;
}
```

linked-list queue

```
class QueueNode {
  ...
  int value;
  QueueNode *next;
};

class Queue {
public:
  Queue();
  ~Queue();
  void enqueue(int value);
  int dequeue();
  bool isEmpty() const;
private:
  QueueNode *front, *back;
};
```



linked-list queue: enqueue

```
class Queue {
    ...
    QueueNode *front, *back;
};

void Queue::enqueue(int value) {
    // one implementation: insert at back
    QueueNode *node = new QueueNode;
    node->value = value;
    if (back) {
        back->next = node;
        back = node;
    } else {
        // other case?
    }
}
```

linked-list queue: dequeue

```
class Queue {
    ...
    QueueNode *front, *back;
};

void Queue::dequeue() {
    if (front) {
        ...
        front = front->next;
        ...
    } else {
        // other case?
    }
}
```

abstract data type

definition: collection of operations
that can be done on data structure

abstract data type

definition: collection of operations
that can be done on data structure

hide implementation details from (library) users

library can change without library users changing code

implementing ADT options

C++ or Java class

just a collection of functions

...

some ADT examples

stacks

queues

lists

multiple reasonable implementations

single set of operations

C++ standard library: stack ADT?

`stack` in C++ standard library

wrapper for several containers

- default: deque (double-ended queue)
- linked list
- vector
- ...

one generic interface!

```
stack<int> s1; // stack based on deque
stack<int, vector<int> > s2; // stack based on vector
stack<int, forward_list<int> > s3; // stack based on singly-li
...
```

ADTs we've seen

stack — operations:

push(Type), pop(Type)
isEmpty(), top()

queue — operations

enqueue(Type), dequeue()
isEmpty()

list — operations

????

list ADT operations

```
// From lab 2 --- selected operations
List someList;
...
bool empty = someList.isEmpty()
someList.makeEmpty();
ListItr iterator  = someList.first() ;
ListItr iterator = someList.last();
someList.insertAfter(value, iterator);
someList.remove(value);
ListItr position = someList.find(value);

// Operations not in the lab
int kthElement = someList.findKth(k);
someList.erase(iterator);
someList.insert(value, index)
```

list ADT operations

```
// From lab 2 --- selected operations
List someList;
...
bool empty = someList.isEmpty()
someList.makeEmpty();
ListItr iterator  = someList.first() ;
ListItr iterator = someList.last();
someList.insertAfter(value, iterator);
someList.remove(value);
ListItr position = someList.find(value);

// Operations not in the lab
int kthElement = someList.findKth(k);
someList.erase(iterator);
someList.insert(value, index)
```

list ADT operations

```
// From lab 2 --- selected operations
List someList;
...
bool empty = someList.isEmpty()
someList.makeEmpty();
ListItr iterator  = someList.first() ;
ListItr iterator = someList.last();
someList.insertAfter(value, iterator);
someList.remove(value);
ListItr position = someList.find(value);

// Operations not in the lab
int kth
someLis
someLis
```

iterator type — internals will depend on implementation
linked list: might contain pointer to node
array: might contain index

list ADT examples

values	34	12	52	16	12	
iterator	a1	a2	a3	a4	a5	a6

```
find(52) == a3

find(2) == a6
    // not found

insert(9999, 2)
    // becomes {34,12,9999,52,16,12}

remove(52)
```

ADT complexity

operation	array*	linked list
find (by value)	linear time	linear time
findKth (by index)	constant time	linear time
first or last	constant time	constant time
insert/erase (with index)	linear time	linear time
insert/erase (with index at end)	constant time	linear time
insert/erase (with iterator)	linear time	constant time

(* fixed-capacity array)

ADT complexity

operation	array*	linked list
find (by value)	linear time	linear time
findKth (by index)	constant time	linear time
first or last	constant time	constant time
insert/erase (with index)	linear time	linear time
insert/erase (with index at end)	constant time	linear time
insert/erase (with iterator)	linear time	constant time

(* fixed-capacity array)

C++ standard library: “sequence container”

`vector`, `list`, `deque` classes in C++ standard library all have:

- iterator type
- `begin()`, `end()`, `front()`, `back()`
- `empty()`, `size()`
- `clear()`, `insert(iterator, Type)`, `erase(iterator)`
- `push_back(Type)`, `pop_back()`

above methods/types sort of an informal ADT

- write template function/class that works any of them!
- this is how `std::stack` works

C++ strings (1)

```
#include <string>
...
    std::string s = "example";

// Mostly same as vector<char>:
    s.size() == 8
    s.at(3) == 'm'
    s[3] == 'm'

    s[0] = 'E';  // string becomes "Example"

    for (string::iterator it = s.begin();
         it != s.end(); ++it) {
        char c = *it;
    }
```

C++ strings (2)

`string` operations not supported by `vector`:

```
s = "some␣string␣constant";
s += "additional␣text";
const char *c_style_string = s.c_str();
cout << s;
    // output: some string constantadditional text
cout << s.substr(1, 3);  // output: ome

if (s == s2) { ... }
if (s < s2) { ... }
...
```

string constants

string constants are pointers to const char arrays

```
const char *hello = "Hello,␣World!";
// BROKEN:
if (hello == "Hello,␣World!") {
    // MAY OR MAY NOT BE TRUE
    // compares addresses and NOT string values
}
```

string constants and std::strings

```
string helloString = "Hello,␣World!";
    // uses string::string(const char*)

if (helloString == "Hello,␣World!") {
    // calls operator==(const string&, const char*)
}

if ("Hello,␣World!" == helloString) {
    // calls operator==(const char*, const string&)
}
```

cin and errors

when `cin` experiences an error,
it doesn't throw an exception (by default)

can test for errors by using `cin` as true/false value
    or `!cin.fail()`

test for EOF (after trying to read there) with `cin.eof()`

```
cin >> number;
if (cin) {  // same as:
    // read 'number' successfully
} else {
    // some sort of error happened
    if (cin.eof()) {
        // the error was trying to read at EOF
    }
}
```

iostreams and failures

```
string s = "old␣value";
cout << "cin.eof()␣=␣" << cin.eof() << "\n";
cin >> s;  // tries to read a word
if (!cin) {
    cout << "cin␣had␣an␣error\n";
}
cout << "s␣=␣" <<s;
cout << "cin.eof()␣=␣" << cin.eof() << "\n";
```

If I just type control-D (“end of file”):

```
$ ./a.out
cin.eof() = 0
^Dcin had an error!
s = old value
cin.eof() = 1
```