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 methods: accessing elements

given `vector<Type>`

```
Type &operator[](int index)
```

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

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

```
Type &at(int index)
```

  throws exception if `index` out of bounds

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

```
Type &front()
```

```
Type &back()
```

vector methods: 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 methods: size and capacity

```
int capacity() const

int size() const

void reserve(int newCapacity)

void resize(int newSize)

void clear()
```

vector methods: 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 / STL

STL = Standard Template Library
- part of the standard library
- (used to be seperate...)

many list-like containers:
- `vector` — dynamic array class
- `string`
- `list`, `slist` — doubly-, and singly-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

standard library recommendation

use `vector`

use `string`

use `stack`

use what's convenient

  certianly what to do in a job

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:

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

methods that take iterators

(assuming `vector<Type>`...)

```
iterator insert(iterator pos, const Type &x)
```

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

```
iterator erase(iterator pos)
```

return iterator pointing to position after the end

```
iterator erase(iterator start, iterator end)
```

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

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 working(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 genreates 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<IntCell> v4(30);  cout << findMax(v4) << 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<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 (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
}
```

compile error (unless `IntCell::operator<` created)!

can't use < on `IntCell`

```
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 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
```

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>()'
```

compiler needs implementation available

required to have imlpementation 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

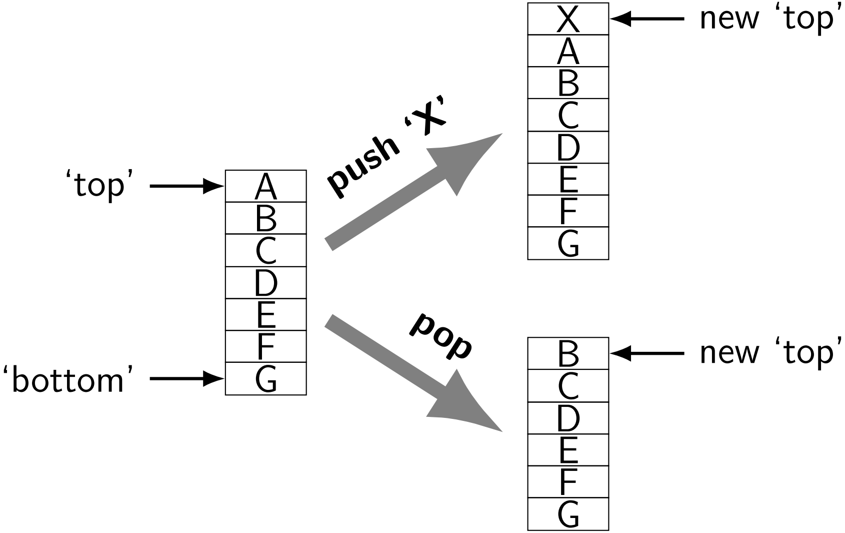
```
#ifndef FINDMAX_H
#define FINDMAX_H
template <typename Comparable>
const Comparable& findMax(
    const vector<Comparable> &a);

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

findmax_impl.h

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

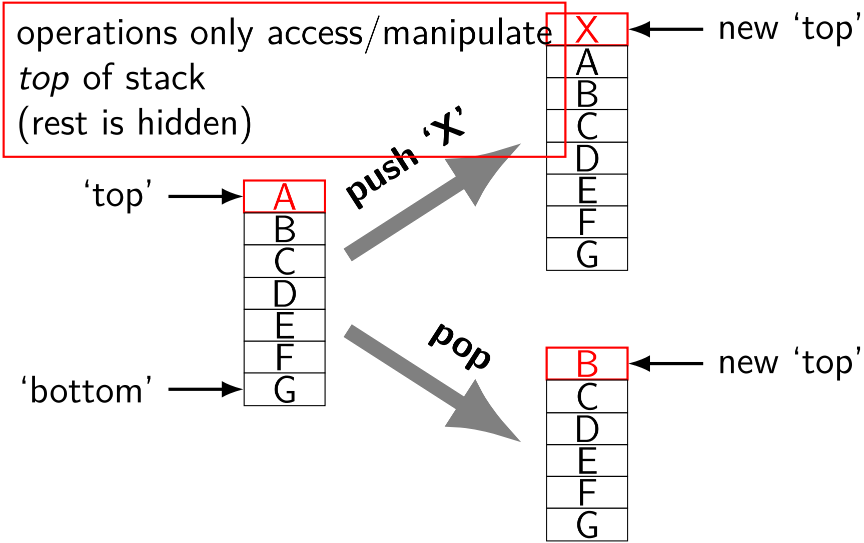
stacks



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

stacks

operations only access/manipulate *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?

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls

some stack applications

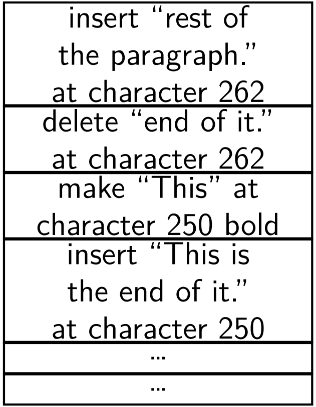
undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls



insert "rest of the paragraph." at character 262

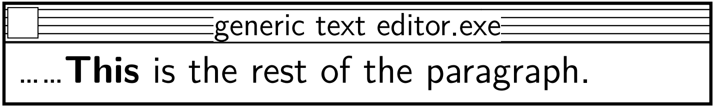
delete "end of it." at character 262

make "This" at character 250 bold

insert "This is the end of it." at character 250

...

...



……**This** is the rest of the paragraph.

some stack applications

undo

parenthesis matching

postfix calculators

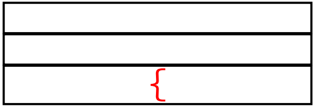
operator precedence

tracking (recursive) function calls

parenthsis matching

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

parenthsis matching



{ [( ) [ ] ( ) }

parenthsis matching

[
{

{ [( ) [ ] ( ) }

parenthsis matching

(
[
{

{ [( ) [ ] ( ) }

parenthsis matching

~~(~~
[
{

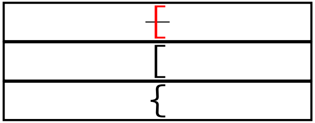
{ [( ) [ ] ( ) }

parenthsis matching

[
[
{

{ [( ) [ ] ( ) }

parenthsis matching



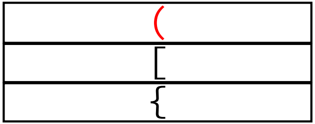
~~[~~

[

{

{ [( ) [ ] ( ) }

parenthsis matching



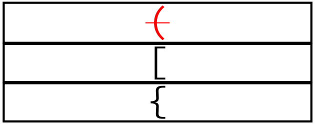
(

[

{

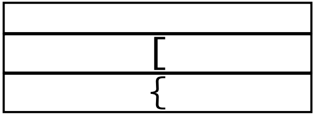
{ [( ) [ ] ( ) }

parenthsis matching



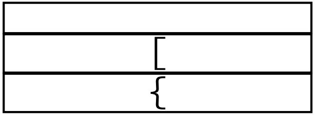
{ [( ) [ ] ( ) }

parenthsis matching



{ [( ) [ ] ( ) }

parenthsis matching



{ [( ) [ ] ( ) }   mismatched!

{ [( ) [ ] ( )] }

some stack applications

undo

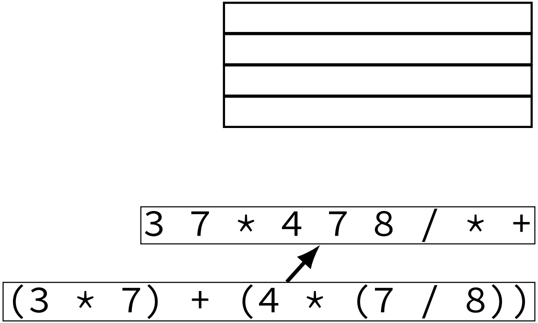
parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls

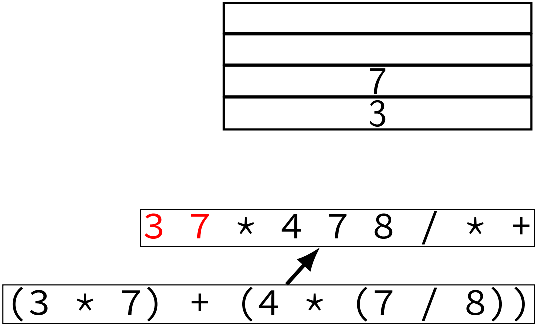
postfix calculations



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

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

postfix calculations



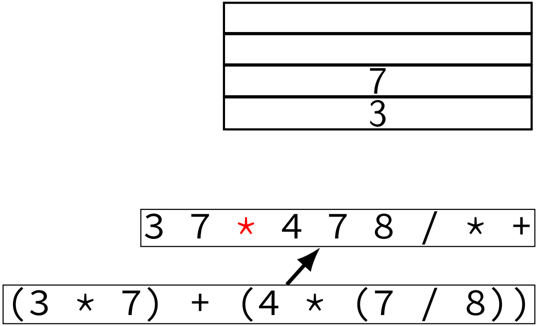
7

3

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

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

postfix calculations



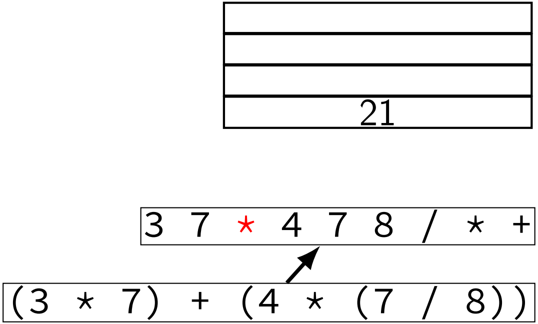
7

3

3 7 * 4 7 8 / * +

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

postfix calculations

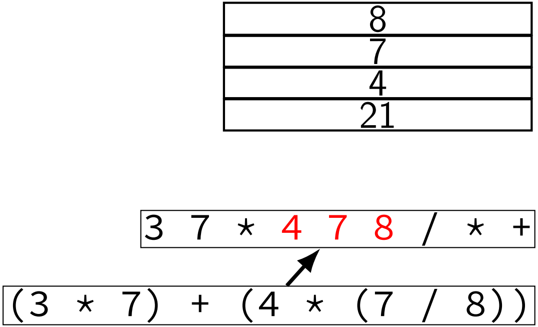


21

`3 7 * 4 7 8 / * +`

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

postfix calculations



8

7

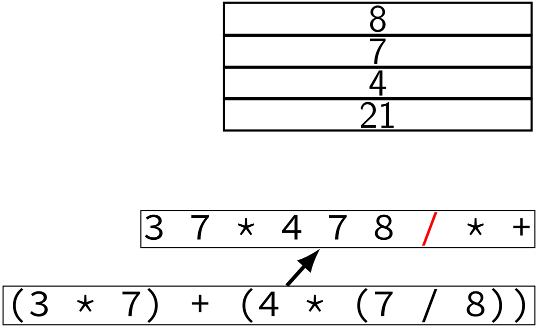
4

21

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

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

postfix calculations

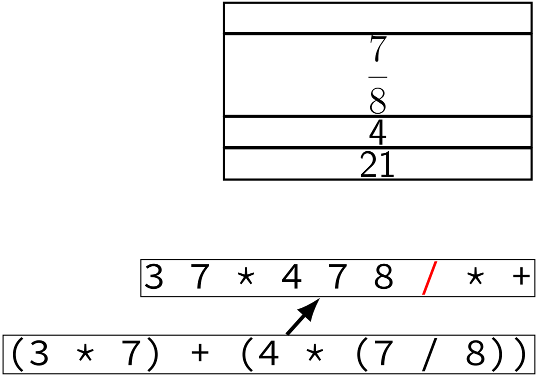


8
7
4
21

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

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

postfix calculations



$\frac{7}{8}$

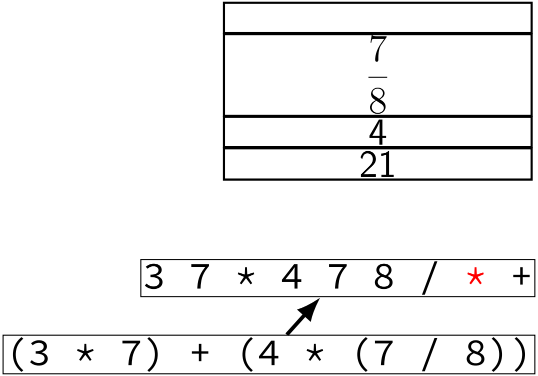
4

21

`3 7 * 4 7 8 / * +`

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

postfix calculations



$\frac{7}{8}$

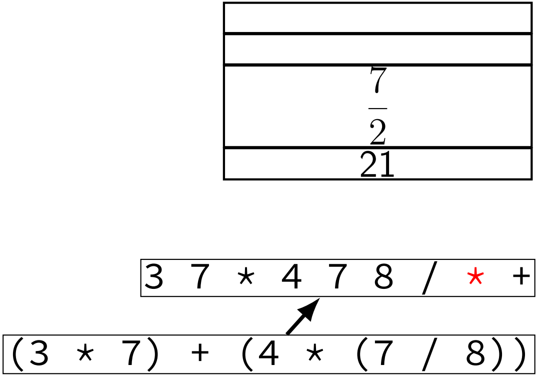
4

21

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

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

postfix calculations

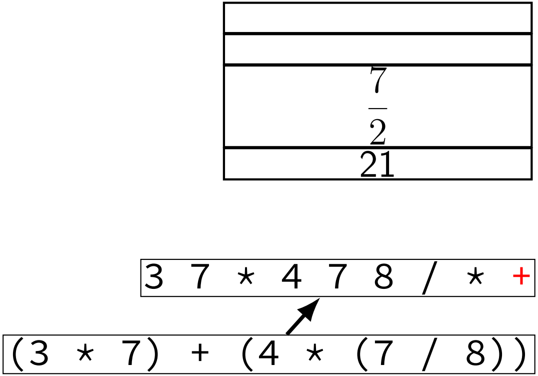


$\frac{7}{2}$
21

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

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

postfix calculations

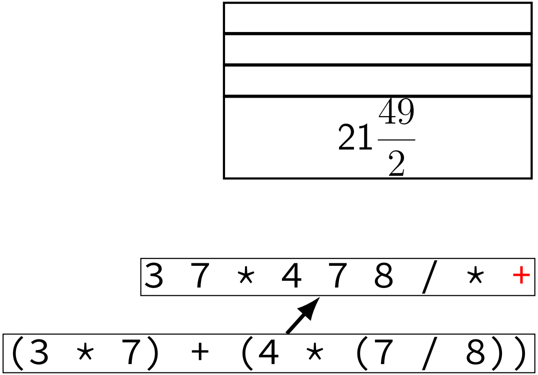


$\frac{7}{2}$
21

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

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

postfix calculations



$21\frac{49}{2}$

3 7 * 4 7 8 / * +

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

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls

some stack applications

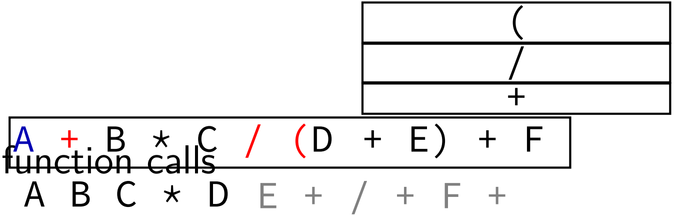
undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls



(

/

+

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

A B C * D E + / + F +

some stack applications

undo

parenthesis matching

postfix calculators

operator precedence

tracking (recursive) function calls

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;
};
```

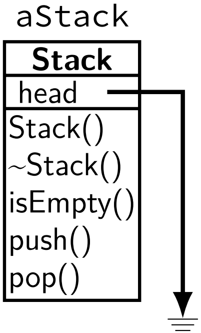
linked list stack of ints

```
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;
```



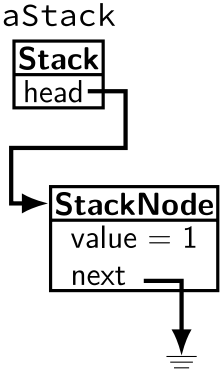
linked list stack of ints

```
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);
```



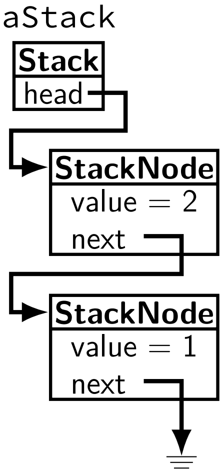
linked list stack of ints

```
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);
}

// ...
```

implementing pop?

```
void Stack::pop() {
    ...
}
```

What could go here?

A. `data.pop_front();`

B. `data.resize(data.size() - 1);`

C. `data.reserve(data.size() - 1);`

D. `data.erase(data.begin());`

E. `data.pop_back();`

implementing pop?

```
void Stack::pop() {
    ...
}
```

What could go here?

A. `data.pop_front();`
B. `data.resize(data.size() - 1);`
C. `data.reserve(data.size() - 1);`
D. `data.erase(data.begin());`
E. `data.pop_back();`

B or E

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];`

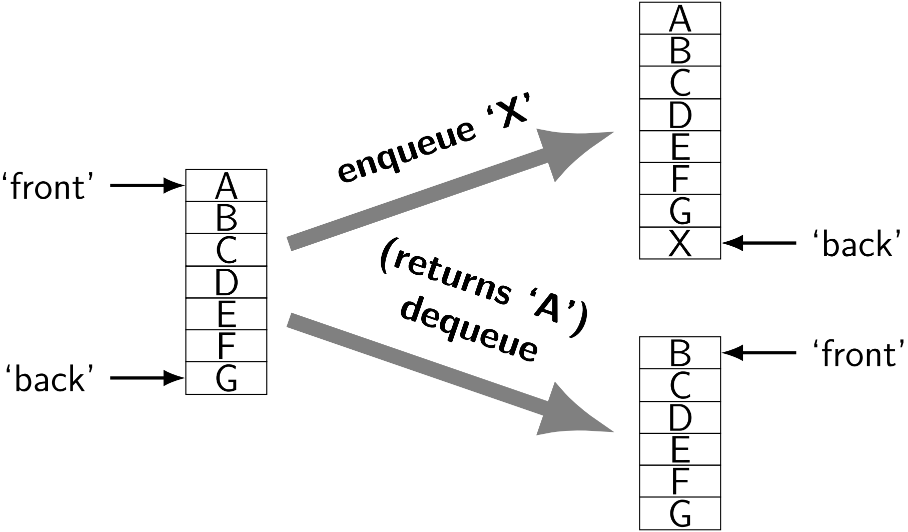
E. `*data.end();`

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’
(returns ‘A’)
dequeue
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, 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;
};
```

```
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;
}
```

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;
}
```

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;
};
```

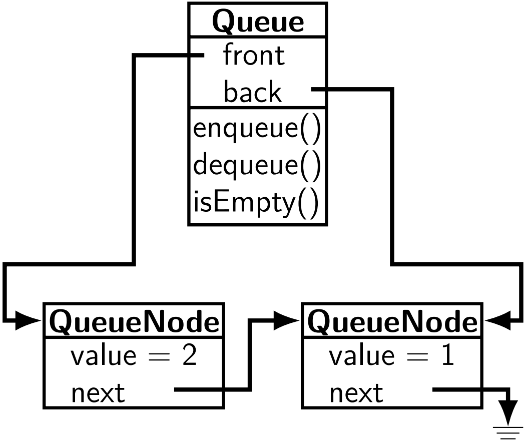
```
void Queue::enqueue(int value) {
  backIndex++;
  if (backIndex >= dataSize)
      ...
  data[backIndex] = value;
}
```

```
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 implementation without library user code changing

functions can be written atop C function with that

implementing ADT options

C++ or Java class

just a collection of functions

…

some ADT examples

stacks

queues

lists

multiple reasonable implementations

same interface

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-linked list
...
```

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('x', 2)
    // becomes {34,12,'x',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
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
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++ strings

```
#include <string>
...
    std::string s, s2;
// Mostly same as vector<char>:
    s.size() == size.length()
    s.at(index), s[index]

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

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