Exceptions (cont.)
Introduction to Linked Lists
Thoughts on Assignment 2

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, September 24, 2012

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- ✓ Pointers & dynamic allocation
- ✓ Managing resources in a class
- ✓ Templates
- ✓ Containers & iterators
- ✓ Error handling
- ✓ Introduction to exceptions
- Introduction to Linked Lists

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- ✓ Testing
- ✓ Some principles

Review
Error Handling

An **error condition** (or “error”) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

- Not necessarily a bug or a user mistake.
- Example: Could not read file.

Three ways to deal with a *possible* error condition in a function:

before — **Prevention**
Client code must prevent the error (precondition).

during — **Containment**
Fix the problem inside the function.

after — **Signal the Client Code**
Idea: When we cannot fulfill our postconditions.

We **like** these two, but they *might* not be feasible

Methods for signaling an error condition to the client code:

- Return an error code
- Set a flag, checked by a separate function
- Throw an **exception**

Review
Introduction to Exceptions — Catching

Exceptions are objects that are “**thrown**”, generally to signal error conditions.

- We **catch** exceptions using a **`try`** … **`catch`** construction.
- “**`throw`**” backs out of blocks & functions, until a matching **`catch`** is found.
- An uncaught exception terminates the program.

```
Foo * makeAFoo() // throw(std::bad_alloc)
{ return new Foo(2, 3); }

void myFunc() // throw()
{
    Foo * p;
    try {
        p = makeAFoo();
    }
    catch (std::bad_alloc & e) {
        allocationSuccessful = false;
        cout << "Oops! Message: " << e.what() << endl;
    }
```

Commented-out **exception specifications**. If uncommented, these are legal C++; I do not recommend using them in release code.

Catch by reference

Review
Introduction to Exceptions — Throwing

We can throw our own exceptions, using "**throw**".

```
class Foo {
public:
    int & operator[](int index)  // May throw std::range_error
    {
        if (index < 0 || index >= arraySize)
            throw std::range_error("Foo: index out of range");
        return theArray[index];
    }
private:
    int * theArray;
    std::size_t arraySize;
};
```

We do not do this very much. And we only do it when we must signal the client code that an error condition has occurred.

Introduction to Exceptions
Example 2

TO DO

- Write a function **`allocate1`** that:
  - Takes a **`size_t`**, indicating the size of an array to be allocated.
  - Attempts to allocate an array of **`int`**s, of the given size.
  - Returns a pointer to this array, using a reference parameter.
  - If the allocation fails, throws **`std::bad_alloc`**.
  - … and has no memory leaks.
- Write a function **`allocate2`** that:
  - Takes a **`size_t`**, the size of **two arrays** to be allocated.
  - Attempts to allocate **two arrays** of **`int`**s, both of the given size.
  - Returns pointer to these arrays, using reference parameters.
  - If the allocation fails, throws **`std::bad_alloc`**.
  - … and has no memory leaks.

Introduction to Exceptions
Final Thoughts

When to Do Things:

- **Throw** when a function you are writing is unable to fulfill its postconditions.
- **Try/Catch** when you can handle an error condition that may be signaled by some function you call.
  - Or simply to prevent a program from crashing.
- **Catch all and re-throw** when you call a function that may throw, you cannot handle the error, but you do need to do some clean-up before your function exits.

Typically we do not do more than one of the above.

- For example, someone else throws, and we catch.

Some people do not like exceptions.

- A *bad reason* not to like exceptions is that they require lots of work.
  - Dealing with **error conditions** is a lot of work. Exceptions are one method of dealing with them. Handling exceptions properly is hard work simply because **writing correct, robust code is hard work**.
- A *good reason* might be that they add hidden execution paths.

Notes on Assignment 2
Overview of Ideas

We have beenn looking at error handling and exceptions. You do *not* need to worry about these on Assignment 2.

You *do* need to be concerned with:

- Pointers & dynamic allocation.
  - Are you doing your dynamic allocation correctly? When you allocate something, is it always freed?
- Managing resources in a class.
  - Class **`KSArray`** should use RAII. This affects how you write it, and how you document it.
- Templates.
  - Class **`KSArray`** is a template. Write and document it appropriately.
- Containers & iterators.
  - Class **`KSArray`** is a generic container. Its member functions **`begin`** and **`end`** return iterators.

Notes on Assignment 2
Thoughts [1/8] — Overall Structure

Your header file should be structured like this:

```
// ksarray.h
...
#ifndef KSARRAY_H
#define KSARRAY_H
...
template <typename T>
class KSArray {

    …
};

...

#endif //#ifndef KSARRAY_H
```

Note: This can (and probably should) be something other than “**`T`**”.

All member-function declarations go here.

There is no file **`ksarray.cpp`**.

All associated global functions and member function definitions (**`operator==`**, etc.) go here.

Notes on Assignment 2
Thoughts [2/8] — Value Type

The template parameter ("**T**" in the code here) is the class's **value type**: the type of all items stored in the container.

- However, you cannot use the template parameter outside the class definition. Make a **typedef**, so that you can.

```
template <typename T>
class KSArray {
    ...
public:
    typedef T value_type;
    ...
private:
    value_type * arrayPtr_;
    ...
};
```

This lets you say "**value_type**" anywhere you mean "the type of the items in the container". For example, **here**.

Other data member(s)?

Notes on Assignment 2
Thoughts [3/8] — Constructors

Array items are *always* default-constructed in C++. You cannot set their values to anything else in an initializer. Therefore, the **copy ctor** will need a loop* in the function body.

```
// copy ctor
KSArray(const KSArray & other)
    :arrayPtr_(new ... ), ...
{ ... }

...
value_type * arrayPtr_;
...
```

Initialize array items with a loop* **here**.

*Or maybe one of the generic algorithms from the STL? (Hint, hint …)

Notes on Assignment 2
Thoughts [4/8] — Copy Assignment

Remember:

- The copy ctor **creates a new object**, which is a copy of some existing object.
- The copy assignment operator **sets an existing object** equal to a copy of some other existing object.

In your copy assignment operator:

- Check for self-assignment.
- If this is not self-assignment, then (1) **deallocate the old array**, (2) do essentially what the copy ctor does.
- Regardless, at the end, return the object assigned.

```
KSArray & operator=(const KSArray & rhs)
{
    if (this != &rhs)
    {
        delete [] arrayPtr_;
        ...
    }
    return *this;
}
```

Code very similar to the copy ctor (except that the copy ctor, being a ctor, can use an initializers, while this cannot).

Note: Later in the semester, we will discuss copy operations further.

Notes on Assignment 2
Thoughts [5/8] — Access to Internal Data

A **`const KSArray`** is supposed to have non-modifiable data. Therefore, if a member function gives access to internal data in a modifiable form, then you will need to **write two versions** of it.

```
... operator[]( ... )
{ ... }
... operator[]( ... ) const
{ ... }

... begin()
{ ... }
... begin() const
{ ... }

... end()
{ ... }
... end() const
{ ... }
```

In each pair, the two functions should be essentially identical, except for (1) the **`const`** at the end of the first line, and (2) the return method. Or, you may use a const_cast trick to call the const version from the non-const version.

Notes on Assignment 2
Thoughts [6/8] — Iterators

Member functions **begin** and **end** are to return iterators.

- These can be pointers. Do not write a separate iterator class.
- Function **begin** returns an iterator to the first array item. You already have a pointer to the first array item (think …); use it.
- Function **end** returns an iterator to just past the last array item. Add a number to the return value of **begin** (what number? …).

```
... begin()
 { return ...; }
...
... end()
{ return ...; }
```

Pointer type.

Notes on Assignment 2
Thoughts [7/8] — Global Functions

If a global function is to use **`KSArray`** in its full generality, then *that function* will need to be a template.

- For example, your **`operator==`**, **`operator<`**, etc. should be able to compare *any* kind of **`KSArray`** (as long as the value type has the proper operator(s) defined). So make your **`operator==`**, **`operator<`**, etc. function templates.

```
template <typename T>
bool operator==(... KSArray<T> ..., ... KSArray<T> ...)
{ ... }

template <typename T>
bool operator<(... KSArray<T> ..., ... KSArray<T> ...)
{ ... }

...
```

These go **outside** the class definition.

Notes on Assignment 2
Thoughts [8/8] — Documentation

We still need **preconditions** and **postconditions** for all functions and **class invariants** for all classes.

In addition, we need **requirements on types** for all templates.

- This means class **`KSArray`** and all global functions; they will all be templates.

```
// Invariants: ...
// Requirements on Types: T must have ...
template <typename T>
class KSArray {
    ...
};

// Pre: ...
// Post: ...
// Requirements on Types: T must have ...
template <typename T>
bool operator==( ... )
{ ... }
```

What has to be true about type **`T`** for this template to be compiled and used successfully? Typically: List member functions or associated global functions that **`T`** needs to have.

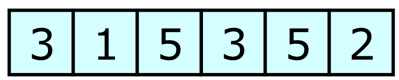
Introduction to Linked Lists
Basics

We now take a brief look at **Linked Lists**.

We discuss Linked Lists in detail later in the semester. For now:

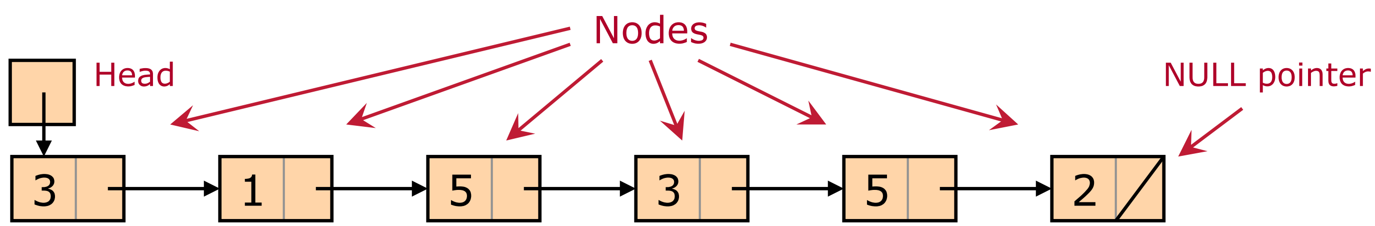
- Like an array, a Linked List is a structure for storing a sequence of items.

Array



- A Linked List is composed of **nodes**. Each has a single data item and a pointer to the next node.

Linked List



- These pointers are the **only** way to find the next data item. Thus, unlike an array, we cannot quickly skip to (say) the 100th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access data structure. Thus, its natural iterator is a **forward iterator**, which has only the ++ operator.

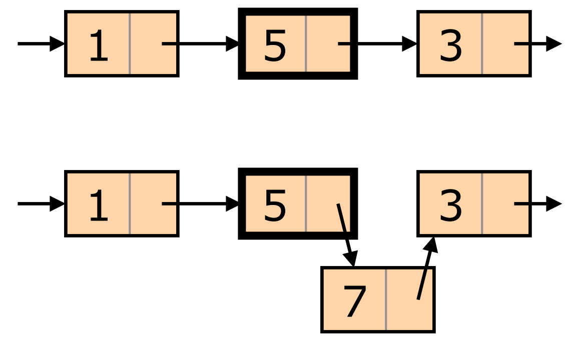
Introduction to Linked Lists
Advantages

Why not always use (smart) arrays?

- One important reason: we can often insert and remove much faster with a Linked List.

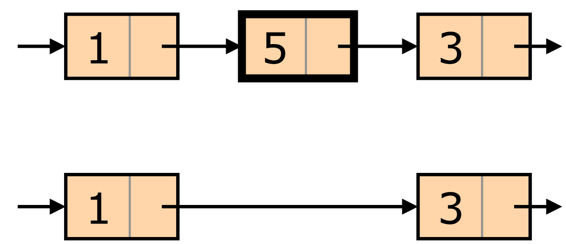
Inserting

- Inserting an item at a given position in an array is slow-ish.
- Inserting an item at a given position (think "iterator") in a Linked List is very fast.
- Example: insert a "7" after the bold node.



Removing

- Removing the item at a given position from an array *is also slow-ish.*
- Removing the item at a given position from a Linked List is very fast.
  - We need an iterator to the *previous* item.
- Example: Remove the item in the bold node.



Introduction to Linked Lists
Implementation

A Linked List node might be implemented like this.

```
template <typename ValueType>
struct LLNode {
    ValueType data_;  // Data for this node
    LLNode * next_;   // Ptr to next node, or NULL if none

    // The following simplify creation & destruction
    LLNode(const ValueType & theData, LLNode * theNext = 0)
        :data_(theData), next_(theNext)
    {}
    ~LLNode()
    { delete next_; }
};
```

Then the head of our list would keep an **(LLNode<…> *)**.

Introduction to Linked Lists
Write Something

TO DO

- Write a function to find the size (number of nodes) of a Linked List, given an **`(LLNode<…> *)`**.