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

25 Sep 2009 CS 311 Fall 2009
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:**

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

- **Containment**
  - Fix the problem inside the function.

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

**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.

```cpp
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.
We can catch **all** exceptions, using “...”.  
- In this case, we do not get to look at the exception, since we do not know what type it is.

```java
try {
    myFunc4(17);
} catch (...) {
    fixThingsUp();
    throw;
}
```

- Inside any `catch` block, we can **re-throw the same exception** using `throw` with no parameters.
Destructors generally should not throw.

- Why? Destructors are called when an automatic object goes out of scope due to an exception. If such a destructor throws, then the program terminates.
- Another reason for this is that a throwing destructor says, “This object cannot be destroyed (now).” Thus, the function that created the object cannot exit, the program cannot end, etc.
Review
Introduction to Exceptions — Example 2

Last time, we did this:

- Write a function `allocate1` that:
  - Takes a `size_t`, indicating the size of an array to be allocated.
  - Attempts to allocate an array of `ints`, 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 `ints`, 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.

See `allocate2.cpp`, on the web page.
When to Do Things:

- **Throw** when a function you are writing is unable to fulfill its postconditions.
- **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.
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 Diagram]

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

  ![Linked List Diagram]

- 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<>*`.

Done. See `list_size.cpp`, on the web page.
Unit Overview
Recursion & Searching

We now begin a unit on recursion & searching.

Major Topics
- Introduction to Recursion
- Search Algorithms
- Recursion vs. Iteration
- Eliminating Recursion
- Recursive Search with Backtracking

We will follow the text a little more closely than we have been.
- Recursion & Searching material is in chapters 2 & 5.

After this, we will look at Algorithmic Efficiency & Sorting, covered in chapter 9.
A recursive algorithm is one that makes use of itself.

- An algorithm solves a problem. If we can write the solution of a problem in terms of the solutions to simpler problems of the same kind, then recursion may be called for.
- At some point, there needs to be a simplest problem, which we solve directly. This is the base case.
Introduction to Recursion Basics — Four Questions

When designing a recursive algorithm or function, consider the following questions (text, page 68):

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

This is critical!
We start with a (somewhat silly) example: Write a recursive function to find the sum of the first \( n \) integers, given \( n \).

- So, given 3, we return \( 1 + 2 + 3 = 6 \).
- We look at this as practice in thinking about recursion. Then we will try a more serious example.
Introduction to Recursion
Sum Example — Four Questions

How can you define the problem in terms of a smaller problem of the same type?

- $1 + 2 + \ldots + n = [1 + 2 + \ldots + (n-1)] + n$.
- Say $f(n) = 1 + 2 + \ldots + n$. Then, for $n > 0$, $f(n) = f(n-1) + n$.
  - This is called a **recurrence relation**.

How does each recursive call diminish the size of the problem?

- It reduces by 1 the number of numbers to be summed.

What instance of the problem can serve as the base case?

- $n = 0$.
  - $n = 1$ would also have worked.

As the problem size diminishes, will you reach this base case?

- Yes, as long as $n$ is nonnegative.
  - Therefore the statement “$n \geq 0$” needs to be a **precondition**.
Next we write specifications.

- Let’s write a recursive function `sumUpTo` that takes a single `int` parameter and returns an `int`.
  - The parameter will be “n”, and the return value will be the sum.
- Preconditions
  - \( n \geq 0 \).
- Postconditions
  - Return \( = 1 + \ldots + n \).

The recurrence relation turns into an algorithm.

- Are we in the base case? If so handle it.
  - If \( n \) is 0, then return 0.
- Otherwise, recurse.
  - Recursive call, parameter: \( n - 1 \).
  - Add \( n \) to the result.
  - Return this.

For \( n > 0 \),
\[
f(n) = f(n-1) + n.
\]
Introduction to Recursion
Sum Example — Coding

Now we write the actual code:

```c
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n) {
    if (n == 0)
        return 0;
    else
        return sumUpTo(n-1) + n;
}
```

How do we know we can make the recursive call?

- Hint: When we call a function, we must satisfy its preconditions.
Introduction to Recursion
Sum Example — Invariants

We know we can make the recursive call because we have an invariant that makes the preconditions for the call true:

```c
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    if (n == 0)
        return 0;
    else // Invariant: n >= 1. (Therefore n-1 >= 0.)
        return sumUpTo(n-1) + n;
}
```

**Here** we have an invariant that says \( n \geq 0 \) (the precondition).

**Here** we leave if \( n \) is exactly 0.

**Result:** If we stay, then \( n \geq 1 \).

**This** is the precondition for this function call.
Introduction to Recursion
Sum Example — Iterative Version

Often we do not really need recursion:

```java
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n) {
    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    return sum;
}
```

This uses iteration (a loop) instead.
And sometimes there is a not-so-obvious way to do things \textit{much} faster:

```c
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + \ldots + n.
int sumUpTo(int n)
{
    return n * (n+1) / 2;
}
```
Search Algorithms
Binary Search Example — What is It?

The sum example from “Introduction to Recursion” was a little silly. Here is one that is not.

Binary Search

- How does it work?
- How would you implement it recursively?
Binary Search is an algorithm to find a given key in a sorted list.
- Here, key = thing to search for. Often there is associated data.
- In computing, sorted = in (some) order.

Procedure
- Pick an item in the middle: the pivot.
- Use this to narrow search to top or bottom half of list. Recurse.

Example: Binary Search for 64 in the following list.

1. Looking for 64 in this list.

   5 8 9 13 22 30 34 37 38 41 60 63 65 82 87 90 91


3. Recurse: Looking for 64 in this list.
Search Algorithms
TO BE CONTINUED ...