# Software Engineering Concepts: Abstraction
# A Little about Linked Lists
# Introduction to Recursion

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 14, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu



Some material contributed by Chris Hartman

# Unit Overview
# Advanced C++ & Software Engineering Concepts

## Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- ✓ Integer types
- ✓ Managing resources in a class
- ✓ Containers & iterators
- ✓ Invisible functions II
- ✓ Error handling
- ✓ Using exceptions
- A little about Linked Lists

## Major Topics: S.E. Concepts

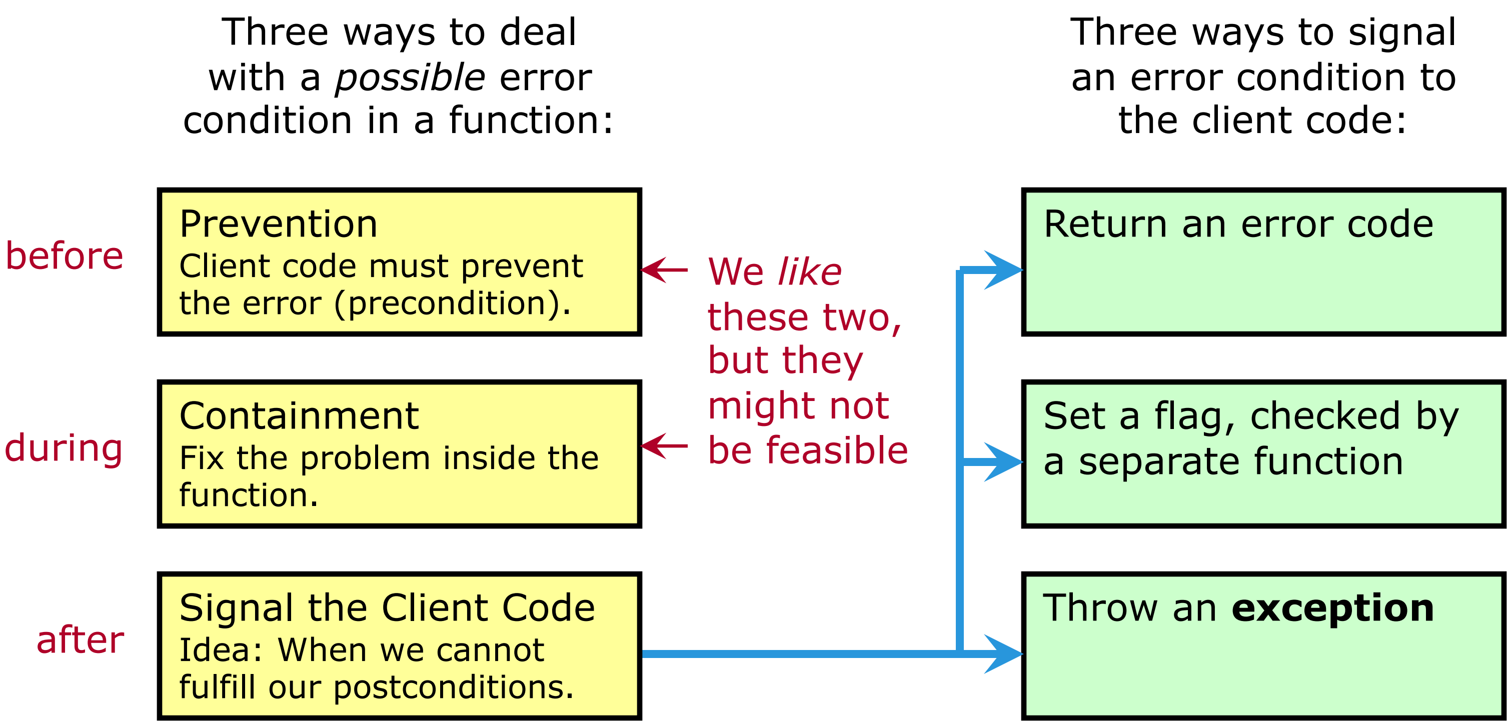
- ✓ Invariants
- ✓ Testing
- Abstraction

# Review

# Review
# Error Handling

An **error condition** (often *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.



## Review
## Using Exceptions [1/3]

**Exception**: an object that is **thrown** to signal an error condition.
To handle an exception, **catch** it using `try` … `catch`.

```
Foo * p;
bool success = true;
try {
    p = new Foo;
}
catch (std::bad_alloc & e) {
    success = false;
    cerr << "Alloc. failed: "
         << e.what() << endl;
}
```

`new` throws `std::bad_alloc` (`<new>`) or a derived class, if memory allocation fails.

How It Works

- When an exception is thrown inside a try-block, control passes to the catch-block that (1) is associated with the smallest possible enclosing try-block, and (2) catches the proper type. Derived classes are handled as usual.
- In all other circumstances, a catch-block is not executed.

## Review
## Using Exceptions [2/3]

**Catch**—when you can handle an error signaled by a function you call.

Catch exceptions by reference.

```
try { … }
catch (std::out_of_range & e) {
```

**Throw**—when your function is unable to fulfill its postconditions.

```
if (ix >= arrsize) throw std::out_of_range("bad index");
```

**Catch all & re-throw**—when you call a throwing function, and you cannot handle the error, but your function must clean up before exiting.

The code contains three dots.

```
try { … }
catch (...) {
    [Clean up here]
    throw; }
```

We generally **only write one** of the three: catch, throw, or catch all & re-throw.

Another might be written by someone else.

# Review
# Using Exceptions [3/3]

**Destructors should not throw.**

Why? Destructors are called when an automatic object goes out of scope due to an exception. If the destructor throws in this context, then the program terminates.

Because of this, generally the destructors in your classes are implicitly marked noexcept, unless you specify otherwise.

- To specify otherwise: "`noexcept(false)`". However, this is *EVIL*. ☹

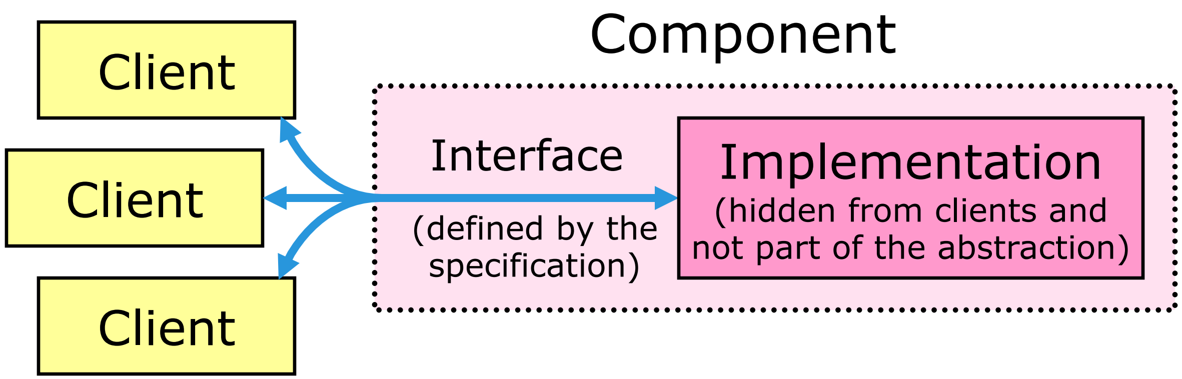
If a noexcept function throws, then the program terminates.

A throwing *constructor* is fine. Throwing is the standard way to signal that an object cannot be successfully constructed.

# Software Engineering Concepts: Abstraction

## Software Engineering Concepts: Abstraction Definitions

**Abstraction**: Considering a software component in terms of *how* and *why* it is used, separate from its implementation.




Here, "**component**" is just a general term for a *thing*: function, class, package, etc.

We use the term "**client**" for *code* that uses a component.

A client is code. A **user** is a person.

## Software Engineering Concepts: Abstraction
## Functional & Data Abstraction [1/2]

**Functional abstraction** means applying the idea of abstraction to a function.

In the second half of the semester, we will talk about **data abstraction**: applying abstraction to data.

You have certainly used these ideas—even if you were not familiar with the terms “functional abstraction” and “data abstraction". *See the next slide for an example.*

# Software Engineering Concepts: Abstraction
# Functional & Data Abstraction [2/2]

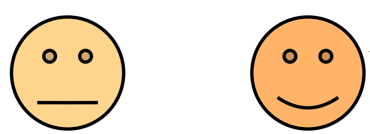
```
void printIntVec(const vector<int> & data)
{
    for (size_t i = 0; i != data.size(); ++i)
        cout << data[i] << " ";
    cout << endl;
}
```

Functional abstraction

Describe this function, in detail.

Function `printIntVec` is given a `vector` of `int`s called `data`, passed by reference-to-const. It executes a `for` loop in which local `size_t` variable `i` is initialized to `0`, the loop continues as long as "`i != data.size()`" evaluates to `true`, and `i` is pre-incremented after each loop iteration. Inside the loop, a reference to an item in `data` is retrieved using the bracket operator, with parameter `i`, and then inserted into `cout`, using overloaded `operator<<`, followed by an array of `char`s of size 2, which contains a blank and a null `char`. After the loop, stream manipulator `endl` is inserted into `cout`. The function then terminates.

Function `printIntVec` prints a given `vector` of `int`s to `cout`. Items are separated by blanks, and followed by a blank and a newline.



# A Little about Linked Lists

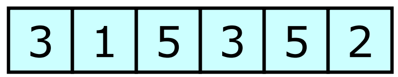
# A Little about Linked Lists
# Basics [1/2]

We now take a brief look at a container called a **Linked List**.

Later in the semester we discuss Linked Lists in detail. For now:

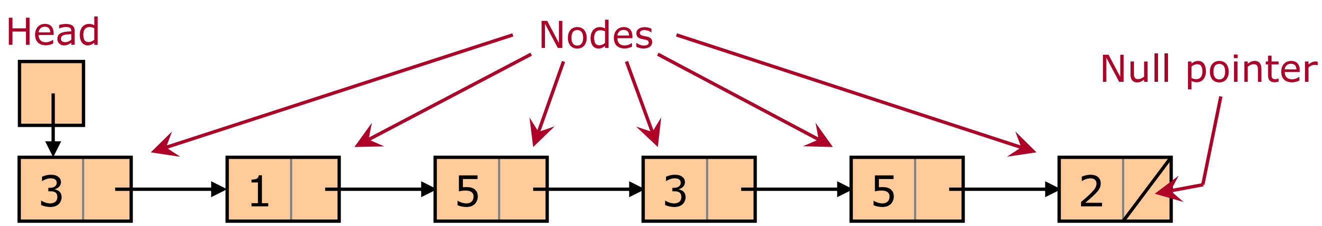
- Like an array, a Linked List stores a sequence of data items.

Array




- A Linked List is made of **nodes**. Each has a single data item and a pointer to the next node, or a null pointer at the end of the list.

Linked List

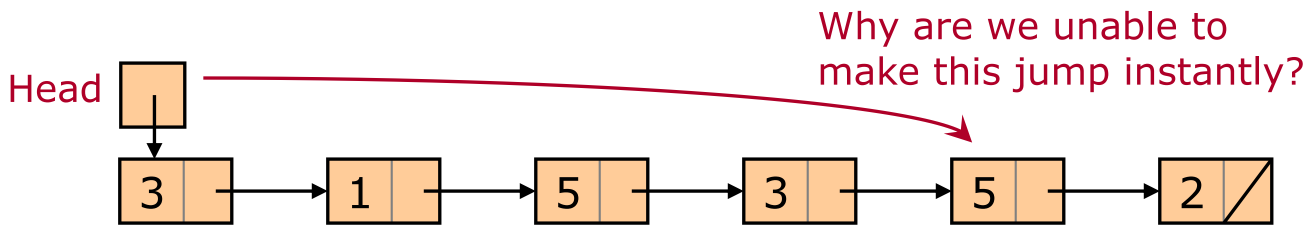



- These pointers are the only way to find the next item. Unlike with an array, we cannot quickly find (say) the 100,000th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access structure. So its iterators are **forward iterators**, which have only the ++ operator.

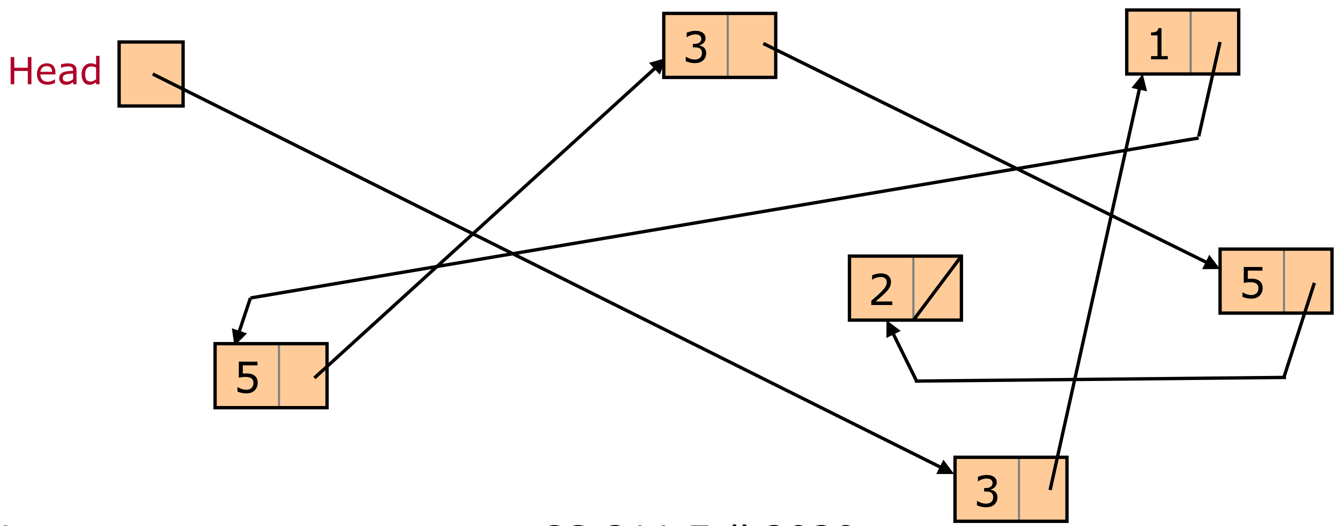
# A Little about Linked Lists
# Basics [2/2]

We *cannot* quickly find a Linked List item, given only its index.

Why not? It certainly looks as if we could.




But the above picture can be deceptive. A Linked List might actually be arranged in memory more like this:



# A Little about Linked Lists
## Advantages

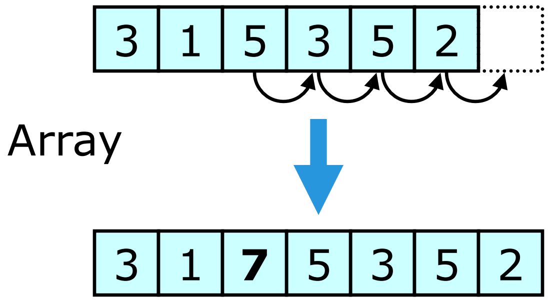
Why not always use (smart) arrays?

One reason: Linked Lists support fast insertion.

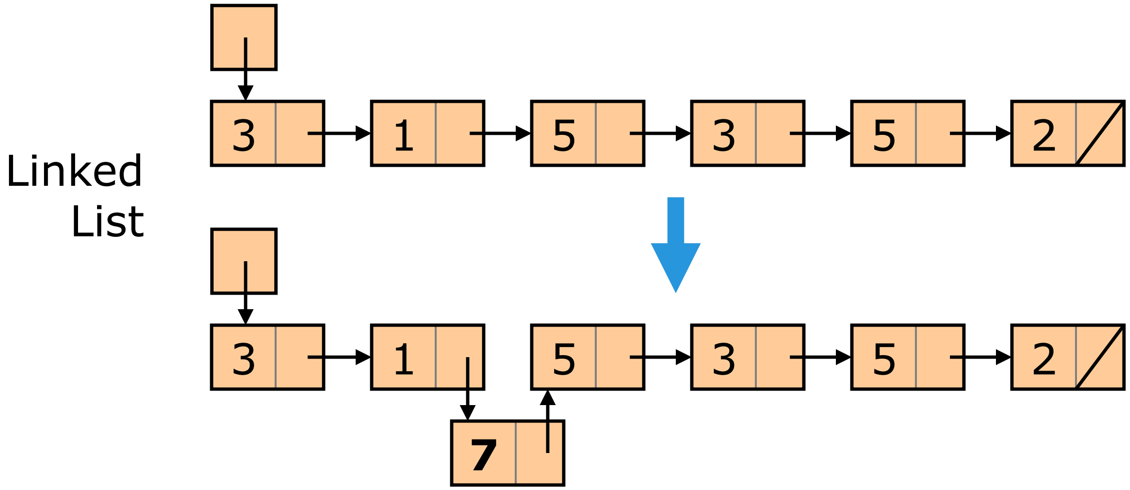
Suppose we have a sequence 3, 1, 5, 3, 5, 2.
We wish to insert a 7 before the first 5.

With an array, we move all later items up.

With a Linked List, *if we know the proper location*, insertion is very fast.




For long sequences, the speed difference can be huge.



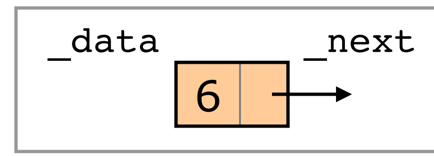
# A Little about Linked Lists
# Implementation

Here is one possible implementation of a Linked List node.

The data members are *public*??!?!?

In practice, only the Linked List package deals with this `struct`, so these are not a problem.

```
template <typename ValType>
struct LLNode {
    ValType   _data;  // Data for this node
    LLNode *  _next;  // Ptr to next node, or nullptr if none
    // The following simplify creation & destruction
    explicit LLNode(const ValType & data,
                    LLNode * next = nullptr)
        :_data(data), _next(next)
    {}
    ~LLNode()
    { delete _next; }
};
```




If `_next` points to a node, then `delete` calls that node's destructor, which will delete *its* `_next` pointer, which calls the destructor of the node after that, etc.

So this destructor calls itself; it is **recursive**. This is convenient! However, it can be problematic if there are lots of nodes. More on this later in the semester.

The head of our Linked List would hold an `(LLNode<…> *)`.

# A Little about Linked Lists
# Implementation — CODE

TO DO

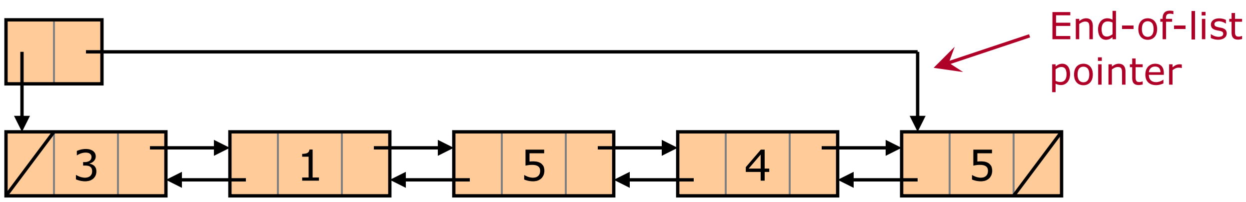
- Write a function to find the size (number of items) of a Linked List, given its head pointer `(LLNode<…> *)`.

*Done. See* `list_size.cpp`*.*
*See* `llnode.h` *for a header that defines* `LLNode`*.*

# A Little about Linked Lists
## Doubly Linked Lists

In a **Doubly Linked List**, each node has two pointers: next-node (null at the end) and previous-node (null at the beginning).

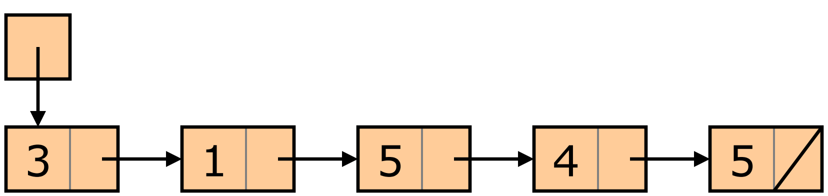
Doubly Linked List




Doubly Linked Lists typically have not only a beginning-of-list pointer, but also an end-of-list pointer.

To make it clear what we are talking about, the one-pointer-per-node Linked List has a longer name: **Singly Linked List**.

(Singly) Linked List



# Unit Overview
# Recursion & Searching

This ends the introductory/review material.

We now begin a short unit on recursion and searching.

Major Topics

- Introduction to recursion
- Search algorithms I
- Recursion vs. iteration
- Search algorithms II
- Eliminating recursion
- Search in the C++ STL
- Recursive backtracking

There will be a number of topics like this one—typically at the end of our coverage of some data structure or algorithmic idea.

After this, we will cover Algorithmic Efficiency & Sorting.

# Introduction to Recursion

# Introduction to Recursion
## Basics — Definitions

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 **smaller** problems of the same kind, then recursion may be called for.
- There must be a smallest problem, which we solve directly. This is a **base case**. (Others are **recursive cases**.)

Similarly, a **recursive** function is one that calls itself.

- Such calls are typically **direct**, but may be **indirect**.
- When a function calls itself, it is making a **recursive call**. We also say it **recurses**.

**Base case**

**Direct** recursion

**Recursive case**

**Indirect** recursion

```
int mult(int a, int b)
{
    if (a <= 1)
        return a == 1 ? b : 0;

    int ax = (a >> 1);
    int m1 = mult(ax, b);
    return m1 + m2(a, ax, b);
}

int m2(int a, int ax, int b)
{
    return mult(a-ax, b);
}
```

# Introduction to Recursion
# Basics — Four Questions

When designing a recursive algorithm or function, consider the following four questions*:

1. How can we solve the problem using solutions to one or more smaller problems of the same kind?
2. How much does each recursive call reduce the size of the problem?
3. What instances of the problem can serve as base cases?
4. As the problem size shrinks, will a base case always be reached?

This is critical!

Every call to a recursive function must eventually reach a base case.

*Adapted from Frank M. Carrano, *Data Abstraction and Problem Solving with C++: Walls and Mirrors*, 4th ed., 2004.

# Fibonacci Numbers — Definitions [1/2]

The **Fibonacci numbers** are the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, …

To get the next Fibonacci number, add the two before it.

We denote the nth Fibonacci number by $F_n$ ($n$ = 0, 1, 2, …). So $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, etc.

Now we can formally define the Fibonacci numbers as follows:

- $F_0 = 0$.
- $F_1 = 1$.
- For $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

Why are we talking about this?

Computing Fibonacci numbers is our first example of a problem that can be solved by multiple algorithms. Some of these algorithms are recursive, and they differ greatly in speed.

# Fibonacci Numbers — Definitions [2/2]

The Fibonacci numbers ($F_n$, for $n$ = 0, 1, 2, …):

- $F_0 = 0$.
- $F_1 = 1$.
- For $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

An equation defining a sequence of numbers in terms of itself, as above, is a **recurrence relation** (often simply **recurrence**).

The values for the start of the sequence are **initial conditions**.

A recurrence often translates nicely into a recursive algorithm.

Let's do such a translation for the Fibonacci numbers: write a recursive function `fibo` that takes an integer $n$ and returns the $n$th Fibonacci number $F_n$.

# Fibonacci Numbers — Four Questions

1. How can we solve the problem using solutions to one or more smaller problems of the same kind?
   - Use the recurrence: $F_n = \boxed{F_{n-2}} + \boxed{F_{n-1}}$.

   Smaller problems of the same kind

2. How much does each recursive call reduce the size of the problem?
   - The first call: by 2. The second call: by 1.

3. What instances of the problem can serve as base cases?
   - Use the initial conditions: $n = 0$, $n = 1$.

4. As the problem size shrinks, will a base case always be reached?
   - Yes, as long as $n$ is nonnegative.
   - So function `fibo` should have "$n \geq 0$" as a precondition.

# Fibonacci Numbers — Design Decisions

Recall: `fibo` takes an integer *n* and returns the *n*th Fibonacci number $F_n$. (I write this as "`F(n)`" in source-code comments.)

What should the parameter and return types for `fibo` be?

- The parameter can be `int`.
- As *n* grows, $F_n$ will grow very quickly. So we need to guard against **numeric overflow**. Let's use a 64-bit unsigned integer for the return type: `std::uint_fast64_t`. A **type alias** could be helpful:

```
using bignum = uint_fast64_t;
```

What pre- and postconditions should `fibo` have?

- Pre: `n >= 0`. Also, `F(n)` is a within the range of values of `bignum`. (Some checking shows that this requires `n <= 93`.)
- Post: `Return == F(n)`.

When we write a recursive function, we usually want to check for the base case(s) first. If we are not in a base case, then we are in a recursive case.

TO DO

- Write recursive function `fibo`, as described.

*Done. See* `fibo_first.cpp`*.*

Function `fibo` turns out to be extremely slow for anything other than small parameters. We will revisit it, rewriting it in several different ways. Most of these will be *much* faster. (Some of the fast versions will be recursive; recursion is not inherently slow!).