

# Arrays & Linked Lists

## Introduction to Recursion

---

CS 311 Data Structures and Algorithms  
Lecture Slides  
Monday, September 16, 2024

Glenn G. Chappell  
Department of Computer Science  
University of Alaska Fairbanks  
[ggchappell@alaska.edu](mailto:ggchappell@alaska.edu)

© 2005–2024 Glenn G. Chappell  
Some material contributed by Chris Hartman

# Unit Overview

## Advanced C++ & Software Engineering Concepts

### Topics: Advanced C++

- ✓ ■ Expressions
- ✓ ■ Parameter passing I
- ✓ ■ Operator overloading
- ✓ ■ Example class
- ✓ ■ Parameter passing II
- ✓ ■ Invisible functions
- ✓ ■ Managing resources in class
- ✓ ■ Containers & iterators
- ✓ ■ Invisible functions II
- ✓ ■ Error handling
- ✓ ■ Using exceptions

### Topics: S.E. Concepts

- ✓ ■ Abstraction
- ✓ ■ Assertions
- ✓ ■ Testing
- ✓ ■ Invariants

**DONE**

---

# Review

# Review

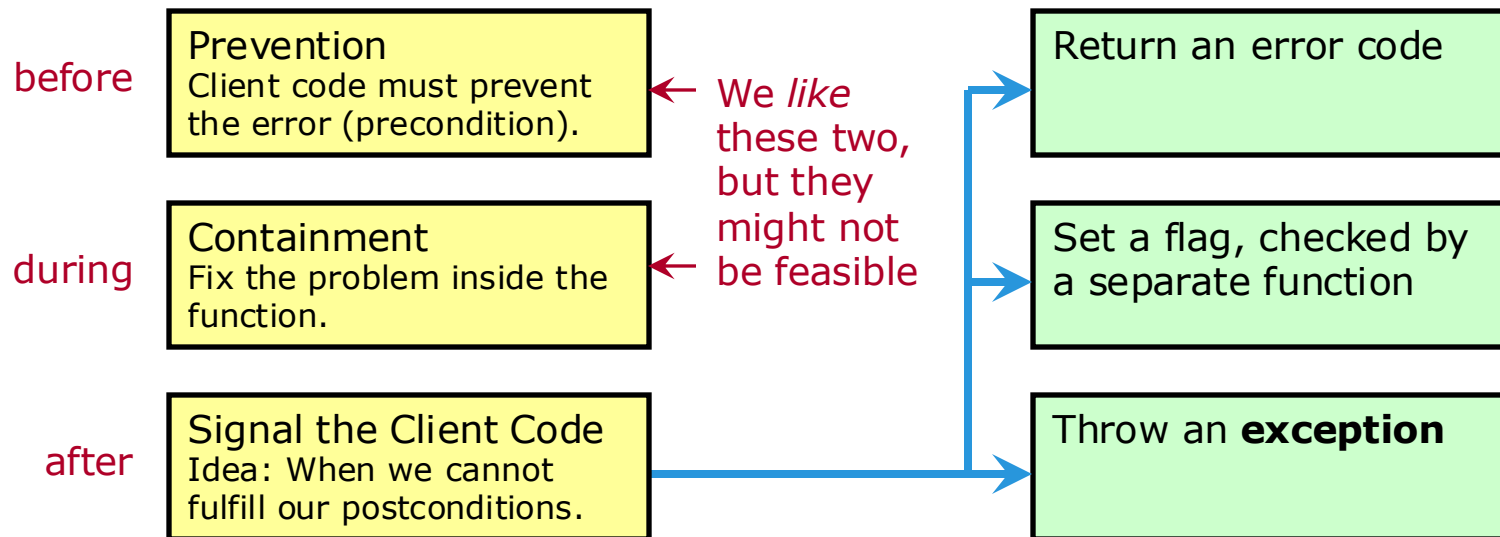
## Error Handling

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.


Three (and only three) ways to deal with a possible error condition in a function:

At least three ways to signal an error condition to client code:



**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 is associated with the smallest possible enclosing try-block that catches the proper type. Derived classes are handled as usual.
- In all other circumstances, a catch-block is not executed.
- An uncaught exception terminates the program.

## Review

### Using Exceptions [2/4]

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

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


 Catch exceptions  
by reference.

**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.

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

 The code contains  
three dots.

We typically **only write one** of the three:  
catch, throw, or catch all & re-throw.  
Another might be written by someone else.

## **Destructors should not throw.**

It is okay for  
constructors  
to 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`—a promise that they will not throw.

Some people do not like exceptions. Some of these people are very vocal about their dislike. But I think that *some* of them dislike exceptions for the wrong reasons.

A bad reason to dislike exceptions is that they require lots of work.

- Dealing with error conditions is work. Writing software that *works* is work. Exceptions are one tool we can use to achieve this goal.
- Handling exceptions properly is hard work because *writing correct software is hard work*.

What *might* be a good reason to dislike exceptions is that they add hidden execution paths. But remember that other error-handling methods have their own downsides—which is why exceptions were invented.



# Unit Overview

## Recursion & Searching

---


This ends the introductory material.

We now begin a unit on recursion and searching.

### Topics

- Arrays & Linked Lists
- 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.

---

# Arrays & Linked Lists

# Arrays & Linked Lists

## Arrays

---

This unit covers recursion and searching. In-depth coverage of data structures will wait until later in the semester. But if we are talking about searching, then we need data structures to search.

The simplest container data structure is the **array**. This stores a sequence of items, placing them one after the other in a contiguous block of memory.

Array     

3	1	5	3	5	2
---	---	---	---	---	---

An array is a random-access structure. Since we know where in memory each item lies, we can look up an item quickly by its index, typically using a bracket operator: `cout << arr[1000];`

In C++, the following hold arrays: `std::vector`, `std::array`, `std::basic_string` (basis of `std::string`), and built-in arrays.

# Arrays & Linked Lists

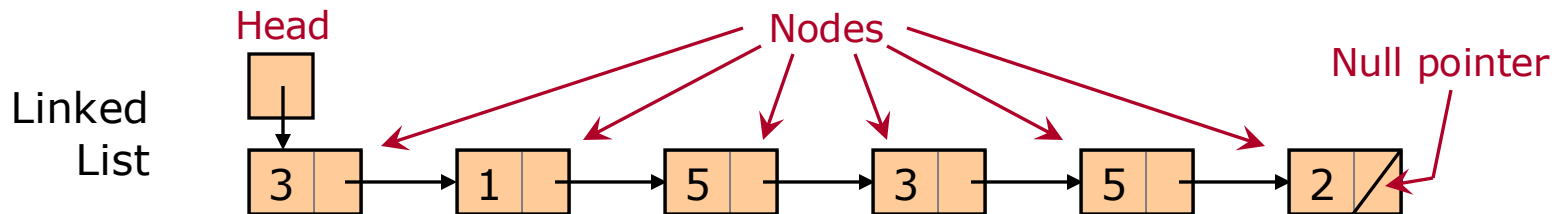
## Linked Lists — Basics [1/2]

Another container data structure is the **Linked List**. Like an array, a Linked List stores a sequence of items.

Array 

3	1	5	3	5	2
---	---	---	---	---	---

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. We keep track of a Linked List using its **head** pointer.

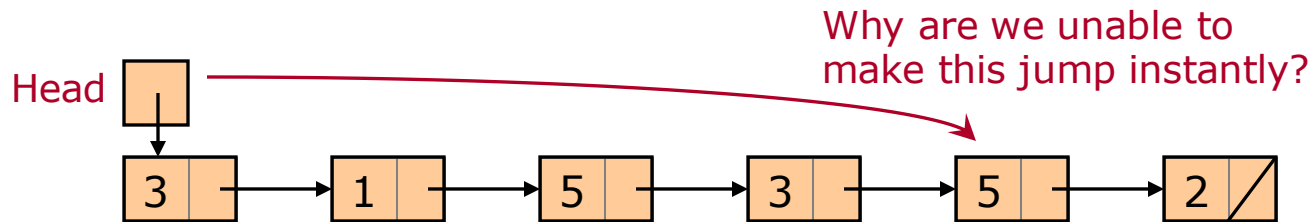


A Linked List is a forward-only sequential-access structure. To find items, we follow pointers through the list. We cannot quickly find the 100,000th item. Nor can we quickly find the previous item.

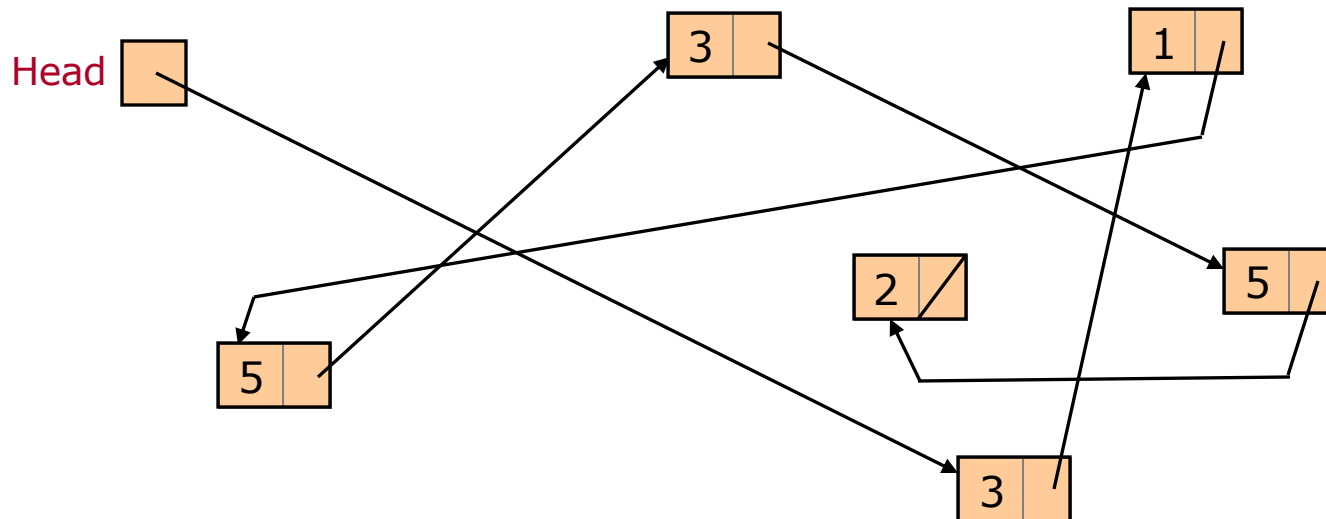
# Arrays & Linked Lists

## 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:



# Arrays & Linked Lists

## Linked Lists — Advantages

Why not always use (smart) arrays?

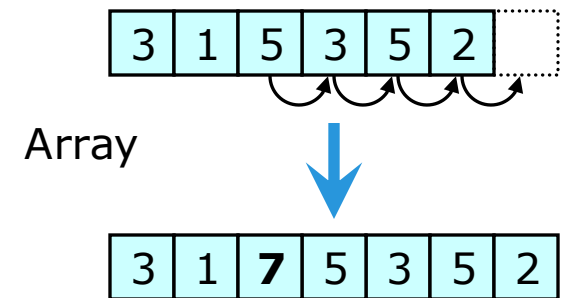
One reason: Linked Lists support fast insertion.

Suppose we have a sequence 3, 1, 5, 3, 5, 2.

We wish to insert a 7 before the first 5.

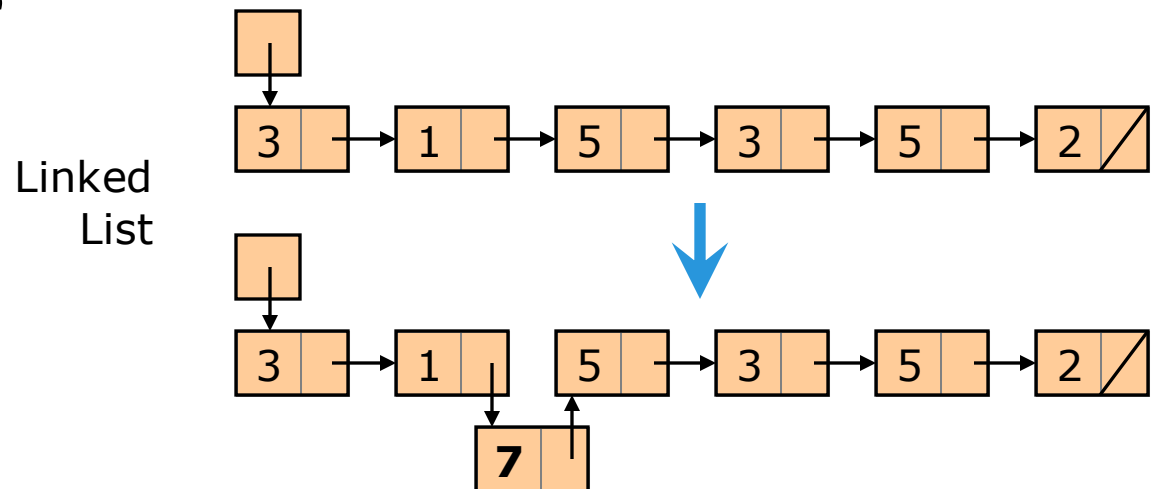
With an array, we move all later items up.

For a large array this can be very slow.



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

For large datasets, the speed-up can be huge.



# Arrays & Linked Lists

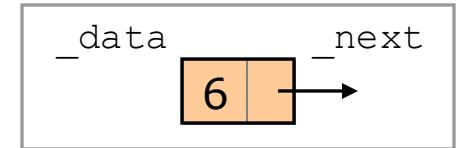
## Linked Lists — Implementation [1/2]

Here is one possible implementation of a Linked List node.

```
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; }
};
```

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

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



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<...> *)`.

# Arrays & Linked Lists

## Linked Lists — Implementation [2/2]

---

### TO DO

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

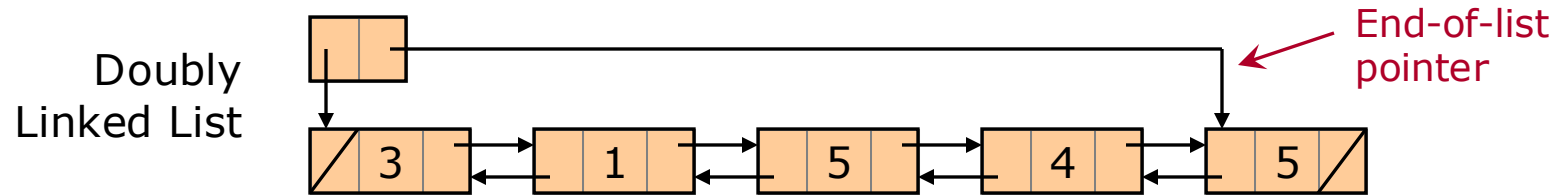
*Done. See `use_list.cpp`.  
The `llnode.hpp` header  
defines `LLNode`.*



# Arrays & Linked Lists

## Linked Lists — Doubly Linked Lists

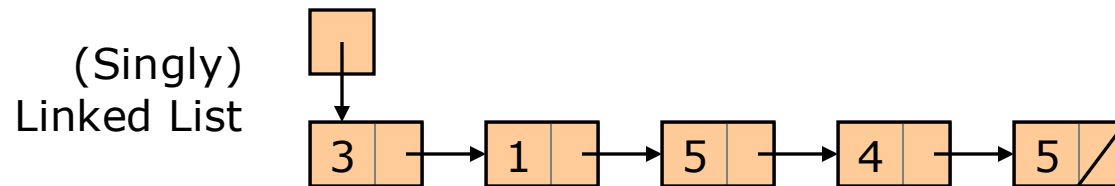
In a **Doubly Linked List**, each node has two pointers: next-node (null at the end) and previous-node (null at the beginning).



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

A Doubly Linked List is a bidirectional sequential-access structure.

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



We cover Linked Lists in more detail later in the semester.

---

# 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**.

```
int mult(int a, int b) Base case
{
```

```
    if (a <= 1)
        return a == 1 ? b : 0;
```

**Direct** recursion

**Recursive case**

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

**Indirect** recursion

```
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.

# Introduction to Recursion

## 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  $n$ th Fibonacci number by  $F_n$  ( $n = 0, 1, 2, \dots$ ).

So  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_2 = 1$ ,  $F_3 = 2$ ,  $F_4 = 3$ , etc.

**Denote** = use notation for something.

So " $F_n$ " is our notation for the  $n$ th Fibonacci number.

Now we can formally define the Fibonacci numbers as follows:

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

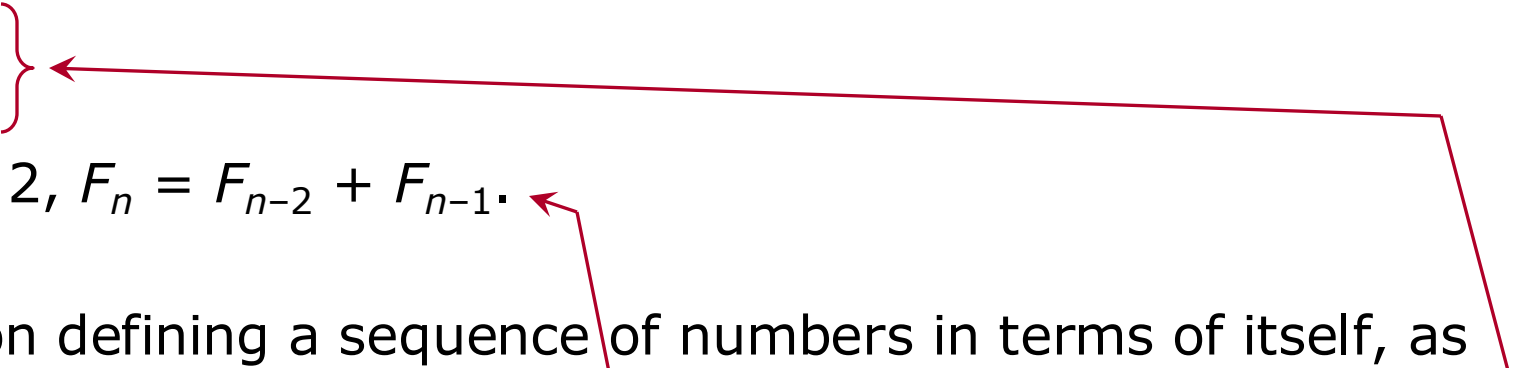
Why are we talking about this?

Computing Fibonacci numbers is our first example of a problem that can be solved by algorithms that differ greatly in performance. And some of these algorithms are recursive.

# Introduction to Recursion

## Fibonacci Numbers — Definitions [2/2]

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

- $F_0 = 0.$
  - $F_1 = 1.$
  - 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: we write a recursive function `fibonacci` that takes an integer  $n$  and returns the  $n$ th Fibonacci number  $F_n$ .

# Introduction to Recursion

## 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:  $F_n = F_{n-2} + F_{n-1}$ Smaller problems of the same kind
2. How much does each recursive call reduce the size of the problem?
  - Say the size is  $n$ . The first call reduces it 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 `fibonacci` should have " $n \geq 0$ " as a precondition.

# Introduction to Recursion

## Fibonacci Numbers — Design Decisions

---

`fibonacci` takes an integer  $n$  and returns the  $n$ th Fibonacci number  $F_n$ .  
I will write  $F_n$  as  $F(n)$  in source-code comments.

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

- The parameter can be `int`.
- As  $n$  grows,  $F_n$  will grow very quickly. We may wish 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 does `fibonacci` have?

- Pre:  $n \geq 0$ . Also,  $F(n)$  is within the range of values of `bignum`. (Some checking shows that this requires  $n \leq 93$  on my system.)
- Post: `Return == F(n)`.



## Introduction to Recursion

### Fibonacci Numbers — CODE

---

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 `fibonacci`, as described.

*Done. See `fibonacci_first.cpp`.*

Function `fibonacci` turns out to be extremely slow for anything other than small parameters. But do not conclude that recursion is slow! We will revisit `fibonacci`, rewriting it in various ways—including fast recursive versions.