Slides by Brian Noble,
Jeff Ringenberg and
Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Procedural Abstraction and Recursion

Abstraction

- Abstraction is a many-to-one mapping that reduces complexity and eliminates unnecessary details by providing only those details that matter.
- For example, there are several ways to implement a multiplication algorithm (table lookup, summing, etc..). Each looks quite different internally than the other, but they do the same thing. In general, a user won't care how it's done, just that it multiplies.
- There are two types of abstraction:
  - Procedural (the topic of the next three weeks)
  - Data

Procedural Abstraction

- Decomposing a program into functions is a way of providing “computational” abstractions.

- Rather than simply being collections of commonly used code, functions provide a useful tool for implementing procedural abstraction within a program.

Procedural Abstraction

- For any function, there is a person who implements the function (the author) and a person who uses the function (the client).
- The author needs to think carefully about **what** the function is supposed to do, as well as **how** the function is going to do it.
- In contrast, the client only needs to consider the **what**, not the **how**. Since how is much more complicated, this is a Big Win for the client!
- In individual programming, you will often be the author and the client. Sometimes it is to your advantage to “forget the details” and only concentrate on higher levels of functionality.

Procedural Abstraction

- Procedural abstractions, done properly, have two important properties:
  - Local: the implementation of an abstraction can be understood without examining any other abstraction implementation.
  - Substitutable: you can replace one (correct) implementation of an abstraction with another (correct) one, and no callers of that abstraction will need to be modified.
- These properties only apply to implementations of abstractions, not the abstractions themselves.
- It is CRITICALLY IMPORTANT to get the abstractions right before you start writing a bunch of code.
- If you change the abstraction that is offered, the change is not local, nor is the new version substitutable.

Procedural Abstraction

- Unfortunately, abstraction limits the scope of change.

- If you need to change what a procedural abstraction does, it can involve many different changes in the program.

- However, if a change can be limited to replacing the implementation of an abstraction with a substitutable implementation, then you are guaranteed that no other part of the project needs to change. **This is vital for projects that involve many programmers.**

Function Definitions vs. Declarations

- In small programs, often we just define functions before they are used.

- In larger programs, it is useful to separate the declaration of a function from its definition. A declaration provides the "type signature" of a function, also called the function header.

- Function headers can be placed in their own file and accessed using the preprocessor directive #include.

Function Definitions vs. Declarations Example

io.h

```
double GetParam(string prompt, double min, double max);
//...
```

declares abstraction

io.cpp

```
#include "io.h"
double GetParam(string prompt, double min, double max)
{ /* ... */ }
```

define abstraction

p1.cpp

```
#include "io.h"
int main() {
  // ...
  GetParam(prompt, min, max);
}
```

uses abstraction

Function Details

- All C++ functions take zero or more arguments and return a result of some type.
- There is a special type, called “void”, that means “no result is returned”. void is still a type, even though it is “the type with no legal values”.
- Typically, a function's signature defines all of the state, in the form of explicit arguments, needed by the procedure to accomplish its goal. However, sometimes there are also implicit arguments: elements of the global environment that are used by the procedure.

Function Details

- The type signature of a function can be considered part of the abstraction – if you change it, customers (callers) must also change.
- However, as long as a new implementation of an abstraction does the “same thing” as some old (correct) implementation, you can replace the old one with the new one.
- Of course, now we need to know what we mean by the “same thing”. This boils down to describing the abstraction (not implementation) of the function. We use **specifications** to do this.

Specifications

- For a procedural abstraction, a specification must answer three questions:
  - What pre-conditions must hold to use the function?
  - Does the function change any inputs (even implicit ones)? If so, how?
  - What does the procedure actually do?

- We answer each of these three questions in a specification comment, and we always include one – with the declaration, if it is separate from definition.

Specification Comments

- There are three clauses to the specification:
  - REQUIRES: the pre-conditions that must hold, if any.
  - MODIFIES: how inputs are modified, if any.
  - EFFECTS: what the procedure computes given legal inputs.

- Note that the first two clauses have an "if any", which means they may be empty, in which case you may omit them.

Specification Comment Example

```
bool isEven(int n);
// EFFECTS: returns true if n is even,
// false otherwise
```

- This function returns true if and only if its argument is an even number. We call functions that return true or false depending on some input property predicates.
- Since the predicate `isEven` is well-defined over all inputs (every possible integer is either even or odd) there need be no REQUIRES clause.
- Since `isEven` modifies no (implicit or explicit) arguments, there need be no MODIFIES clause.

Specification Comment Example

```
int factorial(int n);
// REQUIRES: n >= 0
// EFFECTS: returns n!
```

- The mathematical abstraction factorial is only defined for non-negative integers. So, we REQUIRE that the caller supply an argument greater than or equal to zero.
- Factorial promises to compute n! correctly for non-negative integers only. In other words, the EFFECTS clause is only valid for inputs satisfying the REQUIRES clause.
- Importantly, this means that the implementation of factorial DOES NOT HAVE TO CHECK if $n < 0$! The function specification tells the caller that s/he must pass a non-negative integer.

Example from Project 1: io.h

```
double GetParam(string prompt, double min, double max);
// EFFECTS: Prints the prompt, and reads a double from cin.
// If the value is between min and max, inclusive, returns
// it.  Otherwise, repeats.

void PrintHeader (void);
//  EFFECTS: prints out a nice header for the payment info
//  table.
//  MODIFIES: cout

void PrintMonthlyData (int month, double principal,
                       double interest, double loaned);
//  EFFECTS: prints out a row in the payment info table.
//  MODIFIES: standard output stream
```

More Function Details

- Functions without REQUIRES clauses are considered **complete**; they are valid for all input.
- Functions with REQUIRES clauses are considered **partial**; some arguments that are "legal" with respect to the type system are not legal with respect to the function.
- Whenever possible, it is much better to write complete functions than partial ones.
- When we discuss exceptions, we will see a way to convert partial functions to complete ones.

More Function Details

- What about the MODIFIES clause?

- A MODIFIES clause identifies any function argument or piece of global state that **might** change if this function is called.
  - For example, this can happen with pass-by-reference as opposed to pass-by-value inputs

Text

Specification Comment Example

```
void swap(int &x, int &y);
// MODIFIES: x, y
// EFFECTS: exchanges the values of x and y
```

- The ampersand (&) means that you get a **reference** to the caller's argument, not a **copy** of it. This lets you to change the value of that argument.
- NOTE: If the function could change a reference argument, it must go in the MODIFIES clause. Leave it out only if the function can never change it.
  - This implies you should never use pass-by reference, unless you intend to change the input in some situation.

Call by reference e

```
void swap(int &x, int &y);
// MODIFIES: x, y
// EFFECTS: exchanges the
// values of x and y
```

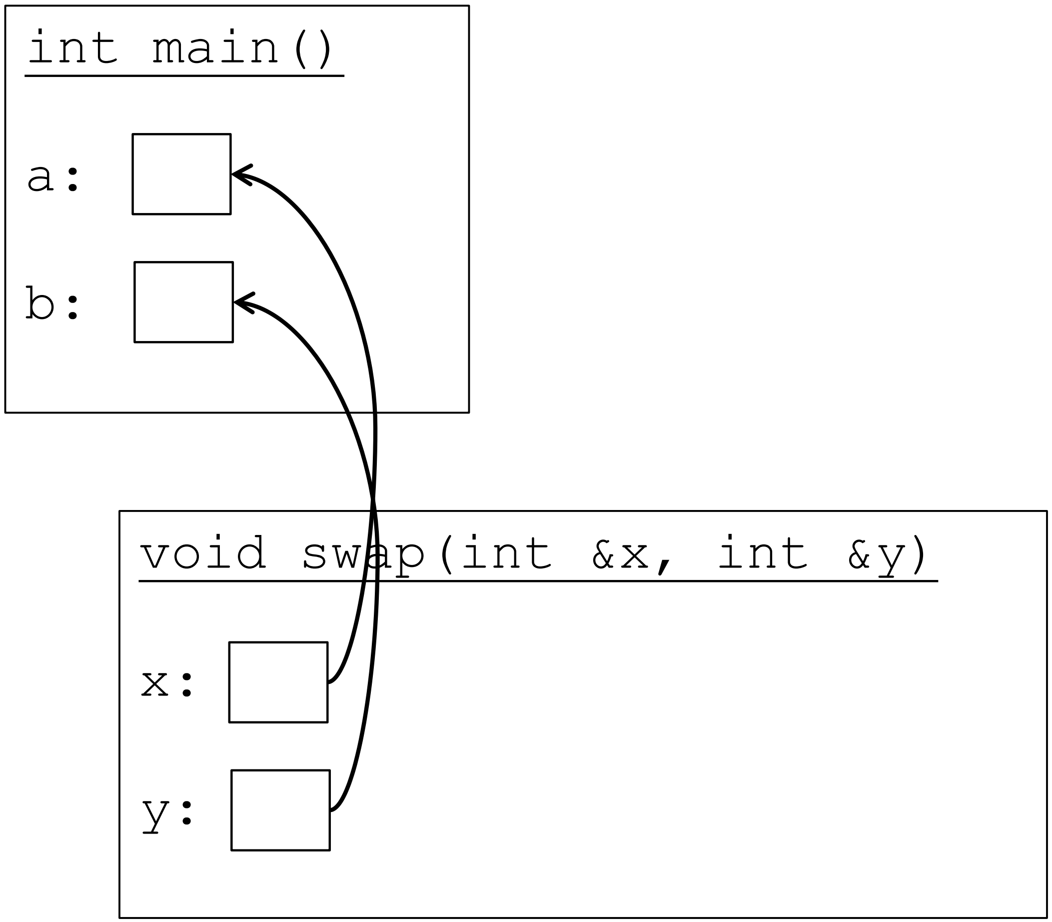
Text

```
int main() {
  int a=4, b=9;
  cout << a << " " << b << endl;
  swap(a,b);
  cout << a << " " << b << endl;
}
```

Text

```
4 9
9 4
```

Call by reference example



Call Stacks: How function calls work

- When we call a function (using pass-by-value semantics) the program follows these steps:

1. Evaluate the actual arguments to the function (order is not guaranteed).
2. Create an “activation record” (sometimes called a “stack frame”) to hold the function's formal arguments and local variables.
3. Copy the actuals' values to the formals' storage space.
4. Evaluate the function in its local scope.
5. Replace the function call with the result.
6. Destroy the activation record.

Call Stacks

- Activation records are typically stored as a **stack**.

- You can think of this process like plates in a cafeteria
  - Each time you clean a plate, you add it to the top of the stack.
  - Each time a new plate is needed, it is taken from the top.

  - Calling a function is like cleaning a plate.
  - Returning from the function is like taking a plate.

Call Stacks

- Activation records work just like the plates example.
- When a function is called, an activation record for **this** invocation is added to the “top” of the stack.
- When that function returns, it's record is removed from the “top” of the stack.
- In the meantime, this function may have called other functions, (which create corresponding activation records). These functions must return (and destroy their corresponding activation records) before this function can return.
- Note: “top” is placed in quotes, because, by convention, stacks are typically drawn growing down rather than up.

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Main starts out with an activation record with room only for the local "result":

main:

result: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Then, main calls plus_two, passing the literal value "0":

main:

result: 0

plus_two:

x: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Which in turn calls plus_one:

main:

result: 0

plus_two:

x: 0

plus_one:

x: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one adds one to x, returning the value 1:

main:

result: 0

plus_two:

x: 0

plus_one:

x: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one's activation record is destroyed:

main:

result: 0

plus_two:

x: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two adds one to the result, and returns the value 2:

main:

result: 2

plus_two:

x: 0

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two's activation record is destroyed:

main:

result: 2

Call Stack Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

main then prints the result:

2

main:

result: 2

Some things to note

- Even though plus_one and plus_two both have formals called “x”, this presents no problem.
  - Since environments are lexically scoped, plus_one cannot see plus_two's x. Instead, a copy of plus_two's x is passed to plus_one, and stored in plus_one's x.

- Neither plus_one nor plus_two can see main's “result”
  - Again, environments are lexically scoped. result is only accessible from within main.

Recursion

A convenient place for using stacks

- “Recursive” just means “refers to itself”.
- So, a function is **recursive** if it calls itself.
- Likewise, a problem is recursive if:
    1. There is (at least) one “trivial” base or “stopping” case.
    2. All other cases can be solved by first solving one (or more) smaller cases, and then combining those solutions with a simple step.

Recursion

A convenient place for using stacks

- Recursive problems are those that are defined in terms of the problem itself and can be solved very elegantly, simply, and (sometimes) efficiently by recursive functions.

- This is the focus of the next few lectures, and the core of the material you will need for project 2.

Recursion Example

- Consider the factorial function:
  - C++ does not have a "factorial" operator (neither do most other programming languages).
  - So, we have to figure out how to solve it.
  - REQUIREMENT: factorial is defined only for the domain of non-negative integers (this will be assumed)

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}. \qquad 3! = \prod_{k=1}^{3} k = 1 * 2 * 3 = 6$$

Recursion Example

- Now consider the recursive definition:

$$n! = \begin{cases} 1 & (n == 0) \\ n * (n\text{-}1)! & (n > 0) \end{cases}$$

- This is a **recursive** definition of factorial; the function factorial is defined in terms of factorial itself.

Recursion Example

$$n! = \begin{cases} 1 & (n == 0) \\ n * (n-1)! & (n > 0) \end{cases}$$

- There are two important features of this definition.
  - First, there is one trivial stopping case that requires no computation:
    $0! = 1$
  - Second, every other case can be solved by first solving a “smaller” problem, where “smaller” means “a problem that is closer to the trivial stopping case,” and then performing a simple additional computation on that smaller result to get the larger one. This is called the “recursive step” (or, sometimes, the inductive step).
- Because of these features, converting to code is easy!

Recursion Example

$$n! = \begin{cases} 1 & (n == 0) \\ n * (n\text{-}1)! & (n > 0) \end{cases}$$

```
int factorial (int n)
  // REQUIRES: n >= 0
  // EFFECTS:  computes n!
{
  if (n == 0) {
     return 1;  // 'base case'
  } else {
     return n*factorial(n-1); // 'recursive step'
  }
}
```

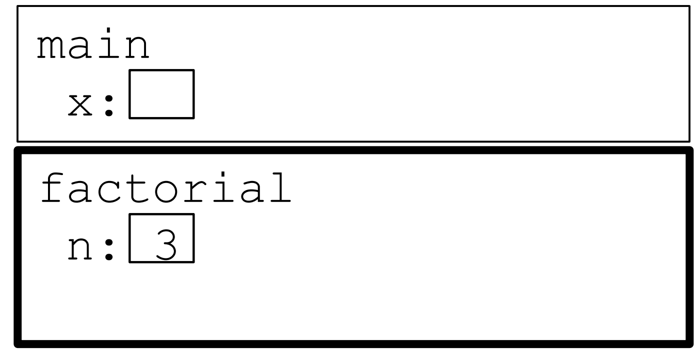
Recursion Example

```
main
  x: [ ]
```

- Suppose we call our function as follows:

```
int main()
{
  int x;
  x = factorial(3);
}
```

Recursion Example



- main() calls factorial with an argument 3.
- We evaluate the actual argument, create an activation record, and copy the actual value to the formal:

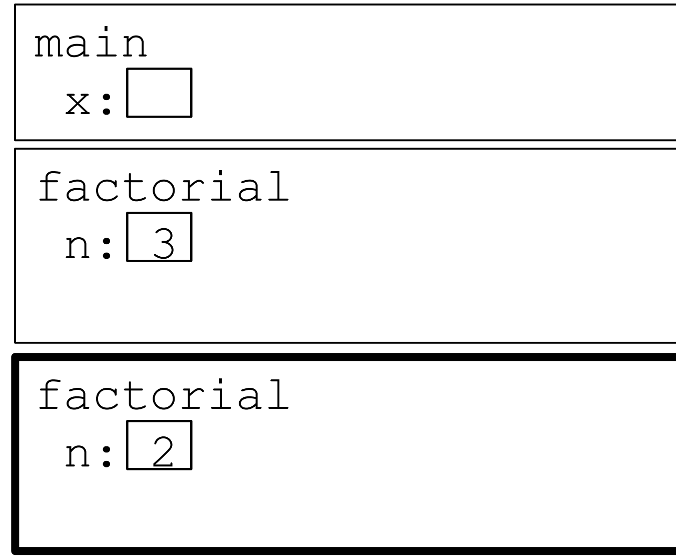
Recursion Example

- Now we evaluate the body of factorial:

  - n is not zero, so we evaluate the alternate arm of the if statement. Substituting for n and simplifying, we get:

    return 3 * factorial(2)

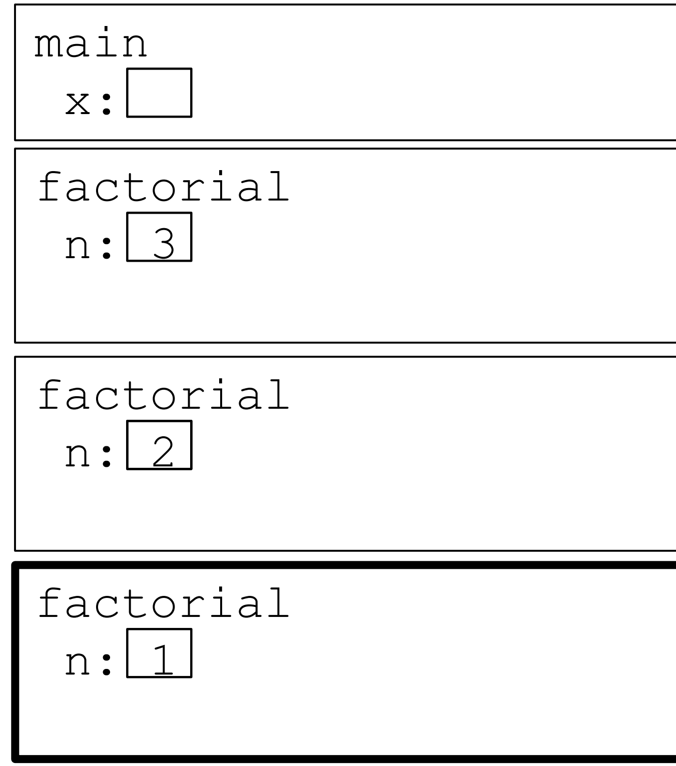
  - So, factorial must call factorial. We follow the “call a function” protocol, and create a **new** activation record for a **new** instance of factorial:



Recursion Example

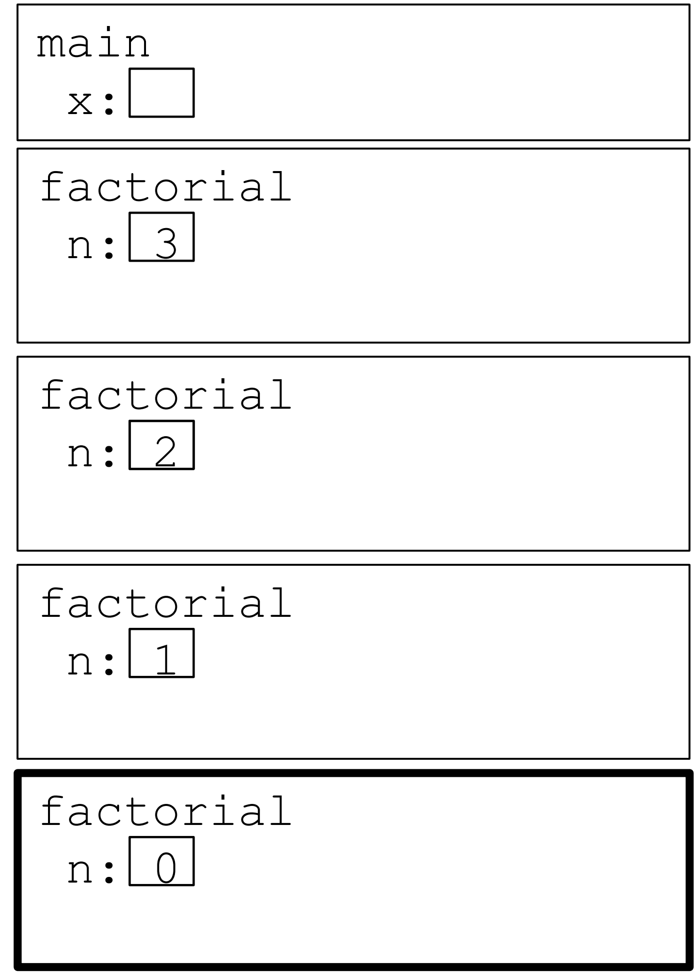
- Again, n is not zero, so we evaluate the arm again:

return 2 * factorial(1)



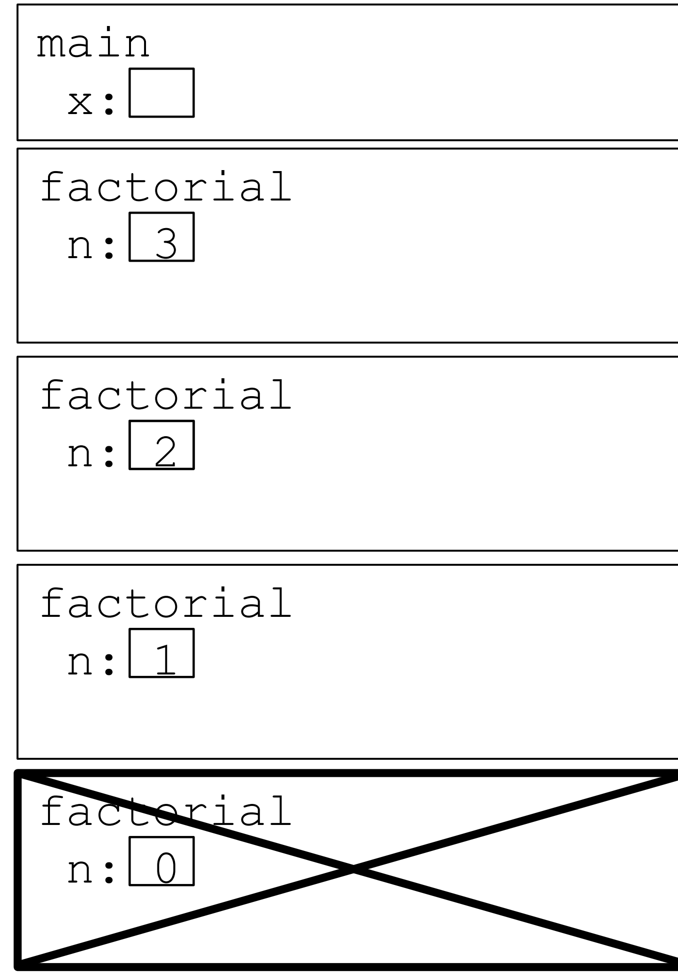
Recursion Example

- And again…



Recursion Example

- This time, n is zero, so we evaluate the consequence rather than the alternative.
- Return the value “1”, popping the most recent activation record off of the stack.



Recursion Example

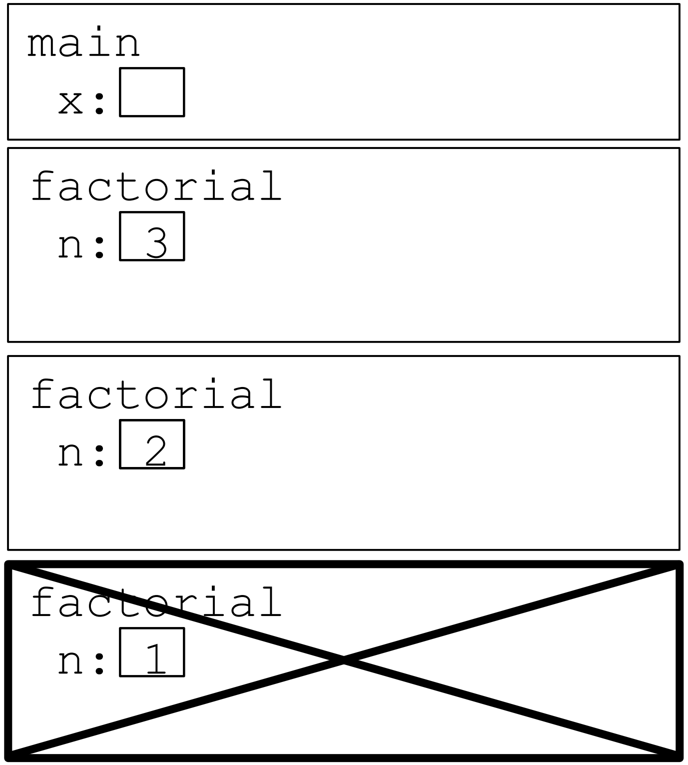
- We called factorial with this statement as follows:

  return 1 * factorial(0)

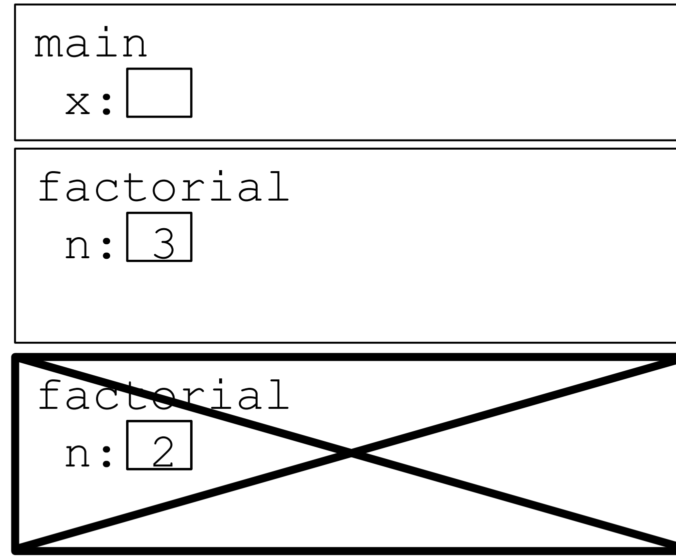
- But, now we know the value of factorial(0), so we can simplify to

  return 1 * 1   => return 1;

- This pops another frame off the stack



Recursion Example



- Allowing us to complete **this** arm:

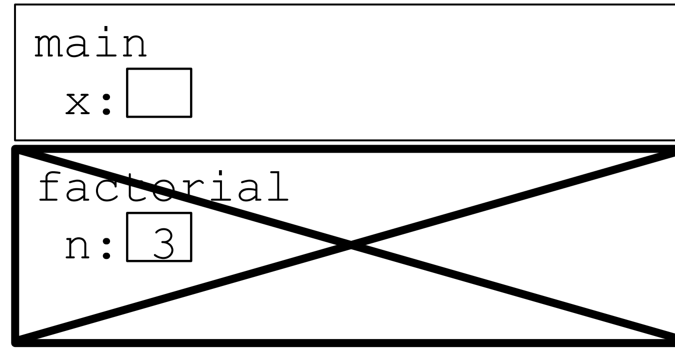
  return 2 * factorial(1) =>

  return 2 * 1 =>

  return 2;

- Now pop off another frame

Recursion Example



- Resolve the last "pending" multiplication:

  return 3 * factorial(2) =>

  return 3 * 2 =>

  return 6

- That's convenient, since it is the correct answer.

- And don't forget that last pop!

Recursion

Writing a function for the general case

- Don’t try to do it all in your head. Instead, treat it like an inductive proof.

- To <u>write</u> a correct recursive function, do two things:
  1. Identify the “trivial” case (or cases), and write them explicitly.
  2. For all other cases, first assume there is a function that can solve smaller versions of the same problem, then figure out how to get from the smaller solution to the bigger one.

Recursion

Another example

- What if you needed to count the "1" bits in the binary representation of a non-negative number?

- There are no "1" bits in the number zero, so that's our base case.
- To figure out the rest of the non-negative integers, let's think about the representation.
- There are 32 bits, from "least" to "most" significant (LSB to MSB). The value of the number is:

$$1*LSB + 2*(2ndLSB) + 4*(3rdLSB) + ... + 2^{31}*(MSB)$$

Recursion
Another example

- Given the representation:

  $1*LSB + 2*(2ndLSB) + 4*(3rdLSB) + ... + 2^{31}*(MSB)$

1. If N is odd, its least significant bit is 1, otherwise it is zero.
2. An even number divided by two has the same number of 1s

   Dividing N by two is the equivalent of shifting its bits one to the right. The least significant gets thrown away, and the most significant is filled with a zero.

- Given these two facts, here is a recursive definition of numOnes:

$$numOnes(N) = \begin{cases} 0 & \text{if } N == 0 \quad \text{(base case)} \\ numOnes(N/2) & \text{if } N>0,\ N \text{ is even (inductive step)} \\ 1 + numOnes((N-1)/2) & \text{if } N>0,\ N \text{ is odd (inductive step)} \end{cases}$$

Make N even by doing this

Recursion

Another example

- Now, write some code for it:

$$
numOnes(N) = \begin{cases} 0 & \text{if } N == 0 \quad \text{(base case)} \\ numOnes(N/2) & \text{if } N>0,\ N \text{ is even (inductive step)} \\ 1 + numOnes((N-1)/2) & \text{if } N>0,\ N \text{ is odd (inductive step)} \end{cases}
$$

- <u>The obvious way:</u>

```
int numOnes(int N)
  // REQUIRES N >= 0
  // EFFECTS returns number of "1"s in N's representation
{
  if (N == 0) return 0;
  else if (N % 2) return 1 + numOnes((N-1)/2);
  else return numOnes(N/2);
}
```

Recursion

Resource Costs

Compare the costs of these two versions

```
int factorial(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * factorial(x-1);
  }
}

// ** Recursive **
```

```
int fact_iter(int x) {
  int result = 1;
  while (x) {
    result *= x;
    x -= 1 ;
  }
  return result;
}
// ** Using loops **
```

Questions for the class, small groups ~5 minutes:

- How many multiplications does each version perform?
- How much space does each one require?