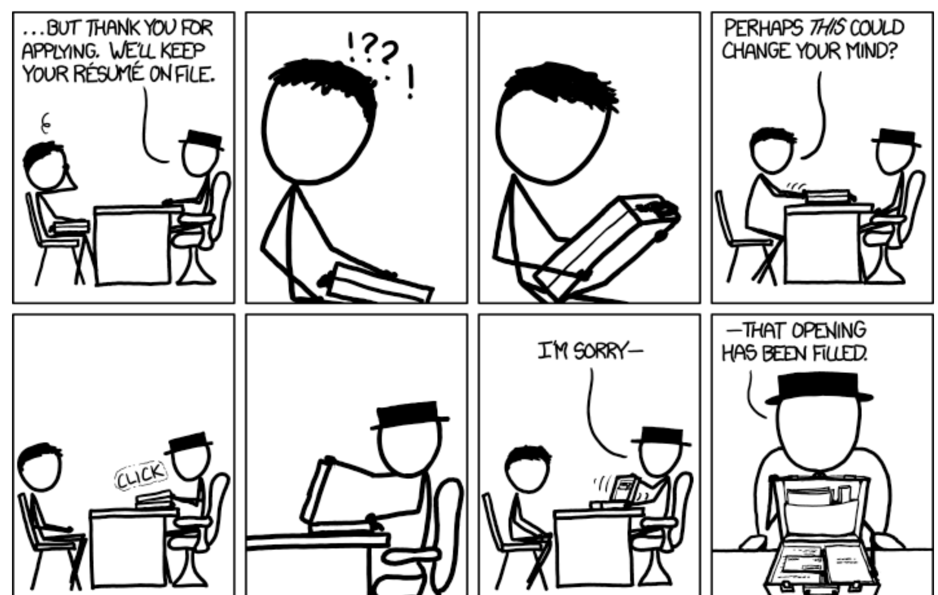
VE280 Programming and Elementary Data Structures

Paul Weng
UM-SJTU Joint Institute

Recursion; Function Pointers; Function Call Mechanism



Learning Objectives

- Understand recursion and know how to write recursive functions
- Understand how to write more general code with function pointers
- Understand function call mechanism
- Understand type inference in C++

Outline

- Recursion
- Function Pointers
- Type Inference
- Function Call Mechanism

Recursion

- Recursion is a nice way to solve problems
  - "Recursive" just means "refers to itself".
  - There is (at least) one "trivial" base or "stopping" case.
  - All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.
- Example: calculate factorial $n!$

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

```
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
}
```

Recursive Helper Function

- Sometimes it is easier to find a recursive solution to a problem if you change the original problem slightly, and then solve that problem using a **recursive helper function**.

```
soln()
{
   …
   soln_helper();
   …
}
```

```
soln_helper()
{
   …
   soln_helper();
   …
}
```

Recursive Helper Function

Example

- A palindrome is a string that is equal to itself when you reverse all characters.
  - For example: rotor, racecar

- Write a function to test if a string is a palindrome.

```
bool is_palindrome(string s);
// EFFECTS: return true if s is
// a palindrome.
```

Palindrome Example

- If a string is empty, it is a palindrome.
- If a string is of length one, it is a palindrome.
- Given a string of length more than one, it is a palindrome, if
  - its first character equals its last one, **and**
  - the substring without the first and the last characters is a palindrome.
- In order to test whether a substring is a palindrome, we define a **helper** function

```
bool is_palindrome_helper(string s,
  int begin, int end);
// EFFECTS: return true if the subtring
// of s starting at begin and ending at
// end is a palindrome.
```

Palindrome Example

```
bool is_palindrome_helper(string s,
  int begin, int end)
// EFFECTS: returns true if the subtring
// of s starting at begin and ending at
// end is a palindrome.
{
  if(begin >= end) return true;
  if(s[begin] == s[end])
    return is_palindrome_helper(s,
      begin+1, end-1);
  else return false;
}
```

Palindrome Example

- With the helper function, `is_palindrome()` can be realized as

```
bool is_palindrome(string s)
// EFFECTS: returns true if s is
// a palindrome.
{
  return is_palindrome_helper(s, 0,
    s.length()-1);
}
```

Which statements are true?

Select all the correct answers.

- **A.** Any loop can be turned into a recursive function.
- **B.** Any recursive function can be turned into a loop.
- **C.** A loop is generally faster than a recursive function.
- **D.** A recursive function is generally faster than a loop.

Outline

- Recursion
- Function Pointers
- Type Inference
- Function Call Mechanism

Function Pointers

Motivation

- If you were asked to write a function to add all the elements in a list, and another to multiply all the elements in a list, your functions would be almost exactly **the same**.

- Writing almost the exact same function twice is a bad idea! Why?
  1. It's wasteful of your time!!
  2. If you find a better way to implement some common parts, you have to change **many different** places; this is prone to error.

Our Example: list_t type

- A list can hold a sequence of zero or more integers.
- There is a recursive definition for the values that a list can take:
  - A valid list is:

either `an empty list`

or `an integer followed by another valid list`

Function Pointers

Background on lists

- Here are some examples of valid lists:

```
( 1 2 3 4 )   // a list of four elements
( 2 5 2 )     // a list of three elements
( )           // an empty list
```

- There are also several operations that can be applied to lists. We will use the following three:
  - list_first( ) takes a list, and returns the first element (an integer) from the list. **REQUIRES: non-empty list!**
  - list_rest( ) takes a list and returns the list comprising all but the first element. **REQUIRES: non-empty list!**
  - list_isEmpty( ) takes a list and returns the Boolean "true" if the argument is an empty list, and "false" otherwise.

Function Pointers

Using lists

- Suppose we want to write a **recursive** function to find the smallest element in a list.
  - The function requires the input list to be non-empty.

Question: how do you do it **recursively**?

- **Answer**:

```
smallest(list) =  the element (if list has
                  only a single element)
               or the minimum of the first
                  element and the smallest
                  element from the rest of
                  the list
```

Function Pointers

Using recursion to find the smallest element in a list

```
int smallest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS:  returns smallest element
  // in the list
{
   int first = list_first(list);
   list_t rest = list_rest(list);
   if(list_isEmpty(rest)) return first;
   int cand = smallest(rest);
   if(first <= cand) return first;
   return cand;
}
```

Function Pointers

Using lists

- Now suppose we want to write a recursive function to find the largest element in a list.
  - The function also requires the input list to be non-empty.

- Recursive definition:

```
largest(list) =  the element (if list has
                 only a single element)
              or the maximum of the first
                 element and the largest
                 element from the rest of
                 the list
```

Function Pointers

Using recursion to find the largest element in a list

```
int largest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS:  returns largest element
  // in the list
{
   int first = list_first(list);
   list_t rest = list_rest(list);
   if(list_isEmpty(rest)) return first;
   int cand = largest(rest);
   if(first >= cand) return first;
   return cand;
}
```

Function Pointers

More Motivation

- `largest` is almost identical to the definition of `smallest`.
- Unsurprisingly, the solution is almost identical, too.
- In fact, the **only** differences between smallest and largest are:
  1. The names of the function
  2. The comment in the EFFECTS list
  3. The polarity of the comparison: <= vs. >=
- It is silly to write almost the same function twice!

Function pointers to rescue!

Function Pointers

A first look

- So far, we've only defined functions as entities that can be called. However, functions can also be referred to by **variables**, and passed as **arguments** to functions.

- Suppose there are two functions we want to pick between: min() and max(). They are defined as follows:

```
int min(int a, int b);
  // EFFECTS: returns the smaller of a and b.
int max(int a, int b);
  // EFFECTS: returns the larger of a and b.
```

Function Pointers

A first look

```
int min(int a, int b);
  // EFFECTS: returns the smaller of a and b.
int max(int a, int b);
  // EFFECTS: returns the larger of a and b.
```

- These two functions have precisely the same type signature:
  - They both take two integers, and return an integer.
- Of course, they do completely different things:
  - One returns a min and one returns a max.
  - **However, from a syntactic point of view, you call either of them the same way.**

Function Pointers

The basic format

- How do you define a **variable** that points to a function taking two integers, and returns an integer?

- Here's how:

```
int  (*foo)(int, int);
```

- You read this from "inside out". In other words:

`foo`	"foo"
`(*foo)`	"is a pointer"
`(*foo)(        );`	"to a function"
`(*foo)(int, int);`	"that takes two integers"
`int  (*foo)(int, int);`	"and returns an integer"

Function Pointers

The basic format

```
int  (*foo)(int, int);
```

- Once we've declared foo, we can **assign** any function to it:

```
foo = min;
```

- Furthermore, after assigning min to foo, we can just call it as follows:

```
foo(3, 5)
```

- …and we'll get back 3!

Function Pointers v.s. Variable Pointers

- For function pointers, the compiler allows us to **ignore** the "address-of" and "dereference" operators.

```
int  (*foo)(int, int);
foo = min; // min() is predefined
foo(5,3);
```

Equivalently:

```
foo = &min;
(*foo)(5, 3);
```

- In contrast, for variable pointers:

```
int  foo;
int *bar;
bar = &foo;
*bar = 2;
```

Function Pointers

Re-write `smallest` in terms of function pointers

```
int compare_help(list_t list, int (*fn)(int, int))
{
    int first = list_first(list);
    list_t rest = list_rest(list);
    if(list_isEmpty(rest)) return first;
    int cand = compare_help(rest, fn);
    return fn(first, cand);
}
int smallest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS: returns smallest element in list
{
  return compare_help(list, min);
}
```

```
int min(int a, int b);
    // EFFECTS: returns the
    // smaller of a and b.
```

Function Pointers

Re-write `largest` in terms of function pointers

```
int compare_help(list_t list, int (*fn)(int, int))
{
    int first = list_first(list);
    list_t rest = list_rest(list);
    if(list_isEmpty(rest)) return first;
    int cand = compare_help(rest, fn);
    return fn(first, cand);
}
int largest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS: returns largest element in list
{
  return compare_help(list, max);
}
```

```
int max(int a, int b);
    // EFFECTS: returns the
    // larger of a and b.
```

Which of the following statements are true?

Select all the correct answers.

- **A.** We can create an array of function pointers.
- **B.** A function can return a function pointer.
- **C.** We can increment a function pointer.
- **D.** We can print a function pointer.

Outline

- Recursion
- Function Pointers
- Type Inference
- Function Call Mechanism

Type Specifier `auto`

Motivation

- A type can become very complex to write
  - E.g., pointers, function pointers, STL containers,…
- `auto` asks the compiler to deduce the type for us
  - `auto var = expression;`
- Examples:
  - `auto pi = 3.14;`
  - `auto total = val1 + val2;`
  - `auto result = f(x, y, z);`
  - `auto *fp = f; // note the need of *`

Type Specifier `auto`

Advantages

- Avoids bugs due to uninitialized variables
  - `auto` variables need to be initialized
- Avoids bugs due to incorrect choice of types
  - E.g., choosing `float` instead of `double`
- Can make refactoring easier
  - Less modification if the type of variable changes
- Can improve legibility
  - `auto *` may be better than `int (*)(int, int)`

Type Specifier `auto`

Disadvantages

- Deduced type of a variable may become not so obvious
  - `auto v = complex_expression;`

- Deduced type of a variable may even be surprising
  - ```
    std::vector<bool> v;
    …
    auto b = v[0];
    ```

- However, in that case, the deduced type can be made explicit
  - `auto v = static_cast<int>(0.5*v.size());`
    ```

Type Specifier `auto`

Examples

- `auto` usually ignores top-level const/ref, but not low-level const/ref
  - ```
    const int ci=i, &cr=i;
    auto a = ci; // int
    auto b = cr; // int; ignores reference
    auto &c = ci; // c reference to const int
    auto *p = &ci; // p pointer to const
    const auto cp = &ci; // const ptr to const
    ```

- `auto` can define several variables in a same statement, but auto-deduced type should be consistent
  - `auto &m = ci, *p = &ci; // OK`
  - `auto &n = i, *p2 = &ci; // Not OK`
    ```

Type Specifier `decltype`

Motivation

- `auto` ignores top-level const and reference, how to get exact type?

- `decltype` provide exact type of expression
  - ```
    decltype(ci) j = ci; // const int
    decltype(cr) k = cr; // const int&
    decltype(f(0)) l = f(1); // f(0) not called
    decltype(i) l = i; // int
    decltype((i)) l = i; // int&!!!
    ```

- Possible to automatically deduce types with `decltype`
  ```
  decltype(auto) m = ci; // const int
  ```
  ```

Outline

- Recursion
- Function Pointers
- Type Inference
- Function Call Mechanism

Call Stacks

How a function call really works

- When we call a function, the program does the following steps:

1. Evaluate the actual arguments to the function (order is not guaranteed). Example: y = add(4-1, 5);
2. Create an "**activation record**" (sometimes called a "**stack frame**") to hold the function's formal parameters and local variables.
   - When call function `int add(int a, int b)`, system creates an activation record: a, b (formal), result (local)
3. Copy the actuals' values to the formals' storage space. a=3 b=5
4. Evaluate the function in its local scope.
5. Replace the function call with the result. y=8
6. Destroy the activation record.

Call Stacks

How a function call really works

- It is typical to have multiple function calls. How the activation records are maintained?
  - Answer: stored as a **stack**.
- Stack: a set of objects which modifies as **last in first out**. Example: a stack of 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, the one at the top is taken **first**

Call Stacks

How a function call really works

- When a function f() is called, its **activation record** is added to the "top" of the stack.
- When the function f() returns, its **activation record** is removed from the "top" of the stack.
- In the meantime, f() may have called other functions.
  - These functions create corresponding activation records.
  - These functions must return (and destroy their corresponding activation records) before f() can return.

Call Stacks

Example

- When a function is called, its **activation record** is added to the "top" of the stack.
- When that function returns, its **activation record** is removed from the "top" of the stack.

double add(double a, double b): a = 1, b = 0, result = 0
double sin(double x): x = 1, result = 0
int main(): x = 1, sinResult = 0

- Note: "top" is placed in quotes, because in reality, stack of activation records grows **down** rather than **up**.

Call Stacks

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 Stacks

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 Stacks

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 Stacks

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 Stacks

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

1

plus_one:

x: 0

Call Stacks

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

1

plus_one:

~~x: 0~~

Call Stacks
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

2

plus_two:

x: 0

Call Stacks

Example

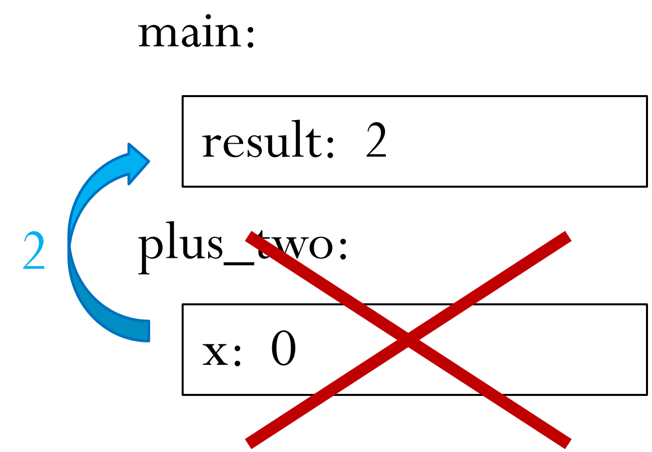
```
int plus_one(int x) {
  return (x+1);
}

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:



Call Stacks

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

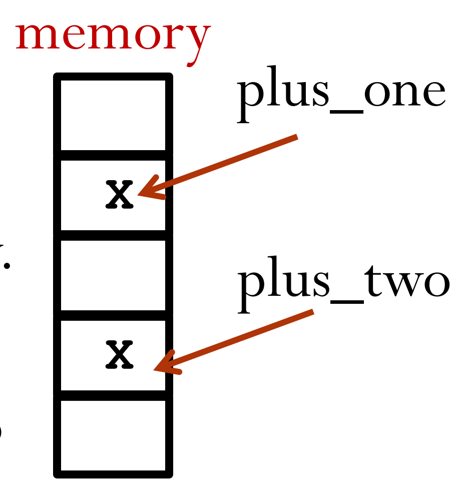
main:

result: 2

Call Stacks

Example: Some things to note

- Even though `plus_one` and `plus_two` both have formal parameters called "x", there is no problem.
  - These two x's are at different locations in memory.
  - `plus_one` cannot see `plus_two`'s x.
  - Instead, the **value** of `plus_two`'s x is passed to `plus_one`, and stored in `plus_one`'s x.



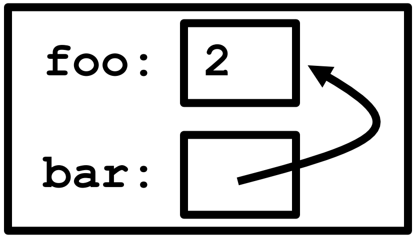
Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Activation record of main:



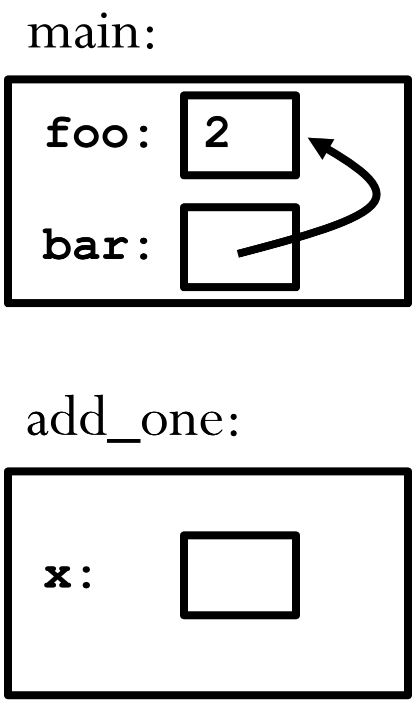
Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Main calls add_one, creating an activation record for add_one



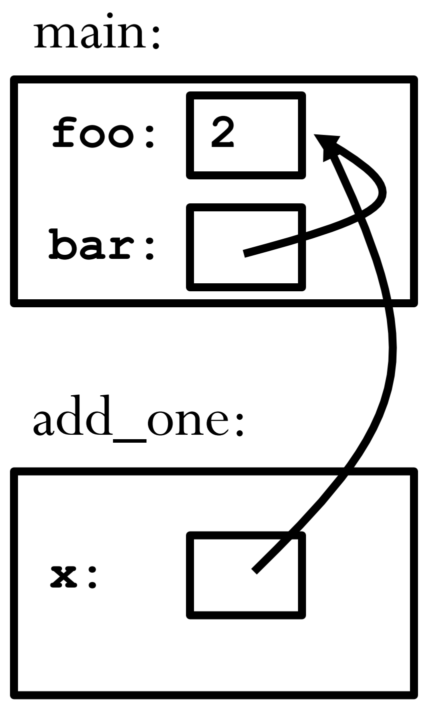
Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Copy the value of bar to add_one's formal parameter x.



Both x and bar point to foo.

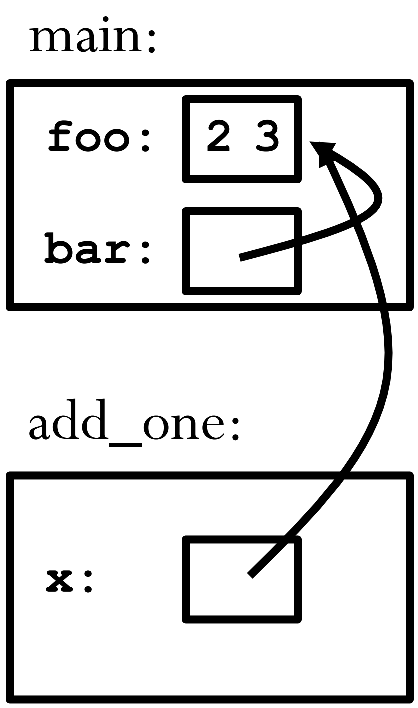
Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

add_one adds 1 to the object pointed to by x.



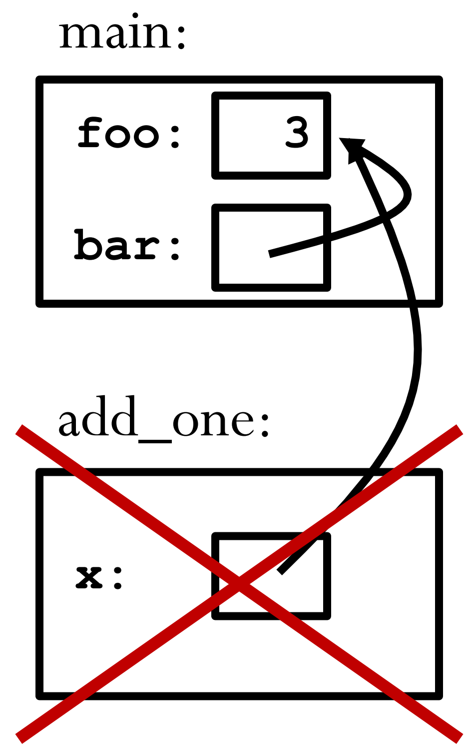
Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

add_one's activation record is destroyed.



Call Stack

Example: Recursion

```
main
  x: [ ]
```

- Suppose we call our function as follows:

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

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

Call Stack

Example: Recursion

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

```
main
  x:
```

```
factorial
  n: 3
  RA: main line #3
```

RA = "Return Address"

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

Call Stack

Example: Recursion

- Now we evaluate the body of factorial:
  - n is not zero, so we evaluate the **else** arm of the if statement:

    return 3 * factorial(2)

  - So, factorial must call factorial. We will create a **new** activation record for a **new** instance of factorial.

```
main
  x: [ ]
```

```
factorial
  n: [3]
  RA: main line #3
```

```
factorial
  n: [2]
  RA: factorial line #2
```

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

Call Stack

Example: Recursion

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

  return 2 * factorial(1)

- This creates a new activation record for factorial

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

```
main
  x: [ ]
```

```
factorial
  n: [3]
  RA: main line #3
```

```
factorial
  n: [2]
  RA: factorial line #2
```

```
factorial
  n: [1]
  RA: factorial line #2
```

Call Stack

Example: Recursion

- And again, we evaluate the **else** arm:

  return 1*factorial(0)

- This creates a new activation record for factorial

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

main
x:

factorial
n: 3
RA: main line #3

factorial
n: 2
RA: factorial line #2

factorial
n: 1
RA: factorial line #2

factorial
n: 0
RA: factorial line #2

Call Stack

Example: Recursion

- In evaluating factorial(0), n is zero, so we evaluate the **if** arm rather than **else** arm.
- Return the value "1"
- Popping the most recent activation record off the stack.

```
main
  x: [ ]

factorial
  n: [3]
  RA: main line #3

factorial
  n: [2]
  RA: factorial line #2

factorial
  n: [1]
  RA: factorial line #2
```

1

~~factorial~~
~~n: 0~~
~~RA: factorial line #2~~

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

Call Stack

Example: Recursion

- In factorial(1), we called factorial(0) as follows:

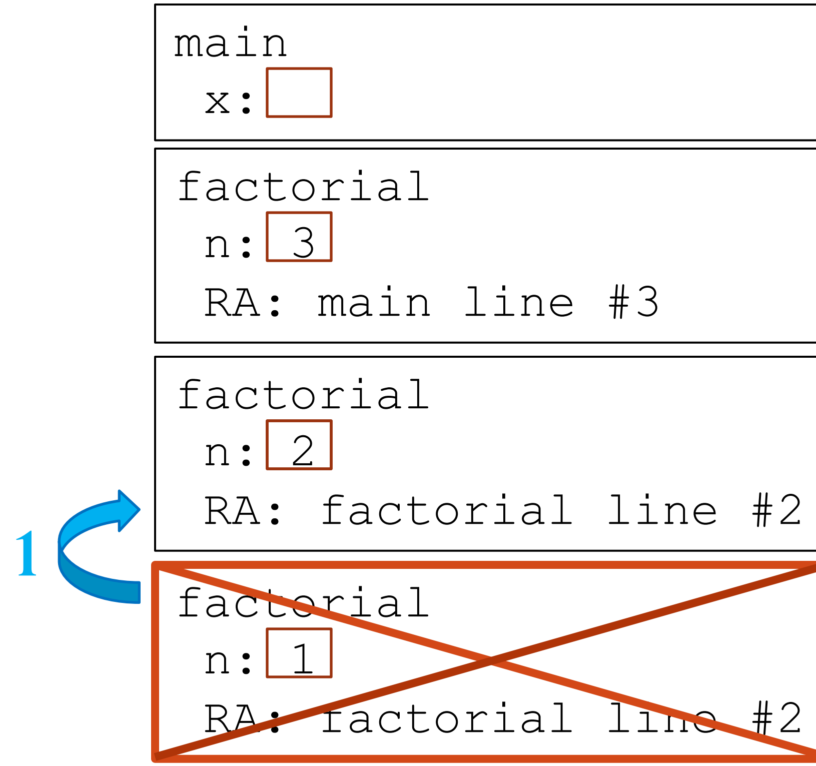
  return 1 * factorial(0)

- Now we know the value of factorial(0), so we complete factorial(1):

  return 1 * 1 => return 1;

  from factorial(1)

- This pops another activation record off the stack



Call Stack

Example: Recursion

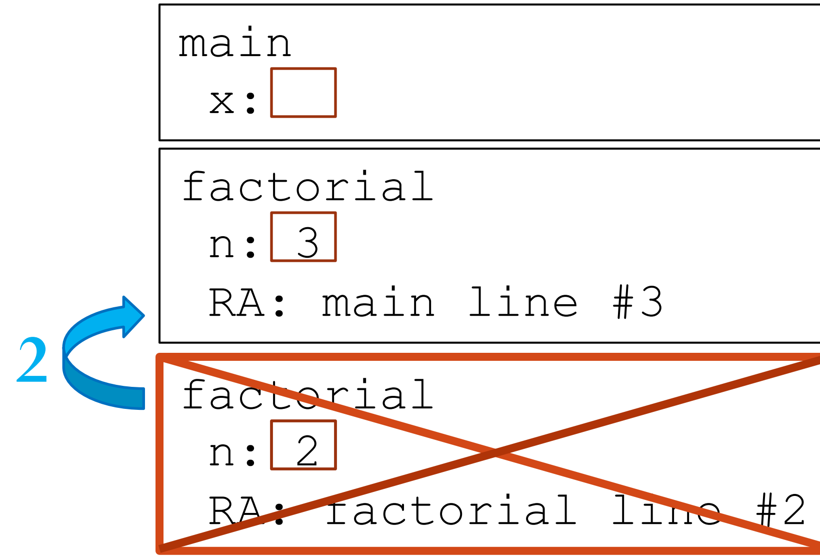
- Now it allows us to complete evaluating factorial(2):

  return 2 * factorial(1) =>

  return 2 * 1 =>

  return 2

  from factorial(2)



- Now pop off another activation record.

Call Stack

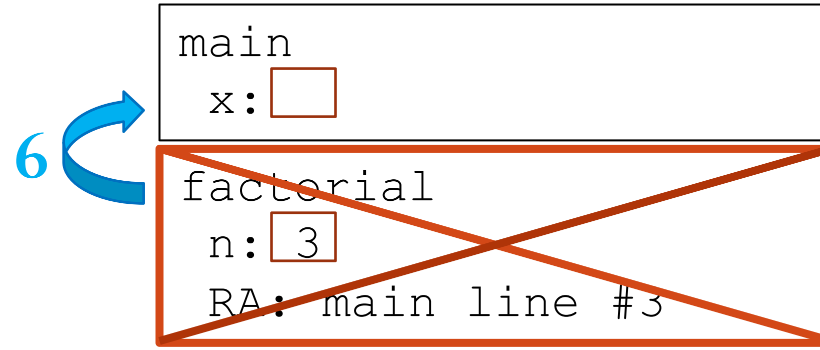
Example: Recursion

- Now we can complete evaluating factorial(3):

  return 3 * factorial(2) =>

  return 3 * 2 =>

  return 6



- That is the correct answer.

- Don’t forget that last pop!

Which of the following statements are true (for nonnegative parameters)?

Select all the correct answers.

- **A.** The number of recursive calls of factorial can be as high as we want.
- **B.** The number of calls of factorial could be reduced by 1.
- **C.** The function factorial always returns a positive number.
- **D.** The function factorial may return a negative number.

Reference

- Recursion
  - Problem Solving with C++, 8th Edition, Chapter 14

- Function pointers
  - C++ Primer (4th Edition), Chapter 7.9