

Ve 280

Programming and Introductory Data Structures

Procedural Abstraction;

Recursion;

Function Pointers

Announcement

- Programming project two announced
 - On recursive function and function pointer
 - Due time: 11:59 pm on June 17th, 2016

Outline

- Procedural Abstraction
- Recursion
- Function Pointers

Procedural Abstraction: Review

- Abstraction
 - Provides only those details that matter.
 - Eliminates unnecessary details and reduces complexity.
- Procedural abstraction: function

Procedural Abstraction

- Procedural abstractions, done properly, have two important properties:
 - **Local**: the **implementation** of an abstraction does not depend on any other abstraction **implementation**.
 - To realize an implementation, you only need to focus **locally**.
 - **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.

Implementation of square() does not depend on **how you implement** multi()

```
int square(int a)
{
    return multi(a,a);
}
```

We can **change** the implementation of multi(). It won't affect square() as long as it does multiplication

Procedural Abstraction

- Locality and substitutability only apply to **implementations** of abstractions, not the **abstractions** themselves.
 - If you change the **abstraction** that is offered, the change is not local.
- It is CRITICALLY IMPORTANT to get the **abstractions** right before you start writing code.

```
int square(int a)
{
    return multi(a,a);
}
```

We cannot change
the abstraction of
“multi” to $2*a*b$.

Procedural Abstraction: Summary

- **Abstraction** and **abstraction implementation** are **different!**
 - Abstraction: tells **what**
 - Implementation: tells **how**
 - **Same** abstraction could have **different** implementations
- If you need to change what an **abstraction** itself, it can involve many different changes in the program.
- However, if you only change the **implementation** of an abstraction, then you are guaranteed that no other part of the project needs to change.
 - **This is vital for projects that involve many programmers.**

Procedural Abstraction and Function

- **Function** is a way of providing procedure abstractions.
- The **type signature** of a function can be considered as **part of the abstraction**
 - Recall: type signature includes return type, number of arguments and the type of each argument.
 - If you change type signature, callers must also change.
- Besides type signature, we need some way to describe **the abstraction (not implementation)** of the function.
 - We use **specifications** to do this.

Procedural Abstraction

Specifications

- We describe procedural abstraction by specification. It answers three questions:
 - What pre-conditions must hold to use the function?
 - Does the function change any inputs (even implicit ones, e.g., a global variable)? 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 **function declaration** (or function definition in case we don't have declaration)

```
...
```

```
// SPECIFICATION COMMENT
```

```
int add(int a, int b);
```

Procedural Abstraction

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.

Procedural Abstraction

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.
- Since the function 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.

Procedural Abstraction

Specification Comment Example

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

- The mathematical abstraction of factorial is only defined for non-negative integers. So, there is a **REQUIRE** clause.
- 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.

Procedural Abstraction

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 (e.g., int) 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.

Procedural Abstraction

More Function Details

- What about the MODIFIES clause?
- A MODIFIES clause identifies any function argument or global state that **might** change if this function is called.
 - For example, it can happen with call-by-reference as opposed to call-by-value inputs.

Procedural Abstraction

Specification Comment Example

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

- NOTE: If the function **could** change a reference argument, the argument must go in the MODIFIES clause. Leave it out only if the function can **never** change it.

Outline

- Procedural Abstraction
- Recursion
- Function Pointers

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 $n!$

```
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  
}
```

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

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 substring  
// 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: return true if the substring  
// 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: return true if s is
// a palindrome.
{
    return is_palindrome_helper(s, 0,
                                s.length()-1);
}
```

Outline

- Procedural Abstraction
- Recursion
- Function Pointers

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.

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

- Note: This implies that `smallest` is not defined for the empty list, just like `list_first` is not.

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 recursive definition is:

`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

- Note: This implies that `largest` is not defined for the empty 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: \leq vs. \geq
- It is silly to write almost the same function twice!

Function pointers to the 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:

<code>foo</code>	“foo”
<code>(*foo)</code>	“is a pointer”
<code>(*foo) (</code>	“to a function”
<code>(*foo) (int, int);</code>	“that takes two integers”
<code>int (*foo) (int, int);</code>	“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);
```

We don't write:

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

- This's not true with 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.
```

Reference

- Procedural abstraction
 - Problem Solving with C++, 8th Edition, Chapter 4.4 and 5.3
- Recursion
 - Problem Solving with C++, 8th Edition, Chapter 14
- Function pointers
 - C++ Primer (4th Edition), Chapter 7.9