VE280 Programming and Elementary Data Structures

Paul Weng
UM-SJTU Joint Institute

Procedural Abstraction

Learning Objectives

- Understand abstraction, procedural abstraction and their importance
- Know how to describe/design procedural abstraction

Abstraction

- Abstraction
  - Provides only those details that matter.
  - Eliminates unnecessary details and reduces complexity.
- Abstraction is like a black box: we know how to use a black box, but we don't know how it operates
- A person using a black box only needs to know **what** it does, NOT **how** it does it
- Example: Multiplication algorithm
  - Many ways to do: table lookup, summing, etc.
  - Each looks quite different, but they do the **same** thing.
  - In general, a user won't care how it's done, just that it multiplies.

Abstraction

- There are two types of abstraction:
  - Procedural ← Focus of this lecture
  - Data

Procedural Abstraction

- **Function** is a way of providing “computational” abstractions.

```
int multi(int a, int b)
{
  // An implementation
  // of multiplication
  …
}
```

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

Using the “multi” abstraction

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 abstraction.

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 **<u>different</u>**!
  - Abstraction: tells **what**
  - Implementation: tells **how**
  - **Same** abstraction could have **different** implementations
- If you need to change 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.**

Which statements about procedural abstraction are true?

Select all the correct answers.

- **A.** It specifies what it does.
- **B.** It specifies how its user should interact with it.
- **C.** It should never provide any information about its implementation.
- **D.** It may sometimes provide some information about its implementation.

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 a **function declaration** (or function definition in case we don't have a 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 needs be no REQUIRES clause.
- Since isEven modifies no (implicit or explicit) arguments, there needs 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.

Which of the following comment(s) should you include?

You are implementing the function that adds an element in a stack: void push(T x, stack s).Select all the correct answers.

- **A.** `// REQUIRES: s to be non-empty`
- **B.** `// REQUIRES: s may be non-empty`
- **C.** `// MODIFIES: s`
- **D.** `// EFFECTS: push x in stack s`

Reference

- Procedural Abstraction
  - Problem Solving with C++, 8th Edition, Chapter 4.4 and 5.3