VE280 2021SU Midterm RC

06 Procedural Abstraction

Concepts & Motivation

Abstraction:

- Provides only those details that matter.
- Eliminates unnecessary details and reduces complexity.
- Only need to know **what** it does, not **how** it does it.

Different roles in programming:

- The author: who **implements** the function
- The client: who **uses** the function
- In individual programming, you are both.
- Example of client: you use `cout` to output, which is written by author of C++. You don't need to worry about how `cout` works.

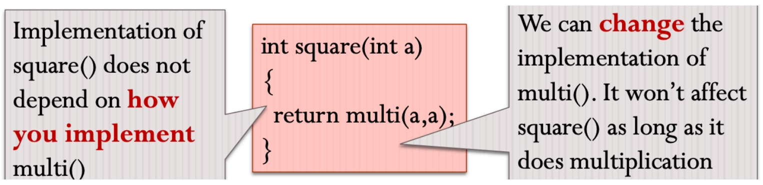
Property

Local

The implementation of an abstraction does not depend on 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.


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

Abstraction & Implementation

- Abstraction: tells **what**
- Implementation: tells **how**
- **Same** abstraction could have **different** implementations

Type Signature

- The type signature of a function can be considered as part of the abstraction.
- Type signature includes **return type, number of arguments and the type of each argument.**
- Type signature does not include the name of the function.
- Type signature is also known as **function prototype**.

For example:

```
int a;
int b;
bool retVal = (*fPtr)(a, b);
```

Then the function pointer `fPtr` has the signature:

```
(bool) (int, int);
// You may declare fPtr as: bool (*fPtr)(int, int)
```

Suppose we now have function:

```
bool greaterThan(int a, int b) { return a > b; }
bool lowerThan(int a, int b) { return a < b; }
```

We can assign both functions to `fPtr` , and the above code can run well.

Specifications

Used to the describe abstraction (not implementation).

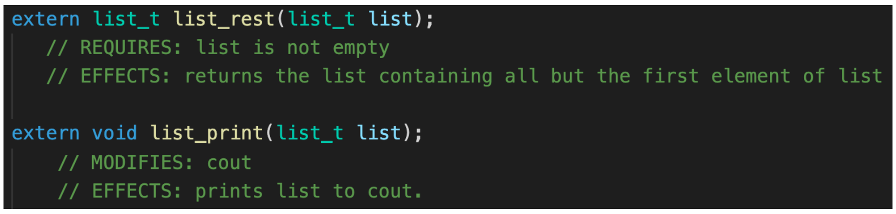
> How to describe implementation: natrual language / pseudo code / real code

There are mainly 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.

Functions without REQUIRES clauses are considered **complete**; they are valid for all input. Functions with REQUIRES clauses are considered partial.

> Note: Specifications are just comments. You cannot really prevent clients from doing stupid things, unless you use exception handling. While in VE280, you can always assume the input is valid if there is a REQUIRES comment.



```
extern list_t list_rest(list_t list);
   // REQUIRES: list is not empty
   // EFFECTS: returns the list containing all but the first element of list

extern void list_print(list_t list);
    // MODIFIES: cout
    // EFFECTS: prints list to cout.
```

> Think: why is cout modified? (not so important, just for understanding)
>
> Hint: http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/

07 Recursion; Function Pointers; Function Call Mechanism

Recursion

- Recursion is a ~~nice~~ way to solve problems: **refers to itself**
- Some problems can only be solved by recursion
- A recursion typically involves:
  - Boundary/base case
  - Recursive call (divide problem to subproblems)
- What kind of problem can be solved using recursion:
  - The problem can be divided into **sub-problems**.
  - Each sub-problem is exactly the same as the orginal one (except size).
  - There is some base case that the recursion can stop.
- Often, we define a recursion problem in a recursive way. For example, factorial can also be defined as $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ if n is not 0, otherwise it is 1. While in a **recursive** way:

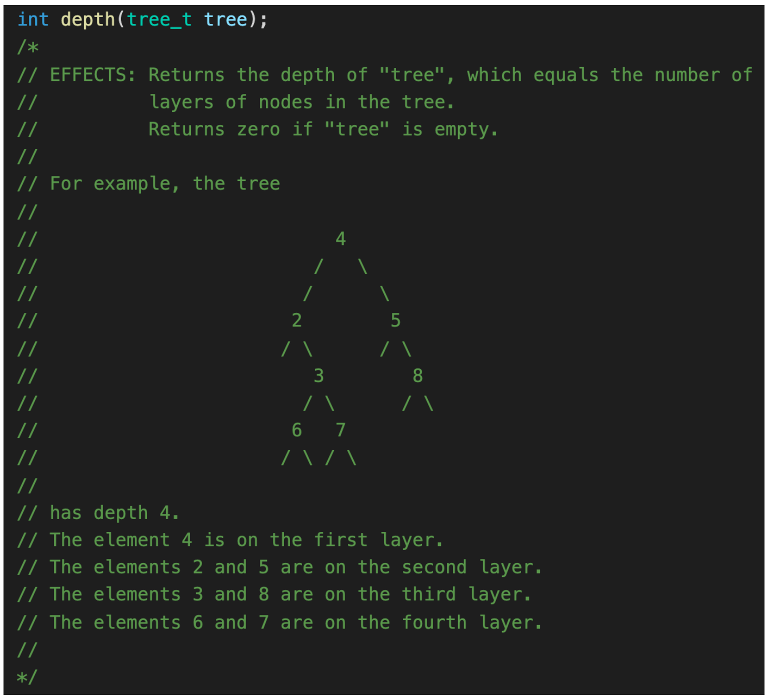
$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Helper Function

- Helper function is recursive
- You want to pass on some extra parameters to help your recursion (indicate the level or status of current recursion)

Examples

- Factorial
- Fibonacci Sequence
- Palindrome
- **Generating Permutation**
- **Recursive Data Structure** (List/Binary Tree)



```
int depth(tree_t tree);
/*
// EFFECTS: Returns the depth of "tree", which equals the number of
//          layers of nodes in the tree.
//          Returns zero if "tree" is empty.
//
// For example, the tree
//
//                           4
//                         /   \
//                        /      \
//                       2        5
//                      / \      / \
//                         3        8
//                        / \      / \
//                       6   7
//                      / \ / \
//
// has depth 4.
// The element 4 is on the first layer.
// The elements 2 and 5 are on the second layer.
// The elements 3 and 8 are on the third layer.
// The elements 6 and 7 are on the fourth layer.
//
*/
```

Function Pointers

Purpose:

- Write less code, make fewer bugs
- Higher level of abstraction

Usage:

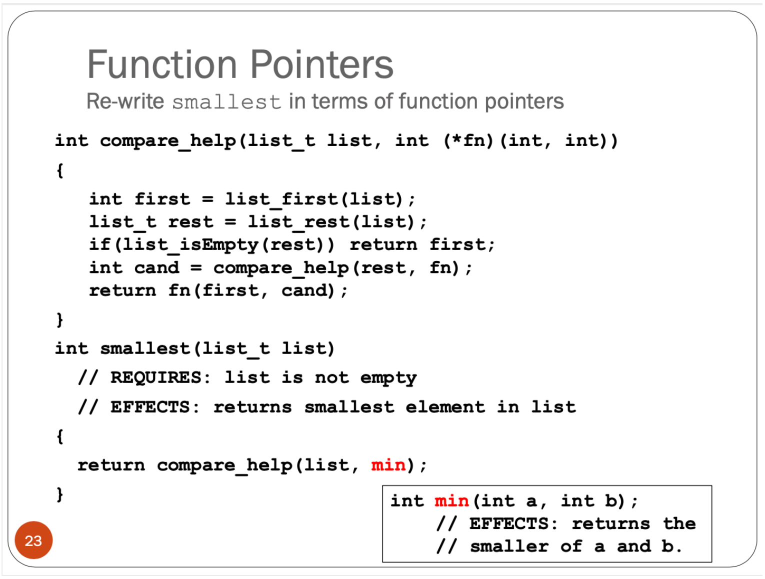
- Define a function pointer **of a specific type** (recall type signature)

`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"

- Assign a function to a function pointer, and use it like a normal function (no need to dereference like a normal variable pointer):

```
int min(int a, int b) { return a < b ? a : b; }
foo = min;     // Do not write: foo = min()
               // Also do not write foo = &min, though it can compile
foo(5,3);      // Do not write (*foo)(5,3) though it can compile
```

- Passing as an argument to other functions:



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);
}
```

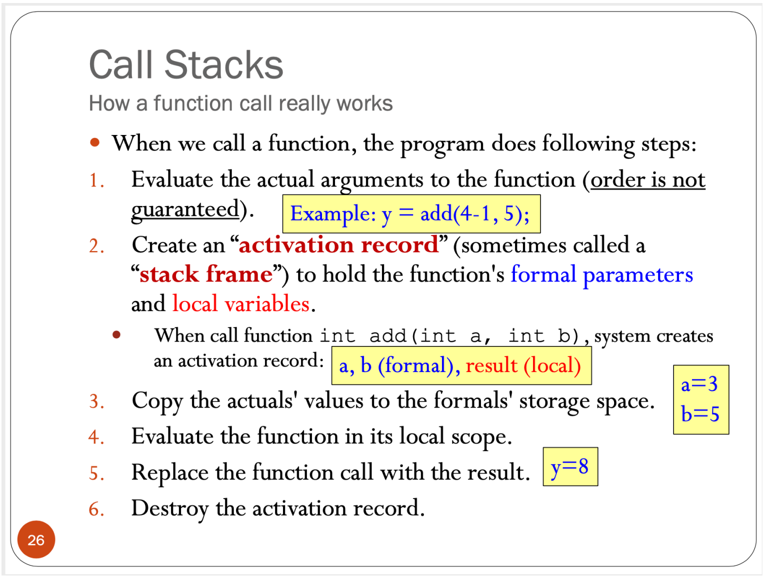
```
int min(int a, int b);
    // EFFECTS: returns the
    // smaller of a and b.
```

23

- *You can also declare an array of function pointers, which is more effective under some situation.

Functional Call Mechanism

General steps of function call (no need to memorize exactly, just understand):



Call Stacks

How a function call really works

- When we call a function, the program does 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.

26

Several important concepts you need to understand:

- Activation record (or stack frame): some space in the stack to hold parameters & variables
- Actual parameters: what you fill in the brackets to call the function
- Formal parameters: after entering the function, the function need to store its parameters in the stack
- Local variables: every function has its own scope (including `main` function). Ordinary variables declared in that scope can only be accessed within that scope.
- Call stack: where the activation records are stored (space is limited)
- **Recursive call**: refer to example in the slide
- Passing pointer/reference: what is the result after function call? Will outside variables be modified?

> A little more on stack:
>
> Stack in data structure: An abstract structure to store data, follows "last in first out".
>
> Stack in computer architecture: a specific place in the memory. Used for function calling. The memory in this place is occupied in a similar way as the stack data structure, therefore it is also called heap.
>
> These two concepts are always mixed since they are very similar. While for "heap", its meanings in the two contexts almost have nothing in common but their name.

L13: Abstract Data Type

What is ADT?

An ADT provides an **abstract description** of values and operations.

The definition of an ADT must combine **both what** that type represents, and **what** operations on the type it supports.

Why we need ADT?

Abstraction hides implementation detail and makes users' life easier.

2 Advantages of ADT:

- Information hiding:
  - The user do not need to know **(and should not need to know) how the object is represented** (is `IntSet` represented by array or linked list? The user do not need to know when using it!).
  - The user do not need to know **(and should not need to know) how the operations on the object are implemented**.
- Encapsulation: combine both data and operations in one entity.

Consider a `Set` ADT which does not contain duplicate elements.

```
Set s;
s.insert(5);
```

User: *"Insert 5 to the set! Easy!"*

Developer: *"The user wants to insert 5, I have to first check if 5 already exists! Ahh, and also the total number may change!"*

C++ "class" provides a mechanism for both **information hiding (**`private/protected` **access specifier)** and **encapsulation (member functions/methods).**

Access Specifier

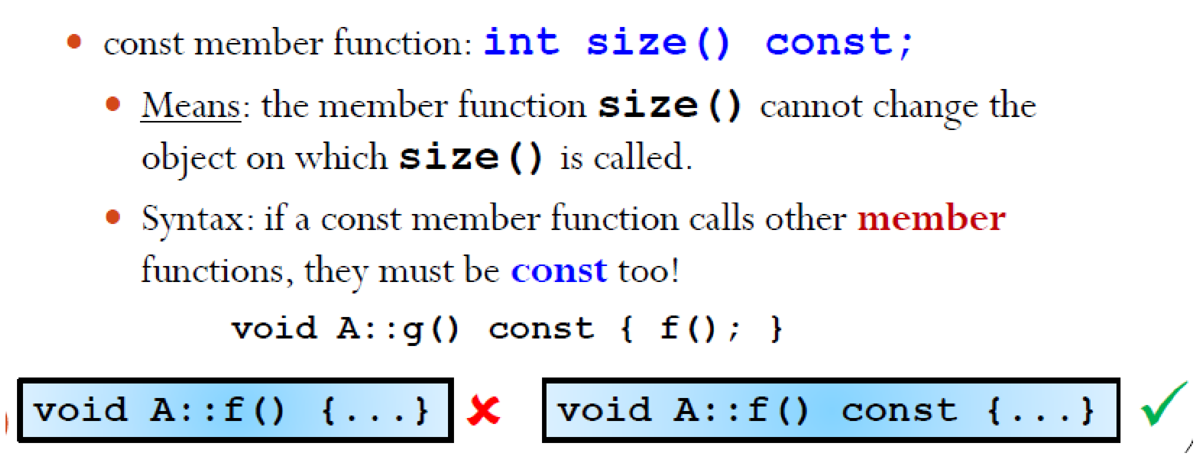
There are three different specifiers, namely `private`, `protected` and `public`.

The accessibility of **members** in class are as follows:

Scope \ Member Specifier	private	protected	public
self	Yes	Yes	Yes
derived classes	No	Yes	Yes
outsiders	No	No	Yes

Const Member Function

Put a `const` specifier both after the **member function declaration and definition**



- const member function: `int size() const;`
  - Means: the member function `size()` cannot change the object on which `size()` is called.
  - Syntax: if a const member function calls other **member** functions, they must be **const** too!

```
void A::g() const { f(); }
```

```
void A::f() {...}
```
✗

```
void A::f() const {...}
```
✓

An ADT Example covered in lecture slides: `IntSet`

```
const int MAXELTS = 100;
class IntSet {
    // OVERVIEW: a mutable set of integers
   public:
    IntSet();
    // MODIFIES: elts, numElts
    // EFFECTS: Default constructor
    void insert(int v);
    // MODIFIES: this
    // EFFECTS: insert v into the set
    void remove(int v);
    // MODIFIES: this
    // EFFECTS: delete v from the set
    bool query(int v) const;
    // EFFECTS: return true if v is in the set, return false otherwise
    int size() const;
    // EFFECTS: return the size of the set

   private:
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
    // REQUIRES: v is in the set
    // EFFECTS: return the index of v in the set
};
```