

# Mid RC part 2

VE280 Teaching Team

October 29, 2024

## Main Content:

- Procedural Abstraction and Specification Comments;
- Recursion;
- Function pointers;
- enum Type.

# Abstractions

Abstraction is a process of emphasizing the separation of "what" and "how". It helps programmers to use a function without knowing how it is implemented.

Two kinds of abstractions:

- procedural abstraction;
- data abstraction.

The main focus of this course is on procedural abstraction.

# Abstractions

Motivation: There're usually two roles in programming: the implementer and the user. The implementer is responsible for implementing the function, and the user is responsible for using the function. For the user, the implementation details are not important. They only need to know the interface of the function.

## Benifits:

- Simplifies Project: Helps manage complex project by focusing on the big picture instead of the detailed implementation;
- Enhances Readability: Hides the implementation details, making the code more readable and understandable;
- Facilitates Maintenance: Encapsulates code for easy maintenance and modification.

## Properties:

- Local: The implementation of an abstraction is independent of any other abstraction implementation;
- Substitutable: The implementation of an abstraction can be replaced by another implementation as long as the interface is the same and the implementation is correct.

# Abstractions

example:

```
// Here the user doesn't need to know how multiply is implemented
int square(int a) {
    return multiply(a, a);
}
```

Figure: Square

```
// Here the implementation of multiply can be replaced by another implementation
// as long as the abstraction is the same and the implementation is correct
int multiply(int a, int b) {
    return a * b;
    // return b * a; // This is also correct
}
```

Figure: Multiply

# Abstractions

Type signature of a function: used to declare the function. It is the function's name, the number of parameters, and the type of each parameter.

example:

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

Figure: Type signature



# Specification Comments

Specification Comments: in order to make your abstraction better understood by users, the following comments are recommended.

- **REQUIRES:** Preconditions that must hold, if any;
- **MODIFIES:** Variables that are modified, if any;
- **EFFECTS:** What the procedure does given legal inputs.

example:

```
int positiveAdd(int a, int b);  
// REQUIRES: a > 0, b > 0  
// MODIFIES: None  
// EFFECTS: Returns the sum of a and b
```

Figure: Specification Comments

# Recursion

A function calling itself is named recursion.  
need:

- base case: break out of recursion;
- recursive step: implement recursion.

```
int factorial(int n){  
    //Requires:  $n \geq 0$   
    //Effects: computes  $n!$   
    if(n==0) return 1; //base case  
    return n*factorial(n-1); //recursive step  
}
```

Figure: Recursion Example

# Recursion

Helper: Usually, due to input/output regulations or other reasons, a helper is more helpful for implementing recursion.

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

```
soln_helper()  
{  
    ...  
    soln_helper();  
    ...  
}
```

Figure: Using a Helper

# Function Pointers

When different functions have only slight differences and are almost identical, repeatedly writing the same code multiple times is a bad choice, as it will make the code less readable, maintainable, and more verbose(also mentioned in abstractions). So function points were introduced to avoid these problems.

# Function Pointers

example:

```
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 largest(list_t list){
    //Requires: List is not empty
    //Effects: returns largest element in list
    return compare_help(list, max);
}
```

Figure: Using a function pointer

# Function Pointers

Unlike variable points, for function points, we write

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

We don't write:

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

# Function Call Mechanism

Call stack: a stack that stores the order of function calls.

Stack: a set of objects which is modified as last in first out.

# Function Call Mechanism

Steps:

- 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;
- $f()$  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.



# Function Call Mechanism

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;  
}
```

Figure: Call stack

# Function Call Mechanism

Example:

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

main

x:

factorial

n:

RA: main line #3

factorial

n:

RA: factorial line #2

factorial

n:

RA: factorial line #2

factorial

n:

RA: factorial line #2





- Clubs 
- Diamonds 
- Hearts 
- Spades 

Figure: Suits

How to categorizing data here?

One way is to assign a number to each suit, like:

- `const int CLUBS = 0;`
- `const int DIAMONDS = 1;`
- `const int HEARTS = 2;`
- `const int SPADES = 3;`

And complete the code based on these assigned numbers later.

A feasible method, but not very convenient.

There is a better way: the enumeration (or enum) type.

# enum Type

Define an enum type:

```
enum Suit_t {CLUBS, DIAMONDS, HEARTS, SPADES};  
bool is_red(Suit_t s){  
    //REQUIRES: suit is one of Clubs, Diamonds, Hearts, or Spades  
    //EFFECTS: returns true if the color of this suit is red.  
    if(s==DIAMONDS || s==HEARTS) return true;  
    else return false;  
}  
int main(){  
    Suit_t k=CLUBS;  
    std::cout<<is_red(k);  
    return 0;  
}
```

Figure: Suit Example

# enum Type

Enum will correspond to numbers:

```
enum Suit_t {CLUBS, DIAMONDS,  
             HEARTS, SPADES};
```

It means

```
CLUBS = 0, DIAMONDS = 1,  
HEARTS = 2, SPADES = 3
```

This can simplify many operations, such as

```
Suit_t s = CLUBS;  
const string suitname[] = {"clubs",  
                           "diamonds", "hearts", "spades"};  
cout << "suit s is " << suitname[s];
```

- 06-Procedural-Abstraction.pdf - ECE2800J FA24
- 07-Recursion-Function-Pointers-and-Call-Mechanism.pdf - ECE2800J FA24
- 08-enum.pdf - ECE2800J FA24
- RC2.md - ECE2800J SP24