Lecture 6: Procedural Abstraction

Abstraction

Abstraction is the principle of separating what something is or does from how it does it.

- Provide details that matters (what)
- Eliminate unnecessary details (how)

There are 2 types of abstractions:

- Data Abstraction
- Procedural Abstraction

The product of *procedural abstraction* is a procedure, and the product of *data abstraction* is an abstract data type (ADT).

Procedural Abstraction

Properties

Functions are mechanism for defining procedural abstractions.

Difference between *abstraction* and *implementation*:

- Abstraction tells what and implementation tells how.
- The same abstraction could have different implementations.

There are 2 properties of proper procedural abstraction *implementation*:

- Local: the implementation of an abstraction does not depend of any abstraction implementation.
- Substitutable: Can replace a correct implementation wit another.

Composition

There are 2 parts of an abstraction:

- Type signature
- Specification

There are 3 clauses in the specification comments:

- REQUIRES: preconditions that must hold.
- MODIFIES: how inputs will be modified.
- EFFECTS: what the procedure is computing.

```c
void log_array(double arr[], size_t size)
// REQUIRES: All elements of `arr` are positive
// MODIFIES: `arr`
// EFFECTS: Compute the natural logarithm of all elements of `arr`
{
    for (size_t i = 0; i < size; ++i){
        arr[i] = log(arr[i]);
    }
}
```

Completeness of functions are defined as follows:

- If a function does not have any `REQUIRES` clauses, then it is valid for all inputs and is complete.
- Else, it is partial.
- You may convert a partial function to a complete one.

Lecture 7: Functions and Recursion

Function Pointers

Definitions

Variable that stores the address of functions are called *function pointers*. By passing them around we could pass functions into functions, return them from functions, and assign them to variables.

Consider when you only need to change one step in a larger function, like changing "adding" all elements in the matrix to "multiplying" all the elements. It is a waste of time and space to repeat the code, thus programmers would consider using a function pointer.

See the following example:

```c
int max(int arr[], size_t size) {
    int maximum = -INT_MAX;
    for (size_t i = 0; i < size; i++){
        maximum = max(maximum, arr[i]);
    }
    return maximum;
}

int min(int arr[], size_t size) {
    int minimum = INT_MAX;
    for (size_t i = 0; i < size; i++){
        minimum = min(minimum, arr[i]);
    }
    return minimum;
}

int avg(int arr[], size_t size) {
    int average = 0;
```

```
    for (size_t i = 0; i < size; i++){
        average += arr[i];
    }
    average /= size;
    return average;
}

int get_stats(int arr[], size_t size, int (*fun)(int[], size_t)){
    return fun(arr, size);
}

int main(){
    int arr[] = {1,2,3,4,5};
    cout << get_stats(arr, 5, min) << endl;
    cout << get_stats(arr, 5, max) << endl;
    cout << get_stats(arr, 5, avg) << endl;
}
```

You may consider functions as a bunch of numbers in the memory, and we can refer to the function by referring to the numbers. These numbers has an address in the memory as well, so we could use that address to refer to the function.

Properties

Given the function declarations,

```
void foo();

int foo(int x, int y);

int foo(int, int);

int *foo(int, char*);

char* foo(int[], int);
```

Write the corresponding function pointers.

```
void (*bar)();

int (*bar)(int, int);

int (*bar)(int, int);

int *(*bar)(int, char*);

char *(*bar)(int[], int);
```

Function pointers are invariant under `*`: Dereferencing a function pointer still gives back a function pointer. Consider the following example.

Given that:

```
int max(int x, int y) { return x > y ? x : y;}
```

What are the values of `a`, `b` and `c`?

```
int (*cmp)(int, int) = max;
int a = cmp(1, 2);
int b = (*cmp)(1, 2);
int c = (*******cmp)(1, 2);
```

It's possible to have an array of function pointers. Consider the following example:

```
int add (int x, int y) {return x + y;}
int multiply(int x, int y) {return x * y;}

int (*fun[])(int, int) = {add, multiply};

int main() {
    int op = 1, x = 0, y = 0;
    cout << "Select your operation (1 = add, 2 = multiply): ";
    cin >> op;
    cout << "Numbers: ";
    cin >> x >> y;
    cout << "ANS = " << fun[op-1](x, y) << endl;
}
```

Function Call Mechanism

To fully understand function call mechanism, you need to understand the memory organization of system, which, is beyond the scope of VE280 (Learn more about it in VE370 / EECS370). For now, you only have to get familiar with the concept of *call stack*.

At each function call, the program does the following:

1. Evaluate the actual arguments.

   For example, your program will convert `y = add(1*5, 2+2)` to `y = add(5, 4)`.

2. Create an activation record (stack frame)

   The activation record would hold the formal parameters and local variables.

   For example, when `int add(int a, int b) { int result = a+b; return result; }` is called, your system would create an activation record to hold:

   - `a`, `b` (formal parameters)
   - `result` (local parameters)

3. Copy the actual values from step 1 to the memory location that holds formal values.
4. Evaluate the function locally.
5. Replace the function call with the result.

   For the same example, your program will convert `y = add(5, 4)` to `y = 9`.

6. Destroy the activation record.

Typically, we come across situations with multiple functions are called and multiple activation records are to be maintained. To store these records, your system applies a data structure called *stack*.

A *stack* is a set of objects. When popping out an element from it, you always get the last element that is pushed into it. This property is referred to as *last in first out* (LIFO).

Consider the following example.

```
int add_2 (int x, int y) {return x + y;}

int add_3 (int x, int y, int z) = {return add_2(x, add_2(y, z));};

int main() {
    int x = 1, y = 2, z = 3;
    int a = add_3(x, y, z);
    cout << a << endl;
}
```

Readers are encouraged to simulate how stack frames are generated and destroyed on paper.

Recursions

Recursion simply means to refer to itself.

For any recursion problem, you may focus on the 2 compositions:

- Base cases: There is (at least) one “trivial” base or “stopping” case.
- Recursive step: All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.

A trivial example would be:

```
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
}
```

A recursive function could be simply recursive or tail recursive.

Recall from the previous section that when a function is called, it gets an activation record.

- A function is simply recursive if it generates a stack frame for each recursive call.
- A function is tail recursive if there is no pending computation at each recursive step.
    - It "re-use" the stack frame rather than create a new one.

The above `factorial` function is an example of a simply recursive function. It can also be redesigned as a tail recursive function.

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

Sometimes it is hard to implement a recursive function directly due to lack of function arguments. In this case, you may find a *helper* function useful.

Instead of

```
recursion(){
    ...
    recursion()
    ...
}
```

One may consider:

```
recursion(){
    ...
    recursion_helper()
    ...
}

recursion_helper(){
    ...
    recursion_helper()
    ...
}
```

An example is to check palindromes.

```
bool is_palindrome_helper(string s,
int begin, int end)
// EFFECTS: returns true if the subtring of s that
// starts at `begin` and ends 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;
}

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

```
    return is_palindrome_helper(s, 0, s.length()-1);
}
```

Lecture 8: Enumeration Types

Why `enum`

`enum` is a type whose values are restricted to a set of values.

Consider the example in your project 2.

```
enum Error_t {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

The advantages of using an `enum` here are:

- Compared to constant `strings`: more efficient in memory.
  - Even the minimum size of a `std::string` is larger than an `int`.
- Compared to constant `int`s or `char`s: more readable and limit valid value set.
  - Numbers or single chars are less readable than a string;
  - When passed to a function, you will always have to specify which integers/chars are valid in `REQUIRES`.

Properties

Enum values are actually represented as an integer types (`int` by default). If the first enumerator does not have an initializer, the associated value is zero.

```
enum Error_t {
    INVALID_ARGUMENT,   // 0
    FILE_MISSING,       // 1
    CAPACITY_OVERFLOW,  // 2
    INVALID_LOG,        // 3
};
```

Making use of this property, one usually writes:

```
int main(){
    const string error_info[] = {
            "The argument is invalid!",
            "The file is missing!",
            "Reaching the maximum capacity!",
            "The log is invalid!"
    };
    enum Error_t err = INVALID_LOG;
    cout << error_info[err] << endl;
}
```

Here are a few important properties of `enum`s.

Comparability:

Enum values are comparable. This means that you can apply <, >, ==, >=, <=, != on enum values.

Designable:

The constant values can also be chosen by programmer.

Consider the following example, what would the output be?

```
enum QAQ {
    a, b, c = 3, d, e = 1, f, g = f+d
};

int main(){
    QAQ err = g;
    cout << static_cast<int>(err) << endl;
}
```

Note that if you directly use `cout << err << endl`, the compiler would complain. You will need to statically cast the enum value to a printable type like `static_cast<int>`.

The correct answer would be:

```
enum QAQ {
    a,      // 0
    b,      // 1
    c = 3,  // 3
    d,      // 4
    e = 1,  // 1
    f,      // 2
    g = f+d // 6
};
```

The underlying integer type can be modified as well. If the range of `int` is too big or too small, you can also use a different underlying integer type.

```
enum Error_t : char {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
    INVALID_LOG,
};
```

Note that `char` is an integer type (Recall your knowledge about ASCII).

Enum Class

You may also define the `enum` as an enum class, *i.e.*

```
enum class Error_t {
    INVALID_ARGUMENT,   // 0
    FILE_MISSING,       // 1
    CAPACITY_OVERFLOW,  // 2
    INVALID_LOG,        // 3
};
```

When defining an variable, do not forget about `Error_t::`.

```
enum Error_t err = Error_t::INVALID_LOG;
```

Credit

SU2019 & SU2020 VE280 Teaching Groups.

VE 280 Lecture 6-8