RC4

SU Zhenxuan

Overview

- Recursion
- Function pointer
- Function call mechanism
- Enum

Recursion

- Definition
  - a function “refers itself”
- When to use?
  - A problem can be easily solved if assuming **sub-problem(s)** are solved
  - Subproblem is the same as the original one
- Do not forget boundary/base
  - Compiler does not even warn you

Example 1. Factorial

- Problem: given non-negative integer n, calculate its factorical $n!$
- Step 1. Determine boundary case:
  - $0! = 1$
- Step 2. Determine what sub-problem(s) should be like:
  - Calculate $m!$, where $m \neq n$
- Step 3. Determine how answer of sub-problem can help solve origin problem:
  - $n! = \frac{(n+1)!}{n+1}$, getting away from boundary case
  - $n! = n(n\ -1)!, n \geq 1$

Recursive helper function

- Scenario:

  You need to implement a function to solve a problem, and you find the problem can be easily solved using recursion. However, the function signature is fixed so you cannot change its argument (i.e. inputs), and you find for recursion, some inputs are useless while some extra inputs is needed.

- In this case, you can use recursive helper function to modify the inputs to whatever you need.

Example 2. Binary search

- Input:
  - A vector a containing integers, whose elements are non-decreasing
  - An integer x
- Problem: is x contained in a?

Example 2. Binary search

- Solution:
  - Step 1. If a is empty, return false. – Boundary case 1
  - Step 2. Find the middle element of a, compare it with x
    - Equal: return true. – Boundary case 2
    - $a[mid] < x$: find x in the left part of a
    - $a[mid] > x$: find x in the right part of a
- Now comes problem: you cannot directly pass “half of vector” recursively
  - It makes your code run slower
  - Helper function is needed

Example 2. Binary search

- One choice of helper function:
  - bool search_in_interval(std::vector<int> a, unsigned int, unsigned int r, int x);
    - EFFECT: search element x in the interval $[l, r)$ of vector a
    - REQUIRES: elements in a are non-decreasing
  - Then modify boundary cases and recursive procedure
  - search_in_interval(a, 0, a.size()) is the answer of original problem

Recursive helper function

- Helper function can also take few arguments

```
struct big_struct{
    int a[100000];
};
int f(big_struct s, int n) {
    if (n == 0) return 0;
    return 1 + f(s, n - 1);
}
int main() {
    big_struct s;
    f(s, 100);
}
```

```
struct big_struct{
    int a[100000];
};
int g(big_struct& s, int n) {
    if (n == 0) return 0;
    return 1 + g(s, n - 1);
}
int f(big_struct s, int n) {
    return g(s, n);
}
int main() {
    big_struct s;
    f(s, 100);
}
```

Function pointers

- Solve several similar problems
  - E.g. given an array, write one function to find max, one to find min
- Writing one function for each problem is boring, and may cause more bugs to appear
- Better way: write a function that takes a function pointer as input
  - By passing different function pointer this function can do different task
  - Higher level of abstraction

Function pointers

- Definition:
  - T0 (*fp)(T1, T2,…);
  - T0 is the return type, T1, T2,… are the parameter type
  - Example: int (*fp)(int, int);
  - Recall type signature and function definition
    - int min(int, int b); (parameter name may and may not be omitted)
    - Change the function name into (*fp)
    - int (*fp)(int, int);

Function pointers

- Assign a function to a function pointer:
  - fp=min; fp=&min;
  - Both work correctly, but we often write in the former way.
- Call function pointer:
  - fp(1, 2); (*fp)(1, 2);
  - Both work correctly, but we often write in the former way.
- Note here is different from normal variable pointers.

Function pointers

- Function pointers as function argument

```
#include <iostream>
int fi_se(int a, int b, int (*fp)(int, int)) {
    return fp(a, b);
}
int fi(int a, int b) {
    return a;
}
int se(int a, int b) {
    return b;
}
int main() {
    std::cout << fi_se(2, 3, fi) << '\n' << fi_se(2, 3, se) << '\n';
}
```

Function pointers

- You can just treat function pointers as other types of variables
  - Declare array of function pointers (sometimes useful)
  - Declare function pointer which is const (almost useless)

Function call mechanism

- When we call a function, the program does:
  1. Evaluate actual arguments (order is not guaranteed)
     - What is the output?

```
#include <iostream>
int output(int x, int y) {
    std::cout << x << ' ' << y << '\n';
    return 0;
}
int f(int x) {
    std::cout << x << '\n';
    return x;
}
int main () {
    output(f(1), f(2));
}
```

Function call mechanism

- When we call a function, the program does:
  1. Evaluate actual arguments (order is not guaranteed)
     - What is the output?

```
#include <iostream>
int output(int x, int y) {
    std::cout << x << ' ' << y << '\n';
    return 0;
}
int f(int x) {
    std::cout << x << '\n';
    return x;
}
int main () {
    output(f(1), f(2));
}
```

Function call mechanism

- When we call a function, the program does:
  1. Evaluate actual arguments (order is not guaranteed)
  2. Store formal parameters and local variables into “activation record” (also called “stack frame”)
  3. Copy the actual values of arguments to formals’ storage space
  4. Evaluate the function (like “step into” function when you use IDE to debug)
  5. Replace the function call with return value
  6. Destroy the activation record

Function call mechanism

- Some import concepts:
  - 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 : a function call itself.
  - Passing pointer/reference: what is the result after function call? Will outside variables bemodified?

Function call mechanism

- Understanding function call mechanism helps you read codes
  - e.g. read recursion as loop.

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

  - From boundary case:
    - $n! = 1 * 2 * 3 * \cdots * n$

Enum

- enum is a **type** whose values are restricted to a set of integer values
- Advantage
  - Use less memory than std::string
  - More readable than const int or char
  - Limit valid value set, so compiler help you find spelling mistakes.

Enum

- Example:

```
#include <iostream>

enum A {
    a, b, c=-1, d, e=5, f, g=a + e, h
};

int main() {
    std::cout << a << ' ' << b << ' ' << c << ' ' << d << ' ‘
              << e << ' ' << f << ' ' << g << ' ' << h << '\n';
}
```

  - Output is 0 1 -1 0 5 6 5 6
- By default the enum value starts from 0, and increments for each value
  - But you can also assign any integer value to them.
- Values in enum (a, b, c,⋯) can be treated as global const int
  - Can be compared.

Enum

- Since enum A is a new type, cin and cout cannot identify them
  - Cast the enum variable to int before print it.

```
#include <iostream>

enum A {
    a, b, c=-1, d, e=5, f, g=a + e, h
};

int main() {
    A A1=a;
    std::cout << static_cast<int>(A1) << '\n';
}
```

Enum

- Enum type:

```
#include <iostream>
Enum A : char {
    a, b, c=-1, d, e=5, f, g=a + e, h
};

int main() {
    std::cout << A::a << ' ' << A::b << '\n';
}
```

- You can change the type if range of int is too big or too small
  - char or long long is OK
  - float or double are not

Enum

- Enum class:

```
#include <iostream>
Enum class A {
    a, b, c=-1, d, e=5, f, g=a + e, h
};

int main() {
    std::cout << A::a << ' ' << A::b << '\n';
}
```

  - a,b,c,⋯ are no longer global.
    - Use A::a, A::b because they are members of class A.