MidRC

L7 Recursion; Function Pointers

Recursion

I think recursion is the most important part in your midexam and final exam:)

Composition

mainly contain 2 parts:

- "stopping" case（base case）to stop the recursion
- recursive step

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1; // 1. BASE CASE, i.e. recursion must have a point of ending
    }
    return n * factorial(n - 1); // 2. recursive call
}
```

Thinking Strategy

1. Find the stopping case (base case).
2. Observe the relationship between large size cases and small size cases.(e.g. from `n-1` to `n`. This is the most difficult step)
3. Write the code!

Pascal Triangle

- Problem Statement

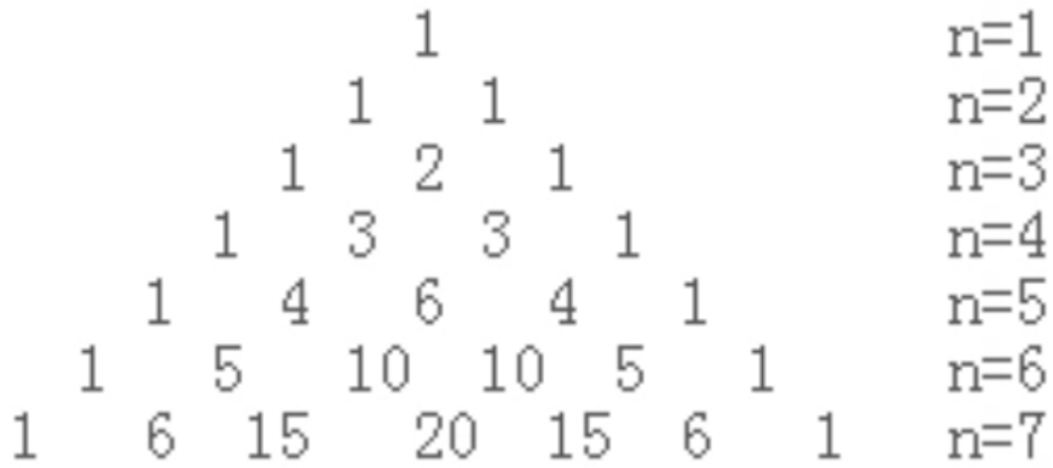


Figure out the value of `i` th row and `j` th column(`i` and `j` start from 0)

- Thinking Strategy
  - Base case: `i == 0` or `j == 0` or `i == j`, the value equals to 1.
  - Recursive step:
    - The value of `(i, j)` equals to the value of `(i - 1, j)` + value of `(i - 1, j - 1)`.
- Code:

```
int pascal(int i, int j)
{
    if(i == 0 || j == 0 || i ==j)
    {
        return 1;
    }
    return pascal(i - 1, j - 1) + pascal(i - 1, j);
}
```

Function Pointers

Motivation

- Save your time and make your code elegant.
- `filter_odd` and `filter_even` -> `filter`、
- Treat **functions as variables**.

Grammar

```
int (*foo)(int a, int b);
foo = max;
foo(5,3);
```

Example

Use function as function parameters:

```
void bubbleSort(int *arr, int n, bool (*cmp)(int, int))
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (cmp(arr[j], arr[j + 1]))
            {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

- If you pass in `bool cmp(int a, int b) { return a < b; }`, the array will be sorted in **ascending order**.
- If you pass in `bool cmp(int a, int b) { return a > b; }`, the array will be sorted in **descending order**.

L8 Function Call Mechanism

Function Call Mechanism

Single Function Call Mechanism

Example:

```
int add(int a, int b)
{
    int result = a + b;
    return result;
}

int main()
{
    int a1 = 1;
    int b1 = 2;
    int c1 = add(a1+b1, b1);
}
```

Steps to call `add`:

1. Evaluate `a1+b1` and `b1`.
2. Create an activation record/stack frame to hold formal parameters a and b, and local variables result.
3. Copy `a1+b1` and `b1` to `a` and `b` respectively.
4. Execute the function body.
5. Replace the function call with return value `result`.
6. Destroy the activation record.

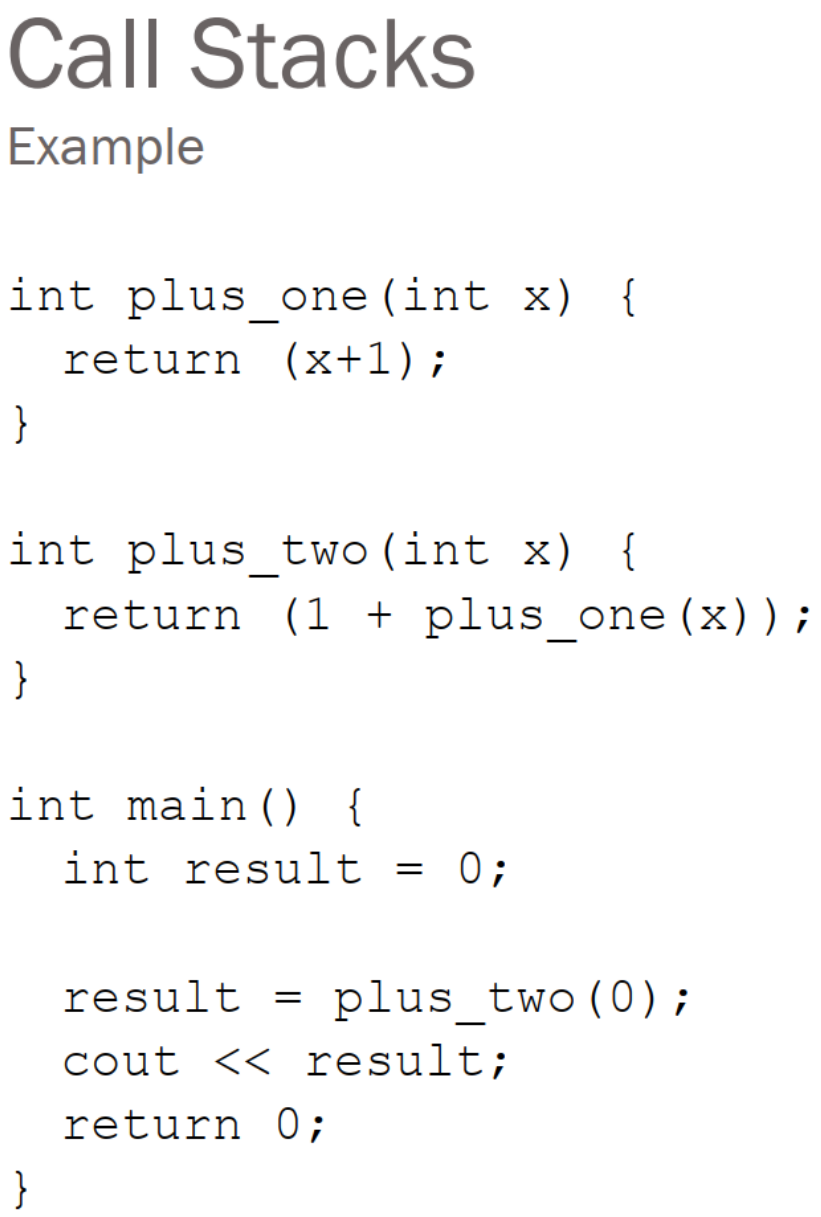
Multiple Function Call Mechanism: Call Stack

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

- When a function is called, its activation record is **added** to the "top" of the stack.
- When that function returns, its activation record is **removed** from the "top" of the stack.

Examples:

Refer to the lecture slide for the ordinary, using pointers, and recursive examples.



L9 Enum; Program Augments;

Enum

Example:

```
enum Suit_t {CLUBS, DIAMONDS, HEARTS, SPADES};
// numerically CLUBS = 0, DIAMONDS = 1, HEARTS = 2, SPADES =3
Suit_t s = CLUBS;
const string suitname[] = {"clubs", "diamonds", "hearts", "spades"};
cout << "suit s is " <<suitname[s]; //use enum type as array index.

bool isRed(Suit_t s)
{
    return s == DIAMONDS || s== HEARTS;
}
```

Keypoints:

- The suits are assigned with default values. `CLUBS = 0, DIAMONDS = 1, HEARTS = 2, SPADES =3`.
- It's OK to use an `enum` as array index, like `suitName[CLUBS] .`

Program Augments

- Know how to write more general programs that can take arguments

Grammar

```
int main(int argc, char *argv[])
```

Keypoints:

- `argc`: number of arguments (including the program name)
- `argv`: array of arguments as C string (including the program name)

Example

mydiff.cpp

```
int main(int argc, char *argv[])
{
    std::cout << argc << std::endl;
    for (int i = 0; i < argc; i++)
    {
        std::cout << argv[i] << std::endl;
    }
// Implementation of diff
}
```

Command in:

```
./mydiff file1 file2
```

Outputs:

```
3         // argc
./mydiff  // argv[0]
file1     // argv[1]
file2     // argv[2]
```

Useful function

```
#include <cstdlib>
int atoi(const char *nptr); // e.g. converts "39" to 39
```

L10 IO

Buffer

- I/O in C++ is buffered: a region of memory that holds data during input or output operations.

The buffer content is cleaned when:

- A newline (e.g., endl or '\n') is inserted into the stream.

```
cout << "ok" << endl;
cout << "ok" << '\n';
```

- The buffer is explicitly flushed.

```
cout << "ok" << flush;
```

- The buffer becomes full.
- The program decides to read from `cin`.
- The program exits.

iostream

- `cin`: standard input (buffered)
- `cout`: standard output (buffered)
- `cerr`: output error messages (not buffered)

fstream

- header file: `#include <fstream>`

Example

```
#include <fstream>
using namespace std;
int main(){
    ifstream ifs;
    ofstream ofs;
    ifs.open("input.txt");
    ofs.open("output.txt");
    char ch;
    while((ch = ifs.get())!=EOF){ // returns a single character if success,
        ofs << ch;                // otherwise -1
    }
    while(ifs.get(ch)){ // returns true if the reading is successful,
        ofs << ch;      // otherwise false
    }
    string s;
    while(getline(ifs,s)){ // returns a reference to its first parameter
        ofs << s;
    }
    ofs << ch << s << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

sstream

- header file: `#include <sstrean>`

```
#include <sstream>
using namespace std;
int main(){
    istringstream is;
    ostringstream os;
    string foo;
    int bar;
    string s = "VE 280.";
    is.str(s);
    is >> foo >> bar;
    os << foo << bar;
    s = os.str();
    return 0;
}
```

L11 Testing

Concepts

- Testing: discover a problem
- Debugging: Fix the problem
- incremental testing: test individual pieces of your program (such as functions) as you write them
    - test smaller, less complex, easier to understand units.
    - You just wrote the code and it is fresh in your mind.

Steps

1. Understand the specification
2. Identify the required behaviors
    - **required behaviors:** For any specification, boil the specification down to a list of things that must happen.(See examples in the lecture slides)
3. Write specific tests
    - **Simple inputs**
    - **Boundary conditions**
    - **Nonsense**
4. Know the answers in advance
5. Include stress tests
    - large test cases
    - long running test cases

Exercise

Write 3 different boundary cases for insert_list in your Project 2. Each case should test a different boundary situation. For each test case, you must provide: a description of the test case, the expected behavior for a correct implementation of the function. Use the provided example as format guideline.

```
list_t insert_list(list_t first, list_t second, unsigned int n)
// REQUIRES: n <= the number of elements in the list "first".
//
// EFFECTS: Returns a list comprising the first n elements of
//          "first", followed by all elements of "second",
//          followed by any remaining elements of "first".
//
// For example: insert (( 1 2 3 ), ( 4 5 6 ), 2)
//              is ( 1 2 4 5 6 3 ).
```

Answer:

```
```

L12 Exception

Concepts

- Motivation: recognize and handle unusual conditions in the program at runtime.
- Exception: something bad that happens in a block of code, preventing the block from continuing to execute.
- Mechanism: If the exception occurs, the program will move to the handler.

Try-Catch Block

- `try`: throws the exception
- `catch`: handles the exception

```
try
{
    if(foo) throw 2.0;
    if(bar) throw 'a';
    if(list) throw list_make();
}
catch (int n) { }
catch (char c) { }
catch (list_t l) { }
catch (...) { } //default handler
```

- If the exception is successfully handled in the catch block, execution continues normally with the first statement following the catch block.

  ```
  void foo(int i)
  {
      try { ... }
      catch (int v) {...}
      ... // Do something next
  }
  ```

Rules:

- You cannot write a catch block unless you have a try block before it.

- Exception will be propagated along the calling function stack. Only the first catch block with the same type as the thrown exception object will handle the exception

Exercise

1. What is the output of the following code?

```
void foo(int x)
{
    try
    {
        bar(x);
    }
    catch(int a)
    {
        cout << "int in foo\n";
    }
    catch(double b)
    {
        cout << "double in foo\n";
    }
    cout << "exit foo\n";
}

void bar(double x)
{
    throw x;
    try
    {
        throw x;
    }
    catch(double a)
    {
        cout << "double in bar\n";
    }
    cout << "exit bar\n";
}

int main()
{
    int x = 6;
    foo(x);
}
```

Your answer:

```

```

Reference

[1] Zhang Wenjing. VE280 RC5. 2023FA.

[2] Weikang Qian. VE280 Lecture 7-12. 2023FA.

[3] Zhongqiang Ren. VE280 Lecture 7-12. 2024SP.

[4] Zhanxun Liu. VE280 RC4. 2023FA.