Functions (II)

Parameter Passing (cont.)

- We know that we can use pointers in C to simulate pass-by-reference.
- However, passing pointers as values is not equivalent to pass-by-reference. We can tell the difference after knowing what is pass-by-reference in C++.

- C++ allows passing by reference. Show swap.cc.

```cpp
#include <iostream>
using namespace std;

int swap (int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
    return 0;
}

int main () {
    int a = 3, b = 5;
    cout << "Before swap a = " << a << " b = " << b << "\n";
    swap(a, b);
    cout << "After swap a = " << a << " b = " << b << "\n";
    return 0;
}
```

- The above swap function passes arguments as references. Using pass-by-reference, the only thing the swap function can do is update the values in the memory referred by the references. It's syntactically incorrect to change the reference values inside the swap function. This is to say it's invalid to do anything like &a = &b in the above swap function.

- Note: passing by reference is not equivalent to passing pointer by value.
  
  • If a function is pass-by-reference, it's unlikely to modify the reference itself in the function be the value referred by the reference.
  
  • If a function is pass-by-value and the value is a pointer, it's possible to reassign another address to the pointer, and the changes on the value referred by the new address won't impact the value stored in the original address.

  • Show ppbv.cc and ask the output.
Another thing you may want to pay attention to pass-by-reference is the aliasing, which means reference to the same entity by different names.

```cpp
#include <iostream>
using namespace std;

int main () {
    int a = 1;
    cout << "Before f, a = " << a << endl;
    f( a );
    cout << "After f, a = " << a << endl;
    cout << "Before g, a = " << a << endl;
    g( &a );
    cout << "After g, a = " << a << endl;
    return 0;
}
```

- Before open_to_problems, x = 10, y = 20
- After open_to_problems, x = 11, y = 21
- Before open_to_problems, x = 11
- After open_to_problems, x = 13
Function Call Implementation

- An easy way to implement factorial? [recursion]

```c
/**
 * factorial.c
 */
#include <stdio.h>
int factorial (int n) {
    if (n < 2)
        return 1;
    else
        return n * factorial (n-1);
}
int main () {
    int n = 3;
    int i = factorial (n);
    printf("factorial(%d) = %d\n", n, i);
    return 0;
}
```

- A key component of a recursive function?
  - The **base case**. Otherwise, the function recurs forever.
  - In the above program, the base case is the if branch.

- How does recursive function work?
  - The factorial function is first called in the main function, and passes value 3 to it.
  - When executing the factorial function, factorial function is called again, and passed value 2 to it.
  - When calculating factorial 2, the function is called for the third time to calculate factorial 1.
  - When get the return value of factorial 1, factorial 2 is calculated.
  - After getting the value of factorial 2, factorial 3 is calculated.
  - Then the final value is returned to the main function.

- This process makes sense logically. But **how the computer handles it? How are those active records saved for recursive calls?**
  - Generally, the computer uses the system or program **stack** to hold active records.
  - **Active records, also called stack frames**, need to hold the following types of information.
    - Return address
    - Return value
    - Arguments
    - Parameters
    - Local variable graphic explains how the stack holds the active records.
  - This stack is a up-side-down stack, and for simplicity, stack frame only holds arguments and return values.