UEE1302 (1102) F10
Introduction to Computers and Programming (I)

Programming Lecture 05
Function (Part II): Parameters, Recursion and Overloading

Learning Objectives

- Parameters
  - call-by-value
  - call-by-reference
  - mixed parameter-lists
- Recursive calls to functions
  - Fibonacci Numbers
  - Hanoin Tower Game
- Overloading and default arguments
  - examples & rules

Passing Parameters

- Two methods of passing arguments as parameters
  - **Call-by-value**
    - copy of value is passed
  - **Call-by-reference**
    - address of actual argument is passed

Call-by-Value Parameters

- Copy of actual argument (value) passed
  - Considered local variable inside function
  - If modified, only local copy (value) changes
    - internally function has no access to actual argument from caller
  - This is the default method
    - used in all examples so far
# Example of Call-by-Value

```cpp
#include <iostream>

using namespace std;

void fSwap(int iObj_a, int iObj_b);

int main()
{
    int iNum_01 = 7;
    int iNum_02 = 9;
    fSwap(iNum_01, iNum_02);
    cout << iNum_01 << iNum_02 << endl;
    return 0;
}

void fSwap(int iObj_a, int iObj_b)
{
    int iObj_c = iObj_a;
    iObj_a = iObj_b;
    iObj_b = iObj_c;
}
```

---

## Call-by-Value Pitfall

- **Common mistake:**
  - Declaring parameters again inside function:
    ```cpp
double fee(int hrsWorked, int minsWorked)
    {
        int quarterHours; // local variable
        int minsWorked; // NO!
    }
    ```
  - Compiler error message ⇒ "redefinition error"

- Value arguments ARE like **local** variables
  - But function gets them automatically

---

## Reference Variables

- A **reference** variable is an **alternative** name for a variable.
  - A reference variable must be initialized to reference another variable.
  - Once the reference is initialized you can treat it just like any other variable.

- To declare a reference variable you precede the variable name with a "&":
  - int& foo;
  - double& cost;

---

## Examples of Reference Variables

```cpp
int count;
int& blah = count;
// blah is the same variable as count
count = 3;
cout << "blah is " << blah << endl;
blah++;
cout << "count is " << count << endl;
```

```
blah is 3
count is 4
```
**Call-by-Reference Parameters**

- Used to provide access to caller’s actual argument $\Rightarrow$ caller’s data can be modified
- Typically used for input function
  - To retrieve data for caller
  - Data is then given to caller
- Specified by ampersand, &, after type in formal parameter list (typically in definition)

Ex: 
```
dataType FuncName(type1& arg_1, 
                   type2& arg_2, ..., type_n& arg_n) 
{ ... } 
```

**Call-by-Reference Details**

- What’s really passed in?
  
  - A **reference** back to caller’s actual argument!
  
    - refers to memory location of actual argument
    - called **address**, which is a unique number referring to distinct place in memory

**Example of Call-by-Reference**

```cpp
#include <iostream>
using namespace std;
void fSwap(int&, int&);

int main() 
{ 
  int iNum_01 = 7;
  int iNum_02 = 9;
  fSwap(iNum_01, iNum_02);
  cout << iNum_01 << endl;
  cout << iNum_02 << endl;
  return 0;
}
```

```cpp
void fSwap(int& Ref_a, int& Ref_b) 
{ 
  int Loc_c = Ref_a;
  Ref_a = Ref_b;
  Ref_b = Loc_c;
}
```

**Constant Reference Parameters**

- Reference arguments inherently "dangerous"
  
  - Caller’s data can be changed
  
    - Often this is desired, sometimes not

- To "protect" data, & still pass by reference:
  
    - Use `const` keyword.
    
      - EX:

      ```cpp
      void sendConstRef(const int& par1, 
                         const int& par2);
      ```

    - Make arguments "read-only" by function
    
      - No changes allowed inside function body
Parameter vs. Argument (revisited)

- Confusing terms, often used interchangeably
- True meanings:
  - formal parameters ⇒ appear in function declaration and function definition
  - arguments ⇒ appear in function call (argument list) and used to fill in a formal parameter
- Call-by-Value vs. Call-by-Reference
  - simply the mechanism used in plug-in process

Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  - \( \text{int mixedCall(int\& p1,int p2,double\& p3)}; \)
- In function call:
  - \( \text{mixedCall(arg1,arg2,arg3)}; \)
  - arg1: integer type & passed by reference
  - arg2: integer type & passed by value
  - arg3: double type & passed by reference

Choosing Formal Parameter Names

- Same rule as naming any identifier:
  - meaningful names!
- Functions as self-contained modules
  - designed separately from the rest of program
  - assigned to teams of programmers
  - all must understand proper function uses
  - OK if formal parameter names are the same as argument names
- Choose function names with the same rules

Learning Objectives

- Parameters
  - call-by-value
  - call-by-reference
  - mixed parameter-lists
- Recursive calls to functions
  - Fibonacci Numbers
  - Hanoi Tower Game
- Overloading and default arguments
  - examples & rules
Recursive Calls

- Functions can **call themselves**!
  - called recursive call (or recursion)
- Recursion is very useful and economical
  - very simple to express a complicated computation recursively.
  - can be converted to nonrecursive functions
- Developing a recursive function
  - **base** step: when the function does not call itself again ⇒ stop condition
  - **recursive** step: compute the return value the help of the function itself ⇒ call itself

Base Step in Recursion

- The base step corresponds to a case in which you’ve known the answer
  - the function returns the value immediately
  - or can easily compute the answer
  - typically, \( f(0), f(1) \) and etc
- If you don’t know a base step, you can’t use recursion!
  - probably do NOT understand the problem
  - often cause **infinite execution** of the program if no base step ⇒ **never stop!**

Recursive Step in Recursion

- Use the recursive call to solve a **sub-problem**
  - the parameters must be **different**
  - typically the input range becomes **smaller**
    ⇒ otherwise, the recursive call will get us no closer to the solution
- Need to do something besides just making recursive calls repeatedly
  - That is the base step

Fibonacci Numbers (revisited)

- Fibonacci sequence:
  \[
  f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}
  \]
- Compute \( \text{fibon}(n) \) by a non-recursive function:
  - how??
- Compute \( \text{fibon}(n) \) by a recursive function:
  ```cpp
  int fibon(int n) {
    int ans;
    if (n==0 || n==1) ans = n;
    else ans = fibon(n-1) + fibon(n-2);
    return ans;
  }
  ```
Hanoi Tower Game

- A popular Math game in Europe, 19th century:
  - consists of three pegs (poles), and a number of disks of different sizes
  - objective is to move the entire stack of disks from 1st peg to 3rd peg

- Game rules:
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg
  - No disk can be placed on top of a smaller disk

Example of Hanoi Tower (1/2)

Example of Hanoi Tower (2/2)

Recursive Step for Hanoi Tower

- Key idea:
  - move 1..(n-1) disks from peg A to peg B
  - move n disk to from peg A to peg C
  - move 1..(n-1) disks from peg B to peg C
Pseudocode for Hanoi Tower

// This is only the pseudocode for Hanoi Tower, // However, C++ code can be developed easily

HanoiTower(n, A, B, C) {
    if (n==1) {
        move disk 1 from peg A to peg C;
    } else {
        HanoiTower(n-1, A, C, B);
        move disk n from peg A to peg C;
        HanoiTower(n-1, B, A, C);
    }
}

Exercises on Recursion

- Rewrite the gcd() function into a recursive version. Hint: by Euclidean algorithm

- Given a series A= {1/2, 1/4, 1/8, …}. Please write a recursive function to compute A(k).

Learning Objectives

- Parameters
  - call-by-value
  - call-by-reference
  - mixed parameter-lists

- Recursive calls to functions
  - Fibonacci Numbers
  - Hanoian Tower Game

- Overloading and default arguments
  - examples & rules

Overloading

- Same function name
  - Different parameter lists

- Two separate function definitions
  - Function "signature"
    - Function name & parameter list
    - Must be "unique" for each function definition

- Allows same task performed on different data
Example of Overloading

- Function computes average of two numbers:
  ```cpp
  double average(double n1, double n2) {
    return ((n1 + n2) / 2.0);
  }
  ```
- Now compute average of three numbers:
  ```cpp
  double average(double n1, double n2, double n3) {
    return ((n1+n2+n3) / 3.0);
  }
  ```
- Same name, two functions

Example of Overloading (cont’d)

- Which function gets called?
  - Depends on function call itself:
    ```cpp
    avg = average(5.2, 6.7);
    ⇒ call "two-parameter average()"
    avg = average(6.5, 8.5, 4.2);
    ⇒ call "three-parameter average()"
    ```
  - Compiler resolves invocation based on the **signature** of function call
    - match call with appropriate function
    - each considered as a separate function

Overloading Pitfall

- Only overload **same-task** functions
  - A mpg() function should always perform same task, in all overloads
  - Otherwise, unpredictable results
- C++ function call resolution:
  - 1st order: looks for **exact** signature
  - 2nd order: looks for **compatible** signature

Overloading Resolution

- **1st exact match**: look for exact signature
  - no argument conversion required
- **2nd compatible match**: look for compatible signature where automatic type conversion is possible:
  - 1st with **promotion** (e.g., int ⇒ double)
    ⇒ no loss of data
  - 2nd with **demotion** (e.g., double ⇒ int)
    ⇒ possible loss of data
### Example of Overloading Resolution

- Given the following functions:
  - (1) int func(int n, double m);
  - (2) int func(double n, int m);
  - (3) int func(int n, int m);
- Consider these calls:
  - \( f(98, 99); \Rightarrow \text{call func (3)} \)
  - \( f(5.3, 4); \Rightarrow \text{call func (2)} \)
  - \( f(4.3, 5.2); \Rightarrow \text{call func ???} \)
- Should avoid such confusing overloading

### Type Conversion in Overloading

- Numeric formal parameters typically made "double" type
- Allows for "any" numeric type
  - any "subordinate" data automatically promoted
    - Ex: int \( \Rightarrow \) double
      - float \( \Rightarrow \) double
      - char \( \Rightarrow \) double
- Avoid overloading for different numeric types

### Example of Automatic Type Conversion

```cpp
def double mpg(double miles, double gallons) {
    return (miles/gallons);
}
```

Examples of function calls:
- mpgComputed = mpg(5, 20); \( \Rightarrow \) convert 5 & 20 to doubles, then passes
- mpgComputed = mpg(5.8, 20.2); \( \Rightarrow \) no conversion necessary
- mpgComputed = mpg(5, 2.4); \( \Rightarrow \) convert 5 to 5.0, then passes values

### Default Arguments

- Allow omitting some arguments
- Specified in function declaration/prototype
  - int showVolume( int length, int width = 1, int height = 1 );
  - last two arguments have default values
- Possible calls:
  - showVolume(2, 4, 6); // all arguments supplied
  - showVolume(3, 5); // height defaulted to 1
  - showVolume(7); // width & height defaulted to 1
Summary (1/2)

- Formal parameter is placeholder, filled in with actual argument in function call
- Call-by-value parameters are local copies in receiving function body
  - actual argument cannot be modified
- Call-by-reference passes memory address of actual argument
  - alternative name for variables
  - actual argument can be modified
  - argument MUST be variable, not constant

Summary (2/2)

- Recursion express a complicated computation recursively
  - a complete recursion consists of (1) base step and (2) recursive step
  - recursion for Fibonacci sequence
  - Hanoi Tower Game
- Multiple definitions of same function name possible: overloading
- Default arguments allow function call to "omit" some or all arguments in list
  - If not provided ⇒ default values assigned