CMPSC 461: Programming Language Concepts, Spring 2024
Assignment 4 Practice Notes Packet

Prof. Suman Saha

October 7, 2025

Problem 1: Calling sequence

Consider the following C program :

```
1   void func1(void);
2   void func2(void);
3   void func3(void);
4
5   void main(){
6       int a,b,c;
7       /* body contents of main*/
8   }
9
10  void func1(void){
11      int b,c,d;
12      /* body contents of func1*/
13  }
14
15  void func2(void){
16      int c,d,e;
17      /* body contents of func2*/
18  }
19
20  void func3(void){
21      int d,e,f;
22      /* body contents of func3*/
23  }
```

Assuming dynamic scoping is used, what variables are visible during the execution of the last called function in the following sequences? Include the name of the function in which the variable was defined along with each visible variable.

1. **main** calls **func1**, **func1** calls **func2**, **func2** calls **func3**
2. **main** calls **func1**, **func1** calls **func3**
3. **main** calls **func2**, **func2** calls **func3**, **func3** calls **func1**
4. **main** calls **func3**, **func3** calls **func3**
5. **main** calls **func1**, **func1** calls **func3**, **func3** calls **func2**

For example, the sequence "**main** calls **func3**, **func3** calls **func2**, **func2** calls **func1**" will lead to the answer :
Visible variables : a(main), b(func1), c(func1), d(func1), e(func2), f(func3)

Solution
In dynamic scoping, the compiler first searches the current block and then successively all the calling functions.This would lead to the following answers :

1. a(main), d(f3), e(f3), f(f3), c(f2), b(f1)

2. a(main), d(f3),e(f3), f(f3), b(f1), c(f1)
3. a(main), b(f1), c(f1), d(f1), e(f3), f(f3)
4. a(main), b(main), c(main), d(f3), e(f3), f(f3)
5. a(main), c(f2), d(f2), e(f2), f(f3), b(f1)

Problem 2: Parameter Passing 1

Observe the following C++ code that swaps two variables :

```
#include<stdio.h>
void swap(int n1, int n2)
{
  int temp = n1;
  n1 = n2;
  n2 = temp;

}
int main() {
    int x = 25;
    int y = 70;
    swap(x, y);
    printf("The numbers after swapping n1 and n2 %d %d \n",x, y);
    return 0;
}
```

How would the output of the program change if the parameters are :

1. Called by value?
2. Called by reference?

Solution

1. When called by value, we are merely passing a copy of x and y into the function. This means that only these "copies" are swapped, and the original variables themselves remain unchanged. Therefore, the output will be (25, 70).
2. When called by address, the memory locations of the variables are passed on to swap(), which results in the intended swap occurring. Therefore, the output will be (70, 25).

Problem 3: Parameter passing 2

Consider the following C-like program. Write down what will be printed out when the parameters are passed (1) by value, (2) by reference and (3) by value return. For each answer, briefly explain how you derived it.

```
1     int x=5, y=6;
2     void foo(int a, int b) {
3         x = a+b;
4         b = a+a;
5     }
6
7     main () {
8         foo(x,y);
9         print x, y;
10    }
```

Solution

1. Call by value : 11,6
   For calling by value, x will be assigned the value (5 + 6) and y will remain as it is in foo(). This leads to the values 11 and 6 for x and y respectively.

2. Call by reference : 11,22
   For calling by reference, the actual locations in memory are passed instead of the "variables" themselves. This means that in foo(), x is assigned the value (5 + 6) and y is assigned the new, changed value of x added to itself. This leads to the values 11 and 22.

3. Call by value return : 5,10
   In call by value return, the scope of the calling function will override the scope of the called function. This would mean that even though x is modified within foo(), the global x = 5 is the value that it will retain after the execution of foo(). The new value of "b" is copied back into y, which produces the above output.

Problem 4: Tail Recursion

Consider the following JAVA program

```
1 public int countNodes(Node<T> start) {
2     if (start == null) {
3         return 0;  // Base case
4     }
5     else {
6         return 1 + countNodes(start.next);
7     }
8 }
```

Write down a **tail recursive** implementation of the function **countNodes** in JAVA language or programming language of your choice. You may use a helper function in your solution.

Solution Note that there are multiple ways to craft the solution for this problem. Focus on the return statement and ensure it is tail-recursive.

```
1  public int countNodes(Node<T> start) {
2      return countNodesHelper(start, 0);
3  }
4  public int countNodesHelper(Node<T> start, int count) {
5      if (start == null) {
6          return count;  // Base case
7      }
8      else {
9          return countNodesHelper(start.next, count + 1);
10     }
11 }
```

The original function provides an abstraction to the helper function where the initial arguments for the helper are set by the programmer (start, 0) in line2. The helper function returns count upon hitting base case otherwise keeps track of the final count when passed as an argument to itself.

Keep in mind the helper function helps ensure we always begin with a count of 0 and increment by 1 whereas without an helper function one can set an initial argument of 100 for the count giving an incorrect result.

Problem 5: Tail-recursion

The Collatz sequence for a number n is generated as follows:

- If n is even, divide it by 2.
- If n is odd, multiply it by 3 and add 1.
- The sequence meet its termination when it first encounters the number 1, as it goes into a short repetition $(1, 4, 2, 1, ...)$

Example: For $n = 6$, the Collatz sequence is: $6, 3, 10, 5, 16, 8, 4, 2, 1$. Therefore, we define the term **Collatz-length** as the length of the Collatz sequence with starting number n before it reaches 1. E.g., the Collatz-length of 6 is 8.

1. Write down a **tail recursive** implementation of the function **collatzlength** in python language preferably or programming language of your choice. You may use the helper function in your solution.
2. Explain why your implementation of Collatzlength is **tail-recursive**?

Solution

1. Code solution in Python:

```
def collatzlength(n):
    return collatzhelper(n, 0)

def collatzhelper(n, steplength):
    if n == 1:
        return steplength
    if n % 2 == 0:
        n = n/2
    else:
        n = (n*3) + 1
    return collatzhelper(n, steplength+1)
```

2. The above provided **Collatzhelper** function avoids the problem of growing recursive stack frames by adding 1 to the steplength and passing it as the parameter for the helper function to keep track of steplength across multiple activation records and finally returns step length. The addition of helper function (private) also ensures that an unknown user doesn't have control over argument steplength in order to avoid incorrect answers for the problem.

Problem 6: Tail Recursion I

Consider the following Python program

```
def sq_sum(lst):
    """Sums a list of numbers squared."""
    if len(lst) == 0:
        return 0
    return (lst[0] * lst[0]) + sum(lst[1:])

def division_mult(number):
    ""Returns the multiplication of any number continuously halved until 1 or less""
    temp = number / 2
    if ((temp == 1) or (temp == 0)):
        return 1
    else:
        return temp * division_mult(temp)
```

1. Write down a **tail recursive** implementation of the function **sq_sum** in python language or programming language of your choice. You may use the helper function in your solution.
2. Write down a **tail recursive** implementation of the function **division_mult** in python language or programming language of your choice. You may use the helper function in your solution.

Note: There are different ways to do achieve this.

Solution

1. Note that there are multiple ways to craft the solution for this problem. Focus on the return statement and ensure it is tail-recursive.
   Here, we introduce a new helper function that is called within sq_sum, this helper has a value variable which keeps track of the current total as it moves along the function. This way, the call doesn't have to wait until the other function is recursively done before returning.

```
def sq_sum(lst):
    """Sums a list of numbers squared."""
    return helper(lst, 0)

def helper(lst, value):
    if len(lst) == 0:
        return value
    temp = (lst[0] * list[0]) + value
    return helper(lst[1: ], temp)
```

2. Note that there are multiple ways to craft the solution for this problem. Focus on the return statement and ensure it is tail-recursive.
   Here, we introduce a new helper function that is called within division_mult, this helper has a value variable which keeps track of the current total as it moves along the function. This way, the call doesn't have to wait until the other function is recursively done before returning.

```
def division_mult(number):
```

```
    """Returns the multiplication of any number continuously halved until 1 """
    return helper(number, 1)

def helper(number, result):
    temp = number / 2
    if ((temp == 0) or (temp == 1)):
        return result
    return helper(temp, temp*result)
```

Problem 7: Tail Recursion II

Consider the following Python program

```
1 def cube_sum(lst):
2     """Sums a list of numbers cubed."""
3     return cube(lst, 0)
4
5 def cube(lst, value):
6     if len(lst) == 0:
7         return value
8     temp = (lst[0] * list[0] * list[0]) + value
9     return cube(lst[1: ], temp)
```

1. Write down a **recursive** implementation of the function **cube_sum** in python language or programming language of your choice. You may use the helper function in your solution.

2. Draw the highest run time stack when the input is 1, 3, 9 for both the **tail recursive and recursive** implementation of the function **cube_sum**. No need to show links and assume global calls the function.

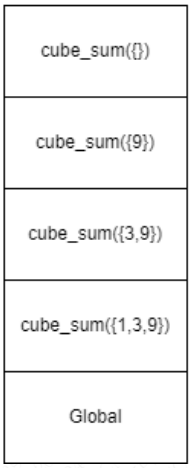
Solution

1. Here, it is very similar to the sq_sum question where it instead of introducing a new variable, we remove the tallying variable and force the function to return something then do the calculating.

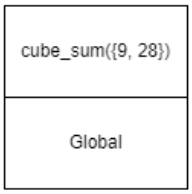
```
1 def cube_sum(lst):
2     """Sums the a list of numbers."""
3     if len(lst) == 0
4         return 0
5     return (lst[0] * lst[0] * lst[0]) + cube_sum(lst[1:])
```

2. See the figure below. This figure shows that tail-recursive reuses the same memory region as its old function stacks. Note that the function stacks are not the same but they reused the memory region.

Recursive



Tail-Recursive



Problem 8: Exception handling

Consider the following code snippet with exceptions. Note that the `main` function calls `foo` in the nested `try` block.

```
1  void foo () {
2      try {
3          throw new Exception1();
4          print ("Firefly");
5          try {
6              throw new Exception1();
7              print ("Dream");
8                  try {
9                      throw new Exception2();
10                     print ("Song");
11                 }
12         }
13
14     }
15     catch(Exception1 e1) {
16         print "handler one";
17     }
18     print ("Edge");
19     throw new Exception2();
20 }
21 void main () {
22     try {
23         print ("Sam");
24         try {
25             print ("Meme");
26             foo();
27         }
28         catch(Exception1 e1) { print "handler two"; }
29         print ("Robin");
30     }
31     catch(Exception2 e2) { print "handler three"; }
32 }
```

a) (7 pt) Write down what will be the output of the following program and briefly justify your answer.

b) (7 pt) Instead of the "replacement" semantics of exception handling used in modern languages (i.e., the semantics introduced in lecture), a very early design of exception handling introduced in the PL/I language uses a "binding" semantics. In particular, the design dynamically tracks a sequence of "catch" blocks that are currently active; a catch block is active whenever the corresponding try block is active. Moreover, whenever an exception is thrown, the sequence of active "catch" blocks will be traversed (in a first-in-last-out manner) to find a matching handler. Furthermore, execution will resume at the statement following the one that throws exception, rather than the next statement after the matching "catch" block as we have seen in the "replacement" semantics.
What will be the output of the following program if the language uses the "binding" semantics? Briefly justify your answer.

Solution

1. (a) In `main()`, ”Sam” is printed.
   (b) ”Meme” is printed.
   (c) `foo()` is called, which throws `Exception1`.
   (d) The `catch(Exception1 e1)` block in `foo()` catches it, prints ”handler one”, and ”Edge”.
   (e) The `catch(Exception2 e2)` block in `main()` catches `Exception2` thrown by `foo()`, and prints ”handler three”.

   It will print out:
   Sam Meme handler one Edge handler three
   Here treat it as it finds the nearest catch for that exception type then jump and continue from that catch block onward.

2. It will print out:
   Sam Meme handler one Firefly handler one Dream handler three Song Edge handler three Robin
   Here treat it as it finds the nearest catch for that exception type then jump to it, execute the block, then jump back to where the exception was thrown and continue from there

Problem 9: Calling Sequence

enumerate

What are function parameters? Briefly explain with an example.

What are function arguments? Briefly explain with an example.

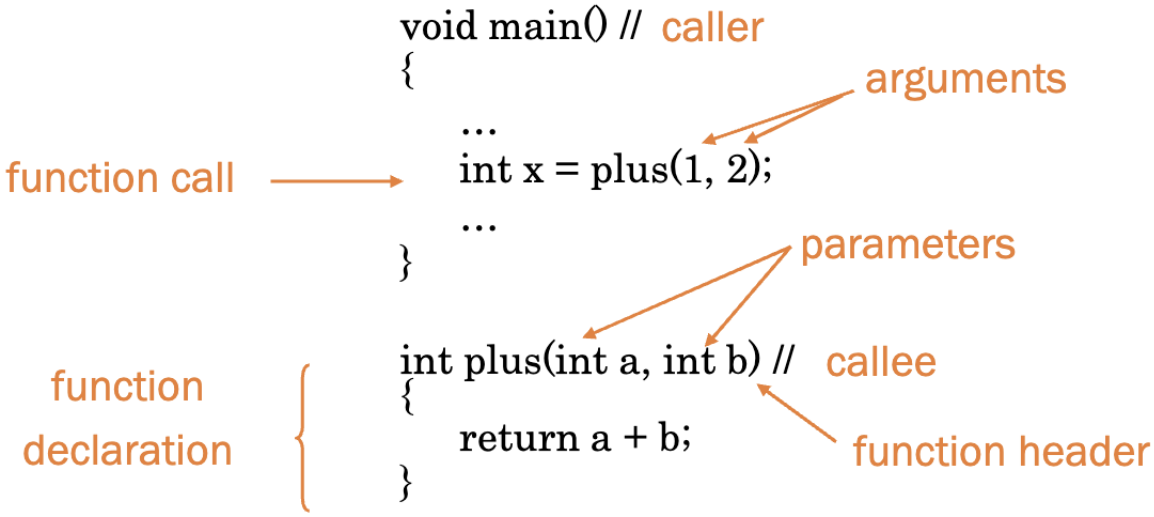
Name the five parameter passing mechanisms.

When passing large arrays/objects, which passing mechanism is the most efficient and why?

What is one downside of always using pass-by-reference?

Solution

1. Parameters are names in the declaration of a function header.



2. Arguments are variables/expressions passed to a function during a function call.
3. Call by value, call by result, call by value-result, call by reference, and call by name.
4. Pass-by-reference - this avoids copying entire array/objects.
5. Possible to accidentally change the value of the referenced parameter.

Problem 10: Calling Sequence

enumerate

How are values passed in call by value (CBV)?

How are values passed in call by result (CBR)?

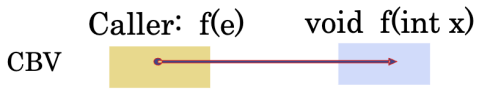
How are values passed in call by value-result (CBVR)?

How are values passed in call by reference (CBR - Reference)?

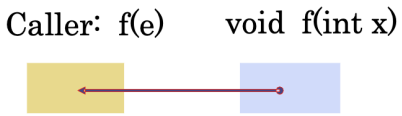
How are values passed in call by value (CBN)?

Solution

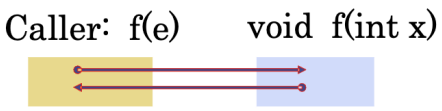
1. In CBV, argument's value is copied into the parameter. Calle cannot modify argument's value as it is a copy.



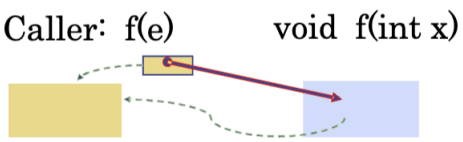
2. In CBR, parameter's computed value is copied back to argument after end of call.



3. CBVR is a combination of CBV and CBR. It first copies the argument value to the parameter then copies back the computed one back to the argument at the end of call.



4. In CBR (reference), the address of the argument is assigned to the parameter. During execution, the argument and paramter the same.



5. In CBN, it substitutes the argument for each occurrence of the parameter in the function without computing the value first.

Problem 11: Exception Handling/Recursion Theory

Briefly explain the following:

1. What is the difference between pre-defined exceptions and user-defined exceptions?
2. List at least two examples of pre-defined exceptions.
3. What happens when a exception is not handled in a function call?
4. Why are local variables in recrusive functions dynamically allocated?
5. What are some benefits of using tail-recursion over traditional recursion?

Solution

1. Pre-defined exceptions are built into the programming language and handle common errors. User-defined exceptions, on the other hand, are custom errors that programmers create to handle specific problems in their own applications.
2. Out of memory, divide by zero and file not found.
3. If the calling function doesn't have a chance to catch and handle the exception, it keeps moving up the chain. If no function along the way handles it, the program will eventually crash, and you'll usually see an error message or stack trace explaining what went wrong.
4. In recursion, variables are dynamically allocated because each function call creates a new instance of the variables. This ensures that each call has its own separate set of variables, allowing the function to work correctly even when called multiple times in a nested manner. Also we are not aware of how many recursive calls are going to be made until run time.
5. Tail-recursion optimizes memory usage and execution speed by reusing the same stack frame for each recursive call, reducing the risk of stack overflow and improving performance.

Problem 12: Exception Handling/Recursion Theory

Answer True or False for the following statements and provide a brief explanation to each question:

1. The order in which catch blocks are listed is not important.
2. If an exception is thrown in a try block, the remaining statements in that try block are executed after executing a catch block.
3. You can have multiple catch blocks for a single try block.

Solution

1. False, since if the catch block has three ellipses declared first, then it will catch all the exceptions and the rest of the catch blocks are ignored.
2. False, following the replacement semantics as mentioned in class lectures, the remaining statements are skipped until a relevant catch block is found and the execution of program continues from after the catch block.
3. True, a try block can have multiple catch blocks, and this is useful when the try block may throw more than one type of exception. Ideally, there shouldn't be any intervening code between the end of the try block and the catch blocks since standard exception handling techniques follow replacement semantics..