Renee Anderson

COSC A361 - Languages and Paradigms

Chapter 9 Assignment

Part 1: Subprograms (15 points)

A subprogram is a small, self-contained section of code that does one specific job. It's like giving a program a mini task to handle, so you don't have to repeat the same code repeatedly. Subprograms make code cleaner, easier to fix, and easier to understand.

Procedures vs. Functions:

A procedure doesn't return a value it just performs an action. A function, on the other hand, gives back a result when it's done.

Example (Python):

```
def greet(name):  # procedure
print(f"Hello, {name}!")

def add(x, y):  # function
   return x + y
```

Advantages: Subprograms save time by letting you reuse code, making programs easier to organize, and helping find and fix bugs faster.

Disadvantages: Too many subprograms can make code harder to track, especially if one change affects others. Also, calling a subprogram adds a small bit of overhead to the program's run time.

Part 2: Parameter Passing (20 points)

There are a few ways to pass data to a subprogram-

- Pass by Value: A copy of the variable is passed. Changes inside the function don't affect the original.
- Pass by Reference: The actual variable is passed. Changes inside the function affect the original.
- Pass by Name: The expression is substituted directly this one isn't common in most modern languages.

Example (C++):

```
// Pass by value
void changeValue(int x) {
   x = 20;
}

// Pass by reference
void changeRef(int &x) {
   x = 20;
}
```

If you call these like this:

```
int num = 10;
changeValue(num); // num stays 10
changeRef(num);  // num becomes 20
```

When to use which:

- Use pass by value for small data when you don't need to change it.
- Use pass by reference when working with large data or when the function needs to modify the original value.

Part 3: Overloaded Subprograms (15 points)

Function overloading means having multiple functions with the same name but different parameters (like data type or number of inputs). It helps make code cleaner and easier to read.

Example (C++):

```
int multiply(int a, int b) { return a * b; }
float multiply(float a, float b) { return a * b; }
```

Benefits: Makes code easier to understand and lets you use one name for similar tasks.

Challenges: Can cause confusion if parameters are too similar or if default values make it unclear which version is being called.

Part 4: Generic Functions (15 points)

A generic function is a flexible function that works with any data type. You don't have to write separate versions for int, float, or string. These are often called templates in C++.

Example (C++):

```
template <typename T>
T getMax(T a, T b) {
  return (a > b) ? a : b;
}
```

Generic vs. Regular:

Regular functions only work for one data type. Generic ones work for many, making code more reusable and cleaner.

Part 5: Functions as Parameters (20 points)

Some functions can take other functions as inputs. This helps make programs more flexible and modular.

Example (Python):

```
def square(x):
  return x * x

def apply_function(func, number):
  return func(number)

print(apply_function(square, 5)) # Output: 25
```

Why it matters: This lets programs do dynamic things, like decide what function to use at runtime. It’s useful for callbacks, filters, and organizing code to avoid repetition.

Part 6: Coroutines (15 points)

A coroutine is like a function that can pause and resume its progress without losing its place. It’s useful when you need to handle multiple tasks without restarting from the beginning for example, in asynchronous programming.

Example (Python):

```
def counter():
  print("Starting coroutine")
  num = 0
  while True:
    received = yield num
    if received:
            num = received
        num += 1

co = counter()
print(next(co))      # Starts coroutine, prints 0
print(co.send(10))   # Sends new value, prints 11
```