Programming Language Concepts

Programming Language Theory

Topics

- Control Structure
  - Expressions and Their Evaluation
  - Statement
  - Control Flow and Recursion
- **Control Abstraction**
  - **Subprogram**

Control Abstraction

- One of the two most important concepts of programming language, along with Data Abstraction.
- For a complex, large software, the main goal or requirement can be achieved,
  - by satisfying many smaller requirements.
  - Concept of Divide and Conquer.

Control Abstraction

- Suppose we're developing a mobile shopping app.
  - We need the following functionalities.
    - Read product data.
    - Display products in the app.
    - Search for products.
    - Manage Customer information.
    - Product Reviews
    - Payment.
    - Manage Purchases.
    - Manage Delivery options.

Control Abstraction

- It is not a good idea to implement all these things in one big program.

- Instead, we can implement ***subprograms*** provide each function.

- We can ***hide implementation details*** and separate them from design.

  - Easy to switch different implementation.

- We only need to know ***how to use*** such subprograms.

Subprogram

- Also called ***procedure*** or ***function***.
- As we talked about before, we will use subprogram, procedure, and function together as synonyms.
- Although *subprogram* is the most general term, let's use function for this lecture.
    - Since it is more familiar, and more similar to what we will describe later in this lecture.

Function

- A function is a piece of code,
  - identified by its *name*,
  - is given a *local environment of its own*,
  - and exchanges information with the other parts of code using *parameters, return value and non-local environment.*
- We can ***define (or declare)*** a function and ***use (or call)*** the function.

Parameter Passing

- Parameter passing is one of the important ways for a subprogram to communicate with the other parts of a program.

- We can consider three kinds of parameters in subprogram's viewpoint.

  - IN parameters: communicate *from the caller to callee*.

  - OUT parameters: *from the callee to the caller*.

  - IN/OUT parameters: *bidirectional* communication.

Parameter Passing

- There are various *parameter passing modes*, which we will discuss in this lecture.
- For further explanation, we need to consider two terms.
  - ***Formal parameters***: these parameters are the ones appeared in the declaration of a function.
  - ***Actual parameters***: also known as ***arguments***. These are the ones passed to a function in a function call.

Call by Value

- It is a mode which corresponds to an IN parameter.
- Actual parameters can be expressions.
  - At the time of a call, actual parameters are evaluated and their ***r-values*** are associated with formal parameters.
  - When the function terminates, formal parameters are destroyed, hence these values are lost.
- Call by value (or pass by value) is the most simple and popular mode.
  - It is the only parameter passing mode in C and Java.

Call by Reference

- In this mode, parameters can be used both input and output (IN/OUT).
- Actual parameters must be expressions with ***l-values***.
- When a function is called, ***actual parameters' l-values*** are associated with formal parameters - ***aliasing***.
- In C++
  - `void func(MyObject& obj);`
  - `MyObject x; func(x);`
- Java has no call by reference: its variable model is reference model, but parameter passing itself is not call by reference.

Call by Reference

- Java has no call by reference: its variable model is reference model, but parameter passing itself is call by value only.
  - class A { int v; } swap(x, y);
  - void swap (A a1, A a2) {
    int tmp = a1.v; a1.v = a2.v; a2.v = tmp; //simulate call by ref.
    A tmp2 = a1; a1 = a2; a2 = tmp2; //What happens?
    }
- Two references of objects in type A are copied and passed to swap().
- You can change the value of their member variables using reference model (a1.v).
- But you can't actually swap two variables x and y.

Call by Constant

- In case of a parameter is not modified in the body of a function,
  - Maintain semantics of call by value, while implementing it using call by reference.
- Parameters are considered ***read-only***.
- In C++
  - `void func(const MyObject& obj);`

Call by Result

- It is a mode implements ***output-only*** communication.
- Actual parameters must be ***expressions with l-values***.
- ***Backward assignment***: the values of formal parameters are *copied to locations obtained using actual parameters' l-values* after a function terminates.
- ```
  void foo(int x) { x = 5; }
  int y = 2;
  foo(y);  //y = 5 after this call.
  ```
  ```

Call by Value-Result

- Combination of Call by Value and Call by Result.
- Implement ***bidirectional*** communication.
- Actual parameters must give ***l-values***.
- At the call, r-values of actual parameters are assigned to formal parameters (Call by Value).
- Then value of formal parameters are copied backward using l-values of actual parameters (Call by Result).
  - ```
    void foo(int x) { x = x + 1; }
    int y = 2;
    foo(y);  //y = 3 after this call.
    ```
    ```

Call by Value-Result

- Difference to call by reference.
  - void foo(int x, int y) {
    x = 3;
    y = 4;
    if(x == y) y = 1; //x == y → true in call by ref.
    }

- int a = 2;
  foo(a, a); //a is 4 after the call.

Call by Name

- This mode is no longer used by modern programming languages.

- Although it is conceptually important, we will not discuss details of the method in this lecture.

- Simply speaking, it is a method to replace formal parameter names in the function body with actual parameter names.

  - ```
    void foo(int x) { x = 1; }
    int y = 2;
    foo(y); → void foo(int y) { y = 1; }
    ```
    ```

Call by Name

- However, simply replacement may cause a problem.

- ```
  int x = 0;
  int foo(int y) {
    int x = 1;
    return x + y;
  }
  int a = foo(x + 1);
  ```

  - a = x + x + 1 = 3

- ```
  int x = 0;
  int foo(int y) {
    int z = 1;
    return z + y;
  }
  int a = foo(x + 1);
  ```

  - a = z + x + 1 = 2

- Hence it is necessary to ***pass actual parameters as well as their evaluation environment***.

Higher-Order Functions

- A function is considered ***higher order***, if
  - it accepts functions as parameters,
  - or it returns a function.
- This mechanism is supported by many programming languages, especially in *functional* languages.

Functions as Parameters

- On the right, there is an example C code using a function as a parameter.
- Function f is passed as a parameter to g.
- Variable x is defined multiple times.
- In function g, which binding of x should be used?
  - Which environment should be checked for name x?

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

Deep and Shallow Binding

- ***Deep Binding***: use the environment active when the ***link between f and h*** **are made.**

- ***Shallow Binding***: use the environment active when the ***call to f (using h) occurs***.

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

Deep and Shallow Binding

- **Static Scope + Deep Binding**
  - h(3) = 4, g(f) = 6
- **Dynamic Scope + Deep Binding**
  - h(3) = 7, g(f) = 9
- **Dynamic Scope + Shallow Binding**
  - h(3) = 5, g(f) = 7

```
int x = 1;                    Static + Deep
int f(int y) {
    return x+y; x = 1
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

Deep and Shallow Binding

- **Static Scope + Deep Binding**
  - h(3) = 4, g(f) = 6
- **Dynamic Scope + Deep Binding**
  - h(3) = 7, g(f) = 9
- **Dynamic Scope + Shallow Binding**
  - h(3) = 5, g(f) = 7

Dynamic + Deep

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x;
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f); x = 4
```

Deep and Shallow Binding

- **Static Scope + Deep Binding**
  - h(3) = 4, g(f) = 6
- **Dynamic Scope + Deep Binding**
  - h(3) = 7, g(f) = 9
- **Dynamic Scope + Shallow Binding**
  - h(3) = 5, g(f) = 7

Dynamic + Shallow

```
int x = 1;
int f(int y) {
    return x+y;
}
int g(function<int(int)> h) {
    int x = 2;
    return h(3)+x; x = 2
}

int main(){
    //Functions as parameters
    int x = 4;
    int z = g(f);
```

What defines the Environment?

- Visibility Rules.
- Exceptions in Visibility Rules - need to consider re-defined names, use names after declaration.
- Scope Rules.
- Parameter Passing Modes.
- Binding Policy.

Functions as Results

- Function can return another function as a result.
- With static scope, call `k()` is actually call `s`, and x = 1 in `s`.
  - Environment is also considered in the returned function.
- Hence the result of `calling_s()` is actually a ***closure***.

```
int x = 1;
int s() {
    return x+1;
}
function<int()> calling_s() {
    return s;
}

int main(){
    //Functions as results
    int x = 4;
    function<int()> k = calling_s();
    int y = k();
```

Closure

- A ***Closure*** is a pair of (expression, environment),
  - which the environment includes all the free variables in the expression.
- ***Free variables*** are the variables used in the expression, but not declared in the environment.
  - In Python, they have a special distinction, so that global variables are not free variables.
  - But usually, global variables are free variables, unless they are declared again in the local environment.

Summary

- Control Abstraction and Subprogram
- Parameter Passing
- Higher-Order Functions