Subprogram Concept

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Introduction

Tow fundamental abstractions

- process abstraction and
- data abstraction.

What is *process abstraction*?

```
int swap(int& a, int& b) {
  int c = a;
  a = b;
  b = c;
}
```

Why process abstraction?

- Memory space efficiency,
- Less development and maintenance time,
- Increase readability, and etc.

General Characteristics

- Each subprogram has a single entry point.
- The caller is suspended during the execution of the callee.
- Control always returns to the caller when callee finishes.

General Characteristics

- Each subprogram has a single entry point.
- The caller is suspended during the execution of the callee.
- Control always returns to the caller when callee finishes.

Really??

General Characteristics

- Each subprogram has a single entry point.
- The caller is suspended during the execution of the callee.
- Control always returns to the caller when callee finishes.

Really??
Not really!!

- Coroutine has multiple entry points.
- Multi-threaded functions run in parallel.
- Callee might abort.

Terminologies

```c
int fun_name(int a0, float b0, char c0);

int main() {
  int a1; float b1; char c1;
  fun_name(a1, b1, c1);
  // why a1 corresponds to a0???
}
```

- a0, b0 and c0 are *formal parameters*, or *parameters*.
- a1, b1 and c1 are *actual parameters*, or *arguments*.
- The collection of number, order and types of formal parameters is called *parameter profile*.
- Parameter profile plus the return type is called *protocol*.

Positional Parameter

Positional Parameter In nearly all PL, the correspondence between actual and formal parameters is done by *position*.

```
int func(int v1, float v2, char v3) { }

int main() {
  func(1, 2.3, 'c');
}
```

This is convenient as long as parameter list is *short*.

Keyword Parameter

Keyword Parameter The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call.

```
def func(v1, v2, v3):
    pass

func(v1 = 1, v2 = 2.3, v3 = 'c')
```

The Boost Parameter Library (BPL) supports keyword in C++.

Mixed Usage

Some PL's support both positional and keyword parameters, e.g., `Python`, `R`, and etc.

```
func <- function(v1, v2, v3) v1 + v2 + v3
func(1, v2 = 3, v3 = 4)
```

The only *restriction* is that after a keyword parameter appears in the list, all remaining parameters *must* be keywords. Why?

Mixed Usage

Some PL's support both positional and keyword parameters, e.g., `Python`, `R`, and etc.

```
func <- function(v1, v2, v3) v1 + v2 + v3
func(1, v2 = 3, v3 = 4)
```

The only *restriction* is that after a keyword parameter appears in the list, all remaining parameters *must* be keywords. Why?

```
func(v3 = 4, v1 = 3, v2 = 3)
func(v3 = 4, v3 = 3, v1 = 3, v2 = 3)     # ERROR
```

Default Value

Some PL's allow parameters to have *default values*, e.g., `Python`, `C++`, `R` and etc.

```
def func(v1, v2, v3 = 0):
    print(v1 * 100 + v2 * 10 + v3)

func(1, 2)
func(1, 2, 3)
```

Some allow mixing parameters with and without default values.

```
func <- function(v1, v2 = 3, v3) v1 + v2 + v3
func(1, v3 = 4)
func(v3 = 4, v1 = 3, v2 = 3)
```

Default Value Cont'd

Javascript, allows parameters to be `undefined`.

```
function foo(v1, v2, v3) {
  v1 = typeof v1 === 'undefined' ? 0 : v1;
  v2 = typeof v2 === 'undefined' ? 0 : v2;
  v3 = typeof v3 === 'undefined' ? 0 : v3;
  console.log(v1 * 100 + v2 * 10 + v3);
}

foo();
foo(1);
foo(1, 2);
foo(1, 2, 3);
```

Variable Number of Parameters

Variadic functions Functions which take a variable number of arguments, e.g., `std::printf` in C++. Usually the parameter is denoted by an ellipsis (...).

```
#include <iostream>
#include <cstdarg>
int add_nums(int count, ...) {
  int result = 0;
  va_list args;
  va_start(args, count);
  for (int i = 0; i < count; ++i)
    result += va_arg(args, int);
  va_end(args);
  return result;
}
int main() {
  std::cout << add_nums(4, 25, 25, 50, 50) << '\n';
}
```

Packed Parameters

In `Python`, we can *pack* and *unpack* positional and keyword arguments.

```
def func(v1, v2):
    print(10 * v1 + v2)

args = (1, 2)                   # a tuple
func(*args)                     # -> func(1, 2)

kwargs = {'v1': 1, 'v2': 2}     # a dict
func(**kwargs)                  # -> func(v1=1, v2=2)
```

Design Issues

1. Are side effects allowed?
2. What types of values can be returned?
3. How many values can be returned?

Generic Subprograms

Why?

Generic Subprograms

Why? Programmers are lazy.

Ad hoc polymorphism Overloaded subprograms need not behave similarly. The only bond is the name.

Subtype polymorphism A variable of type T can access any object of type T or any type derived from T.

Parametric polymorphism A subprogram that takes generic parameters that are used in type expressions that describe the types of the parameters of the subprogram. E.g., `C++` template.

More general polymorphism as provided by script languages.

Outline

Definition

Closure A *subprogram* and the *referencing environment* where it was defined. The referencing environment is needed if the subprogram is called from any arbitrary place in the program.

Unlimited Extend Variable Lifetime is that of the whole program which usually means they must be heap-dynamic, rather than stack-dynamic.

Closure in C++

```
#include <functional>
#include <cstdio>
auto func(int b) {
  int a = 100;
  int bb = b;
  return f = [&a, b](int c) { a += 100; return a + b + c; };
}
int main() {
  auto f = func(20);
  printf("%d\n", f(3));
  printf("%d\n----------------\n", f(3));
  auto f2 = func(40);
  printf("%d\n", f2(3));
  printf("%d\n----------------\n", f2(3));
  auto f3 = func(40);
  printf("%d\n", f3(3));
  printf("%d\n", f3(3));
}
```

Closure in Javascript

```
function genf (a) {
  var b = 30;

  return function(c) {
    return a + b + c;
  };
}

var f = genf(100);
console.log(f(1));
console.log(f(2));
```

E.g., Extensively used in D3.

Outline

Semantic Models

Formal parameters are characterized by one of three distinct semantics models:

In Mode
: They can receive data from the actual parameter.

Out Mode
: They can transmit data to the actual parameter.

Inout Mode
: They can do both.

Data Transfer

- Either an *actual value* is copied back and forth or

```
int foo(int v) { }
int main() {
  int a = 3;
  foo(a);
}
```

- or an *access path* is transmitted, e.g., pointers, references.

```
int foo(int& v) { }
int main() {
  int a = 3;
  foo(a);
}
```

Pass-by-Value

- Implements *in mode* semantics.
- The value of the actual parameter is used to initialize the corresponding formal parameter
- which then acts as a local variable in the subprogram

It may be implemented by *copy* or *write-protected access path*.

Pass-by-Result

- Implements *out mode* semantics.
- No value is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable and
- Its value is transmitted back to the actual parameter

Pass-by-Result Cont'd

```
function foo(a, b) {
  a = 0; b = 1;
}
```

- What if actual parameters are literals?

```
foo(3, 3)
```

- What if actual parameters collide?

```
var v = 0;
foo(v, v);
```

- When to evaluate the address of the actual parameter?

```
var v = [1, 2]; var p = 0;
foo(v[p], p);
```

Pass-by-Copy

AKA Pass-by-Value-Result.

- Implements *in-out mode* semantics.
- A combination of pass-by-value and pass-by-result.

Pass-by-Reference

Access path is passed to formal parameters.

- Access to the formal parameters is slower.
- Aliasing problem.

Pass-by-Name

Literally substitute formal parameters with actual parameters.

- Formal parameter is bound to an access method at the time of the subprogram call.
- Actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.

In C++, it is used at *compile time*, i.e., *macro* and *template*.

```
#define min(x, y) ((x) < (y) ? (x) : (y))
int main() {
  min(3, 4);
  min(3 + 4, 4 + 100);
}
```

Copy-on-Write (COW)

AKA *implicit sharing*, is an an optimization strategy.

```r
foo <- function(x) {
    ## now formal parameter and actual parameter uses the same buffer
    ## until
    x <- 3
    ## now they use different buffer
}
```

It is not restricted to parameter passing.

```cpp
#include <string>
int main() {
  std::string x{"Hello"};
  std::string y = x;  // y and x use the same buffer
  y += ", World!";    // now y uses a different buffer, x still uses
                      // the same old buffer
}
```

Outline

Summary

1. Actual and formal parameter
2. Closure
3. Parameter passing