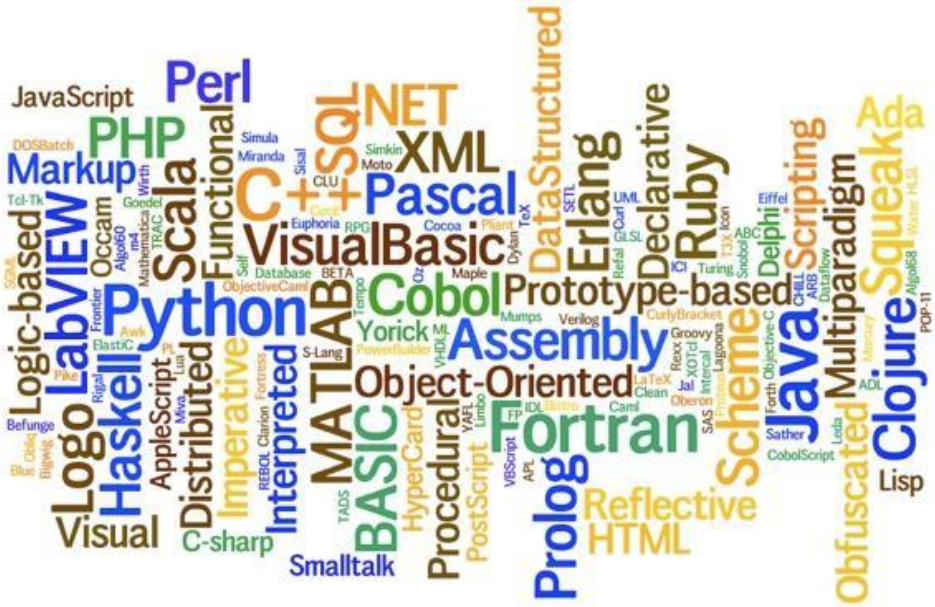

Programming Languages
Subprograms II

Programming Languages
Module 9 (Chapter 9 - PART I)

Dr. Tamer ABUHMED
College of Computing

SUNG KYUN KWAN UNIVERSITY

Topics to be covered

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Overloaded Subprograms
- Generic Subprograms
- Design Issues for Functions
- User-Defined Overloaded Operators
- Closures

Part 2

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

Multidimensional Arrays as Parameters: C and C++

- storage-mapping function for row major order for matrices:
  address(mat[i, j]) = address(mat[0,0]) + (i * number_of_columns) + j
- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter

```
void printArray1(int arr[][3], int m, int n)
{
int i, j;
for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      cout << arr[i][j] << "  ";
}
```

- Formal parm. `arr[][3]` disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

Multidimensional Arrays as Parameters: C and C++

```
void printArray2(int *arr, int m, int n)
   {
int i, j;
for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      cout<< *(arr + (i*n) + j)<<"  ";
   }


int main () {

int arr[][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
int m = 3, n = 3;
printArray1(arr, m, n);
printArray2((int *)arr, m, n);

return 0;
   }
```

Multidimensional Arrays as Parameters: Java

- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java) that is set to the length of the array when the array object is created

```
public static void printArray(int[][] array) {
  for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++)
                  System.out.println(array[i][j]);
   }
}
public static void main(String[] args) {
  int array[][] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9 } };
  printArray(array);
}
```

Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

Pass Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
- C++/C

```
void func(void(*f)(int)) {
  for (int ctr = 0; ctr < 5; ctr++) {
    (*f)(ctr);  }
  }
void print(int x) {
cout << x << endl;
}

int main () {

func(print);

return 0;
}
```

Pass Parameters that are Subprogram Names: Python

```
def sub1() :

    def sub2():
        print(x) # print the value of x

    def sub3():
        x = 3
        sub4(sub2);

    def sub4(subx) :  #Pass Parameters that are Subprogram
        x = 4
        subx();

    x = 1;
    sub3()

sub1()
```

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
- *Deep binding*: The environment of the definition of the passed subprogram
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

Consider the execution of sub2 when it is called in sub4.

- **For shallow binding**, the referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4.
- **For deep binding**, the referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x in sub1, and the output is 1.
- **For ad hoc binding**, the binding is to the local x in sub3, and the output is 3.

Calling Subprograms Indirectly

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not know until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers

```
void my_int_func1(int x)
{
cout << "my_int_func1"<<endl;
cout<< x<<endl;
  }
```

```
int main () {

void(*foo)(int);
foo = &my_int_func;

// call way1
foo(2);
/* call way 2
(*foo)(2);
  return 0;
  }
```

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

How about this overloading at C++?

```
void sum(float a,  float b);
float sum(float a,  float b);
```

cannot overload functions distinguished by return type alone

How about this overloading at C++?

```
void fun(float b = 0.0);
void fun();
```

'fun' : ambiguous call to overloaded function

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  - A cheap compile-time substitute for dynamic binding

Generic Subprograms (continued)

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a **`template`** clause that lists the generic variables, which can be type names or class names

```
template <class Type>
  Type max(Type first, Type second) {
  return first > second ? first : second;
  }
```

Generic Subprograms (continued)

- Java 5.0

  - Differences between generics in Java 5.0 and those of C++:

  1. Generic parameters in Java 5.0 must be classes
  2. Java 5.0 generic methods are instantiated just once as truly generic methods
  3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
  4. Wildcard types of generic parameters

Generic Subprograms (continued)

- Java 5.0 (continued)

```
public static <T> T doIt(T[] list) { … }
```

- The parameter is an array of generic elements (`T` is the name of the type)
- A call:

```
doIt<String>(myList);
```

Generic parameters can have bounds:

```
public static <T extends Comparable> T
  doIt(T[] list) { … }
```

The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms (continued)

- Java 5.0 (continued)
  - Wildcard types

`Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
    for (Object e: c) {
      System.out.println(e);
    }
  }
```

- Works for any collection class

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class
  - Lua allows functions to return multiple values

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute `x + y , x.__add__(y)`

- C++ example

```
Complex operator+(Complex  &num1, Complex &num2)
{
    double result_real = num1.real + num2.real;
    double result_imaginary = num1.imag + num2.imag;
    return Complex( result_real, result_imaginary );
}
```

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn't need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

Closures (Python)

```
def f(x):
    def g(y):
        return x + y
    return g

def h(x):
    return lambda y: x + y

a = f(1)
b = h(1)
print(a(5))       # 6
print(b(5))       # 6
print (f(1)(5))   # 6
print(h(1)(5))    # 6
```

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A closure is a subprogram and its ref. environment