Chapter 9 Subprograms

A piece of program that can be executed from various places in a program.

Subprogram is useful:

1. It eliminates the need to replicate its code everywhere that code needs to be executed.
2. It serves as a process abstraction: the programmer can make use of its operation without being further concerned with how that operation is implemented.

There are three types of subprograms:

1. Procedures
2. Functions
3. Methods

- Parameter-Passing Methods
- Parameters That Are Subprogram Names
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators

Formal Parameter Default Values

In certain languages (e.g., C++, Ada), formal parameters can have default values (if not actual parameter is passed)

```
#include <iostream.h>
void Func( int one, int two=2, int three=3);
main ()    {
    Func(10, 20, 30);
    Func(10, 20);   // Let the last parm default
    Func(10);       // Just provide the required parm.
}
void Func( int one, int two, int three)    {
    cout << "One   = " << one   << endl;
    cout << "Two   = " << two   << endl;
    cout << "Three = " << three << endl;
}
```

Basic rules

- When a default is provided, all remaining parameters must be given defaults.
- Default values must be of the correct type.

Java and C# methods can accept a variable number of parameters as long as they are of the same type.

Variable Length Parameter Lists

• Suppose we wanted to create a method that processed a different amount of data from one invocation to the next.

Example: let's define a method called **average** that returns the average of a set of integer parameters

```
// one call to average three values
mean1 = average (42, 69, 37);

// another call to average seven values
mean2 = average (35, 43, 93, 23, 40, 21, 75);
```

Downside: we'd need a separate version of the method for each parameter count

Java provides a convenient way to create *variable length parameter lists*

• Using special syntax in the formal parameter list, we can define a method to accept any number of parameters of the same type

Indicates a variable length parameter

```
public double average (int ... list)
{
  // whatever
}
```

element type

array name

```
public double average (int ... list){
   double result = 0.0;
   if (list.length != 0)   {
      int sum = 0;
      for (int num : list)
         sum += num;
      result = (double)sum / list.length;}

   return result;}
```

- The type of the parameter can be any primitive or object type

```
public void printGrades (Grade ... grades)
{
   for (Grade letterGrade : grades)
      System.out.println (letterGrade);
}
```

- A method that accepts a variable number of parameters can also accept other parameters

```
public void test (int count, String name, double ... nums)
{
   // whatever
}
```

- The varying number of parameters must come last in the formal arguments

- C# allows methods to accept a variable number of parameters, as long as they are of the same type.
- In C#, the mechanism for specifying that a method accepts a variable number of arguments is by using the *params* keyword as a qualifier to the last argument to the method which should be an array.

```
class ParamsTest{

public static void PrintInts(string title, params int[] args){

    Console.WriteLine(title + ":");

    foreach(int num in args)
        Console.WriteLine(num);

    }

public static void Main(string[] args){
    PrintInts("First Ten Numbers in Fibonacci Sequence", 0, 1,
1, 2, 3, 5, 8, 13, 21, 34);
    }
}
```

Design Issues for Subprograms

- What parameter passing methods are provided?
- Are parameter types checked?
- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- Can subprograms be overloaded?
- Can subprogram be generic?

Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms

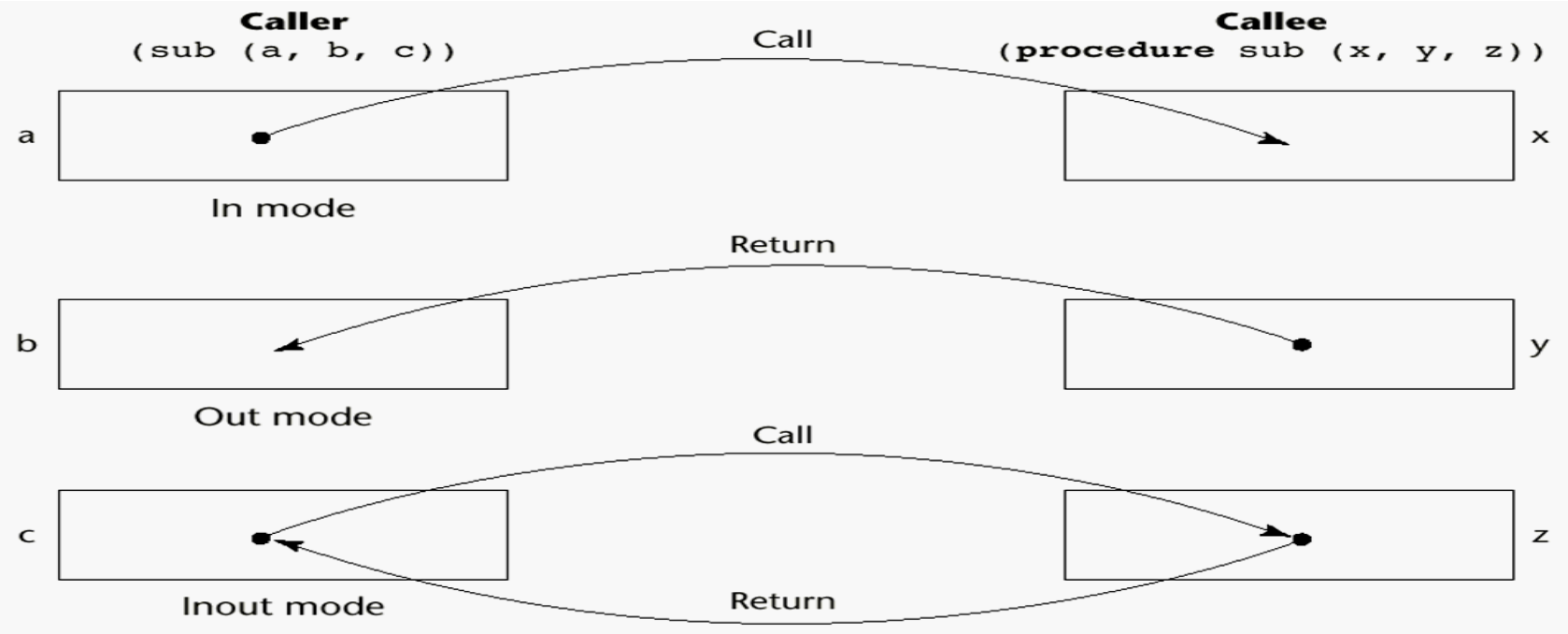
Formal parameters are characterized by three distinct models:

(1) in mode (Pass-by-value): They receive data from the corresponding actual parameter;

(2) out mode (Pass-by-result): They can transmit data to the actual parameter;

(3) inout mode (Pass-by-reference): they can do both (1) and (2).

Models of Parameter Passing



Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying

This mechanism implemented in C/C++/Java/C#.

```
void test ( int );
int my_num = 5;
test(my_num);  //value of actual parameter is passed in


test(int local) {
  result = 100/local; }
```

- When copies are used, additional storage is required
- Storage and copy operations can be costly

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller

```
int my_num;
test(&my_num); //the actual parameter has no value when passed in
printf("\n num: %d ", my_num);    //  ???
```

```
void test ( int * num ){
   int local;     //requires a local variable
   local = 20;
   *num = local;
}
```

Pass-by-Reference (Inout Mode)

- Pass an access path, usually just an address to the called subprogram.

```
int num = 5;
test(&num); //a value holding an address to actual parameter is assed in
printf("\n num: %d ", num);    //  ???
```

```
void test ( int * num ){   //the only local copy is a pointer
      *num += 100; //changes are direct ; nothing is passed out
}
```

- Advantages

– Passing process is efficient (no copying)

- Disadvantages

– Aliases will be created.

```
#include <stdio.h>
#include <stdlib.h>

void stuff( int *);

int main() {
  int i = 6;
  int * iptr = &i;
  printf("\n const i: %d ",i);
  stuff(iptr);
  printf("\n const i: %d \n",i);
  return 0;
}

void stuff ( int * px){
  * px = 5;
}

/* output:
const i: ?
const i: ?
*/
```

Pointer and reference:

With call-by-reference in C++ you cannot change the location to which a reference refers.
No way to re-assign a reference; no reference arithmetic.

Parameter Passing Methods of Major Languages

C
Everything is pass-by-value
Pass-by-reference is "simulated" by using pointers as parameters

It's sometimes said that C uses pass-by-value except for arrays, which are passed by reference. But that's not true.

Array-valued variables are actually pointer variables that point to the first element of the array.

C++
A special pointer type called reference type for pass-by-reference

Java
Passed by value
Passed by reference

Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal, FORTRAN 90, Java, and Ada: it is always required

- ANSI C and C++: choice is made by the user.

In C89, the formal parameters of function can be defined in two ways:

(1) They can be as original C, the name of the parameters are listed in parenthesis and the type declaration for them follow.

```
double sin(x)        ///no type check
    double x;
    { …… }
```

(2) prototype method:

The formal parameter type are included in the list.

```
double sin(double x)       //type checked
{   ……}
```

```
value = sin(count);  //count is int type
```

It is also legal. The type is checked and **int** is coerced to **double**.

Thus, in C89, the user chooses whether parameter are to be type checked.

In C99 and C++, all functions must have their formal parameters in prototype form. However, type checking can be avoided for some of the parameters by replacing the last part of the parameter list with an ellipsis,

```
int printf(const char* format_string, …);
```

A call to printf must include at least one parameter. Beyond that, anything is legal.