CMPS 3500

Programming Languages

Dr. Chengwei Lei
CEECS
California State University, Bakersfield
Chapter 9

Subprograms
Type Checking Parameters

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible
Multidimensional Arrays as Parameters

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function.
Multidimensional Arrays as Parameters: C and C++

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- 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: Java and C#

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (length in Java, Length in C#) that is set to the length of the array when the array object is created
#include <stdio.h>
void display(int age)
{
    printf("%d", age);
}

int main()
{
    int ageArray[] = { 2, 3, 4 };  //Passing array element ageArray[2] only.
    display(ageArray[2]);
    return 0;
}
```c
#include <stdio.h>

float average(float age[]);

int main()
{
    float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18};
    avg = average(age); // Only name of an array is passed as an argument
    printf("Average age = %.2f", avg);
    return 0;
}

float average(float age[])
{
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < 6; ++i) {
        sum += age[i];
    }
    avg = (sum / 6);
    return avg;
}
```
#include <stdio.h>

void displayNumbers(int num[2][2]);

int main()
{
    int num[2][2], i, j;
    printf("Enter 4 numbers:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);
    // passing multi-dimensional array to displayNumbers function
    displayNumbers(num);
    return 0;
}

void displayNumbers(int num[2][2])
{
    // Instead of the above line,
    // void displayNumbers(int num[][2]) is also valid
    int i, j;
    printf("Displaying:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            printf("%d\n", num[i][j]);
}
#include <stdio.h>

void myfuncn( int *var1, int var2)
{
  for(int x=0; x<var2; x++)
  {
    printf("Value of var_arr[%d] is: %d \n", x, *var1);
    /*increment pointer for next element fetch*/
    var1++;
  }
}

int main()
{
  int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
  myfuncn(var_arr, 7);
  return 0;
}
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
Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters

- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?
Parameters that are Subprogram Names: Referencing Environment

- **Shallow binding**: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- **Deep binding**: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- **Ad hoc binding**: The environment of the call statement that passed the subprogram
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 known until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers
In C#, method pointers are implemented as objects called delegates

- A delegate declaration:

```csharp
public delegate int Change(int x);
```

- This delegate type, named Change, can be instantiated with any method that takes an `int` parameter and returns an `int` value

A method: `static int fun1(int x) { ... }

Instantiate: `Change chgfun1 = new Change(fun1);

Can be called with: `chgfun1(12);

- A delegate can store more than one address, which is called a multicast delegate
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
  - 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
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
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

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

  ```java
  public static <T> T doIt(T[] list) { ... }
  
  - The parameter is an array of generic elements (T is the name of the type)
  - A call:
    ```java
doIt<String>(myList);
    ```
  
  Generic parameters can have bounds:

  ```java
  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
    ```java
    void printCollection(Collection<?> c) {
        for (Object e: c) {
            System.out.println(e);
        }
    }
    ```
    - Works for any collection class
C# 2005
- Supports generic methods that are similar to those of Java 5.0
- One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
  - Another – C# 2005 does not support wildcards
Generic Subprograms (continued)

- F#

- Infers a generic type if it cannot determine the type of a parameter or the return type of a function – automatic generalization

- Such types are denoted with an apostrophe and a single letter, e.g., ‘a

- Functions can be defined to have generic parameters

```fsharp
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

- `%A` is a format code for any type

- These parameters are not type constrained
Generic Subprograms (continued)

- F# (continued)
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+
Operators can be overloaded in Ada, C++, Python, and Ruby

A Python example

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

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

- A JavaScript closure:

```javascript
function makeAdder(x) {
    return function(y) { return x + y; }
}
...

var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) + "<br />
   
   add 5 to 20: " + add5(20) + "<br />
   
- The closure is the anonymous function returned by makeAdder
```
Closures (continued)

- C#
  - We can write the same closure in C# using a nested anonymous delegate
  - `Func< int, int>` (the return type) specifies a delegate that takes an `int` as a parameter and returns an `int`

```csharp
static Func< int, int> makeAdder( int x) {
    return delegate( int y) { return x + y;};
}
...
Func< int, int> Add10 = makeAdder(10);
Func< int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```
Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called symmetric control: caller and called coroutines are on a more equal basis
- A coroutine call is named a resume
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped
Coroutines Illustrated: Possible Execution Controls

(a)
Coroutines Illustrated: Possible Execution Controls

(b)
Coroutines Illustrated: Possible Execution Controls with Loops
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
- A coroutine is a special subprogram with multiple entries