Concepts of Programming Languages

6th Week

Subprograms and Parameters

Prof. Dr. Peter Heusch

Subprograms

- Subprograms historically were the oldest fundamental abstraction in programming
  - Available already in Charles Babbage's Analytical Engine (designed in the 1840s)!
  - Data abstraction came up much later with abstract data types and object orientation
- A subprogram is a collection of statements that can be re-used
  - Saves memory space and coding time
  - Enhances readability and hides implementation detail

Types of Subprograms

- Procedures: Return no value, side effects only (e.g., changes to variables that are visible to both procedure and caller)
- Functions: Return a value, are not supposed to have side effects (since modelled on mathematics), but usually do anyway
- Methods: Like functions, but special in their dependence on a calling object

Running Subprograms

1. Each subprogram has a single entry point
2. The calling program is suspended during execution of the called subprogram
3. Control always returns to the caller when the called subprogram's execution terminates

But:

- Coroutines can have multiple entry points
- Multi-threading tasks run simultaneously and do not necessarily return to the caller

Coroutines

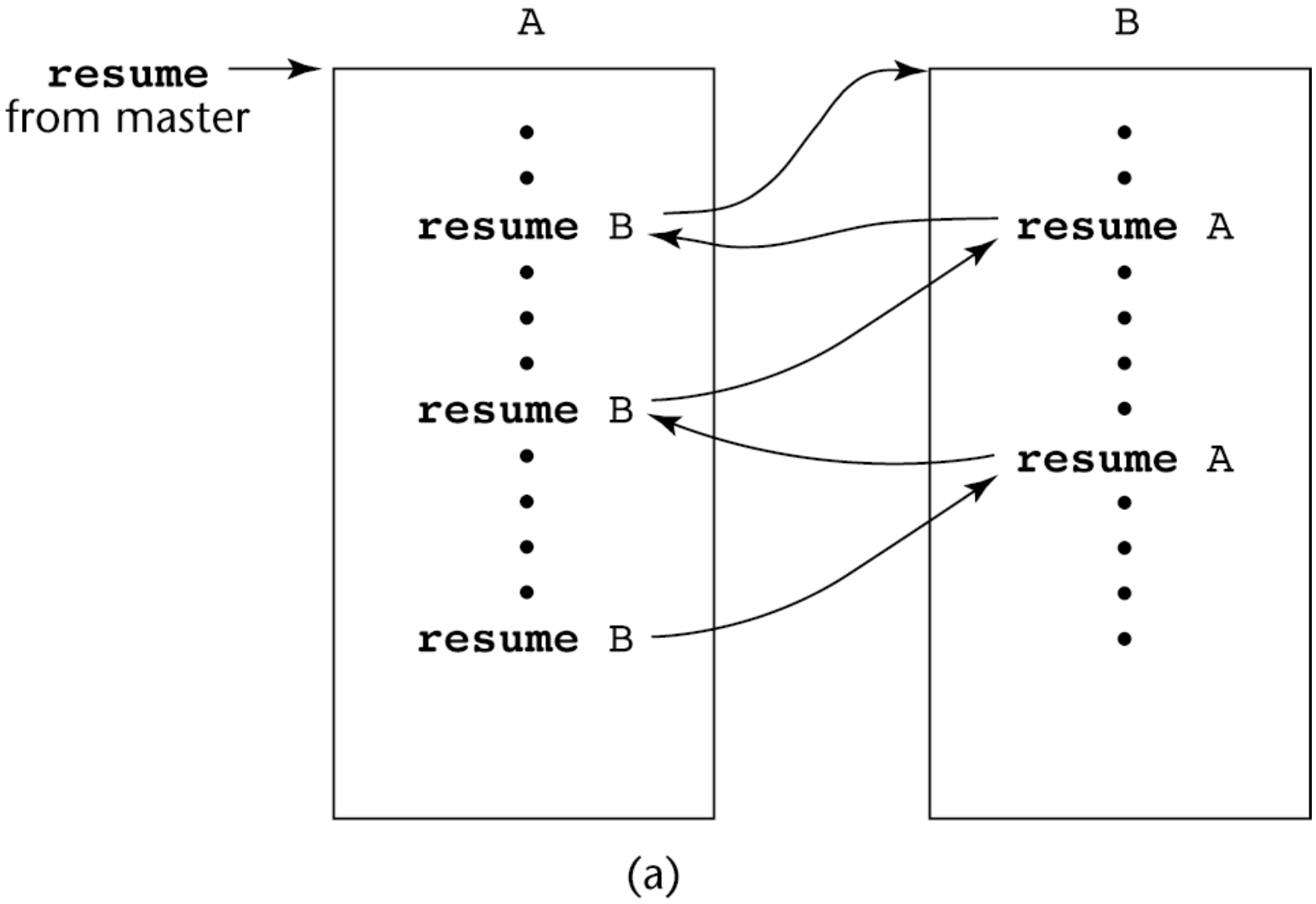
- A coroutine is a subroutine allowing explicit suspending and resuming at defined points
- Coroutines are well suited for consumer / producer problems:

```
def producer():
 while true:
   produce(item)
   enqueue(item)
   resume(consumer)
```

```
def consumer():
  while true:
    dequeue(item)
    consume(item)
    resume(producer)
```

- In C, coroutines can be implemented using setjmp and longjmp
- Lua supports coroutines

Possible Execution Paths



(a)

Subprogram Basics

Header (up to {)

Protocol: Signature+ return type

Signature: #, order, types

```
public static void main (String[] args) {
   System.out.println(„Hello World“);
}
```

Call

Subprogram Basics

- Subprogram definitions describe the interface to and the actions of a subprogram
- A *call* is an explicit request to execute the subprogram
- A *header* is the first part of the definition, containing the name, the kind, and the formal parameters of the subprogram
- The *signature* of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's signature and, if it returns a value, its return type

Declarations and Definitions

- A subprogram declaration provides the protocol but not the body of the subprogram
  - In C and C++, declarations are called prototypes
- If a subprogram declares parameters, there are always two outlooks:
  - Inside the subprogram declaration: *Formal parameter*, used to access the parameters from inside the subprogram
  - The corresponding *actual parameter* is the value or address used in the subprogram call statement
- By calling the subprogram, the actual parameter is linked to the formal parameter

Design Issues for Subprograms

- Different languages provide very different subprogram designs:
  - **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 subprograms be generic?**

Parameter Uses

- Pass values to the subprogram (in mode)
- Pass results from the subprogram (special „result“ variable instead of a return value) (out mode)
- Read value from parameter and modify it (inout mode)

Parameter Passing

- The transmission of parameters to and from the called subprograms is called parameter passing
- Five different techniques have been used:
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

Parameter Passing by Value

- Used to communicate values to the function only: in mode
- The value of the actual parameter is used to initialize the corresponding formal parameter
  - Normally implemented by copying from the actual to the formal parameter
  - Can be implemented by transmitting an access path but not recommended (write protection)
- When copies are used, additional storage is required
  - Storage and copy operations can be costly (for large arrays, structures)

Parameter Passing by Result

- Used to communicate results back to the caller: out mode
- Upon calling, no value is transmitted to the subprogram for the formal parameter
- The formal parameter is a local variable; its value is transmitted to the caller's actual parameter upon returning to the caller
- Requires extra storage location and copy operation
- Potential problem: Which value is returned when x(p,p) is called?

Parameter Passing by Value/Result

- Combines both preceding techniques (inout):
  - Actual parameter is copied to formal one at call
  - Formal parameter is copied to actual one at return
- Sometimes called passing by copy
- Needs additional storage and copy operations when passing large structures
- Advantage: all local variables are guaranteed to be free of any synchronization problems

Parameter Passing by Reference

- The subprogram gets an access path (typically a memory address, all modes)
  - Only a constant amount of information must be passed to the subprogram (max. 64 bit)
- Access to variables is slower than with passing by value, an additional indirection is needed
- Aliases (bad for readability) are possible: sub(p,p)
- Unintended changes may be made to globally visible variables and for parallel running programs, synchronization issues may appear

Parameter Passing by Name

- Actual parameters textually replace formal variables: no binding to a value until the parameter is first evaluated (in mode)
- Realized in ALGOL (by the compiler) and C (by the preprocessor), but complex and inefficient
- Can be implemented in Python by passing the expression to be evaluated as string:

```
def sum(f, t, e):
   r = eval(e)
   while f < t:
      f = f + 1; r = r + eval(e)
   return r
print(sum(1,100,"1.0/f"))
```

Implementing Parameter Passing

- Parameters are passed:
  - Through registers (SPARC, z/Series)
  - Through the stack (x86)
- Pass by reference passes an address onto the stack or inside a register
- Pass by value and/or result passes the whole data through the stack (unless very small)

Parameter Passing Conventions

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers
- Java
  - All parameters are passed by value
  - Object parameters pass a reference by value

Parameter Passing Conventions (cont.)

- Ada
  - Three semantics modes of parameter transmission: in, out, in out; in is the default mode
  - Parameter transmission also determines read- and writeability
- C#
  - Default method: pass-by-value
  - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with ref
- PHP: very similar to C#

Design Issues for Subprograms

- Different languages provide very different subprogram designs:
  - **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 subprograms be generic?**

Parameter Type Checking

- Considered very important for reliability
- FORTRAN 77 and Kernighan/Ritchie-C: **none**
- Pascal, FORTRAN 90, Java, and Ada: **always**
- Procedural part of ANSI C and C++ **user choice**:
  - Function with prototypes (protocol is known) are checked
  - Function without prototypes are unchecked (and have implicitly return type int)
- Weakly typed and dynamically typed languages (Perl, Python, JavaScript, PHP) **do not require** type checking

Subprograms as Parameters

- Natural concept in functional programming, more unusual in imperative programming
- C & C++ allow passing subprogram names as parameters:

```
void qsort(void *data, int count, int size, int
(*comparator)(void *, void *));
```

- Questions:
  - Are parameters checked for the subprogram?
  - What about the variables in the static scope?
- Java avoids these problems by always passing objects implementing some interface
  - Or use lambda expressions/closures since Java 8

Closures

- A closure is a combination of a subprogram and the referencing environment in which it was defined
    - Subprogram can now be called anywhere in the code, with all necessary variables visible and usable
- Supported in functional and scripting languages
    - These languages also have static scoping and allow subprograms values

Design Issues for Subprograms

- Different languages provide very different subprogram designs:
  - **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 subprograms be generic?**

Subprogram Overloading

- Subprograms are overloaded if more than one subprogram shares the same name
- Every version of an overloaded subprogram has a unique protocol (or signature)
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls, in C++ and Java this is forbidden
- Python does not have overloaded subprograms but dynamic parameters

Default Parameters

- A very simple variant of overloading is given by the possibility to introduce default parameters
- Parameters are called default, if the compiler / interpreter interpolates predefined values unless no explicit value is given:
  - In C++ at the point of declaration
  - In Python/PHP at the point of definition
- Default values must be literals or constants
- If some parameter has a default value, all of the following ones must have default values, too

Overloading and Defaults

- Overloading and Defaults can introduce ambiguities that are only visible when using classes:

```
int foo() ...
int foo(int value=42) ...
```

- Question: Which version of the function is to be used when calling `foo()`?
- Defaults can be replaced by overloading with trampolines:

```
int foo() { return foo(42); }
int foo(int value) ...
```

Named Parameters

- Some programming languages allow to link the arguments to parameters explicitly by denoting the formal parameter names to their values
- In conjunction with default values, named parameters allow to specify only those values that really deviate from their defaults
- Named parameters are heavily used in Visual Basic, where a single function is implemented to perform a great variety of tasks

Case Study: Python

- Python does not allow multiple definitions of the same function
- Since Python is dynamically typed, there is no need to predeclare the parameters, the set of formal parameters therefore only denotes the minimal number of actual parameters
- Further parameters are processed by:
  - A single parameter preceeded by * that catches additional positional parameters inside a tuple
  - A single parameter preceeded by ** that catches additional named parameters inside a hash

Design Issues for Subprograms

- Different languages provide very different subprogram designs:
  - **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 subprograms be generic?**

Generic Subprograms

- A generic or polymorphic subprogram takes parameters of different types on different activations
    - Overloaded subprograms provide *ad hoc polymorphism*
- If a subprogram takes a generic type parameter which is used in a type expression describing the type of the parameters, the subprogram provides *parameterized polymorphism*

Parametric Polymorphism in C++

- The following C++ template can be instantiated for any type for which operator > is defined:

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

- Instantiation works similar to text replacement

```
int max (int first, int second) {
      return first > second? first :
     second;
   }
```

Parametric Polymorphism in Java

- In Java a generic function requiring certain methods of the parameters must do so by giving the implementing interfaces in the declaration:

```
<T extends Comparable> T max(T first, T
  second) {
     return first.compareTo(second) > 0 ?
          first : second;
   }
```

-

Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be functions or procedures
- Three models of parameter passing: in mode, out mode, and inout mode, five passing methods
- Some languages allow subprogram overloading
- Subprograms can be generic