SIENAcollege
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2019

Topic Notes: Subprograms

Our next familiar programming language construct to examine in more detail is the *subprogram*. Whether called subroutines, procedures, functions, or methods, many of the fundamentals are the same.

We typically categorize subprograms into these groups:

- *procedures* or *subroutines* that do not return a value
- *functions* that return a single value
- *methods* are associated with an object (see Chapters 11-12, which we will likely only touch on briefly)

Even in early programming languages, the use of subprograms was emphasized to promote

- code reusability
- modularity – a high-level problem decomposition
- resource efficiency
  - memory space for code
  - reduce programming time

The general behavior of subprograms is well known to us as programmers:

- a subprogram has a single entry point (the beginning)
- the caller's execution stops while the subprogram executes
- control returns to the caller upon completion

And the basic terminology:

- subprogram *definition* – the declaration and implementation of a subprogram
- subprogram *call* – the explicit activation statement of a subprogram
- subprogram *header* – the first part of the definition

  - sometimes called a *specification* or *interface*
  - specifies *type* of subprogram, its *name*, and its *parameters* (if any)

- subprogram *body* – statements which implement the subprogram
- subprogram *parameter profile* or *signature* – the number, order and types of the subprogram's formal parameters
- subprogram *protocol* – a subprogram's parameter profile plus its return type
- subprogram *declaration* – a mini "pre-definition", stating protocol information (*e.g.*, C/C++ Prototypes)
- *formal parameter* – name listed in the subprogram header and used in the subprogram like a local variable
- *actual parameter* – a value or address used in the subprogram call statement

How do actual parameters get matched to the appropriate formal parameters on a subprogram call?

We are likely most familiar with *positional parameters* – the mapping of actual to formal parameters is based on the order in the parameter list:

```
int sub(int x, int y) {

  return x - y;
}

...

  sub(5, 2);
```

Here, we all know that `x` is assigned `5` and `y` is assigned `2` on the example function call.

You may or may not have seen examples of languages that use *keyword parameters*, where the actual parameters are explicitly matched to formal parameters by name.

This means parameters can be listed in any order, avoiding transposition errors. However, it requires that callers must know the formal parameter names.

Ada (of course) is one of the languages that supports this. The text shows an example from Python.

Fortran 90 does this as well:

On the web: F90 Keyword arguments and default arguments at
`https://www.nsc.liu.se/~boein/f77to90/c8.html`

The above also shows how Fortran 90 deals with *default parameters*. In these cases, a parameter can be left off and given a default value when it is not specified.

C++ supports this, but since its parameters are specified only positionally, any optional parameters must come at the end of the parameter list.

See Example:
`/home/cs340/examples/cppdefault`

Many languages also support variable-length argument lists (*variadic functions*). We know that C must support this if functions like `printf` could work, since it can take any number of parameters, depending on the number of specifiers in the format string.

But how does it do it?

See Example:
`/home/cs340/examples/varargs/varargs.c`

For examples in many languages:

On the web: Wikipedia article "Variadic function" at
`http://en.wikipedia.org/wiki/Variadic_function`

Java's mechanism:

See Example:
`/home/cs340/examples/varargs/Varargs.java`

Perl passes all parameters to subprograms in a special "parameter array" called `@_`.

On the web: Using the Parameter Array at
`http://www.cs.cf.ac.uk/Dave/PERL/node50.html`

Models of Parameter Passing

The text discusses many design issues for subprograms, but we will focus on just a few.

The first is the semantics of parameter passing. That is, when we have a subprogram, how does information pass through the parameters between the caller and the callee?

There are three major models:

- *in mode* – information flows through the parameter from caller to callee
- *out mode* – information flows through the parameter from the callee back to the caller
- *inout mode* – information flows through the parameter from the caller to the callee, then back from the callee to the caller

---

General Semantics

In most cases, parameter passing occurs through a run-time stack.

A subroutine call generally involves:

- setting up and initializing memory for parameters
- stack-dynamic allocation of local variables
- saving of the execution status of calling program
- transfer of control and arrange for the return

and on return:

- in mode and inout mode parameters must have their values returned
- deallocation of stack-dynamic local variables
- restoration of execution status
- return of control to the caller

The *subprogram linkage*, which is the entire call and return process, most often depends upon an *activation record* placed on the program's run-time call stack.

An activation record for simple subprograms consists of three parts:

- space for local variables
- space for parameters
- the return address

If we have support for stack dynamic local variables (which is the case for the modern languages that support recursion), we also need a register which will hold a base address (often called a *frame pointer*). The frame pointer is the base to which the offsets of all local variables and parameters are added to compute their actual address.

In these cases, a *dynamic link address*, which points to the start of the activation record of the caller, is part of the activation record so it can be restored on return.

We will look later at more details of how this works.

These are accomplished using a number of methods for parameter passing.

Call by Value

With *call by value*, the formal parameter is initialized by actual parameter. No changes to the formal parameter in the subprogram should propagate back to the actual parameter in the caller.

It is normally implemented by copying, but can be done by providing a write-protected *access path* to the actual parameter.

- With copying, write protection of the actual parameter is easy – the subprogram has no access to it
- However, copying requires extra space, as there are now 2 copies, and extra time, as the copy must be performed
- With an access path, there is the expense of enforcing write protection and access is slower through indirect references

This is the parameter passing method of choice for most modern languages, including C/C++, C#, Java, Pascal, Ruby, Scheme.

Call by Reference

With *call by reference*, the formal parameter is a reference to the memory location of the actual parameter.

This is used in Pascal (with the `var` keyword) and C++ with the `&` operator.

See Example:
`/home/cs340/examples/callbyref/callbyref.cpp`

This is efficient, eliminating the copy and extra storage needed for call by value, but access is slower because of the indirection.

It also introduces potential side effects and aliases.

C appears to have a call by reference, but it is really a call by value where the values are pointers.

See Example:
`/home/cs340/examples/callbyref/cpointers.c`

All object parameters in Java are passed by reference.

Call by Result

With *call by result*, no information is initially transmitted to the subprogram through the parameter. The formal parameter acts like a local variable in the subprogram, then its final value is sent back to the actual parameter (by copying).

A few problems that can arise:

- If we have

  ```
  f(x, x);
  ```

  which formal parameter from inside `f` will be copied back to `x` last?
- If we have

```
f(a[i], i);
```

do we use the original `i` to find the appropriate array entry to copy back to, or the potentially-modified `i`?

Both C# and Ada provide call by result by specifying the `out` keyword to a parameter.

Call by Value-Result

A parameter passed by `value-result` is a combination of call by value in that the formal parameter is initialized using the actual parameter's value, and call by result in that the formal parameter's final value is copied back to the actual parameter at the end of the subprogram.

This differs from call by reference in that the formal parameters have local storage during the execution of the subprogram.

Ada supports this by using both the `in` and `out` keywords.

Call by Name

With *call by name* parameters, parameters are passed by a textual substitution. However, it is difficult to implement and is not used by any major language.

The idea is sometimes used at compile time, for example in C, we can use the `#define` mechanism to define macros that act like call by name.

```
#define SAFE_MALLOC(v,type,size) \
 {  v = (type) malloc(size) ; \
    if ( v == NULL) { \
       fflush(stdout); \
       fprintf(stderr,"in file %s, line %d, failed to allocate %ld bytes",\
               __FILE__,__LINE__,size); \
       exit(1); \
    } \
 }
```

ALGOL 60 did implement call by name. This allowed for a programming technique called *Jensen's Device*.

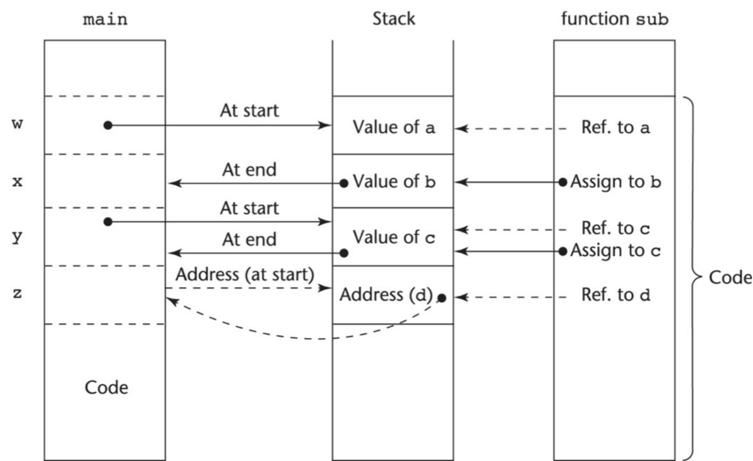
On the web: Wikipedia article "Jensen's Device" at
`http://en.wikipedia.org/wiki/Jensen's_device`

We can use the C preprocessor to implement this idea:

See Example:
`/home/cs340/examples/jensens`

Implementing Parameter-Passing

In most languages parameter communication takes place through the run-time stack.



Pass-by-reference are the simplest to implement; only an address is placed in the stack.

Multidimensional Arrays as Parameters

In a language like C, a one-dimensional array can be passed to a function as a pointer to the first element of the array, and from the type of the array and an index, pointer arithmetic can be used to calculate the addresses of array elements.

However, with a multidimensional array, a single pointer is not sufficient, as the address will depend on the length of the array in each dimension. C can overcome this by forcing all but the first subscript to have a length specified in the formal parameter definintion, or by passing an extra parameter and performing the pointer arithmetic manually rather than using subscript notation.

Other languages, like Java, implement multidimensional arrays as single-dimensional arrays of single-dimensional arrays, avoiding this problem.

Overloaded Subprograms

An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment.

This is included in languages including Ada, Java, C++, and C#. The resolution of which should be used for a specific call is based on which definition's protocol matches the call.

See Example:
`/home/cs340/examples/overload`

Indirect Subprogram Calls

Sometimes, we do not know until runtime which subprogram will need to be called for a particular run of a program.

In C and C++, this is accomplished through *function pointers*. Examples where this is used include

- *callback functions*, which are functions passed as parameters to a function that are to be called by that function to perform part of their task

  On the web: "Callback" on Wikipedia at
  `http://en.wikipedia.org/wiki/Callback_(computer_programming)`

  You will work with callbacks in C in a lab exercise.

- the function to call upon creation of a new thread (as in POSIX threads)

  See Example:
  `/home/cs340/examples/pthreadhello`

Generic Subprograms

A *generic* or *polymorphic subprogram* takes parameters of different types on different activations.

We saw exampels of overloaded subprograms, which provide *ad hoc polymorphism*.

With *subtype polymorphism*, a parameter of type `T` can access any object of type `T` or any type derived from `T` (in object-oriented programming languages).

We will primarily consider *parametric polymorphism*, where a subprogram can take a *type parameter*.

The text described generic subprograms in C++, which uses the `template` keyword:

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

The function `max` can then be called with any datatype for its parameters, and it will return the appropriate type. We can try it:

See Example:
`/home/cs340/examples/cppgeneric`

Notice that in C++, the `Type` can be either a primitive type or a class.

In Java, where generic have been supported since version 5.0, type parameters must be classes. With *autoboxing* and *autounboxing*, where needed conversions are automatically made between the primitive types (*e.g.*, `int`, `double`, *etc.*) and their `Object` containers (*e.g.*, `Integer`, `Double`, *etc.*), this is less of a restriction than it may at first seem.

Some examples:

See Example:
`/home/cs340/examples/javageneric`

This example also includes generic classes in addition to generic methods. Worth a look now, even though it's not part of the subprogram topic.

Overloaded Operators

Operators are overloaded in many languages – an example you know well is the + operator being used for addition and string concatenation in Java. Some languages allow user-defined functions to define new overloads for operators. For example, the C++ program below includes several *overloaded operators*.

See Example:
`/home/cs340/examples/opoverload`

Coroutines

One final topic of interest from Chapter 9 of Sebesta concerns *coroutines*. These are somewhat like traditional subroutines, as they call each other, and return, but the caller and called coroutines are on a more equal footing, they have *symmetric control*.

The idea is that when a coroutine returns to its caller, it retains its state information and resumes after the last statement executed in the previous execution, forming a sort of "quasi-concurrency" possibility. As control is transferred back and forth (using a *resume* operation in place of a traditional subroutine call) between coroutines, the result is an interleaving (but not true concurrency, where executions would overlap).

A few graphics demonstrating this can be found in the text's Figures 9.3 and 9.4, p. 405.

More on Subroutine Implementation

Chapter 10 describes implementation issues arising with subroutines, and we will touch on a few of those.

We discussed earlier the general semantics of subroutine calls and returns – passing parameters, allocating local variables, transfer of control, and deallocation.

For what Sebesta calls "simple" subprograms, no real call stack is needed: the information for all subroutines, both code and data, is available at all times in the activation record instance.

However, such simple subprograms have significant limitations, notably that they do not support recursion. Since there is only one copy of the parameters, local variables, and return address for each subroutine, if more than one copy of any subroutine is "active" at the same time, later invocations will overwrite data from earlier invocations.

The text shows an example of how a language such as C supports recursion using stack dynamic variable allocation. The introduction of a *dynamic link* allows management of multiple instances of the activation records for each subroutine to exist on the stack. An *environment pointer* (EP) (sometimes called a *frame pointer*) is used to locate the current activation record on the stack for access to its parameters and local variables. Copies of the EP are stored on the run-time stack as part of activation records so the EP can be restored to the caller's context when a subroutine returns.

Many of the specifics come down to the architecture to which the language is being compiled. Local variables might be allocated to registers and only saved to the stack when necessary because of a subsequent subroutine call. For example, consider this code using the MIPS ISA:

See Example:
`/home/cs340/examples/mips-factorial`

And a greatest common denominator function in Motorola 68000 assembler:

See Example:
`/home/cs340/examples/m68k-gcd`

Finally, we can see how the gcc x86 compiler performs function calls by running gcc with the `-S` option to generate a `.s` file.

See Example:
`/home/cs340/examples/gcd`

We will not be concerned with nested subprogram implementation as described in Section 10.4.

Section 10.5 describes how block variable storage can be implemented. The key idea is that activation record space can be shared among variables whose lifetimes are guaranteed not to overlap.