CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by: Ira Woodring

Subprograms

(Chapter 9 in Sebesta)

Recall that there are two types of abstractions in a programming language - data abstraction and process abstraction.

Process abstraction has been around for some time; data abstraction came much later.

Process abstraction allows us to reuse code - typically through the use of a subprogram.

A **subprogram** is a binding of a grouping of statements to a process. This binding may or may not have a name.

Subprograms help us to divide our code into logical units of repeated processes. Subprograms have the following:

- each subprogram has a single entry point
- the code that calls the subprogram is suspended while the subprogram runs
- control of the program returns to the calling code when the subprogram finishes

(Notice here we are not talking about any kind of concurrency).

The interface and the process of the subprogram are called the **subprogram definition**. A **subprogram call** asks our runtime to create a new instance of a subprogram and to execute it.

When our subprogram is in the process of executing it is said to be **active**.

A **subprogram header** provides the details about the subprogram, such as the type of subprogram it is. A subprogram can be of two types:

- **procedures** - can be thought of as defining new statements; these do not return values.
- **functions** - are subprocesses that can return values (they could also be void).

A **method** is a subprogram that is associated with a class or object

We learned about methods during our lectures on OOP

The header also provides the name for the subprogram (if it is given a name), and may also list any parameters the subprogram may take.

We may often call subprogram headers a **signature**.

Every language that implements subprograms decides upon the syntax of creating a function. In C and C++ for instance we might have:

```
void doStuff(int a, double c);
```

Javascript and PHP begin each subprogram with the `function` keyword:

```
// Javascript
function doStuff(a, c){ }

// PHP
function doStuff($a, $c){ }
```

Perl uses the sub keyword:

```
sub doStuff {
  my($a, $c) = @_;
}
```

Ruby and Python both use the `def` keyword:

```
// Ruby
def doStuff(a, c)
  ...
// Python
def doStuff(a, c):
  ...
```

Though Python requires the `:` to denote a new scope

Interestingly, in Python the `def` statements are executable code that causes a name to be given to the function. Since we can execute the function definition we can do some interesting things:

```
if i==1:
  def doStuff(a, b):
    return a + b
else:
  def doStuff(a, b):
    return a * b
```

This causes the program to make a choice about how to define `doStuff()`, and can lead to some interesting behavior in a program.

A subprogram may have a **parameter profile** which provides the

- number
- order
- type(s)

of its formal parameters.

Formal parameters are listed in the subprogram header. The variables listed in this header are sometimes called dummy variables; when defining the subprogram they aren't instantiated or usually even bound. Typically this happens when the subprogram is called.

Even then the dummy variables may just be bound to other program variables.

When we call a subprogram and pass parameters we are passing **actual parameters.**

```
// Formal parameters
int doStuff(int a, float c);

...

int main(int argc, char** argv){
  int x = 42;
  float y = 1701.0;
  // Actual parameters
  doStuff(x, y);
}
```

There are different ways to relate actual parameters to formal parameters. Most languages do so through inspection of the parameter positions (these are, unsurprisingly enough, called **positional parameters**).

However, some languages also provide for **keyword parameters**:

In Python for instance:

```
def calcVolume(height, width, depth=1):
  return height * width * depth

// All of these are valid calls
calcVolume(height=10, width=5)
calcVolume(width=5, height=10)
calcVolume(height=10, width=5, depth=3)
calcVolume(width=5, height=10, depth=3)
calcVolume(10, 5)
calcVolume(10, 5, 3)
// Even mixed! (Though once we use a keyword we
// must continue them.  No more positional after that!
calcVolume(5, depth=8, width=8)
```

Notice that in the above example `depth` was given a default value. This meant that we did not need to expressly provide a value for depth. This would allow us, for instance, to reuse this function for 2d or 3d objects easily.

Some languages allow us to provide a variable number of parameters. In C for instance, the `printf` function has the following function header:

```
int printf ( const char * format, ... );
```

The ellipsis here means that the function can take any number of parameters:

```
printf("The meaning is %d.\n", meaning);
printf("%d + %d * %d = %d\n", a, b, c, d);
```

We certainly don't want to write a separate function for each possible number of parameters passed in.

We can do this with variable argument lists (defined in stdarg.h):

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

void print_args(const char *str, ...){
        va_list ap;
        va_start(ap, str);
        while(str){
                printf("%s\n", str);
                str = va_arg(ap, const char *);
        }
        va_end(ap);
}

int main(int argc, char** argv){
        print_args("Ira", "Jacob", "Woodring", NULL);
}
```

We can do the same thing in Python with *args or **kwargs (depending on whether we need positional or keyword arguments):

```
def printStuff(*args):
  for i in args:
    print i

def printStuff2(**kwargs):
  for key in kwargs:
    print key,":",kwargs[key]
```

Overloaded subprograms are available in some languages. This allows us to have multiple subprograms with the same name in the same referencing environment. We have seen these in Java and C before; for instance we might have the following functions:

```
public int addMe(int x, int y);
public float addMe(float x, float y);
```

Though both of these have different signatures, they have the same name.

Generic subprograms may work the same way on different types of data. For instance, we may want to create a function that returns the maximum of two values, regardless of type. In C++ we might do this:

```
template <typename T>
T chooseMax(T a, T b){
  if(a<b)
    return b;
  return a;
}
```

As long as a and b are the same type here, and have a operator< defined this will work.

Similarly, Java has generic capabilities as well:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

(taken from https://en.wikipedia.org/wiki/Generics_in_Java)

Variables that are defined inside of a subprogram are called **local variables**. They may be static or stack dynamic (exist throughout the programs lifecycle or are bound and unbound with the lifecycle of the subprogram).

As we talked about with variables, stack dynamic are the most flexible. They are essential for recursive subprograms. However they are slower than static variables

When designing a language we must decide how to pass parameters to subprograms. There are three models:

- in mode
- out mode
- inout mode

- in mode parameters receive data from actual parameters
- out mode transmit data to the parameter
- inout mode can do either mode

To implement these methods of parameter passing, a few different models exist.

- Pass-by-value
- Pass-by-result
- Pass-by-value-result
- Pass-by-reference
- Pass-by-name

Pass-by-value uses the actual parameter to initialize a corresponding formal parameter. The formal parameter will then be a local variable to the function.

This is a way to perform in-mode passing.

Usually pass-by-value is implemented by copying but we could do it by passing an access path to the actual parameter, if the memory for it was write-protected.

Pass-by-value is fast for small objects, but for large objects or large collections of objects this can be slow. Additional overhead is needed for storage.

Pass-by-result does not pass a value to a subprogram; this is a method of performing out-mode passing. The formal parameter to the function serves as a local variable. When the function returns, the copy of the local variable gets copied into the actual parameter.

Pass-by-result has the same issues as pass-by-value, and can additionally fail if a parameter collision occurs. Consider:

```
sub(var1, var2){ ... }

sub(p1, p1);  // Function call
```

When the function returns var1 and var2 will be overwritten with p1. Or, function call worthless if p1 is read-only.

Pass-by-value-result provides inout-mode passing. It is a combination of pass-by-value and pass-by-result. The actual parameters are used to initialize the formal parameters which are then local to the function. At the return from the function that formal parameters are passed to the actual parameters.

This is sometimes called Pass-by-copy.

Pass-by-reference provides in-out mode passing as well. Instead of copying though, an access path is passed to the subprogram.

This is very efficient; no copying needs to be done and no extra storage space is needed.

However, it isn't perfect. It is slower for a subprogram to use these references since the variables aren't local. It also can harm readability and reliability.

This is due to the fact that we are creating aliases; our formal parameters become aliases for our actual parameters.

Pass-by-name is another inout mode passing technique. This technique essentially places the actual parameter in-line of the code where the formal parameters are used.

For the languages we have discussed in this class, C uses pass-by-value (in-mode) but allows us to simulate pass-by-reference (in-out mode) with pointers. C stole this idea from ALGOL 68 which used the same methods.

C++ has all the same methods the C has, but adds reference types as well. Reference types provide an additional in-out mode for C++.

Java uses pass by value, but objects can only be accessed via references in Java. This makes it seem as if Java does pass by reference. Java does not have a facility for passing primitives by reference.

Ada and Fortran (since Fortran 95) can specify in, out, or inout mode for each formal parameter.

C# is pass-by-value, much like Java. C# can do pass-by-reference though, by putting the `ref` keyword before a formal and actual parameter.

```
void sumer(ref int oldSum, int newOne) { ... }
...
sumer(ref sum, newValue);
```

C# also provides supprt for out-mode parameters via pass-by-reference parameters that do not need any initial values. We create these with the `out` keyword.

PHP is pass-by-value, but we can perform pass-by-reference by putting an ampersand before our parameters.

```
function doStuff(&$var){
...
}
```

Most programming languages allow only a single return value, but there are those that allow more. Lua for instance, allows multiple returns through a comma-separated list:

```
return 3, sum, index
```

Python pretends to allow multiple values. It will allow the return of a tuple (which is just a single object), but it makes it easy to get the data out of the tuple:

```
def doStuff(){
    ...
    return (one, two, three)
}

a,b,c = doStuff()
```

Or, if we don't care about some of the data being returned we can ignore it:

```
def doStuff(){
  ...
  return (one, two, three)
}

_,b,_ = doStuff()
```

Go allows this as well:

```go
package main

import "fmt"

func vals(int, int){
  return 3,7
}

func main() {
  a,b := vals()
  fmt.Println(a)
  fmt.Println(b)

  _, c := vals()
  fmt.Println(c)
}
```

(Taken from https://gobyexample.com/multiple-return-values)

It is often helpful for us to pass subprograms to subprograms. Perhaps we want to provide a custom sorting function. Our book notes that we may want to write a function that performs mathematical integration. To have it work on any function we may wish to pass in the function it is integrating as a parameter.

In C, we use function pointers to pass a subprogram to a subprogram. For instance, in the following code we choose between either an add or mult function depending on what operation the user requests:

```
int add(int a, int b){ return a + b; }

int mult(int a, int b){ return a * b; }

int main(int argc, char** argv){
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        int c = atoi(argv[3]);

        int (*fun)(int,int) = &add;
        if(c != 0){
                fun = &mult;
        }
        int result = (*fun)(a,b);
        printf("%d\n", result);
}
```

Many newer languages make this far easier:

Javascript for instance:

```
function add (a, b){
  return a + b;
}

function mult(a,b){
  return a * b;
}

function doStuff(a, b, fun){
  return fun(a,b);
}
```

Python:

```python
def add(a, b):
  return a + b

def mult(a,b):
  return a * b

def doStuff(a, b, fun):
  return fun(a, b)
```

As of Java 8 we have lambda expressions, and can do something like the following:

```
public class Calculator {
    interface IntegerMath { int operation(int a, int b);   }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
```

Previous example taken from

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax

In languages that allow nested subprograms and passing subprograms as arguments, another complication arises

When the nested subprogram is finally called, possibly in a different scope from where it was defined, what referencing environment does the subprogram use?

Consider the following example from your textbook

```
function sub1() {
    var x;
    function sub2() { alert(x); };
    function sub3() {
      var x;
      x = 3;
      sub4(sub2);
      };
    function sub4(subx) {
      var x;
      x = 4;
      subx();
      };
    x = 1;
    sub3();
    };
```

This can be resolved three ways the referencing environment can be chosen:

- *shallow binding* -- use environment of calling statement that enacts subprogram
- *deep binding* -- use environment of definition of subprogram
- *ad hoc binding* -- use environment of statement that passed subprogram as an actual parameter

```
function sub1() {
    var x;
    function sub2() { alert(x); };
    function sub3() {
      var x;
      x = 3;
      sub4(sub2);
      };
    function sub4(subx) {
      var x;
      x = 4;
      subx();
      };
    x = 1;
    sub3();
    };
```

You will typically see deep binding -- it is the most appropriate choice for static-scoped languages

Shallow binding is sometimes used in dynamic-scoped languages

Ad hoc binding is never used

One issue with using deep binding is that the referencing environment may no longer exist

If function in which nested subroutine was created has returned, any stack-dynamic variables should be deallocated

Closures are subprograms *and* the referencing environment where they were defined. We need the referencing environment so they can access variables in the scope in which they were created.

Consider:

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

var add10 = makeAdder(10);
var add5 = makeAdder(5);
alert("10 + 20 = ", + add10(20));
alert("5 + 20 = ", + add5(20));
```

(Taken from the book). Here add10 retains its binding of 10 to x, and add5 retains its x binding to 5. Those bindings must be available when the functions are called.

Note that the programmer did not keep track of the closure directly