CSE 3302 Notes 7: Control Abstraction

(Last updated 11/1/15 12:35 PM)

References: Gabbrielli-Martini: 7; Dybvig: 5.5, 5.7

7.1. SUBPROGRAMS

Interfacing Concepts:

Parameters:

Mathematics:	Parameters	Arguments
Computing:	Formal Parameters	Actual Parameters

Are the types of the actual parameters checked against the types of formal parameters?

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/separate1.c`
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/separate2.c`

Positional (conventional)

Named - formal parameter names may be used in caller -
`http://en.wikipedia.org/wiki/Named_parameter`

Default values - specified with the procedure
`http://en.wikipedia.org/wiki/Default_argument`

Flexible arity -

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/poly.cpp` (varargs, notes 04)
JavaScript - every function has a local variable `arguments` that accesses (and aliases) the argument list (Crockford, p. 31)

Return Values: Are they part of function's signature?
(`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/notes07.return.cpp`)

State Changes - input/output parameters, encapsulated data structure/object, globals, database/files . . .

Aside: Relationship with caller - if any, or existence as a process/thread

From SR (Synchronizing Resources), a concurrent/distributed language:

SR Mechanisms	`proc` (procedure)	`in`
`call` (synchronous)	Procedure call	Rendezvous/handshake with process (from Ada, single-slot buffer)
`send` (asynchronous)	Thread creation	Message passing (queue)

http://ranger.uta.edu/~weems/NOTES4351/04notes.pdf, p. 16-18
"An overview of the SR language and implementation",
http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=42192.42324

Functional Abstraction (Software Component) Concepts:

Process and Data Abstraction . . . reusability, modularity . . . "software ICs"
http://www.cs.dartmouth.edu/%7Edoug/components.txt
(Data - Cardelli & Wegner *ACM Computing Surveys* (1985) article,
http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=6041.6042 for Notes 8/9/10)
Process - W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design",
IBM Systems Journal, Vol 13 (2), 1974, 115-139,
http://ieeexplore.ieee.org.ezproxy.uta.edu/stamp/stamp.jsp?tp=&arnumber=5388187
Cohesiveness - coincidental, logical, temporal, communicational, sequential, functional (p. 121)
Lambda calculus / function application as a model of computation:
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/lambdaLand.rkt
(Scary aside: http://en.wikipedia.org/wiki/Ladder_logic)

Parameter-Passing Mechanisms:

Call-by-value - a copy of the object is created

Call-by-reference - a pointer to the object is passed, so the object may be modified
(Java situation for objects, C for arrays, is occasionally referred to as call-by-sharing)

Pascal - both -by-value and -by-reference are available for all objects

By-value - even arrays, structures, and sets will be copied
(asides: compiler avoidance or https://en.wikipedia.org/wiki/Persistent_data_structure)

To pass by reference - `var` keyword before parameter

Not difficult to mix these together

```
procedure c(var x:integer)
begin
end;

procedure d(x:integer)
begin
end;

procedure b(var x:integer)
begin
  c(x);
  d(x);
end;

procedure a;
var x:integer;
begin
  b(x);
end;
```

C

By-value - scalars, structures

By-reference - arrays

Mixing these is messy (but C++ uses `&` in parameter list to simulate `var`)

```
void c(int *x)
{
}

void d(int x)
{
}

void b(int *x)
{
  c(x);
  d(*x);
}

void a()
{
  int x;
  b(&x);
}
```

```
void c(int &x)
{
}

void d(int x)
{
}

void b(int &x)
{
  c(x);
  d(x);
}

void a()
{
  int x;
  b(x);
}
```

Call-by-name - behaves as though the argument *expression* is substituted into function

No concern for scope (involved variables are by-reference in original scope)

Simple examples look like call-by-reference

Like macros (but "dynamic" . . . "textual substitution") - don't take the call to the function, *take the function to the call*

Each argument expression may have associated code ("thunk") to support the use of the expression as an r-value and as an l-value

Relationship between arguments

```
swap(x,y)
  t=x;
  x=y;
  y=t;
```

```
swap(i,a[i])
  t=i;
  i=a[i];
  a[i]=t
```

```
swap(a[i],i)
  t=a[i];
  a[i]=i;
  i=t;
```

Two things to read carefully in Gabbrielli:

p. 175 - copy rule, especially the *does not capture variables* part

p. 178 - Jensen's Device

http://en.wikipedia.org/wiki/Man_or_boy_test (1964) brings recursion to the party ...

Passing arrays for C:

```
int a[10][20][30],***c;

void printMat1(int m,int n,int p,int ***c) {
int i,j,k;

for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    for (k=0;k<p;k++)
      printf("[%d][%d][%d]=%d\n",i,j,k,c[i][j][k]);
}

void printMat2(int m,int n,int p,int c[][20][30]) {
int i,j,k;

for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    for (k=0;k<p;k++)
      printf("[%d][%d][%d]=%d\n",i,j,k,c[i][j][k]);
}

printMat1(10,20,30,c);
printMat2(10,20,30,a);
#error "1"
printMat2(10,20,30,c);
#error "2"
printMat1(10,20,30,a);
```

7.2. HIGHER-ORDER FUNCTIONS

Simple (pointer to) function passing for C, due to simple scoping

Classic UNIX/C data structure examples:

```
void qsort(void *base, size_t nmemb, size_t size,
     int(*compar)(const void *, const void *));

void *bsearch(const void *key, const void *base, size_t nmemb,
     size_t size, int (*compar)(const void *, const void *));
```

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/functions.c

Traditional Pascal - avoids issues by requiring passed procedures to have only by-value parameters (and procedures/functions cannot be returned)

Scheme (aside) - http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/notes07.ho.rkt

Languages with nested functions, lexical/static scope, procedures as arguments - deep binding

1. When a function is passed, its referencing environment is committed (e.g. by passing an *additional pointer* with the appropriate static link value, in addition to the address of the function's code).

2. The called function has no reason to use the additional pointer directly since the compile-time symbol tables have different referencing environments.

3. Whenever the passed function is called, the new activation record will have its static link set to the *additional pointer*.

This necessity is known as the *downward fun*(ctional) *arg*(ument) *problem*.

The previous extension works as long as the function address (and additional pointer) cannot be used for a call after an activation record on the static chain is gone, known as the *upward fun*(ctional) *arg*(ument) *problem*:

1. In general, the involved data/activation records need heap allocation (and a garbage collection mechanism).

2. Without other complications (e.g. recursion), garbage collection might be avoided.

(This mechanism is also effective for dealing with *continuations*)

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES07/notes07.funarg.rkt`

7.3. EXCEPTIONS

Aside: Multilevel returns/Signals (`setjmp`/`longjmp`)/Exceptions in C (`http://ranger.uta.edu/~weems/NOTES3302/signal.c`)

For JavaScript: `http://ranger.uta.edu/~weems/NOTES3302/exception.html`

C++

Exception Hierarchy (`http://en.cppreference.com/w/cpp/error/exception`)

RAII - resource acquisition is initialization

Well-designed C++ code should avoid explicit destructor calls and depend on scope

Thrown exceptions going through several levels of calls will not lead to resource leaks

Implementations

Simple - for each encountered `try`/`catch` block: entry registers (pushes) and exit removes (pops) the handler and targeted exceptions from a list. Handling an exception involves traversing this list and activation records.

Low run-time overhead (in absence of throws) - Each step that would follow an exception list link (in the simple method) is replaced by a binary search (using the program counter as key) of a table whose entries are the beginning of code for functions and `try`/`catch` blocks.