CSE 3302 Notes 9: Subprograms

(Last updated 4/2/14 7:23 PM)

9.1. Introduction

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

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

Lambda calculus / function application as a model of computation:

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES09/lambdaLand.rkt

9.2. Fundamentals of Subprograms

Interfacing -

Input - Parameters (passed by value)

Output - return value or parameters (passed by reference)

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

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

Identity ("key") - parameter profile/signature/prototype/protocol . . . *overloading*

Given a number of similar definitions (including inheritance), what are the possibilities for matching a call under the presence of coercions (C++)?

Parameters -

Positional

Keyword

Flexible arity -

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES05/poly.c (varargs, notes 05)

JavaScript - every function has a local variable `arguments` that accesses (and aliases) the argument list

9.3. Design Issues

List of questions regarding subprogram support in a PL (Sebesta, p. 397):

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

http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES09/separate1.c
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES09/separate2.c

9.4. Local Variables

Nothing new . . .

9.5. Parameter Passing Methods

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
(aside: compiler "avoidance" or "persistent" data structures?)

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`)

```
c(int *x)
{
}

d(int x)
{
}

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

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 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;
```

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

Passing arrays:

```
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);
```

9.6. Parameters that are Subprograms

Binding time of referencing environment for a subprogram

- Shallow Binding - ignore (goes along with dynamic scoping, e.g. Perl . . . stack-like)
- Deep Binding
  - Static/lexical binding
    - Static chain pointer created *at same time as* reference to function (*closure*)
    - Problem when reference to function persists longer than stack frame
  - Solution - anything needed for closure gets heap allocation (and garbage collection)
    - Like object systems
    - Like functional languages
- JavaScript (`http://ranger.uta.edu/~weems/NOTES3302/closure.html`)

9.7. CALLING SUBPROGRAMS INDIRECTLY

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 *));
```

Pascal - avoids issues by requiring passed procedures to have only by-value parameters

9.8. OVERLOADED SUBPROGRAMS (http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES05/poly.c)

9.9. GENERIC SUBPROGRAMS

Options:

Macros - C (http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES09/Qmacro.c)

- stringify token ## - concatenate tokens

```
// Macro definition
#define Q(name,n,typ) \
typ Q##name[(n)+1]; \
int Q##name##Tail=0,Q##name##Head=0;   \
int Qempty##name() \
{ return Q##name##Tail==Q##name##Head; } \
\
int Qenqueue##name(typ x) \
{ \
  Q##name[Q##name##Tail++]=x; \
  if (Q##name##Tail==(n)+1) \
  { \
    printf("Qenqueue%s error\n",#name); \
    Q##name##Tail=0; \
  } \
  if (Qempty##name()) \
    exit(0); \
} \
\
typ Qdequeue##name() \
{ \
  typ temp; \
  if (Qempty##name()) \
  { \
    printf("Qdequeue%s error\n",#name); \
    exit(0); \
  } \
  temp=Q##name[Q##name##Head++]; \
  if (Q##name##Head==(n)+1) \
    Q##name##Head=0; \
  return temp; \
}
```

```
// Macro application

Q(TenInts,10,int)

...

for (i=0;i<10;i++)
  QenqueueTenInts(i);
for (i=0;i<10;i++)
  if (i!=QdequeueTenInts())
    exit(0);
if (QemptyTenInts())
  printf("TenInts works\n");
```

Dynamic Binding

Concept

Write general code in terms of a base class

New situations are handled by subclassing/method overriding

Runtime penalty for homogeneous situations?

Type errors are left to run-time

Generics

Concept

Similar to ML lists - all items have same type (oversimplified)

Still necessary to have methods/functions for new types

Can be defined with several involved type variables

Type variables could still involve dynamic binding

Type issues detected at compile time

Java - still only one bytecode block of code, regardless of number of instantiations

C++ (templates) - version of code for each instantiation

9.10. Design Issues for Functions

Side effects and nature of returned values

9.11. User-Defined Overloaded Operators

Ada, C++, ML

Issues: Arity, types, associativity, infix/functional notation, precedence

Fixed set of symbols to be overloaded? Or can new "original" operators be created?

9.12. Closures

9.13. Coroutines

`http://en.wikipedia.org/wiki/Named_pipe`