CSE 3302 Notes 7: Subroutines & Control Abstraction

(Last updated 10/22/12 2:01 PM)

8.1. Review of Stack Layout

Skim - only slightly deeper than notes 3

8.2. Calling Sequences

Skim - only slightly deeper than notes 3

8.3. Parameter Passing

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
(Scott calls Java situation for objects call-by-sharing)

Pascal - both are available for all objects

By-value - even arrays, structures, and sets will be copied
(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

Simple examples look like call-by-reference

Like macros - don't take call to the function, take function to the call

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` brings recursion to the party ...

Figure 8.3 on page 404

8.4. Generic Subroutines & Modules

Options:

- Macros - C (`Qmacro.c`)
  - # - stringify token
  - ## - concatenate tokens

```
#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; \
}
```

```
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

8.5. Exception Handling

JavaScript -

Many errors lead to error values (`undefined`, `NaN`, `null` . . .)

Very flexible on what gets thrown (`exception.html`)

Others -

Capability of declaring exception objects, but . . .