parameter-passing and memory allocation models

CSE 341

16 August 1999

a quick outline

1. how to talk about procedures
2. how to think about procedures
3. more than one way to pass a parameter
4. stack vs. heap allocation

For more details, read Sethi, Chapter 5.

our old friend the procedure

- reusable piece of code
- has a well-defined purpose
- often takes parameters
- sometimes returns a value

nitpicking: procedures vs. functions

- only functions...

  return a value
  can be used in expressions

- sometimes not distinguished

  C folks call both "functions."
  Pascal, Visual Basic have different keywords.

parameters: formals vs. actuals

- *actual (parameter)*

  what's passed in
  can be an expression

- *formal (parameter)*

  placeholder in body

respect your procedures

Quick! Give me four good reasons not to take procedures away!

- *abstraction*
  informative name hides implementation details

- *implementation hiding*
  can change later without changing interface

- *modularization*
  managing complexity, scale

- *libraries*
  extend language and reuse code

the activation record

...the most common way to get a stack frame

- *activation*
  procedure call, execution of body

- *activation record*
  "storage associated with an activation" (Sethi)
  - one record per activation
  - each with own set of locals, formals

an example in C/C++

```
// function definition
double square(float x)
{
  return x * x;
}


// function call; from some other function
const double PI_SQUARED = square(M_PI);
```

stack frames from C/C++ local scopes

```
int main(void)
{
  int x = 8;

  // a new scope for the sake of example
  {
     // declare local to this scope
     // "shadows" above x
     int x = 11;

     printf("inner scope x == %d\n", x);
  }

  printf("outer scope x == %d\n", x);
  return 0;
}
```


passing by value

- most common of three we'll see today (e.g. C/C++, except when using C++'s `&`)

1. formal is an evaluated copy* of the actual

   *in C++, copy constructor is called, not operator=

2. body executed

```
// classic failed swap in C
void brokenSwap(int x, int y)
{
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```


passing by reference

- FORTRAN
- Pascal and C++ using special syntax:

```
procedure swap(var x : integer, var y : integer);
void referenceSwap(int &x, int &y);
```

"formal parameter becomes [an alias] for the location of an actual parameter" (Sethi)

"Every time you see a formal in the body, replace it in your mind with the actual." (me)

(It doesn't make sense to pass an expression.)

```
// a real, working swap (C++)
void referenceSwap(int &x, int &y)
{
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```


Brace yourselves. This can mess with your mind:

Java passes (references) by value

Remember, everything is a reference or primitive type.

All parameters are passed *by value*, but parameters are often references.

Some experiments to unmess your mind...

```
public class SwapTest {
  public static void main(String[] args) {
    String str1 = new String("August");
    String str2 = new String("11");

    System.out.println("before maybeSwap:  "
                       + str1 + " " + str2);
    maybeSwap(str1, str2);
    System.out.println("after maybeSwap:   "
                       + str1 + " " + str2);

    int n1 = 8;
    int n2 = 11;

    System.out.println("before perhapsSwap:  "
                       + n1 + " " + n2);
    perhapsSwap(n1, n2);
    System.out.println("after perhapsSwap:   "
                       + n1 + " " + n2);
  }

  public static void maybeSwap(String x, String y) {
     String tmp;
     tmp = x;
     x = y;
     y = tmp;
  }

  public static void perhapsSwap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
  }
}
```


SwapTest output

```
before maybeSwap:  August 11
after maybeSwap:   August 11
before perhapsSwap:  8 11
after perhapsSwap:   8 11
```

Hopefully, your brain doesn't hurt too much.

passing by value-result

a.k.a. "passing by copy-in/copy-out"

If actual is not "assignable," just like passing by value. Otherwise...

1. *copy-in* (as when passing by value)
   actual is evaluated, assigned to formal;
   location is saved for later

2. *copy-out*
   final formal value assigned to actual

How's it different? In the presence of...
- ◦ abnormal termination
- ◦ aliasing

by reference vs. by value-result

```
program

    ...

    procedure foo(x, y); begin
        i := y
    end;

    ...

    begin
       i := 2; j := 3;
       foo(i, j);
    end.
```

Remember how Java Slide No. 11 said something about primitive types on the stack, objects on the heap?

stack allocation

- activation records in C/C++, Java, most imperative languages
  - C/C++, can include objects
  - Java, never includes objects
- activation records pile up as in a stack
- locals, formals deallocated with activation record automatically
- C++ ScopeTracer class

stupid (useful, actually) stack allocation tricks

```
class ScopeTracer {
protected:
  char scopeIdentifier[255];

public:
  ScopeTracer(char id[])
  {
     strcpy(scopeIdentifier, id);
     cerr << "ScopeTracer:  entering "
          << scopeIdentifier << endl;
  }

  ~ScopeTracer()
  {
    cerr << "ScopeTracer:  exiting "
         << scopeIdentifier << endl;
  }
};
```


using ScopeTracer

```
void f(void)
{
  ScopeTracer t("f");
}


int main(void)
{
  ScopeTracer t("main");

  {
    ScopeTracer t("main inner");
  }

  f();
}
```

heap allocation

- anything dynamically allocated via `malloc` or `new` in C/C++
- most allocation in Java, except references, primitive types
- heap storage remains allocated until "no longer necessary"
  - not necessarily upon end of activation
  - garbage collected
- can't have closures, first-class functions with stack-allocated activation records

What have we learned today?

1. that Greg really cares about your education
2. the language of procedures: *procedure, function, actual, formal, activation record, stack frame*
3. passing *by value*, *by reference*, *by value-result*
4. allocation on the *stack* and *heap*