CMSC 330: Organization of Programming Languages

Exceptions

Parameter Passing

Preconditions

- Functions often have requirements on their inputs

```
// Return maximum element in A[i..j]
int findMax(int[] A, int i, int j) { ... }
```

  - A is nonempty
  - A isn't null
  - i and j must be nonnegative
  - i and j must be less than A.length
  - i < j (maybe)
- These are called *preconditions*

Dealing with Errors

- What do you do if a precondition isn't met?
- What do you do if something unexpected happens?
  - Try to open a file that doesn't exist
  - Try to write to a full disk

Signaling Errors

- Style 1: Return invalid value

```
// Returns value key maps to, or null if no
// such key in map
Object get(Object key);
```

  - Disadvantages?

Signaling Errors (cont'd)

- Style 2: Return an invalid value and status

```
static int lock_rdev(mdk_rdev_t *rdev) {
  ...
  if (bdev == NULL)
    return -ENOMEM;
  ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```

Problems with These Approaches

- What if all possible return values are valid?
  - E.g., `findMax` from earlier slide
  - What about errors in a constructor?
- What if client forgets to check for error?
  - No compiler support
- What if client can't handle error?
  - Needs to be dealt with at a higher level
- Poor modularity- exception handling code becomes scattered throughout program
- 1996 Ariane 5 failure classic example of this …

Ariane 5 failure

Design issues: In order to save funds and ensure reliability, and since the French Ariane 4 was a successful rocket, the Inertial Reference System (SRI) from Ariane 4 was reused for the Ariane 5.

What happened?: On June 4, 1996 the Ariane 5 launch vehicle failed 39 seconds after liftoff causing the destruction of over $100 million in satellites.

Cause of failure: The SRI, which controls the attitude (direction) of the vehicle by sending aiming commands to the rocket nozzle, sent a bad command to the rocket causing the nozzle to move the rocket toward the horizontal.

The vehicle tried to switch to the backup SRI, but that failed for the same reason 72 millisec earlier.

The vehicle had to then be destroyed.

Why Ariane 5 failed

- SRI tried to convert a floating point number out of range to integer. Therefore it issued an error message (as a 16 bit number). This 16 bit number was interpreted as an integer by the guidance system and caused the nozzle to move accordingly.
  - The backup SRI performed according to specifications and failed for the same reason.
- Ada range checking was disabled since the SRI was supposedly processing at 80% load and the extra time needed for range checking was deemed unnecessary since the Ariane 4 software worked well.
- The ultimate cause of the problem was that the Ariane 5 has a more pronounced angle of attack and can move horizontally sooner after launch. The “bad value” was actually the appropriate horizontal speed of the vehicle.

Better approaches: Exceptions in Java

- On an error condition, we *throw* an exception
- At some point up the call chain, the exception is *caught* and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
  - Like goto, but structured

Throwing an Exception

- Create a new object of the class Exception, and throw it

```
if (i >= 0 && i < a.length )
  return a[i];
throw new ArrayIndexOutOfBounds();
```

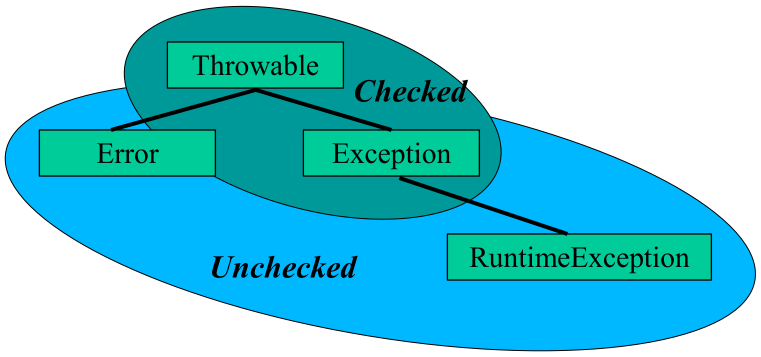
- Exceptions thrown are part of the return type in Java
  - When overriding method in superclass, cannot throw any more exceptions than parent’s version

Method throws declarations

- A method declares the exceptions it might throw
  - public void openNext() throws UnknownHostException, EmptyStackException { … }

- Must declare any exception the method might throw
  - Unless it is caught in (masked by) the method
  - Includes exceptions thrown by called methods
  - Certain kinds of exceptions excluded

Exception Hierarchy



Unchecked Exceptions

- Subclasses of RuntimeException and Error are unchecked
  - Need not be listed in method specifications
- Currently used for things like
  - NullPointerException
  - IndexOutOfBoundsException
  - VirtualMachineError
- Is this a good design?

Exception Handling

- First catch with supertype of the exception catches it
- finally is always executed

```
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBounds e) {
            System.out.println(“a[] out of bounds”); }
catch (MyOwnException e) {
            System.out.println(“Caught my error”); }
catch (Exception e) {
            System.out.println(“Caught” + e.toString());  throw e;  }
finally { /* stuff to do regardless of whether an exception */
            /* was thrown or a return taken */  }
```

Implementing Exceptions in Java

- JVM knows about exceptions, and has built-in mechanism to handle them

```
public class A {
    void foo() {
        try {
            Object f = null;
            f.hashCode();
        }
        catch (NullPointerException e) {
            System.out.println("caught");
        }
    }
}
```

Implementing Exns in Java

```
void foo();
  Code:
   0:    aconst_null
   1:    astore_1
   2:    aload_1
   3:    invokevirtual #2;
         //hashCode
   6:    pop
   7:    goto    19
```

```
 10:  astore_1
 11:  getstatic #4; //System.out
 14:  ldc #5;  //String caught
 16:  invokevirtual #6; //println
 19:  return
Exception table:
 from   to  target type
   0     7    10   NullPointerExn
```

- Exception table tells JVM what handlers there are for which region of code
  - Notice that putting this "off to the side" keeps it out of the main code path
    - (Though less important for an interpreted language)

Implementing Exns in C++

- Design battle: resumption vs. termination
  - Resumption: an exception handler can resume computation at the place where the exception was thrown
  - Termination: throwing an exception terminates execution at the point of the exception
- C++ settled on termination
  - What do you think?

CMSC 330: Organization of Programming Languages

Parameter Passing
and More on Scoping

Order of Evaluation

- Will OCaml raise a Division_by_zero exception?

```
let x = 0

if x != 0 && (y / x) > 100 then
  print_string "OCaml sure is fun"

if x == 0 || (y / x) > 100 then
  print_string "OCaml sure is fun"
```

  – No: && and || are *short-circuiting* in OCaml
    - e1 && e2 evaluates e1. If false, it returns false. Otherwise, it returns the result of evaluating e2
    - e1 || e2 evaluates e1. If true, it returns true. Otherwise, it returns the result of evaluating e2

Order of Evaluation (cont'd)

- C, C++, Java, and Ruby all short-circuit &&, ||
- But some languages don't, like Pascal:

```
x := 0;
...
if (x <> 0) and (y / x > 100) then
  writeln('Sure OCaml is fun');
```

  – So this would need to be written as

```
x := 0;
...
if x <> 0 then
  if y / x > 100 then
    writeln('Sure OCaml is fun');
```

Call-by-Value

- In *call-by-value* (*cbv*), arguments to functions are fully evaluated before the function is invoked
  - Also in OCaml, in let x = e1 in e2, the expression e1 is fully evaluated before e2 is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;
int add(int x, int y) { return r + x + y; }
int set_r(void) {
  r = 3;
  return 1;
}
add(set_r(), 2);
```

Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print?

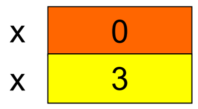
```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

  - Prints 0

Call-by-Value in Imperative Languages, con't.

- Actual parameter is copied to stack location of formal parameter

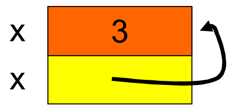
```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Reference

- Alternative idea: Implicitly pass a *pointer* or *reference* to the actual parameter
  - If the function writes to it the actual parameter is modified

```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Reference (cont'd)

- Advantages
  - The entire argument doesn't have to be copied to the called function
    - It's more efficient if you're passing a large (multi-word) argument
    - Can do this without explicit pointer manipulation
  - Allows easy multiple return values
- Disadvantages
  - Can you pass a non-variable (e.g., constant, function result) by reference?
  - It may be hard to tell if a function modifies an argument
  - What if you have *aliasing*?

Aliasing

- We say that two names are *aliased* if they refer to the same object in memory
  - C examples (this is what makes optimizing C hard)

```
int x;
int *p, *q; /*Note that C uses pointers to
                 simulate call by reference */
p = &x;  /* *p and x are aliased */
q = p;   /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }
struct list *p, *q;
...
q = p;   /* *q and *p are aliased */
         /* so are p->x and q->x */
         /* and p->next->x and q->next->x... */
```

Call-by-Reference (cont'd)

- Call-by-reference is still around (e.g., C++), but seems to be less popular in newer languages
  - Older languages (e.g., Fortran, Ada, C with pointers) still use it
  - Possible efficiency gains not worth the confusion
  - "The hardware" is basically call-by-value
    - Although call by reference is not hard to implement and there may be some support for it

Call-by-Value Discussion

- Call-by-value is the standard for languages with side effects
  - When we have side effects, we need to know the order in which things are evaluated, otherwise programs have unpredictable behavior
  - Call-by-reference can sometimes give different results
  - Call-by-value specifies the order at function calls
- But there are alternatives to call by value and call by reference ...

Call-by-Name

- *Call-by-name* (*cbn)*
  - First described in description of Algol (1960)
  - Generalization of Lambda expressions (to be discussed later)
  - Idea simple: In a function:

    Let add x y = x+y

    add (a*b) (c*d)

    Example:
    add (a*b) (c*d) =
    (a*b) + (c*d) ← executed function

    Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively
  - But implementation: Highly complex, inefficient, and provides little improvement over other mechanisms, as later slides demonstrate

Call-by-Name (cont'd)

- In *call-by-name* (*cbn*), arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y

let z = add (add 3 1) (add 4 1)
```

OCaml; cbv; arguments evaluated here

Haskell; cbn; arguments evaluated here

```
add x y = x + y

z = add (add 3 1) (add 4 1)
```

Call-by-Name (cont'd)

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; cbv; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; cbn; never evaluated because parameter is never used

Call by Name Examples

1. P(x) {x = x + x;}
   Y = 2;
   P(Y); ← means Y = Y+Y = 4
   write(Y)

2. But if F(m) {m = m + 1; return m;}
   What is:
   int A[10];
   m = 1;
   P(A[F(m)])

P(A[F(m)]) → A[F(m)] = A[F(m)]+A[F(m)] → A[m++] = A[m++]+A[m++]
→ A[2] = A[3]+A[4]

But: Call by Name Anomalies

3. Write a program to exchange values of X and Y: (e.g., swap(X,Y))
   Usual way: swap(x,y) {t=x; x=y; y=t;}
   – Cannot do it with call by name. Cannot handle both of following: swap(m, A[m]) swap(A[m],m)
   – One of these must fail. Why?

Two Cool Things to Do with CBN

- CBN is also called *lazy evaluation*
  – (CBV is also known as *eager evaluation*)

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build “infinite” data structures

```
integers n = n::(integers (n+1))
take 10 (integers 0)  (* infinite loop in cbv *)
```

Three-Way Comparison

- Consider the following program under the three calling conventions
  - For each, determine i's value and which a[i] (if any) is modified

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

Example: Call-by-Value

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]	f	g
1	0	1	2		
				1	1
				5	2

Example: Call-by-Reference

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2		10	
2		10	

Example: Call-by-Name

```
int i = 1;

void p(int f, int g) {
  g++;        } i++;
  f = 5 * i;  } a[i] = 5*i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]
1	0	1	2
2			10
2			10

The expression a[i] isn't evaluated until needed, in this case after i has changed.

Other Calling Mechanisms

- *Call-by-result*
  - Actual argument passed by reference, but not initialized
  - Written to in function body (and since passed by reference, affects actual argument)
- *Call-by-value-result*
  - Actual argument copied in on call (like cbv)
  - Mutated within function, but does not affect actual yet
  - At end of function body, copied back out to actual
- These calling mechanisms didn't really catch on
  - They can be confusing in cases
  - Recent languages don’t use them

CBV versus CBN

- CBN is flexible- strictly more programs terminate
  - E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate
- Order of evaluation is really hard to see in CBN
  - Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)
- Call-by-name is more expensive since:
  - Functions have to be passed around
  - If you use a parameter twice in a function body, its thunk (the unevaluated argument) will be called twice
    - Haskell actually uses *call-by-need* (each formal parameter is evaluated only once, where it's first used in a function)

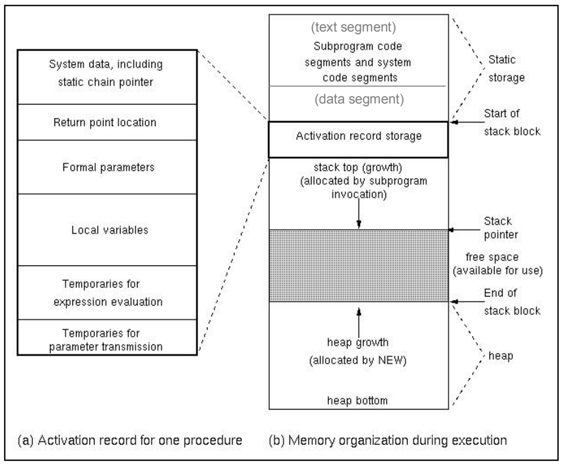
CBV versus CBN (cont'd)

- Call-by-name isn't very "mainstream"
  - Haskell solves these issues by not having side effects
  - But then someone invented "monads" (constructed values that simulate side effects) so you can have side effects in a lazy language
- Call-by-name's benefits may not be worth its cost

How Function Calls Really Work

- Function calls are so important they usually have direct instruction support on the hardware
- We won't go into the details of assembly language programming
  - See CMSC 212, 311, 412, or 430
- But we will discuss just enough to know how functions are called

Machine Model (Generic UNIX)


System data, including static chain pointer
Return point location
Formal parameters
Local variables
Temporaries for expression evaluation
Temporaries for parameter transmission
(text segment)
Subprogram code segments and system code segments
(data segment)
Activation record storage
stack top (growth) (allocated by subprogram invocation)
heap growth (allocated by NEW)
heap bottom
Static storage
Start of stack block
Stack pointer
free space (available for use)
End of stack block
heap

(a) Activation record for one procedure (b) Memory organization during execution

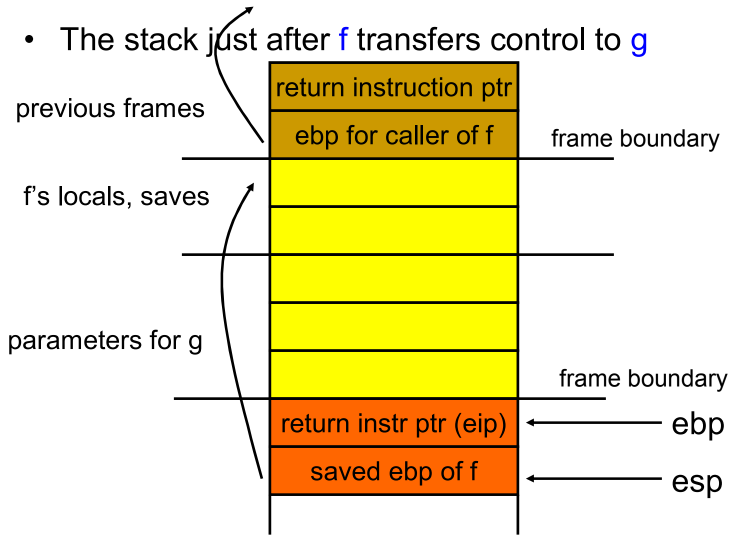
- The *text segment* contains the program's source code
- The *data segment* contains global variables, static data (data that exists for the entire execution and whose size is known), and the heap
- The *stack segment* contains the activation records for functions

Machine Model (x86)

- The CPU has a fixed number of *registers*
  - Think of these as memory that's really fast to access
  - For a 32-bit machine, each can hold a 32-bit word
- Important x86 registers
  - eax generic register for computing values
  - esp pointer to the top of the stack
  - ebp pointer to start of current stack frame
  - eip the program counter (points to next instruction in text segment to execute)

The x86 Stack Frame/Activation Record

- The stack just after f transfers control to g



Based on Fig 6-1 in Intel ia-32 manual

x86 Calling Convention

- To call a function
  - Push parameters for function onto stack
  - Invoke CALL instruction to
    - Push current value of eip onto stack
      - I.e., save the program counter
    - Start executing code for called function
  - Callee pushes ebp onto stack to save it
- When a function returns
  - Put return value in eax
  - Invoke LEAVE to pop stack frame
    - Set esp to ebp
    - Restore ebp that was saved on stack and pop it off the stack
  - Invoke RET instruction to load return address into eip
    - I.e., start executing code where we left off at call

Example

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

int main(void) {
  int x;

  x = f(3, 4);
}
```

gcc -S a.c

```
f:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:
        ...
        subl    $8, %esp
        pushl   $4
        pushl   $3
        call    f
     l: addl    $16, %esp
        movl    %eax, -4(%ebp)
        leave
        ret
```

Lots More Details

- There’s a whole lot more to say about calling functions
  - Local variables are allocated on stack by the callee as needed
    - This is usually the first thing a called function does
  - Saving registers
    - If the callee is going to use eax itself, you’d better save it to the stack before you call
  - Passing parameters in registers
    - More efficient than pushing/popping from the stack
  - Etc...
- See other courses for more details