Programming Languages

Sessions 9 & 10 – Main Theme
Control Abstractions
Concurrency
Dynamic Allocation and Garbage Collection

Dr. Jean-Claude Franchitti
New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

1. Session Overview
2. Control Abstractions
3. Concurrency
4. Dynamic Allocation and Garbage Collection
5. Conclusion

Adapted from course textbook resources
Programming Language Pragmatics (3rd Edition)
Michael L. Scott, Copyright © 2009 Elsevier
What is the course about?

- Course description and syllabus:
  - [http://www.nyu.edu/classes/jcf/g22.2110-001](http://www.nyu.edu/classes/jcf/g22.2110-001)

- Textbook:
  - *Programming Language Pragmatics (3rd Edition)*
    - Michael L. Scott
    - Morgan Kaufmann

- Additional References:
  - Osinski, Lecture notes, Summer 2010
  - Grimm, Lecture notes, Spring 2010
  - Gottlieb, Lecture notes, Fall 2009
  - Barrett, Lecture notes, Fall 2008

Session Agenda

- Session Overview
- Control Abstractions
  - Generic Programming and Templates (C++, Java, ML, etc.)
  - Containers and Iteration
  - Exception Handling
  - Continuations
- Concurrency
  - Threads (Ada, Java, etc.)
  - Tasks synchronization
  - Communication
- Dynamic Allocation and Garbage Collection
  - Mark/sweep, copying, reference counting
- Conclusion
Icons / Metaphors

- Information
- Common Realization
- Knowledge/Competency Pattern
- Governance
- Alignment
- Solution Approach

Session 7 & 8 Review

- Program Structure
  - Key Concepts
    - Review of Session 2
  - Software Complexity
  - Modules
  - Modularity Constructs
  - Visibility – Koenig Lookup
- Object-Oriented Programming
  - Key Concepts
    - Review Session 6
  - What is OOP
  - OOP Features
  - OOP Programming
  - Encapsulation and Inheritance
  - Initialization and Finalization
  - Dynamic Method Binding
  - Multiple Inheritance
- A Quick Survey of Various Languages
- Conclusions
Agenda

1. Session Overview
2. Control Abstractions
3. Concurrency
4. Dynamic Allocation and Garbage Collection
5. Conclusion

Control Abstractions

- Generic Programming and Templates (C++, Java, ML, etc.)
- Containers and Iteration
- Exception Handling
- Continuations
- Textbook:
  » Sections 8.4, 8.5
Generic Programming

- Let’s us abstract over types and other non-value entities.
- Examples:
  - A sorting algorithm has the same structure, regardless of the types being sorted
  - Stack primitives have the same semantics, regardless of the objects stored on the stack.
- One common use:
  - algorithms on containers: updating, iteration, search
- Language models:
  - C: macros (textual substitution) or unsafe casts
  - Ada: generic units and instantiations
  - C++, Java, C#: templates
  - ML: parametric polymorphism, functors

---

Parametrizing Software Components

<table>
<thead>
<tr>
<th>Construct</th>
<th>parameter(s) bound to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>bounds, element type</td>
</tr>
<tr>
<td>subprogram</td>
<td>values (arguments)</td>
</tr>
<tr>
<td>Ada generic package</td>
<td>values, types, packages</td>
</tr>
<tr>
<td>Ada generic subprogram</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ class template</td>
<td>values, types</td>
</tr>
<tr>
<td>C++ function template</td>
<td>values, types</td>
</tr>
<tr>
<td>Java generic</td>
<td>classes</td>
</tr>
<tr>
<td>ML function</td>
<td>values (including other functions)</td>
</tr>
<tr>
<td>ML type constructor</td>
<td>types</td>
</tr>
<tr>
<td>ML functor</td>
<td>values, types, structures</td>
</tr>
</tbody>
</table>
Templates in C++

template <typename T>
class Vector {
public:
    explicit Vector (size_t); // constructor
    T& operator[] (size_t);     // subscript operator
    ... // other operations
private:
    ... // a size and a pointer to an array
};

Vector<int> V1(100);    // instantiation
Vector<int> V2;          // use default constructor

typedef Vector<employee> Dept; // named instance

Class and Value Parameters

template <typename T, unsigned int i>
class Buffer {
    T v[i];               // storage for buffer
    unsigned int sz;      // total capacity
    unsigned int count;   // current contents
public:
    Buffer () : sz(i), count(0) { }
    T read ();
    void write (const T& elem);
};

Buffer<Shape *, 100> picture;
Template Hopes for the Best

```cpp
template <typename T> class List {
    struct Link { // for a list node
        Link *pre, *succ; // doubly linked T val;
        Link (Link *p, Link *s, const T& v) : pre(p), succ(s), val(v) { }
    };
    Link *head;
    public:
    void print (std::ostream& os) {
        for (Link *p = head; p; p = p->succ) // operator<< must exist for T
            // if print will be used.
            os << p->val << "\n";
    }
};
```

Function Templates

- Instantiated implicitly at point of call:

```cpp
template <typename T>
void sort (vector<T>&) { ... }

void testit (vector<int>& vi) {
    sort(vi); // implicit instantiation
    // can also write sort<int>(vi);
}
```
Functions and Function Templates

- Templates and regular functions overload each other:

```cpp
template <typename T> class Complex {...};

template <typename T> T sqrt (T); // template

template <typename T> Complex<T> sqrt (Complex<T>); // different algorithm
double sqrt (double); // regular function

void testit (complex<double> cd) {
    sqrt(2); // sqrt<int>
    sqrt(2.0); // sqrt<double>: regular function
    sqrt(cd); // sqrt<complex<double>>
}
```

Iterators and Containers

- Containers are data structures to manage collections of items
- Typical operations: insert, delete, search, count
- Typical algorithms over collections use:
  - Imperative languages: iterators
  - Functional languages: map, fold

```
interface Iterator<E> {
    boolean hasNext (); // returns true if there are
    // more elements
    E next (); // returns the next element
    void remove (); // removes the current element
    // from the collection
};
```
The Standard Template Library

- STL: A set of useful data structures and algorithms in C++, mostly to handle collections
  - Sequential containers: list, vector, deque
  - Associative containers: set, map
- We can iterate over these using (what else?) iterators
- Iterators provided (for vector<T>):

  ```
  vector<T>::iterator
  vector<T>::const_iterator
  vector<T>::reverse_iterator
  vector<T>::const_reverse_iterator
  ```

- Note: Almost no inheritance used in STL

Iterators in C++

- For standard collection classes, we have member functions begin and end that return iterators
- We can do the following with an iterator p (subject to restrictions):
  - "p "Dereference" it to get the element it points to
  - ++p, p++ Advance it to point to the next element
  - --p, p-- Retreat it to point to the previous element
  - p+i Advance it i times
  - p-i Retreat it i times
- A sequence is defined by a pair of iterators:
  - the first points to the first element in the sequence
  - the second points to one past the last element in the sequence
- There is a wide variety of operations that work on sequences.
Iterator Example

```cpp
#include <vector>
#include <string>
#include <iostream>

int main () {
    using namespace std;
    vector<string> ss(20); // initialize to 20 empty strings
    for (int i = 0; i < 20; i++)
        ss[i] = string(1, 'a'+i); // assign "a", "b", etc.
    vector<string>::iterator loc = find(ss.begin(), ss.end(), "d"); // find first "d"
    cout << "found: " << *loc
            << " at position " << loc - ss.begin() << endl;
}
```

STL Algorithms – Part I

- STL provides a wide variety of standard "algorithms" on sequences
- Example: finding an element that matches a given condition

```cpp
// Find first 7 in the sequence
list<int>::iterator p = find(c.begin(), c.end(), 7);

// Find first number less than 7 in the sequence
bool less_than_7 (int v) {
    return v < 7;
}

list<int>::iterator p = find_if(c.begin(), c.end(), less_than_7);
```
Example: doing something for each element of a sequence
It is often useful to pass a function or something that acts like a function:

```cpp
template <typename T>
class Sum {
    T res;
public:
    Sum (T i = 0) : res(i) {} // initialize
    void operator() (T x) { res += x; } // accumulate
    T result () const { return res; } // return sum
};

void f (list<double>& ds) {
    Sum<double> sum;
    sum = for_each(ds.begin(), ds.end(), sum);
    cout << "the sum is " << sum.result() << endl;
}
```

Function Objects

```cpp
template <typename Arg, typename Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

struct R { string name; ... };

class R_name_eq : public unary_function<R, bool> {
    string s;
public:
    explicit R_name_eq (const string& ss) : s(ss) {}
    bool operator() (const R& r) const { return r.name == s; }
};

void f (list<R>& lr) {
    list<R>::iterator p = find_if(lr.begin(), lr.end(),
        R_name_eq("Joe");
    ...
}
Binary Function Objects

```cpp
template <typename Arg, typename Arg2, typename Res>
struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};

template <typename T>
struct less : public binary_function<T,T,bool> {
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
};
```

Currying with Function Objects

```cpp
template <typename BinOp>
class binder2nd
    : public unary_function<typename BinOp::first_argument_type, typename BinOp::result_type> {
protected:
    BinOp op;
    typename BinOp::second_argument_type arg2;
public:
    binder2nd (const BinOp& x, const typename BinOp::second_argument_type& v) :
        op(x), arg2(v) {}
    return_type operator() (const argument_type& x) const {
        return op(x, arg2);
    }
};
template <typename BinOp, typename T>
binder2nd<BinOp> bind2nd (const BinOp& op, const T& v) {
    return binder2nd<BinOp> (op, v);
}```
“Is this readable? ... The notation is logical, but it takes some getting used to.” – Stroustrup, p. 520

Equivalent to the following in ML:

```ml
fun f xs limit =
    let val optNum = List.find (fn x => x < limit) xs
    val num = Option.getOpt (optNum, limit)
in ...
end
```

Templates in C++ allow for arbitrary computation to be done at compile time!

```cpp
template <int N> struct Factorial {
    enum { V = N * Factorial<N-1>::V };}

template <> struct Factorial<1> {
    enum { V = 1 };}

void f () {
    const int fact12 = Factorial<12>::V;
    cout << fact12 << endl; // 479001600
}
Generics in Java

- Only class parameters
- Implementation by type erasure: all instances share the same code

```java
interface Collection <E> {
    public void add (E x);
    public Iterator<E> iterator ();
}

Collection <Thing> is a parametrized type
Collection (by itself) is a raw type!
```

Generic Methods in Java

```java
class Collection <A extends Comparable<A>> {
    public A max () {
        Iterator<A> xi = this.iterator();
        A biggest = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (biggest.compareTo(x) < 0)
                biggest = x;
        }
        return biggest;
    }
    ...
}
```
Functors in ML

- Why functors, when we have parametric polymorphic functions and type constructors (e.g., containers)?
  - Functors can take structures as arguments. This is not possible with functions or type constructors.
  - Sometimes a type needs to be parameterized on a value. This is not possible with type constructors.

Example Functor – The Signature

```
signature SET =
sig
  type elem
  type set
  val empty : set
  val singleton : elem -> set
  val member : elem * set -> bool
  val union : set * set -> set
  ...
end
```
Example Functor – The Implementation

functor SetFn (type elem
   val compare : elem * elem -> order) : SET =

structure
  type elem = elem
  datatype set = EMPTY
  | SINGLE of elem
  | PAIR of set * set

  val empty = EMPTY
  val singleton = SINGLE

  fun member (e, EMPTY) = false
  | member (e, SINGLE e') = compare (e, e') = EQUAL
  | member (e, PAIR (s1,s2)) = member (e, s1) orelse member (e, s2)

  ...
end

Example Functor – The Instantiation

structure IntSet =
    SetFn (type elem = int
            compare = Int.compare)

structure StringSet =
    SetFn (type elem = string
            compare = String.compare)

fun cmp (is1, is2) = ...

structure IntSetSet = SetFn (type elem = IntSet.set
                               compare = cmp)

- Compare functor implementation with a polymorphic type: how are element comparisons done?
Generics in Ada95

- I/O for integer types.
- Identical implementations, but need separate procedures for strong-typing reasons.

```ada
generic
  type Elem is range <>; -- any integer type
package Integer_IO is
  procedure Put (Item: Elem);
  ...
end Integer_IO;
```

A Generic Package

```ada
generic
  type Elem is private; -- parameter
package Stacks is
  type Stack is private;
  procedure Push (X: Elem; On: in out Stack);
  ...
private
  type Cell; -- linked list
  type Stack is access Cell; -- representation
  type Cell is record
    Val: Elem;
    Next: Ptr;
  end record;
end Stacks;
```
The syntax is: type T is ...;

- Within the generic, the operations that apply to any type of the class can be used.
- The instantiation must use a specific type of the class.
A Generic Function

```plaintext
generic
type T is range <>; -- parameter of some integer type
type Arr is array (Integer range <>) of T;
    -- parameter is array of those
function Sum_Array (A: Arr) return T;

-- Body identical to non-generic version
function Sum_Array (A: Arr) return T is
  Result: T := 0; -- some integer type, so 0 is legal
begin
  for J in A'range loop -- array: 'range available
    Result := Result + A(J); -- integer: "+" available
  end loop;
  return Result;
end;
```

Instantiating a Generic Function

```plaintext
type Apple is range 1..2**15 - 1;
type Production is array (1..12) of Apple;
type Sick_Days is range 1..5;
type Absences is array (1..52) of Sick_Days;

function Get_Crop is new Sum_Array (Apple, Production);
function Lost_Work is new Sum_Array (Sick_Days, Absences);
```
Generic Private Types

- The only available operations are assignment and equality

```plaintext
generic
  type T is private;
  procedure Swap (X, Y: in out T);

procedure Swap (X, Y: in out T) is
  Temp: constant T := X;
begin
  X := Y;
  Y := Temp;
end Swap;
```

Subprogram Parameters

- A generic sorting routine should apply to any array whose components are comparable, i.e., for which an ordering predicate exists. This class includes more than the numeric types:

```plaintext
generic
  type T is private;
with function "<" (X, Y: T) return Boolean;
  type Arr is array (Integer range <>) of T;
procedure Sort (A: in out Arr);
```
Supplying Subprogram Parameters

- The actual must have a matching signature, not necessarily the same name:

```plaintext
procedure Sort_Up is
  new Sort (Integer, "<", ...);

procedure Sort_Down is
  new Sort (Integer, ">", ...);

type Employee is record ... end record;
function Senior (E1, E2: Employee) return Boolean;
function Rank is new Sort (Employee, Senior, ...);
```

Value Parameters

- Useful to parameterize containers by size:

```plaintext
generic
  type Elem is private; -- type parameter
  Size: Positive; -- value parameter
package Queues is
  type Queue is private;
  procedure Enqueue (X: Elem; On: in out Queue);
  procedure Dequeue (X: out Elem; From: in out Queue);
  function Full (Q: Queue) return Boolean;
  function Empty (Q: Queue) return Boolean;
private
  type Contents is array (Natural range <>) of Elem;
  type Queue is record
    Front, Back: Natural;
    C: Contents (0 .. Size);
  end record;
end Queues;
```
Packages as Parameters

- We also want to define a package for elementary functions (sin, cos, etc.) on complex numbers. This needs the complex operations, which are parameterized by the corresponding real value.

The Instantiation Requires an Instance of the Package Parameter

- Instantiate complex types with long_float components:

  ```plaintext
package Long_Complex is
  new Generic_Complex_Type (long_float);
  ```

- Instantiate complex functions for long_complex types:

  ```plaintext
package Long_Complex_Functions is
  new Generic_Complex_Functions (long_complex);
  ```
Generics Review

- Basic difference between Java, C++, and Ada styles
- Benefits and limitations of Java style
- When do you use bounded types in contracts – e.g., \(<T \text{ extends } X>\)?
  - When the generic method needs to call a method in \(X\)
- When do I use wildcard types in interfaces e.g. \(\text{List<}\text{? extends Foo}<\text{>}\)?
  - When a list of any subclass of \(\text{Foo}\) will work; I'm reading \(\text{Foo}\)'s from them.
- What are the type-checking rules for parameterized types? Can I assign \(\text{List<ColoredPoint>}\) to variable of type \(\text{List<Point>}\)?
  - No. But you may assign \(\text{List<ColoredPoint>}\) to a variable of type \(\text{List<}\text{? extends Point}<\text{>}\)
  - There may be, and in this case, there is, an operation add. You don't want to add a \(\text{Point}\) to a variable of type \(\text{List<Point>}\) that actually has a value of \(\text{class List<ColoredPoint>}\)
- When do I use generic methods – e.g.,
  - \(\text{public }<\text{X}> \text{List<X>} \text{makeSingle(X element);}\)
  - When there is a constraint between the types, in this case, the element type and the \(\text{List}\) type.
- What different things can I do with Ada generics?

Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors
Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception

- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions
  - explicitly in the program everywhere they might occur

Exceptions

- General mechanism for handling abnormal conditions

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>How raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>predefined</td>
<td>constraint violations,</td>
<td>by the runtime system</td>
</tr>
<tr>
<td></td>
<td>I/O errors,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>communication errors,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>other illegalities</td>
<td></td>
</tr>
<tr>
<td>user-defined</td>
<td>pop from empty stack</td>
<td>explicitly by user code</td>
</tr>
<tr>
<td></td>
<td>explicit illegalities</td>
<td></td>
</tr>
</tbody>
</table>

- Exception handlers specify remedial actions or proper shutdown
- Exceptions can be stored and re-raised later
One way to improve robustness of programs is to handle errors. How can we do this?

We can check the result of each operation that can go wrong (e.g., popping from a stack, writing to a file, allocating memory)

Unfortunately, this has a couple of serious disadvantages:

1. it is easy to forget to check
2. writing all the checks clutters up the code and obfuscates the common case (the one where no errors occur)

Exceptions let us write clearer code and make it easier to catch errors

---

Defined in Standard:

- Constraint_Error : value out of range
- Program_Error : illegality not detectable at compile-time:
  - unelaborated package, exception during finalization, etc.
- Storage_Error : allocation cannot be satisfied (heap or stack)
- Tasking_Error : communication failure

Defined in Ada.IO Exceptions:

- Data_Error, End_Error, Name_Error, Use_Error, Mode_Error, Status_Error, Device_Error
### Handling Exceptions

- Any begin-end block can have an exception handler:

```plaintext
procedure Test is
  X: Integer := 25;
  Y: Integer := 0;
begin
  X := X / Y;
exception
  when Constraint_Error =>
    Put_Line("did you divide by 0?");
  when others =>
    Put_Line("out of the blue!");
end;
```

### A Common Idiom

```plaintext
function Get_Data return Integer is
  X: Integer;
begin
  loop
    begin
      Get(X);
      return X;  -- if got here, input is valid,  -- so leave loop
    exception
      when others =>
        Put_Line("input must be integer, try again");
        -- will restart loop to wait for a good input
    end;
  end loop;
end;
```
User-Defined Exceptions

package Stacks is
  Stack_Empty: exception;
  ...
end Stacks;

package body Stacks is
  procedure Pop (X: out Integer;
                 From: in out Stack) is
  begin
    if Empty(From)
      then raise Stack_Empty;
    else ...
  end Pop;
  ...
end Stacks;

The Scope of Exceptions

- An exception has the same visibility as other declared entities: to handle an exception it must be visible in the handler (e.g., caller must be able to see Stack_Empty)
- An others clause can handle unnamable exceptions partially

when others =>
  Put_Line("disaster somewhere");
  raise; -- propagate exception,
          -- program will terminate
**Exception Run-Time Model**

- How to propagate an exception:
  1. When an exception is raised, the current sequence of statements is abandoned (e.g., current Get and return in example)
  2. Starting at the current frame, if we have an exception handler, it is executed, and the current frame is completed
  3. Otherwise, the frame is discarded, and the enclosing dynamic scopes are examined to find a frame that contains a handler for the current exception (want dynamic as opposed to static scopes because those are values that caused the problem)
  4. If no handler is found, the program terminates
- Note: The current frame is never resumed

**Exception Information**

- An Ada exception is a label, not a value: we cannot declare exception variables and assign to them
- But an exception occurrence is a value that can be stored and examined
- An exception occurrence may include additional information: source location of occurrence, contents of stack, etc.
- Predefined package Ada.Exceptions contains needed machinery
package Ada.Exceptions is
  type Exception_Id is private;
  type Exception_Occurrence is limited private;

  function Exception_Identity (X: Exception_Occurrence) return Exception_Id;
  function Exception_Name (X: Exception_Occurrence) return String;

  procedure Save_Occurrence
    (Target: out Exception_Occurrence;
     Source: Exception_Occurrence);
  procedure Raise_Exception (E: Exception_Id;
                            Message: in String := "")
    ...
end Ada.Exceptions;

begin
  ...
exception
  when Expected: Constraint_Error =>
    -- Expected has details
    Save_Occurrence(Event_Log, Expected);

  when Trouble: others =>
    Put_Line("unexpected " &
             Exception_Name(Trouble) &
             " raised");
    Put_Line("shutting down");
    raise;
end;
Exceptions in C++

- Similar runtime model,...
- But exceptions are bona-fide values,
- Handlers appear in try/catch blocks

```cpp
try {
    some_complex_calculation();
} catch (const RangeError& e) {
    // RangeError might be raised
    // in some_complex_calculation
    cerr << "oops\n";
} catch (const ZeroDivide& e) {
    // same for ZeroDivide
    cerr << "why is denominator zero?\n";
}
```

Defining and Throwing Exceptions

- The program throws an object. There is nothing needed in the declaration of the type to indicate it will be used as an exception

```cpp
struct ZeroDivide {
    int lineno;
    ZeroDivide(...) { ... } // constructor
    ...
};
...
if (x == 0)
    throw ZeroDivide(...); // call constructor
    // and go
A handler names a class, and can handle an object of a derived class as well:

```java
class Matherr {}; // a bare object, no info
class Overflow : public Matherr {...};
class Underflow : public Matherr {...};
class ZeroDivide : public Matherr {...};

try {
    weatherPredictionModel(...);
} catch (const Overflow& e) {
    // e.g., change parameters in caller
} catch (const Matherr& e) {
    // Underflow, ZeroDivide handled here
} catch (...) {
    // handle anything else (ellipsis)
}
```

- Model and terminology similar to C++:
  - Exceptions are objects that are thrown and caught
  - Try blocks have handlers, which are examined in succession
  - A handler for an exception can handle any object of a derived class

- Differences:
  - All exceptions are extensions of predefined class Throwable
  - Checked exceptions are part of method declaration
  - The finally clause specifies clean-up actions
    - in C++, cleanup actions are idiomatically done in destructors
Any class extending Exception is a checked exception
System errors are extensions of Error; these are unchecked exceptions
- Checked exceptions must be either handled or declared in the method that throws them; this is checked by the compiler.

```
public void replace(String name, Object newValue) throws NoSuch
{
    Attribute attr = find(name);
    if (attr == null) throw new NoSuch(name);
    newValue.update(attr);
}
```
Mandatory Cleanup Actions

- Some cleanups must be performed whether the method terminates normally or throws an exception.

```java
public void parse (String file) throws IOException {
    BufferedReader input =
    new BufferedReader(new FileReader(file));
    try {
        while (true) {
            String s = input.readLine();
            if (s == null) break;
            parseLine(s); // may fail somewhere
        }
    } finally {
        if (input != null) input.close();
    } // regardless of how we exit
}
```

Exceptions in ML

- Runtime model similar to Ada/C++/Java
- Exception is a single type (like a datatype but dynamically extensible)
- Declaring new sorts of exceptions:

  ```ml
  exception StackUnderflow
  exception ParseError of { line: int, col: int }
  ```

- raising an exception:

  ```ml
  raise StackUnderflow
  raise (ParseError { line = 5, col = 12 })
  ```

- handling an exception:

  ```ml
  expr1 handle pattern => expr2
  ```

- If an exception is raised during evaluation of `expr1`, and pattern matches that exception, `expr2` is evaluated instead.
Typing issues:
- The type of the body and the handler must be the same
- The type of a raise expression can be any type (whatever type is appropriate is chosen)

Call-With-Current-Continuation

- Available in Scheme and SML/NJ; usually abbreviated to call/cc
- In Scheme, it is called call-with-current-continuation.
- A continuation represents the computation of “rest of the program”.
- call/cc takes a function as an argument. It calls that function with the current continuation (which is packaged up as a function) as an argument. If this continuation is called with some value as an argument, the effect is as if call/cc had itself returned with that argument as its result
- The current continuation is the “rest of the program”, starting from the point when call/cc returns.

```
(call/cc (lambda (c) (c 5))) ;; returns 5
(call/cc (lambda (c) 5)) ;; so does this
(call/cc (lambda (c) (+ 1 (c 5)))) ;; ditto
```
The Power of Continuations

- We can implement many control structures with call/cc:
  - return:
    
    ```scheme
    (lambda (x)
      (call/cc (lambda (ret)
        ... ;; body of function
        (ret 76) ;; call continuation with result
        ...
      ))
    )
    ```

- goto:
  
  ```scheme
  (begin
    ...
    (call/cc (lambda (k) (set! here k)) ;; set label
    ...
    (here ()) ;; ‘goto’ here
    ...
  )
  )
  ```

Exception Via Call / CC

- Exceptions can also be implemented by call/cc:
  - Need global stack: handlers
  - For each try/catch:
    
    ```scheme
    (call/cc (lambda (k)
      (begin
        (push handlers (lambda ()
          (begin
            (pop handlers)
            (catch-block)
            (k ())))))
        (try-block)
        (pop handlers)))))
    ```

- For each raise:
  
  ```scheme
  ((top handlers))) ; call the top function on the handlers stack
  ```
One word, two different meanings – the word “serialization”

- In databases:
  » The property of transactions that they appear as if they had executed in a serial order, that is, without overlap
  » Part of the requirement for atomicity
- In distributed RPC/RMI/message passing:
  » A technique for converting a message or argument list containing a set of typed objects on one machine into a bitstring that can be sent to another machine, parsed, and reconstructed as a corresponding message or argument list in another machine
  » Also called marshalling, and its counterpart demarshalling

Agenda

1. Session Overview
2. Control Abstractions
3. Concurrency
4. Dynamic Allocation and Garbage Collection
5. Conclusion
Introduction to Concurrency

- Background and Motivation
- Concurrent programming
- Declaration, creation, activation, termination
- Synchronization and communication
- Time and delays
- Conditional communication
- Non-determinism

Textbook:
» Section 12

What are safe, regular, and atomic registers? What read results are possible for a given sequence of writes for each kind?

- What is a test-and-set or compare-and-swap instruction?
- What is a semaphore? What are the P and V operations? How can they be implemented using compare-and-swap?
- What is a critical region?
- How are critical regions used for safe programming?
- How are producer-consumer queues used for safe programming?
Concurrency, continued

- What is a monitor?
- How is that implemented in C++, Java, Ada?
- Understand these concepts:
  » Synchronized method (Java)
  » Ada entry
  » Rendezvous call
  » Guarded select statement
- How are concurrent, conditionally available resources programmed in Java, Ada?
- Be able to write a guarded select in Ada, or a wait/notify in Java, and to spot and correct bugs in these programs
- What is a memory model? What are the advantages of a memory model with strong guarantees? What are the disadvantages?
- How can you assure thread safety by programming patterns? By compile-time enforcement?

Background and Motivation

- A PROCESS or THREAD is a potentially-active execution context
- Classic von Neumann (stored program) model of computing has single thread of control
- Parallel programs have more than one
- A process can be thought of as an abstraction of a physical PROCESSOR
Background and Motivation

- Processes/Threads can come from
  - multiple CPUs
  - kernel-level multiplexing of single physical machine
  - language or library level multiplexing of kernel-level abstraction

- They can run
  - in true parallel
  - unpredictably interleaved
  - run-until-block

- Most work focuses on the first two cases, which are equally difficult to deal with

Background and Motivation

- Two main classes of programming notation
  - synchronized access to shared memory
  - message passing between processes that don't share memory

- Both approaches can be implemented on hardware designed for the other, though shared memory on message-passing hardware tends to be slow
Race conditions

» A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen

» Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue)

Race conditions (we want to avoid race conditions):

» Suppose processors A and B share memory, and both try to increment variable X at more or less the same time

» Very few processors support arithmetic operations on memory, so each processor executes
  – LOAD X
  – INC
  – STORE X

» Suppose X is initialized to 0. If both processors execute these instructions simultaneously, what are the possible outcomes?
  • could go up by one or by two
**Background and Motivation**

- **Synchronization**
  - SYNCHRONIZATION is the act of ensuring that events in different processes happen in a desired order.
  - Synchronization can be used to eliminate race conditions.
  - In our example, we need to synchronize the increment operations to enforce MUTUAL EXCLUSION on access to X.
  - Most synchronization can be regarded as either:
    - Mutual exclusion (making sure that only one process is executing a CRITICAL SECTION [touching a variable, for example] at a time), or as
    - CONDITION SYNCHRONIZATION, which means making sure that a given process does not proceed until some condition holds (e.g., that a variable contains a given value).

- **One might be tempted to think of mutual exclusion as a form of condition synchronization (the condition being that nobody else is in the critical section), but it isn't**
  - The distinction is basically existential v. universal quantification.
  - Mutual exclusion requires multi-process consensus.

- **We do NOT in general want to over-synchronize**
  - That eliminates parallelism, which we generally want to encourage for performance.

- **Basically, we want to eliminate "bad" race conditions**, i.e., the ones that cause the program to give incorrect results.
Historical development of shared memory ideas

To implement synchronization you have to have something that is ATOMIC
- that means it happens all at once, as an indivisible action
- In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses)
- In early machines, reads and writes of individual memory locations were all that was atomic

To simplify the implementation of mutual exclusion, hardware designers began in the late 60's to build so-called read-modify-write, or fetch-and-phi, instructions into their machines

Background and Motivation

Tasking

- concurrent programming
- declaration, creation, activation, termination
- synchronization and communication
- time and delays
- conditional communication
- non-determinism
Synchronous and asynchronous models of communication
Description of concurrent, independent activities
A task is an independent thread of control, with own stack, program counter and local environment.
Ada tasks communicate through
  » rendezvous (think "meeting someone for a date")
  » protected objects
  » shared variables
Java threads communicate through shared objects (preferably synchronized)
C++ has no core language support for concurrency

SCHEDULERS give us the ability to "put a thread/process to sleep" and run something else on its process/processor
  » start with coroutines
  » make uniprocessor run-until-block threads
  » add preemption
  » add multiple processors
Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.

Coroutines can be used to implement:
- iterators (Section 6.5.3)
- threads (to be discussed in Chapter 12)

Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack.

Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)
Figure 12.5 Life time of concurrent threads. With co-begin, parallel loops, or launch-at-execution (a), threads are always properly nested. With fork/join (b), more general patterns are possible.

Figure 12.6 Two-level implementation of threads. A thread scheduler implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler implemented in the operating system kernel, multiplexes processes on top of one or more physical processors.
Coroutines

- Multiple execution contexts, only one of which is active
- Transfer (other):
  - save all callee-saves registers on stack, including ra and fp
  - *current := sp
  - current := other
  - sp := *current
  - pop all callee-saves registers (including ra, but NOT sp!)
  - return (into different coroutine!)
- Other and current are pointers to CONTEXT BLOCKs
  - Contains sp; may contain other stuff as well (priority, I/O status, etc.)
- No need to change PC; always changes at the same place
- Create new coroutine in a state that looks like it's blocked in transfer. (Or maybe let it execute and then "detach". That's basically early reply)

Run-until block threads on a single process

- Need to get rid of explicit argument to transfer
- Ready list data structure: threads that are runnable but not running

```plaintext
procedure reschedule:
  t : cb := dequeue(ready_list)
  transfer(t)
```

- To do this safely, we need to save 'current' somewhere - two ways to do this:
  - Suppose we're just relinquishing the processor for the sake of fairness (as in MacOS or Windows 3.1):
    ```plaintext
    procedure yield:
      enqueue (ready_list, current)
      reschedule
    ```
  - Now suppose we're implementing synchronization:
    ```plaintext
    sleep_on(q)
    enqueue(q, current)
    reschedule
    ```
- Some other thread/process will move us to the ready list when we can continue
Preemption

» Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields

» Requires that we protect the scheduler data structures:

```plaintext
procedure yield:
  disable_signals
  enqueue(ready_list, current)
  Reschedule
  re-enable_signals
```

» Note that reschedule takes us to a different thread, possibly in code other than yield. Invariant: EVERY CALL to reschedule must be made with signals disabled, and must re-enable them upon its return

```plaintext
disable_signals
if not <desired condition>
  sleep_on <condition queue>
re-enable_signals
```
**Multiprocessors**

Disabling signals doesn't suffice:

```plaintext
procedure yield:
  disable_signals
  acquire(scheduler_lock)  // spin lock
  enqueue(ready_list, current)
  reschedule
  release(scheduler_lock)
  re-enable_signals

  disable_signals
  acquire(scheduler_lock)  // spin lock
  if not <desired condition>
    sleep_on <condition queue>
  release(scheduler_lock)
  re-enable signals
```

**Task Declaration**

- A task type is a limited type

```plaintext
task type Worker:     -- declaration;  
                     -- public interface

  type Worker_Id is access Worker;

  task body Worker is  -- actions performed in lifetime
                       begin
                          loop   -- Runs forever;
                          compute;  -- will be shutdown
                          end loop;  -- from the outside.
                       end Worker;
```
More Task Declarations

- A task type can be a component of a composite
- Number of tasks in a program is not fixed at compile-time

```plaintext
W1, W2: Worker; -- two individual tasks
type Crew is array (Integer range <>) of Worker;
First_Shift: Crew (1 .. 10); -- group of tasks
type Monitored is record
  Counter: Integer;
  Agent: Worker;
end record;
```

Task Activation

- When does a task start running?
  - if statically allocated => at the next begin
  - if dynamically allocated => at the point of allocation

```plaintext
declare
  W1, W2: Worker;
  Joe: Worker_Id := new Worker; -- Starts working now
  Third_Shift: Crew(1..N); -- N tasks
begin    -- activate W1, W2, and the Third_Shift
  ...
end;    -- wait for them to complete
         -- Joe will keep running
```
### Task Services

- A task can perform some actions on request from another task.
- The interface (declaration) of the task specifies the available actions (entries).
- A task can also execute some actions on its own behalf, without external requests or communication.

```plaintext
task type Device is
  entry Read (X: out Integer);
  entry Write (X: Integer);
end Device;
```

### Synchronization – The Rendezvous

- Caller makes explicit request: entry call
- Callee (server) states its availability: accept statement
- If server is not available, caller blocks and queues up on the entry for later service
- If both present and ready, parameters are transmitted to server
- Server performs action
- Out parameters are transmitted to caller
- Caller and server continue execution independently
Simple mechanism to prevent simultaneous access to a critical section: code that cannot be executed by more than one task at a time

```
task type semaphore is
  entry P; -- Dijkstra’s terminology
  entry V; -- from the Dutch
  -- Proberen te verlangen (wait) [P];
  -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
  loop
    accept P;
    -- won’t accept another P
    -- until a caller asks for V
    accept V;
  end loop;
end semaphore;
```

A task that needs exclusive access to the critical section executes:

```
Sema : semaphore;
...
Sema.P;
-- critical section code
Sema.V;
```

If in the meantime another task calls Sema.P, it blocks, because the semaphore does not accept a call to P until after the next call to V: the other task is blocked until the current one releases by making an entry call to V

Programming hazards:
» someone else may call V => race condition
» no one calls V => other callers are livelocked
Delays and Time

- A delay statement can be executed anywhere at any time, to make current task quiescent for a stated interval:

```
delay 0.2; -- type is Duration, unit is seconds
```

- We can also specify that the task stop until a certain specified time:

```
delay until Noon; -- Noon defined elsewhere
```

Conditional Communication

- Need to protect against excessive delays, deadlock, starvation, caused by missing or malfunctioning tasks
- Timed entry call: caller waits for rendezvous a stated amount of time:

```
select
    Disk.Write(Value => 12,
                Track => 123); -- Disk is a task
or
    delay 0.2;
end select;
```

- if Disk does not accept within 0.2 seconds, go do something else
Conditional Communication (Continued)

- Conditional entry call: caller ready for rendezvous only if no one else is queued, and rendezvous can begin at once:

```pascal
select
  Disk.Write(Value => 12, Track => 123);
else
  Put_Line("device busy");
end select;
```

- Print message if call cannot be accepted immediately

Conditional Communication (Continued)

- The server may accept a call only if the internal state of the task is appropriate:

```pascal
select
  when not Full =>
    accept Write (Val: Integer) do ... end;
  or
  when not Empty =>
    accept Read (Var: out Integer) do ... end;
  or
  delay 0.2; -- maybe something will happen
end select;
```

- If several guards are open and callers are present, any one of the calls may be accepted – non-determinism
Concurrency in Java

- Two notions
  - class Thread
  - interface Runnable
- An object of class Thread is mapped into an operating system primitive

```java
interface Runnable {
    public void run();
}
```

- Any class can become a thread of control by supplying a run method

```java
class R implements Runnable {
    ...
}
Thread t = new Thread(new R(...));
t.start();
```

Threads at Work

```java
class PingPong extends Thread {
    private String word;
    private int delay;
    PingPong (String whatToSay, int delayTime) {
        word = whatToSay; delay = delayTime;
    }
    public void run () {
        try {
            for (;;) { // infinite loop
                System.out.print(word + " ");
                sleep(delay); // yield processor
            }
        } catch (InterruptedException e) {
            return; // terminate thread
        }
    }
}
```
Activation and Execution

- Call to start activates thread, which executes run method
- Threads can communicate through shared objects
- Classes can have synchronized methods to enforce critical sections

```java
public static void main (String[] args) {
    new PingPong("ping", 33).start();  // activate
    new PingPong("pong", 100).start();  // activate
}
```

Implementing Synchronization

- Condition synchronization with atomic reads and writes is easy
  - You just cast each condition in the form of "location X contains value Y" and you keep reading X in a loop until you see what you want
- Mutual exclusion is harder
  - Much early research was devoted to figuring out how to build it from simple atomic reads and writes
  - Dekker is generally credited with finding the first correct solution for two processes in the early 1960s
  - Dijkstra published a version that works for N processes in 1965
  - Peterson published a much simpler two-process solution in 1981
Repeatedly reading a shared location until it reaches a certain value is known as SPINNING or BUSY-WAITING.

A busy-wait mutual exclusion mechanism is known as a SPIN LOCK:
- The problem with spin locks is that they waste processor cycles.
- Synchronization mechanisms are needed that interact with a thread/process scheduler to put a process to sleep and run something else instead of spinning.
- Note, however, that spin locks are still valuable for certain things, and are widely used.
  - In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead.

SEMAPHORES were the first proposed SCHEDULER-BASED synchronization mechanism, and remain widely used.

CONDITIONAL CRITICAL REGIONS and MONITORS came later.

Monitors have the highest-level semantics, but a few sticky semantic problem - they are also widely used.

Synchronization in Java is sort of a hybrid of monitors and CCRs (Java 3 will have true monitors.)

Shared-memory synch in Ada 95 is yet another hybrid.
A semaphore is a special counter

- It has an initial value and two operations, P and V, for changing that value
- A semaphore keeps track of the difference between the number of P and V operations that have occurred
- A P operation is delayed (the process is de-scheduled) until #P-#V <= C, the initial value of the semaphore
Implementing Synchronization

```pascal
type semaphore = record
  N : integer  -- usually initialized to something nonnegative
  Q : queue of threads
end;

procedure (ref S : semaphore)
  disable_signals
  acquire_lock(scheduler_lock)
  S.N := 1
  if S.N < 0
    sleep_on(S.Q)
  release_lock(scheduler_lock)
  reenable_signals

procedure (ref S : semaphore)
  disable_signals
  acquire_lock(scheduler_lock)
  S.N := 1
  if S.N <= 0
    -- at least one thread is waiting
    enqueue(ready_list, dequeue(S.Q))
  release_lock(scheduler_lock)
  reenable_signals
```

Figure 12.14 Semaphore operations, for use with the scheduler code of Figure 12.12.

---

Implementing Synchronization

```pascal
shared buf : array [1..SIZE] of bdata
shared next,full,next,empty : integer := 1,1
shared mutex : semaphore := 1
shared empty_slots,full_slots : semaphore := SIZE,0

procedure insert(d : bdata)
  P(empty_slots)
  P(mutex)
  buf[next,empty] := d
  next,empty := next,empty + mod SIZE + 1
  V(mutex)
  V(full_slots)

function remove : bdata
  P(full_slots)
  P(mutex)
  d := buf[next,full]
  next,full := next,full + mod SIZE + 1
  V(mutex)
  V(empty_slots)
  return d
```

Figure 12.15 Semaphore-based code for a bounded buffer. The mutex binary semaphore protects the data structure proper. The full_slots and empty_slots general semaphores ensure that no operation starts until it is safe to do so.
Implementing Synchronization

- It is generally assumed that semaphores are fair, in the sense that processes complete P operations in the same order they start them.

- Problems with semaphores
  - They're pretty low-level.
    - When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs (because you do a V inside an if statement, for example, as in the use of the spin lock in the implementation of P).
  - Their use is scattered all over the place.
    - If you want to change how processes synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone.

Language-Level Mechanisms

- Monitors were an attempt to address the two weaknesses of semaphores listed above.

- They were suggested by Dijkstra, developed more thoroughly by Brinch Hansen, and formalized nicely by Hoare (a real cooperative effort!) in the early 1970s.

- Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism.
  - None, incorporates the precise semantics of Hoare's formalization.
A monitor is a shared object with operations, internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time.

- A process that calls a busy monitor is delayed until the monitor is free.
  - On behalf of its calling process, any operation may suspend itself by waiting on a condition.
  - An operation may also signal a condition, in which case one of the waiting processes is resumed, usually the one that waited first.

The precise semantics of mutual exclusion in monitors are the subject of considerable dispute. Hoare's original proposal remains the clearest and most carefully described.

- It specifies two bookkeeping queues for each monitor: an entry queue, and an urgent queue.
- When a process executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first process on the appropriate condition queue obtains control of the monitor.
- When a process leaves a monitor it unblocks the first process on the urgent queue or, if the urgent queue is empty, it unblocks the first process on the entry queue instead.
Language-Level Mechanisms

- Building a correct monitor requires that one think about the "monitor invariant". The monitor invariant is a predicate that captures the notion "the state of the monitor is consistent."
  - It needs to be true initially, and at monitor exit
  - It also needs to be true at every wait statement
  - In Hoare's formulation, needs to be true at every signal operation as well, since some other process may immediately run
- Hoare's definition of monitors in terms of semaphores makes clear that semaphores can do anything monitors can
- The inverse is also true; it is trivial to build a semaphores from monitors (Exercise)

Agenda

1 Session Overview
2 Control Abstractions
3 Concurrency
4 Dynamic Allocation and Garbage Collection
5 Conclusion
For most languages, the amount of memory used by a program cannot be determined at compile time
  » Earlier versions of FORTRAN are exceptions!

Some features that require dynamic memory allocation:
  » recursion
  » pointers, explicit allocation (e.g., new)
  » higher order functions

In imperative languages, space for local variables and parameters is allocated in activation records, on the stack
  » This is because the lifetime of such values follows a FIFO discipline – when the routine returns, we don’t need its locals or arguments any more

This is not the case for:
  » Space allocated with new
  » Space for local variables and parameters in functional languages

The lifetime (aka extent) of these entities may be longer than the lifetime of the procedure in which they were created

These are allocated on the heap
The heap is finite – if we allocate too much space, we will run out

Solution: deallocate space when it is no longer necessary

Methods:
- Manual deallocation, with e.g., free, delete (C, Pascal)
- Automatic deallocation via garbage collection (Java, C#, Scheme, ML, Perl)
- Semi-automatic deallocation, using destructors (C++, Ada)
  - Automatic because that the destructor is called at certain points automatically
  - Manual because that the programmer writes the code for the destructor

Manual deallocation is dangerous (because not all current references to an object may be visible)

Two basic methods:
- Free list – typically for manual and semi-automatic deallocation
- Heap pointer – typically for automatic deallocation

Free list method:
- A linked list of unused blocks of memory is maintained (the free list)
- Allocation: a search is done to find a free block of adequate size; it is removed from the free list
- Deallocation: the block is put on the free list

Problems:
- May take some time to find a free block of the right size
- Memory eventually becomes fragmented
Allocation – Heap Pointer

- Heap pointer method:
  - Initially, the heap pointer is set to bottom of heap
  - Allocation: the heap pointer is incremented an appropriate amount
  - Deallocation: some sort of garbage collection
- Problems:
  - Requires moving of live objects in memory

Automatic Deallocation

- Basic garbage collection methods:
  - Reference counting – usually done by programmer
  - Mark/sweep – needs run-time support
    - variant: compacting
    - variant: non-recursive
  - Copying – needs run-time support
    - variant: incremental
    - variant: generational
An object $x$ is live (i.e., can be referenced) if:
- $x$ is pointed to by a variable
  - on the stack (e.g., in an activation record)
  - a global
- There is a register (containing a temporary or intermediate value) that points to $x$
- There is another object on the heap (e.g., $y$) that is live and points to $x$

All live objects in the heap can be found by a graph traversal:
- Start at the roots – local variables on the stack, global variables, registers.
- Any object not reachable from the roots is dead and can be reclaimed

Each object has an extra bit called the mark bit
- Mark phase: the collector traverses the heap and sets the mark bit of each object encountered
- Sweep phase: each object whose mark bit is not set goes on the free list

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
</tr>
</thead>
</table>
| GC()    | for each root pointer $p$ do  
          |   mark($p$);  
          |   sweep();  |
| mark($p$) | if $p$→mark /= 1 then  
          |   $p$→mark = 1;  
          |   for each pointer field $p$→x do  
          |     mark($p$→x);  |
| sweep() | for each object $x$ in heap do  
          |   if $x$.mark = 0 then insert($x$, free_list);  
          |   else $x$.mark = 0;  |
Copying Garbage Collection

- Heap is split into 2 parts: FROM space, and TO space
  - Objects allocated in FROM space
  - When FROM space is full, garbage collection begins
  - During traversal, each encountered object is copied to TO space
  - When traversal is done, all live objects are in TO space
  - Now we flip the spaces – FROM space becomes TO space and vice versa
    - Note: since we are moving objects, any pointers to them must be updated
- This is done by leaving a forwarding address
- Heap pointer method used for allocation – fast

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
</tr>
</thead>
</table>
| `GC()` | for each root pointer `p` do  
  `p := traverse(p);`  
  if *p contains forwarding address then  
  `p := *p; // follow forwarding address`  
  return `p;`  
  else {  
  `new_p := copy (p, TO_SPACE);`  
  `*p := new_p; // write forwarding address`  
  for each pointer field `p->x` do  
  `new_p->x := traverse(p->x);`  
  return `new_p;`  
  }

traverse(p)
### Generational Garbage Collection

- A variant of a copying garbage collector
- Observation: the older an object gets, the longer it is expected to stay around.
  - Why?
    - Many objects are very short-lived (e.g., intermediate values)
    - Objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program
- Idea: instead of 2 heaps, use many heaps, one for each “generation”
  - Younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - When a generation is traversed, live objects are copied to the next-older generation
    - When a generation fills up, we garbage collect it

### Reference Counting

- The problem:
  - We have several references to some data on the heap
  - We want to release the memory when there are no more references to it
  - Sadly, we don’t have garbage collection “built-in”
- Idea: Keep track of how many references point to the data, and free it when there are no more.
  - Set reference count to 1 for newly created objects
  - Increment reference count whenever we make a copy of a pointer to the object
  - Decrement reference count whenever a pointer to the object goes out of scope or stops pointing to the object
  - When an object’s reference count becomes 0, we can free it
Costs of various methods:
- \( L \) = amount of storage occupied by live data
- \( M \) = size of heap
- Mark/sweep: \( O(L) + O(M) = O(M) \) since \( M > L \)
- Copying: \( O(L) \)
  - experimental data for LISP: \( L \) is around 0.3 * \( M \)
- Harder to compare with reference counting, but mark/sweep and copying are generally faster.

A chief reason C++ has destructors is to enable implementation of reference counting.
class C {
public:
    C () : p(NULL) { }
    C (const C& c) : p(c.p) { if (p) p->refCount++; }
    ~C () { if (p && --p->refCount == 0) delete p; }
    C& operator= (const C&);
    ...
private:
    struct RefCounted {
        int refCount;
        ...
        RefCounted (...) : refCount(1), ... { ... }
    };
    RefCounted *p;
}

const C& C::operator= (const C& c) {
    if (c.p)
        c.p->refCount++;
    if (p)
        p->refCount--;
    p = c.p;
    return *this;
}
Agenda

1. Session Overview
2. Control Abstractions
3. Concurrency
4. Dynamic Allocation and Garbage Collection
5. Conclusion

Assignments & Readings

- Readings
  » Chapter Sections 8.4, 8.5, 12
- Assignment #6
  » See Programming Assignment #4 posted under “handouts” on the course Web site
  » Due on May 19, 2011