Terminology

- Exception: An error, or more generally, an unusual condition.
- Raise, Throw, Signal: A statement is said to "raise" (or "throw" or "signal") an exception if the execution of this statement leads to an exception. ("throw" is the term used in C++/Java language descriptions, "raise" is used in SML.)
- Catch: A catch statement is used in C++/Java to declare a handler. The keyword "handle" is used in SML to denote the same concept.

Terminology (contd.)

- Resumption model: After the execution of the handler, control returns back to the statement that raised the exception.
  - Example: signal handling in UNIX/C.
- Termination Model: Control does ***not*** return to that statement after the handler is executed.
  - Example: Exception handling in most programming languages (C++, Java and SML).

Exception Handling

- Uncaught exceptions are propagated up the call stack.
- Example: f calls g, which in turn calls h
- if h raises an exception and there is no handler for this exception in h, then g gets that exception.
- If there is a handler for the exception in g, the handler is executed, and execution continues normally after that.
- otherwise, the exception is propagated to f.

Exception Handling in C++/Java

Syntax: <blockWithHandler> ::= try <block> <match>

<match> ::= <handler> ... <handler>

<handler> ::= catch (<parameter decl>) { <block> }

Exception Handling in C++/Java(contd.)

Example:

```
int fac(int n) {
      if (n <= 0) throw (-1) ; else if (n > 15) throw ("n too large");
      else return n*fac(n-1); }
void g (int n) { int k;
      try { k = fac (n) ;}
      catch (int i) { cout << "negative value invalid" ; }
      catch (char *s) { cout << s; }
      catch (...) { cout << "unknown exception" ;}
```

- use of g (-1) will print "negative value invalid"
  g (16) will print "n too large"
- If an unexpected error were to arise in evaluation of fac or g, such as running out of memory, then "unknown excpetion" will be printed

Exception Vs Return Codes

- Exceptions are often used to communicate error values from a callee to its caller. Return values provide alternate means of communicating errors.

Example use of exception handler:

```
float g (int a, int b, int c) {
  float x = 1./fac(a) + 1./fac(b) + 1./fac(c) ;  return x ; }
main() {
  try {  g(-1, 3, 25); }
  catch (char *s) { cout << "Exception `" << s << "'raised, exiting\n"; }
  catch (...) { cout << "Unknown exception, eixting\n"; }
```

We do not need to concern ourselves with every point in the program where an error may arise.

Exception Vs Return Codes(contd.)

```
float g(int a, int b, int c) {
    int x1 = fac(a) ;
    if (x1 > 0) {
      int x2 = fac(b) ;
      if (x2 > 0) {
        int x3 = fac(c) ;
        if (x3 > 0) {
          return 1./x1 + 1./x2 + 1./x3 ;
        }
      }
    }
    return …; // there was an error
}
main() {
    int x = g(-1, 2, 25);
    if (x < 0) { /* identify where error
    occurred, print */ }
```

- If return codes were used to indicate errors, then we are forced to check return codes (and take appropriate action) at every point in code.

Use of Exceptions in C++ Vs Java

- In C++, exception handling was an after-thought.
  - Earlier versions of C++ did not support exception handling.
  - Exception handling not used in standard libraries
  - Net result: continued use of return codes for error-checking
- In Java, exceptions were included from the beginning.
  - All standard libraries communicate errors via exceptions.
  - Net result: all Java programs use exception handling model for error-checking, as opposed to using return codes.

Implementation of Exception Handling in Programming Languages

- Exception handling can be implemented by adding "markers" to ARs to indicate the points in program where exception handlers are available.
- Entering a try-block at runtime would cause such a marker to be put on the stack
- When exception arises, the RTE gets control and searches down from stack top for a marker.
- Exception then "handed" to the catch statement of this try-block that matches the exception
- If no matching catch statement is present, search for a marker is contined further down the stack

Heap management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality
- Models
  - Explicit allocation and free (C, C++)
  - Explicit allocation, automatic free (Java)
  - Automatic allocation and free (SML, Python, Javascript)

Fragmentation

- Internal fragmentation
  - When asked for x bytes, allocator returns y > x bytes; y-x represents internal fragmentation
- External fragmentation
  - When (small) free blocks of memory occur in between used blocks
    - the memory manager may have a total >> N bytes of free memory available, but none may be large enough to satisfy a request of size N.

Reducing Fragmentation

- Use blocks of single size (early LISP)
  - Limits data-structures to use less efficient implementations.
- Use bins of fixed sizes, e.g., 2^n for n=0,1,2,...
  - When you run out of blocks of a certain size, break up a block of next available size
  - Eliminates external fragmentation, but increases internal fragmentation
- Maintain bins as LIFO lists to increase locality
- malloc implementations (Doug Lea)
  - For small blocks, use bins of size 8k bytes, $0 < k < 64$
  - For larger blocks, use bins of sizes 2^n for $n > 9$

Coalescing

- What if a program allocates many 8 byte chunks,frees them all and then requests lots of 16 byte chunks?
  - Need to coalesce 8-byte chunks into 16-byte chunks
  - Requires additional information to be maintained
    - for allocated blocks: where does the current block end, and whether the next block is free

Explicit Vs Implicit Management

- Explicit memory management can be more efficient, but takes a lot of programmer effort
- Programmers often ignore memory management early in coding, and try to add it later on
  - But this is very hard, if not impossible
- Result:
  - Majority of bugs in production code is due to memory management errors
    - Memory leaks
    - Null pointer or uninitalized pointer access
    - Access through dangling pointers

Managing Manual Deallocation

- How to avoid errors due to manual deallocation of memory
  - Never free memory!!!
  - Use a convention of object ownership (owner responsible for freeing objects)
    - Tends to reduce errors, but still requires a careful design from the beginning. (Cannot ignore memory deallocation concerns initially and add it later.)
  - Smart data structures, e.g., reference counting objects
  - Region-based allocation
    - When a bunch of objects having equal life time are allocated

Garbage Collection

- Garbage collection aims to avoid problems associated with manual deallocation
  - Identify and collect garbage automatically
- What is garbage?
  - Unreachable memory
- Automatic garbage collection techniques have been developed over a long time
  - Since the days of LISP (1960s)

Garbage Collection Techniques

- Reference Counting
  - Works if there are no cyclic structures
- Mark-and-sweep
- Generational collectors
- Issues
  - Overhead (memory and space)
  - Pause-time
  - Locality

Reference Counting

- Each heap block maintains a count of the number of pointers referencing it.
- Each pointer assignment increments/decrements this count
- Deallocation of a pointer variable decrements this count
- When reference count becomes zero, the block can be freed

Reference Counting

- Disadvantages
  - Does not work with cyclic structures
  - May impact locality
  - Increases cost of each pointer update operation
- Advantages
  - Overhead is predictable, fixed
  - Garbage is collected immediately, so more efficient use of space

Mark-and-Sweep ("Trace-based")

- Mark every allocated heap block as "unreachable"
- Start from registers, local and global variables
- Do a DFS, following the pointers
  - Mark each heap block visited as "reachable"
- At the end of the sweep phase, reclaim all heap blocks still marked as unreachable

Issues with GC

- Memory fragmentation
  - Memory pages may become sparsely populated
  - Performance will be hit due to excessive virtual memory usage and page faults
  - Can be a problem with explicit memory management as well
- Solution:
  - Compacting GC
    - Copy live structures so that they are contiguous
  - Copying GC

Copying Collection

- Instead of doing a sweep, simply copy over all reachable heap blocks into a new area
- After the copying phase, all original blocks can be freed
- Now, memory is compacted, so paging performance will be much better
- Needs up to twice the memory of compacting collector, but can be much faster
  - Reachable memory is often a small fraction of total memory

Generational GC

- Take advantage of the fact that most objects are short-lived
- Exploit this fact to perform GC faster
- Idea:
  - Divide heap into generations
  - If all references go from younger to older generation (as most do), can collect youngest generation w/o scanning regions occupied by other generations
    - Need to track references from older to younger generation to make this work in all cases

Garbage collection in Java

- Generational GC for young objects
- "Tenured" objects stored in a second region
  - Use mark-and-sweep with compacting
- Makes use of multiple processors if available
- References
  - http://java.sun.com/javase/technologies/hotspot/gc/g
  - http://www.ibm.com/developerworks/java/library/j-jtp

GC for C/C++

- Cannot distinguish between pointers and nonpointers
  - Need “conservative garbage collection”
- The idea: if something “looks” like a pointer, assume that it may be one!
  - Problem: works for finding reachable objects, but cannot modify a value without being sure
    - Copying and compaction are ruled out!
- Reasonable GC implementations are available, but they do have some drawbacks
  - Unpredictable performance
  - Can break some programs that modify pointer values before storing them in memory