Garbage Collection

Martin Kero

Martin.Kero@ltu.se

SMD163 - Compiler Construction

Contains material generously provided by Johan Nordlander and Mark P. Jones

Outline

Run-Time Systems
- Why use a Run-Time System?
- Libraries and Linking
- Pros and Cons

Memory Management
- Dynamic Memory Allocation
- Garbage Collection

Outline

Run-Time Systems
- Why use a Run-Time System?
- Libraries and Linking
- Pros and Cons

Memory Management
- Dynamic Memory Allocation
- Garbage Collection

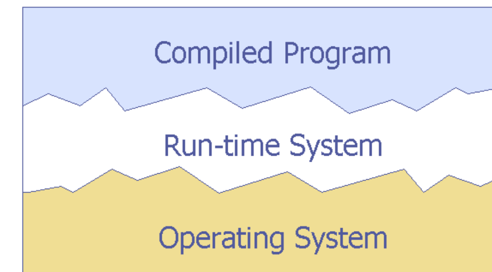
Motivation to use a Run-Time System

- Every programming language provides a different view of what the computer can do:
  - Built-in types, and associated operations;
  - I/O facilities;
  - Concurrency and multiple threads;
  - Dynamic memory management;
  - Etc..
- Different computers, and different operating systems, do not necessarily support all of these features directly, and will often use different interfaces, and even different semantics.

Examples from Java:

- Not all CPUs have built-in support for floating point division, or for standard mathematical functions like square root, sin, or log.
- Unix systems use ”File descriptors” to identify files, and do not know anything about the `File`, `OutputStream`, etc. objects used in Java.
- Many operating systems include support for multiple threads of execution, but they do not all use the same interface.

Using a Run-Time System



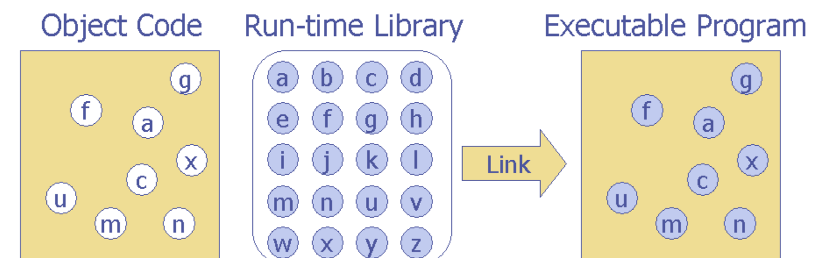
- A run-time system bridges the gap between:
  - The language designer's expectation of what facilities the underlying system will provide; and
  - The set of features that are actually supported by the machine or its operating system.

Run-Time Libraries:

- Conceptually, a run-time system has two parts:
  - A set of conventions about the way that different kinds of value are represented, and about the data structures that are used;
  - A library of code to implement the required features, or to wrap up operating system features according to the conventions of the run-time system.
- Compiled programs:
  - Must follow the conventions of the run-time system;
  - May include references to code in the run-time library.

Linking:

- A linking process is used to build executable programs by connecting compiled object files to the appropriate run-time system libraries.
- The goal is to fill each reference in an object file with the corresponding code from the library.



Static Linking:

- Static linking produces executable code by inserting sections of run-time library code into each executable program.
- Thus portions of the run-time library may be duplicated many times over, which takes extra space on disk, and in memory (if multiple processes are executing).
- If there is a bug in a run-time library routine, then all compiled programs will need to be rebuilt.

Dynamic Linking:

- With dynamic linking, libraries are separate units that can be loaded into memory and connected to executable code as needed.
- It is enough to have just one copy of a dynamically linked library (DLL) on disk. A single copy in memory to be shared between multiple programs.
- In effect, DLLs behave like extensions of the operating system.
- But programs using DLLs won't run properly without suitable versions of their libraries.

Run-time System Pros:

Portability: A run-time system isolates a program from the details of a particular operating environment, and so makes it easier to port code between different platforms.

Reuse: A run-time system provides standardized libraries and abstractions that can be used in many different programs.

Abstraction: A run-time system can package up low-level features in a way that makes them easier to understand and use.

Run-time System Cons:

Size: A comprehensive run-time system may be quite large, and so require a lot of effort to implement and port.

Overhead: A run-time system can add overhead, increasing the size of executable programs.

Coding Difficulties: The implementation of some features may conflict with the semantics of features in the underlying system, and so require an inefficient or indirect encoding.

Outline

Run-Time Systems
- Why use a Run-Time System?
- Libraries and Linking
- Pros and Cons

Memory Management
- Dynamic Memory Allocation
- Garbage Collection

Dynamic Memory Allocation

- Dynamic memory allocation is used when the amount of memory that will be needed to store a program's data cannot be predicted at compile-time.
- Examples of programs where this is useful include: compilers, web browsers, word processors, ...

Allocation in Run-time Systems:

- Some languages do not support dynamically allocated memory; instead, programmers must anticipate/guess the requirements at compile-time and pre-allocate storage accordingly.
- Many operating systems do not support (fine-grained) dynamic memory allocation well ... but many languages require it.
- As a result, dynamic memory allocation is one of the most commonly supported features in modern run-time systems.

Explicit Allocation

- Different languages provide different ways to allocate memory:

```
bytes   =  (int*)malloc(120);
ints    =  new int[30];
expr    =  new IntExpr(120);
list    =  x:xs
(cons   x  xs)
```

Allocating from a Heap:

- Where does dynamically allocated memory come from?
- When the run-time system is initialized, it requests a large block of memory from the operating system, which is known as the *heap*.
- The run-time system maintains a *heap pointer* that identifies the "next free location."

- To allocate n bytes, we return the current heap pointer setting, and advance the heap pointer by n.

Memory Usage in a Browser:

- Each time you visit a new web page, the browser needs to allocate memory to store the text, images, and other items on that page.
- You might run a browser for a long time and visit many web pages.
- If the browser doesn't take steps to reclaim memory, then, eventually, your browser will not be able to load any new web pages.

Reclaiming Memory Explicitly:

- In some languages, programmers can tell the run-time system that they have finished with a piece of memory, and that it can be recycled.

```
free(bytes);    /* C */
destroy ints;   /* C++ */
```

- This can be risky; the programmer must ensure that:
  - The specified memory was allocated dynamically;
  - No part of the program will attempt to access that section of memory again.

Reclaiming Memory in a Browser:

- It is easy to manage memory in a browser:
  - Keep data for the web pages that can be reached using the "Back" and "Forward" buttons.
  - If memory gets tight, we can reclaim the storage used by some of the web pages: we can store the data on disk, or download it again if it is needed.
- In other words, it is easy to see where the calls to `free` or `destroy` should go.

Reclaiming Memory Automatically:

- In general, however:
  - It is hard to know when memory can be reclaimed;
  - If it is too early, the run-time system's structures will be corrupted, and the program could crash;
  - If memory is reclaimed too late, then the program will have a ”space leak” and use more memory than it needs.
- Incorrect attempts to reclaim memory are one of the biggest sources of bugs in C++ programs.
- Could a run-time system do better in deciding when memory can be reclaimed?

Garbage Collection

- *Garbage collection* is the term used to describe automatic reclamation of computer storage.
- An object is *garbage* if it will not be used again – in other words, if it is not alive.
- Conceptually, garbage collection is a two phase process:
  - **Detect:** Distinguish live objects from those which are garbage.
  - **Reclaim:** Reclaim memory used by garbage so that the running program can reuse it.
- In practice, these phases may be interleaved.

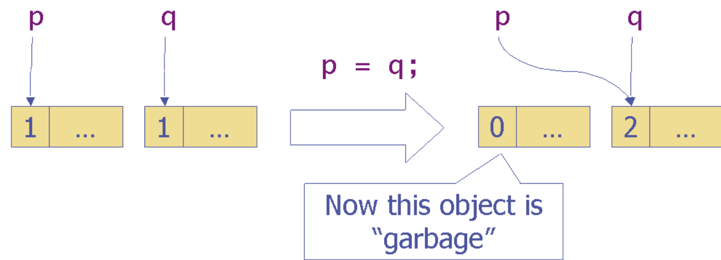
How do we Detect Garbage?

- Optimally, an object should be detected as garbage immediately after its last use ... however, determining if an access is the last time is very difficult, sometimes even impossible.
- Approximations:
  - Detecting when an object cannot be used by the program anymore.
  - Naïve solution: Count the number of pointers to each object.
    ... this leads us to the technique of *reference counting*.

Reference Counting:

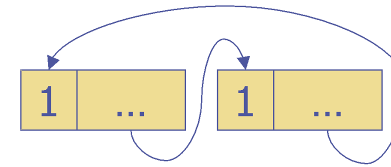
- A run-time system can attach a reference count to each chunk of memory that is allocated.
- The reference count is the number of pointers to this object from elsewhere in the program.
- Every time we duplicate a pointer to an object, we increment the reference count.
- Every time we eliminate a pointer to an object, we decrement the reference count.
- When the reference count is zero, the object can be reclaimed.

Example:



A compiler can generate code to maintain the reference counts, but the overhead is usually quiet high.

The Problem With Cycles:



- The two objects shown here are garbage; i.e. neither one can be reached from anywhere in the program.
- But neither one has a zero reference count, so neither one will be reclaimed ... a memory leak!
- Thus additional steps must be taken to deal with cycles ... one major reason why reference counting is not very popular in run-time systems.

How do we Detect Garbage?

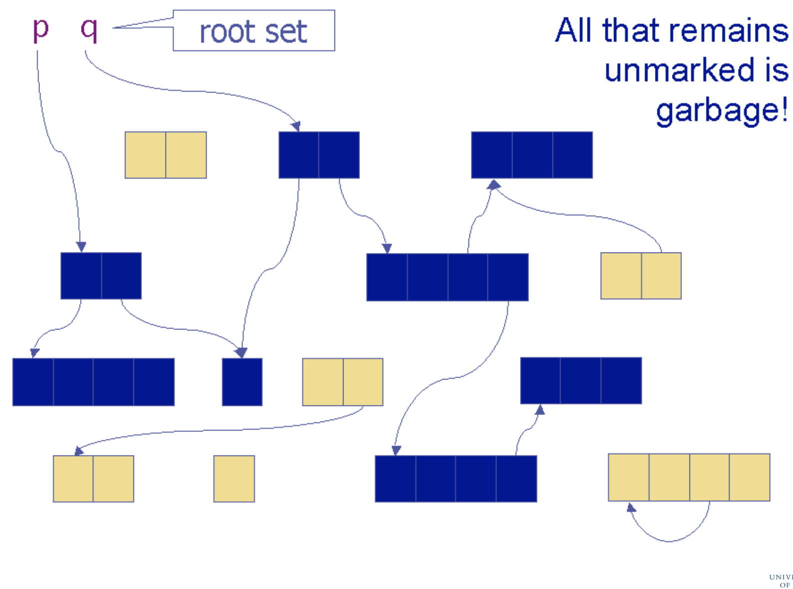
- Approximations:
  - Detecting when an object cannot be used by the program anymore.
  - First attempt: An object is garbage if there are no pointers to it.
    ... this leads us to the technique of *reference counting*.
  - But an object can be garbage, even if there are pointers to it ... if those pointers are in other pieces of garbage.
  - Second attempt: An object is garbage if it is *unreachable*.

    This is still conservative, but usually works quite well.

Reachability:

- Suppose that we could interrupt a MiniJava computation at any stage. Which objects might be live at that point?
- We can identify a set of *roots* for live data:
  - Any object that is pointed to from a global variable (i.e., a static field in a class, but we don’t have those in MiniJava);
  - Any object that is pointed to from an active frame on the stack.
- Any object that can be reached from one (or more) of the roots might be used in a future computation.

Understanding Reachability:



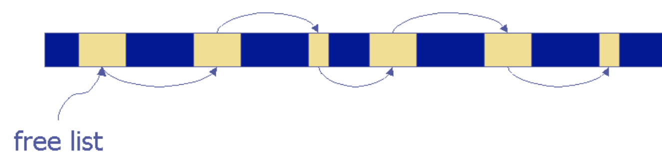
Tracing Garbage Collection:

- Traverse the graph, starting at the roots, and *keep* every object that is reached;
- The rest of the heap is then garbage.
- Example: *Mark-and-Sweep*
  - Mark every object that is reached;
  - Sweep the heap for unmarked objects and reclaim their storage.
  - The time to garbage collect is proportional to the size of the heap: we have to sweep the whole heap to find unmarked objects.

Fragmentation:

- Once the marking phase is over, the heap will typically be broken into a mixture of marked and unmarked areas:

- We can deal with this by linking together the unused areas in a *free list*:

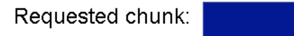


Allocating From a Free List:

- Now we must allocate memory from the free list too; we cannot just advance a heap pointer.
- To allocate n bytes:
  - Search for the first free chunk with $\geq$n bytes in it;
  - Allocate the required memory from that chunk;
  - Return any unused portion to the free list.
  - This allocation strategy is called *first-fit*.
- There are some techniques that we can use to make this more efficient.
- Exactly the same problems occur in systems with explicit memory reclamation (malloc()...)

A Problem Due To Fragmentation:

- A serious problem here is the risk of failing to allocate, which may happen when the heap is broken into small pieces that are hard to reuse.
- For example, this allocation will fail:

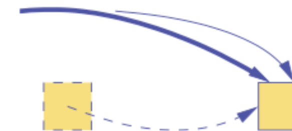


Although there is enough unused memory, in total, it is not available in one *contiguous* block.

- Several *compaction* techniques have been developed to overcome this problem.

Compacting the Heap:

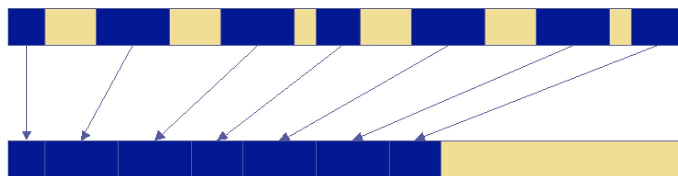
- The main problem when compacting the heap is that we need to *move* live objects.
- If an object has more than one live pointer pointing at it, we need to notify the others when the object has been moved.
- Ex:



- We need to install a *forwarding pointer* in the old location.

A Copying Collector:

- The heap is divided into two parts, where only one is active at the same time.
- A copying collector works by copying all the reachable data from the currently active part (*from space*) into the inactive one (*to space*), and then switching the labels (i.e. the active one becomes inactive and vice versa).



Pros and Cons:

- A copying collector ensures that the heap is compacted at each garbage collection:
  - No fragmentation!
  - We can go back to allocating using a simple heap pointer.
- The time to garbage collect is proportional to the amount of memory that is reachable, which may be much less than the size of the heap.
- We have to split the available memory resources between two large heaps, even though we only use one at a time.

Extending The Copying Collector (Generations):

- To avoid copying long-lived objects back and forth we can divide the heap in more than two parts. This is called *generational collection*:
  - Long-lived objects are promoted from the *younger* heap to the *older*.
  - Only one heap is garbage collected at the same time.

Extending The Copying Collector (Incremental):

- We have so far assumed that the main computation is put on ”hold” while the garbage collection is taking place.
- For an interactive program with a large heap size, this might cause a noticeable pause in execution.
- For real-time applications, a long pause at random is not acceptable.
- Much effort has been invested in the design of more sophisticated, *incremental* garbage collection algorithms that solve these problems by interleaving garbage collection and the main computation(s).

The Cost of Garbage Collection:

- Appel has argued that garbage collection can sometimes be cheaper than stack allocation. The basic idea is that garbage on the stack must be popped of, while a copying collector only processes reachable data.
- Other estimates suggest that the use of garbage collection can increase execution time by 10%.
- In any case:
  - The cost depends on the ”quality” of the garbage collector, and on the program that uses it.
  - There are also overheads with schemes for explicitly reclaimed memory...
  - Perhaps the overheads of garbage collection are justified by the reduction in bugs?

Further Reading:

- There is a large literature on garbage collection; we have only scraped the surface here!
- There is some material in Appel's book.
- There is a book devoted to garbage collection by Richard Jones and Rafael Lins:
  `http://www.cs.kent.ac.uk/people/staff/rej/`
- Another great source is Paul Wilson's survey:
  `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`