**Memory Allocation– Part 2**

**Slide 1**

In part 2 of this presentation on memory allocation we will discuss some other issues related to the way in which memory is allocated.

**Slide 2**

First we define two terms, dangling reference and dangling object that are two problems related to memory allocation.

A dangling reference or a dangling pointer is a reference or pointer to unallocated memory. It can occur either because a pointer has never been assigned allocated memory or the assigned memory has been deallocated. Even when a language resets a pointer when deallocation is performed, a dangling reference can still occur when two pointers point to an object and only one is deallocated. The other will now be pointing to unallocated memory.

Dangling references can only occur in languages with explicit deallocation. They can never occur in languages like Java in which the run-time system manages the deallocation.

A dangling object or garbage is allocated memory that can no longer be referenced. It occurs when an only pointer to an object is reassigned. Like dangling references, dangling objects can only arise when the language requires the programmer to explicitly perform the deallocation.

**Slide 3**

C is a language in which dynamic memory allocation and deallocation is the responsibility of the programmer. So let’s consider a program that illustrates how dangling references and objects can be created in a language like C

In the first three lines a pointer p is declared and assigned a block of dynamically allocated memory large enough to hold an integer. The value of 1 is stored in that location and then the memory is explicitly deallocated. At that point the pointer remains pointing to that memory.

In the next highlighted block of three lines, a pointer q is declared and again assigned memory in the first line. A value is stored in that memory location in the next line and in the third line a new block is allocated and assigned to the pointer q. The result is that the block of memory originally assigned to that pointer is a dangling object or garbage, because it can no longer be accessed.

**Slide 4**

Next we consider several other issues related to pointers.

Although some newer C and C++ compilers may consider uninitialized variables to be an error, early compilers never performed such a check. The most problematic kind of uninitialized variable is a pointer. Dereferencing an uninitialized pointer results in accessing an unknown memory location.

It is another kind of dangling reference and a potential problem that can be hard to find because the effect will be unpredictable and depend upon what memory location is being referenced.

Another issue that arises in C and C++ results from the fact that in those languages, pointers can point to variables on the compiler’s run-time stack.

Allowing such pointers creates the possibility that a pointer can point to a variable whose lifetime has ended--another instance of a dangling reference.

Most languages that allow pointers only allow them to point to memory on the heap, which is the case with Ada’s access variables.

**Slide 5**

Let’s consider a simple C program that contains an uninitialized pointer.

In these two lines a pointer p is declared but never initialized. In the next line, it is dereferenced and assigned a value. The result of this assignment is that the value 1 is being stored in whatever memory location p points to, which may be different depending upon when the program is run.

**Slide 6**

Next, we examine a C function that returns a pointer to a local variable. This function accepts a C string of length 10 as a parameter and copies it to a local string converting it to all upper case.

The local C string result is where the converted string will be stored.

This for loop makes that copy while converting each character to upper case.

What is being returned is a pointer to the local variable result. Its lifetime ends when the function ends, so it will become a dangling reference. Although it may contain the desired result if accessed immediately after the call, if it is accessed after a call to another function it may then be overwritten. Some newer C compilers flag this situation with a warning.

**Slide 7**

We conclude this video on these issues related to memory allocation with a discussion of automatic garbage collection.

LISP was the first language to perform automatic garbage collection

Unlike most imperative languages in which dynamic memory allocation is explicit, in LISP it occurs implicitly, which is why implicit deallocation was a natural choice.

In C++, for example, dynamic allocation is done explicitly with new and deallocation is done explicitly with delete. We have already seen examples of the problems that can arise when a language requires the programmer to explicitly perform deallocation.

In Ada, the compiler designers had the choice of whether to include automatic garbage collection.

Java included it in the very first version of the language. So although dynamic memory allocation in Java is still explicit whenever new is used, the deallocation is done implicitly.