Safe and Secure Software

An Invitation to **Ada 2005**

7

Safe Memory Management

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

The memory of the computer provides a vital part of the framework in which the program resides. The integrity of the memory contents is necessary for the health of the program. There is perhaps an analogy with human memory. If the memory is unreliable then the proper functioning of the person is seriously impaired.

There are two main problems with managing computer memory. One is that information can be lost by being improperly overwritten by other information. The other is that the memory itself can become filled and irrecoverable, so that no new information can be stored. This is the problem of memory leaks.

Memory leak is an insidious fault since it often does not show up for a long time. There was an example of a chemical control program that seemed to run flawlessly for several years. It was restarted every three months because of some external constraints (a crane had to be moved which necessitated stopping the plant). But the schedule for the crane changed and the program was then allowed to run for longer – it crashed after four months. There was a memory leak which slowly gnawed away at the free storage.

Buffer overflow

Buffer overflow is almost a generic term used to denote the violation of the security of information. Buffer overflow enables information to be overwritten or read mistakenly or maliciously.

This is a common fault with C and C++ programs and is typically caused by the absence of checks in those languages regarding writing or reading outside the bounds of an array. We illustrated this problem in the chapter on Safe Typing when discussing the example of throwing a pair of dice.

This problem cannot normally arise in Ada because there are checks that an array index does not lie outside the range of allowed values. These checks can be suppressed if we are absolutely sure that the program is perfect, but this is perhaps an unwise thing to do unless the program has been proved to be correct by analysis tools such as the SPARK Examiner mentioned in Chapter 11.

Although the absence of range checks is the ultimate cause of buffer overflow problems in C, it is exacerbated by other language features such as the choice of indicating the end of a string with a zero byte. This means that programmers have to test for this value (directly or indirectly) in many string manipulation routines. It is easy to make mistakes in performing such tests and in any event the zero value might be accidentally overwritten itself. These secondary problems are often the key to loopholes which enable viruses to enter a system.

Another common way in which data can be accidentally destroyed is through the use of incorrect pointers. Pointers in C are treated as addresses and

arithmetic can be performed on them. It is therefore easy for a pointer to have a miscomputed value and so to point to the wrong thing. Writing through the pointer then destroys some other data.

In the chapter on Safe Pointers we saw that Ada guards against this by applying strong typing to all pointers, and through the accessibility rules which ensure that objects do not vanish while being referenced by other objects.

Therefore, basic features of Ada guard against the accidental loss of data through overwriting memory. The remainder of this chapter addresses the issue of losing memory itself.

Heap control

Programming languages are typically implemented using three sorts of data storage

- global data that exists throughout the life of the program and can thus be allocated permanently (and often statically),
- data stored on a stack which grows and contracts as the flow of control passes through various subprograms,
- data allocated in a heap and used and discarded in a manner not directly tied to the flow of control.

Fortran global common is the primeval example of global static storage (this relates to Fortran as it was in the early days of programming). But global static storage exists in all languages. In Ada if we declared

```
package Calendar_Data is
  type Month is (Jan, Feb, Mar, ... , Nov, Dec);
  Days_In_Month: array (Month) of Integer :=
    (Jan => 31, Feb => 28, Mar => 31, Apr => 30,
    May => 31, Jun => 30, Jul => 31, Aug => 31,
    Sep => 30, Oct => 31, Nov => 30, Dec => 31);
end;
```

then storage for the array Days_In_Month would naturally be declared in fixed global storage.

The stack is an important storage structure in all modern programming languages. Note that we are here talking about the underlying stack used by the implementation and not an object of the type Stack used for illustration in an earlier chapter. The stack is used for parameter passing in subprogram calls (actual parameters, the return address, saved registers, and so on) as well as for local variables within a subprogram. In a multitasking program where several threads of activity occur in parallel, each task has its own stack.

Now consider the function Nfv_2000 used in the program for interest rates in the chapter on Safe Pointers

```
function Nfv_2000 (X: Float) return Float is
  Factor: constant Float := 1.0 + X/100.0;
begin
  return 1000.0 * Factor**2 + 500.0 * Factor – 2000.0;
end Nfv;
```

The object Factor will typically be stored in the stack. It will come into existence when the function is called and will cease to exist when the function returns. This is all managed safely and automatically by the call/return mechanism. Note that although Factor is marked as a constant nevertheless it is not static since each call of the function will provide a different value for it. Moreover, the function might be called by two different tasks at the same time in a multitasking program and so Factor certainly cannot be stored globally.

The values of any actual parameters such as X are also stored on the stack.

Now consider a more elaborate subprogram which declares a local array whose size is not known until the program executes – consider for example a function to return an arbitrary array in reverse order. In Ada we might write

```
function Rev(A: Vector) return Vector is
  Result: Vector(A'Range);
begin
  for K in A'Range loop
    Result(K) := A(A'First+A'Last–K);
  end loop;
  return Result;
end Rev;
```

where the type Vector is declared as

```
type Vector is array (Natural range <>) of Float;
```

This notation indicates that Vector is an array type but the bounds are not given except that they must be within the subtype Natural (and so in the range 0 to Integer'Last). When we declare an actual object of the type Vector we must supply bounds. So we might have

```
L: Integer := ... ;
My_Vector, Your_Vector: Vector(1 .. L);          -- L need not be static
...
Your_Vector := Rev(My_Vector);
```

In most programming languages we would be forced to place an object such as the local variable Result on the heap rather than the stack because its size is not known until the program executes. This is certainly not necessary because a

stack is flexible and storage for local variables can always be managed on a last-in–last-out basis.

But the heap is often used because it requires a bit of thought to design and manage dynamically sized data efficiently and without care the subroutine calling mechanism can suffer a loss of performance. Implementations of Ada always use the stack for local data – an efficient technique is to use both ends of the stack, one end for return links and fixed local data and the other end for dynamically sized local data. This enables the location of return addresses to be computed more efficiently and yet keeps full flexibility. Furthermore, Ada systems usually guard against the stack running out of storage and raise the exception Storage_Error if it does (or rather if it is about to).

The above example illustrates a number of nice points about Ada. By contrast it is quite tricky to write in C. This is because C has no proper abstraction for arrays and so we cannot pass an array as a parameter but only a pointer to an array. Moreover C cannot return a result which is anything other than a scalar value and so cannot pass back the reversed array either. We could of course simply declare a function that reverses the argument *in situ* and leave it to the user to make a copy first. But doing the reverse *in situ* is tricky since we have to take care not to destroy the values as we swap them. So perhaps it is best to pass pointers to both the original array and the result as distinct parameters. The other difficulty is that C does not know how long its arrays are and so we have to pass the length of the array as well (or maybe the upper bound). This is yet another hazard since it is all too easy to pass a length that does not correspond to that of the array. So we might have

```
void rev(float *a, float *result, int length);
{
  for (k=0; k<length; k++)
    result[k] = a[length–k–1];
}
...
float my_vector[100], your_vector[100];
...
rev(my_vector, your_vector, 100);
```

Although this chapter is meant to be about storage management it is perhaps worth pausing to list some of the risks and difficulties in the above C code.

- Arrays in C always have lower bound 0 and so if the application has a different natural lower bound such as 1 then confusion can arise. Ada allows any lower bound.
- The length of the array has to be passed separately, there is a risk of getting the length wrong and confusing the length with the upper bound. In Ada the attributes of the array are passed as part of the array itself.

- The address of the result array has to be passed separately. There is the danger of confusing the two arrays which cannot happen in Ada because the assignment clarifies which is which.
- The loop has to be written out explicitly whereas the Ada notation ties it to the range of the array automatically.

However, we have strayed from the topic. The key point is that if we did declare a local array in C++ whose size was not static as in

```
void f(int n, ... );
{   float a[] = new float [n];
...
}
```

then the array a will be placed in the heap and not on the stack. In C we would have to use malloc which does explicitly reveal the use of the heap.

The general danger of using the heap is that storage might be deallocated when it is still in use or left allocated when it is not needed. Because Ada allows dynamically sized objects on the stack, the heap is basically only used when allocators are invoked as mentioned in the chapter on Safe Pointers. This results in better performance and less chance of memory leaks.

Storage pools

We now turn to the use of the heap in Ada. The proper term is storage pool. If we do an allocation such as in the procedure Push discussed in the chapter on Safe Object Construction thus

```
procedure Push(S: in out Stack; X: in Float) is
begin
   S := new Cell'(S, X);
end Push;
```

then the space for the new Cell will be taken from a storage pool. There is always a standard storage pool but we can declare and manage our own storage pools as well.

LISP was the first language to take storage management out of the hands of the programmer, and to incorporate a garbage collector in order to reclaim storage. This approach is used in a number of other languages including Python and Java. The presence of a garbage collector simplifies programming substantially, but has its own problems. For example, the garbage collector may interrupt the execution of the program at unpredictable times, and is therefore unusable in a real-time environment. A programmer of a real-time system must retain fine control over memory and deallocation and must be able to reclaim

memory at some precise time rather than waiting for the garbage collector to do it. As a consequence a garbage collector is not appropriate for a general purpose language and especially to one used for low-level, real-time and safety-critical applications.

Ada provides the user with a choice of mechanisms. Storage control can be done

- by hand. That is by programming the release of storage on an individual basis.
- by using storage pools. Individual items can be deleted from a specific pool and the whole pool can be discarded when no longer required.
- by a garbage collector. This might not be available in all implementations.

In order to return a lump of storage that is no longer used we call an instantiation of a predefined generic function called Unchecked_Deallocation. In order to do this we have to use a named access type so we will suppose that the type Cell is declared by

```
type Cell;
type Cell_Ptr is access all Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Float;
  end record;
```

Note that we have an intrinsic circularity here which is broken by first giving an incomplete declaration of the type Cell. We now write

```
procedure Free is new Unchecked_Deallocation(Cell, Cell_Ptr);
```

In order to deallocate storage we simply call the procedure Free with an access value referring to the storage concerned. Thus the procedure Pop should now be written as

```
procedure Pop(S: in out Stack; X: out Float) is
  Old_S: Stack := S;
begin
  X := S.Value;
  S := S.Next;
  Free(Old_S);
end Pop;
```

Note that we are here using the version of the type Stack that is limited private and not the version that is controlled.

It might seem that the use of Free is risky. In general it might be that there was another reference to the deallocated storage. But in this example the user's view of the type is limited and so the user cannot have made a copy of the structure. Moreover, the user cannot see the details of the type Stack and in particular cannot see the types Cell and Cell_Ptr at all and therefore cannot call Free. Thus once we have assured ourselves that Pop is correct then no trouble is possible. Finally, the instantiation of Unchecked_Deallocation provides a cross-check by requiring the use of named access types and thus checks that the parameters match.

We must also change Clear as well. The easy way is to write

```
procedure Clear(S: in out Stack) is
  Junk: Float;
begin
  while S /= null loop
    Pop(S, Junk);
  end loop;
end Clear;
```

Although this technique ensures that storage is deallocated properly whenever Pop and Clear are called, there is still the risk that the user might declare a stack and leave its scope when it is not empty. Thus

```
procedure Do_Something ...
  A_Stack: Stack;
begin
  ...                    -- play with A_Stack
  ...                    -- is it empty as we leave?
end Do_Something;
```

If A_Stack were not null when Do_Something is left then the storage would be lost. We cannot leave the onus on the user to take care not to lose storage so we should make the stack a controlled type as illustrated at the end of the chapter on Safe Object Construction. We can then declare our own procedure Finalize perhaps simply as

```
overriding
procedure Finalize(S: in out Stack) is
begin
  Clear(S);
end Finalize;
```

Note the use of the overriding indicator just to ensure that we have not misspelled Finalize or mistyped its formal parameters.

Ada also permits users to declare their own storage pools. This is straightforward but would take too much space to explain in detail here. But the

general idea is that there is a predefined type Root_Storage_Pool (which itself is a limited controlled type) and we can declare our own storage pool type by deriving from it thus

```
type My_Pool_Type(Size: Storage_Count) is
                    new Root_Storage_Pool with private;
overriding
procedure Allocate( ... );
overriding
procedure Deallocate( ... );
-- also overriding Initialize( ... ) and Finalize( ... );
```

The procedure Allocate is automatically called when a new object is allocated by an allocator and Deallocate is automatically called when an object is discarded by calling Free. The user then writes appropriate code to manage the pool as desired. Since a pool type is also controlled the procedures Initialize and Finalize are automatically called when the whole pool is declared and finally goes out of scope.

In order to create a pool we then declare a pool object in the usual way. And finally we can link a particular access type to use the pool.

```
Cell_Ptr_Pool: My_Pool_Type(1000);              -- pool size is 1000
for Cell_Ptr'Storage_Pool use Cell_Ptr_Pool;
```

An important advantage of declaring our own pools is that the risk of fragmentation can be minimized by keeping different types in different pools. Moreover, we can write our own storage allocation mechanisms and even do some storage compaction if we so wish. A further point is that if the access type concerned is declared locally then the pool can be local as well and will automatically be discarded so that there can be no possibility of storage being lost.

Finally, there is a safeguard against misuse of Unchecked_Deallocation and that is that since it is a predefined library unit, any unit we write that calls it will have

```
with Unchecked_Deallocation;
```

written boldly at the start of the text. This will then be clearly visible to anyone reviewing the program and especially to our Manager.

Restrictions

There is a general mechanism for ensuring that we do not use certain features of the language and that is the pragma Restrictions. Thus if we write

```
pragma Restrictions(No_Dependence => Unchecked_Deallocation);
```

then we are asserting that the program does not use Unchecked_Deallocation at all – the compiler will reject the program if this is not true.

There are over forty such restrictions in Ada 2005 which can be used to give assurance about various aspects of the program. Many are rather specialized and relate to multitasking programs. Others which concern storage generally and are thus relevant to this chapter are

```
pragma Restrictions(No_Allocators);
pragma Restrictions(No_Implicit_Heap_Allocations);
```

The first completely prevents the use of the allocator **new** as in **new** Cell'(...) and thus all explicit use of the heap. Just occasionally some implementations might use the heap temporarily for objects in certain awkward circumstances. This is rare and can be prevented by the second pragma.

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com

Courtesy of
AdaCore
The GNAT Pro Company