ACM *Ada Letters* **XIII**,3 (May/June 1993), 91-95.

# How to Steal from a Limited Private Account — Why Mode **IN OUT** Parameters for Limited Types *Must* be Passed by Reference

Henry G. Baker

Nimble Computer Corp., 16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

*Every* use of a mode **IN OUT** parameter which is implemented by means of copy-in, copy-out renders the resource protected by a limited type vulnerable to compromise through programming errors or deliberate attacks. Since the implementation of mode **IN OUT** with by-reference semantics trivially closes this loophole, and since only by-reference semantics "make sense" for a resource or pool of resources, we strongly recommend that Ada9X require by-reference semantics for mode **IN OUT** parameters, at least for those of a limited type.

## Introduction

Whatever their original motivation, Ada's *limited* types have been extensively used to protect the integrity of various *resources*, e.g., the processor resources of *tasks*, the secondary storage resources of *files*, the storage resources of heap-allocated *lists*, the resources protected by "passwords" or unique "handles", the monetary resources of financial *accounts*. In other words, designers of private types are drawn to the use of a *limited* type when the a resource has a unique integrity that would be violated if the object were copied through the use of an assignment operation.

The lack of the assignment operation for a limited type allows the designer of the type to completely control any copying operations, and therefore preserve the uniqueness of the protected resource's "object identity".

Unfortunately, while the designer of a limited type can deny the ability to copy to a user of the type, he cannot deny the ability to copy to the Ada implementation itself, and a careless or unscrupulous user of the type can thus take advantage of an Ada's copy-in, copy-out implementation of the **IN OUT** parameter passing mode to destroy the integrity of the limited type.

## A Simple `ACCOUNT` Limited Type

A favorite example of Ada textbook writers for the use of limited private types is a simple accounting system. A bank account is a resource that clearly has to be protected from arbitrary access, in order for the accounting system to have any value. Since money is a resource which must be *conserved* by all financial transactions, it makes no sense to allow arbitrary assignments, since an assignment does not conserve money, and can therefore be trivially be used to steal money from the account.

Consider the following example from the (otherwise) excellent paper by Seidewitz [Seidewitz92]:

```
package Bank is
 type ACCOUNT is limited private;
 procedure Open(The_Account: in out ACCOUNT; With_Balance: MONEY);
 procedure Deposit(Into_Account: in out ACCOUNT; The_Amount: MONEY);
 procedure Withdraw(From_Account: in out ACCOUNT; The_Amount: MONEY);
 function Balance_Of(The_Account: ACCOUNT) return MONEY;
private
 type ACCOUNT is
  record
   Balance: MONEY := 0.00;
  end record;
 end Bank;
```

Copyright © 1992 by Nimble Computer Corporation. 

```
with Bank,Ada_IO; use Bank,Ada_IO;
procedure PETTY_THIEF is
 anAccount: ACCOUNT;                                    -- The target account.
 procedure Steal_10_From(account1: in out ACCOUNT) is
  procedure Accomplice(account2: in out ACCOUNT) is
   begin
    Withdraw(account2,10.00); Put_Line("Legitimately withdrew $10.00");
    Withdraw(account1,10.00); Put_Line("Stole $10.00");
   end Accomplice;
  begin
   Accomplice(account1);
  end Steal_10_From;
 begin
  Open(anAccount,12.00);          -- Open the account with a trivial amount.
  Steal_10_From(anAccount);
  Put("Balance: $"); Put(Balance_Of(anAccount)); New_Line; -- Prints $2.00.
 end PETTY_THIEF;
```

Our petty thief has opened an account with $12.00, withdrawn $10.00, stolen $10.00, and the account still has $2.00 in it! Furthermore, the bank itself is not aware of what is going on, either. But this is just petty theft. Consider the following *professional* thief:

```
with Bank,Ada_IO; use Bank,Ada_IO;
procedure WILLY_SUTTON is
 anAccount: ACCOUNT;                                    -- The target account.
 procedure Steal_Amount_From(account1: in out ACCOUNT; amount: MONEY) is
  amount_to_steal: MONEY := amount;                   -- Amount left to steal.
  procedure Accomplice(account2: in out ACCOUNT; amount: MONEY) is
   begin
    Withdraw(account1,amount); Put("Stole $"); Put(amount); New_Line;
   end Accomplice;
  begin
   while amount_to_steal > 100.00 loop
    Accomplice(account1,100.00);
    amount_to_steal := amount_to_steal-100.00;
    end loop;
   if amount_to_steal > 0.00 then
    Accomplice(account1,amount_to_steal); amount_to_steal := 0.00;
   end if;
  end Steal_Amount_From;
 begin
  Open(anAccount,102.00);           -- Open the account with a small amount.
  Steal_Amount_From(anAccount,100000.00);
  Put("Balance: $"); Put(Balance_Of(anAccount)); New_Line; -- Prints $102.00.
 end WILLY_SUTTON;
```

What's going on here? How is it that we can steal an arbitrary amount of money from a limited private bank account?

The problem stems from Ada83's laxity regarding the implementation of **`IN OUT`** parameters. Ada83 allows an implementation to implement **`IN OUT`** either by "reference", or by "copy-in, copy-out", or either, in a completely non-deterministic manner. If an implementation implements "by-reference" **`IN OUT`** parameters for limited types, then the bank will have no problem, because all of the accounts—`anAccount`, `account1` and `account2`—are *all the same*, and the bank will not give out more money than the account possesses.

The theft occurs when **`IN OUT`** parameters for the limited private type are passed by "copy-in, copy-out". The problem begins with `account1` and `account2`. `account2` is initialized from `account1` with the correct amount of money, but since this initialization leaves `account1` with its

Copyright © 1992 by Nimble Computer Corporation. 

money intact, our robber can steal from `account1`. Later, when `account2` is copied back into `account1`, the theft is covered up, and our robber can go through the same cycle ad infinitum.

But the bank's Ada lawyer will cry "Fowl!", because our bank robber has done something "erroneous". The bank robber has *aliased* the names `account1` and `account2`, and then accessed `account1` within the scope of `account2`. Sutton's dependence upon this aliasing is "erroneous", according to the Ada Language Reference Manual [6.2/13], but the bank's dependence upon "by-reference" semantics is also erroneous [6.2/7].

But bank robbers usually know that what they are doing is "erroneous"—that's why they try to escape and hide! The looters in the recent L.A. riot knew that what they were doing was "erroneous". The "erroneousness" of double parking doesn't seem to have any effect on Boston or New York drivers, either. In other words, a warning of such "erroneousness" without an effective enforcement mechanism is like shouting in a hurricane.

Of course, the bank can guarantee "by-reference" semantics by using an access type to every account record. The additional level of indirection will solve the aliasing problem by making `anAccount`, `account1` and `account2` all refer to an account *pointer*, rather than directly to an account record. Using this policy, the account pointers will never be modified—only the accounts themselves—and therefore the aliasing cannot cause any difficulties. But if the pointers will never be modified, we don't need the complexity of mode **`IN OUT`** anymore, since the value copied out will always be the same as the value copied in; mode **`IN`** would be simpler, more efficient, and even more error-resistant, since copy-in, copy-out **`IN OUT`** parameters still have other aliasing problems.

Suppose that our bank "wised up" and reprogrammed its accounts to use pointers. However, even after going to this trouble, our bank is still not secure from copy-in, copy-out implementations of mode **`IN OUT`** parameters which the user programs himself. Even relatively benign uses of these mode **`IN OUT`** parameters can still create havoc. Consider the following program:

```
with Bank,Ada_IO; use Bank,Ada_IO;
procedure ABU_NIDAL is
 Account1,Account2: ACCOUNT;                             -- Some bank accounts.
 procedure BENIGN1(x,y: in out ACCOUNT) is
  begin Open(x,10.00); end BENIGN1;
 procedure BENIGN2(x,y: in out ACCOUNT) is
  begin Open(y,10.00); end BENIGN2;
 begin
  BENIGN1(Account1,Account1);   -- One of these two will lose an open account.
  BENIGN2(Account2,Account2);   --  "        "          "
 end ABU_NIDAL;
```

What happens in this example is that each `Account` variable is default-initialized to the `null` pointer. Within the `BENIGN1` procedure, the first argument account is opened with an initial balance of $10.00, while within the `BENIGN2` procedure, the second argument account is opened with an initial balance of $10.00. In the process of opening an account, the Bank allocates an account record from the heap and assigns it to the `Account` variable. However, the caller of `BENIGN1` has aliased the two parameters so that the `Account1` variable is copied in-to both `x` and `y`, and after the procedure has finished, both `x` and `y` are copied back into `Account1`. Depending upon the order of copying, `Account1` will be left with either the initialized account or the uninitialized account. Similarly, `BENIGN2` checks the other ordering of copy-out. Thus, if a compiler always uses the same ordering for copy-out, then we are *guaranteed* that one or the other of the two calls will cause an initialized account record to be lost. In other words, if accounts are repeatedly being aliased and opened, we can make the bank "lose" as many account records as we please, and eventually cause the Bank to crash from insufficient record space. Of course, it has cost us $10.00 for each record we cause the Bank to lose, but we can make this amount arbitrarily small to maximize the confusion caused per dollar invested.

Copyright © 1992 by Nimble Computer Corporation. 

Thus, even if we allow the Bank even a single use of **`IN OUT`**—the parameter to `Open`—then the Bank can be compromised. We have claimed, above, that virtually *every* use of limited types protects a resource of some sort, and we now claim that virtually <u>every resource protected by such a limited type can be compromised by the aliasing of **`IN OUT`** parameters</u>, as in the examples given above. Thus, it is *never* safe to use **`IN OUT`** parameters with a limited type, so long as Ada allows **`IN OUT`** parameters to be passed by copy-in, copy-out.[1] In other words, the use of **`IN OUT`** mode *always* puts the resource at risk, and therefore should *always* be avoided.

## Ada9X Should Resurrect the Original Requirement to Pass Limited Types by Reference

If the most obvious combination of a language's features is inherently unsafe, then something is terribly wrong with the language. In the case of Ada83, the problem lies squarely with its refusal to require by-reference behavior for **`IN OUT`** mode parameters of a limited type.

Many of the usual reasons for not using by-reference semantics for **`IN OUT`** parameters do not exist for limited private types. All derivations of such a type outside the defining package must have the same representation, because no external deriver can know enough about the type to give a proper representation clause. Thus, any type conversions will always be trivial, and can be performed without requiring any physical change in the representation. Thus, efficient by-reference semantics can always be achieved for limited private types whose representations are composite.

Another use of copy-in, copy-out is for a distributed system, so that pointers do not constantly have to be dereferenced across a communication link. However, since limited types in such a system are protecting a *global* resource (why else communicate values of the type across the link in the first place), which can be seen by many different users, it is even *more* imperative that by-reference semantics be used in this case, because the temptation to use aliases is even greater in a distributed system due to "efficiency" pressures.

Thus, we argue that not only will the requirement to make mode **`IN OUT`** for limited types always by-reference be completely upwards compatible in that every non-erroneous program will continue to work, we make the stronger claim that previous users of **`IN OUT`** mode for limited types will now achieve a much greater level of security, since the Willy Suttons of the world will not be able to rob their resource banks.

## Discussion

The STEELMAN requirements for Ada83 were quite adamant about the problem of aliasing:

> "The language shall attempt to prevent aliasing (i.e., multiple access paths to the same variable or record component) that is not intended, but shall not prohibit all aliasing. Aliasing shall not be permitted between output parameters nor between an input-output parameter and a nonlocal variable. Unintended aliasing shall not be permitted between input-output parameters. A restriction limiting actual input-output parameters to variables that are nowhere referenced as nonlocals within a function or routine, is not prohibited. All aliasing of components of elements of an indirect type shall be considered intentional." [STEELMAN78,7I].

Unfortunately, those responsible for STEELMAN greatly underestimated the cost of detecting aliasing, and therefore Ada83 attempts to prevent aliasing through jaw-boning alone. They also underestimated the havoc that would result from copy-in, copy-out in the presence of such undetected and unpunished aliasing. STEELMAN's other admonitions give better advice:

> "The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors." [STEELMAN78,1B].

---

[1]Even worse, Ada83 *requires* limited scalar types to be passed by copy, so such types are inherently unsafe. Of course, scalar limited types are rarely used anyway, because the type designer cannot guarantee initialization in Ada83.

Copyright © 1992 by Nimble Computer Corporation.

> "*There shall be no language restrictions that are not enforceable by translators.*" [STEELMAN78,1F].
>
> "The language shall be completely and unambiguously defined." [STEELMAN78,1H].

Since a by-reference implementation removes the sting from aliasing, Ada9X should "declare a victory and move on", by allowing aliasing but requiring by-reference implementation. The current Ada83 strategy is "broke", and so it is incumbent upon the Ada9X committee to "fix it".

## REFERENCES

Ada83LRM. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Beidler, John. "Relaxing the Constraints on Ada's limited private Types through Functional Expressions". Ada Letters XII, 2 (Mar/Apr 1992),57-61.

STEELMAN Dept. of Defense STEELMAN requirements for high order computer programming languages. June, 1978.

Copyright © 1992 by Nimble Computer Corporation. 