Access Types

university.adacore.com

Dynamic Memory

Access types design

- Java references, or C/C++ pointers are called access types in Ada
- An object is associated to a pool of memory
- Different pools may have different allocation / deallocation policies
- Without doing unchecked deallocations, and by using pool-specific access types, access values are guaranteed to be always meaningful
- In Ada, access types are typed

```
type Integer_Access is access all Integer;

V : Integer_Access := new Integer;
```

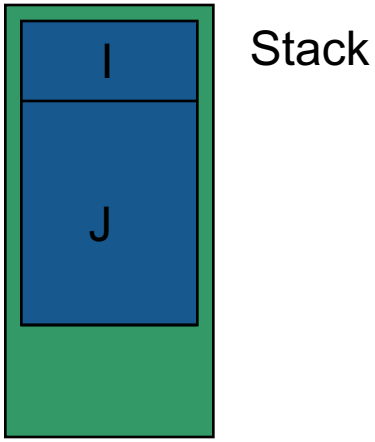
```
/* in C */
int * V = malloc (sizeof (int));

/* or in C++ */

int * V = new int;
```

Access types are dangerous

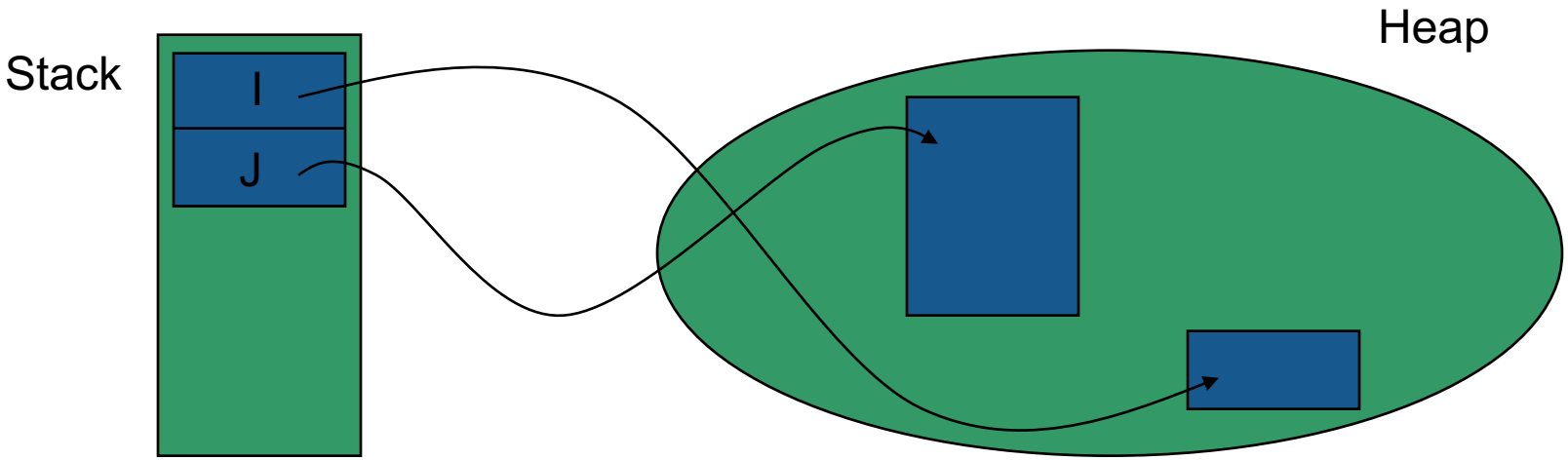
- **Multiple memory issues**
  - Leaks / corruptions
- **Introduces potential random failures that are complicated to analyze**
- **Increase the complexity of the data structures**
- **May decrease the performance of the application**
  - Dereferences are slightly more expensive than direct access
  - Allocations are a lot more expensive than stacking objects
- **Ada avoids the use of addresses as much as possible**
  - Arrays are not pointers
  - Parameters are implicitly passed by reference
- **Only use them when needed**

Stack vs Heap

```
I : Integer := 0;
J : String := "Some Long String";
```



```
I : Access_Integer := new Integer'(0);
J : Access_String := new String'("Some Long String");
```



Ada Access Types

General access types

- **Can point to any pool (including the stack)**

```
type T is […]
type T_Access is access all T;
V : T_Access := new T;
```

- **Access Types are distinct types**

```
type T_Access_2 is access all T;
V2 : T_Access_2 := T_Access_2 (V);
```

- **Note – there is a pool-specific access type (without all), introducing additional constraints discussed later**

```
type T is […]
type T_Access is access T;
V : T_Access := new T;
```

Null values

- **A pointer that does not point to any actual data has a null value**
- **Without an initialization, a pointer is null by default**
- **null can be used in assignments and comparisons**

```
   type Acc is access all Integer;

   V : Acc;
begin
   if V = null then
      --  will go here
   end if

   V := new Integer'(0);
   V := null; --  semantically correct, but introduces a leak
```

Allocations

- **Objects are created with the "new" reserved word.**
- **The created object must be constrained with the constraint given during the allocation**

```
V : String_Access := new String (1 .. 10);
```

- **The object can be created by copying an existing object – using a qualifier**

```
V : String_Access := new String'("This is a String");
```

Deallocation

dependency on the generic subprogram

```
with Ada.Unchecked_Deallocation;

procedure P is
   type An_Access is access all A_Type;

   procedure Free is new Ada.Unchecked_Deallocation (A_Type, An_Access);

   V : An_Access := new A_Type;
begin
   Free (V);

end P;
```

V is null after the call

Creation of the deallocation procedure (instance of the generic subprogram)

Specify first the type and then the access type.

Dereferencing pointers

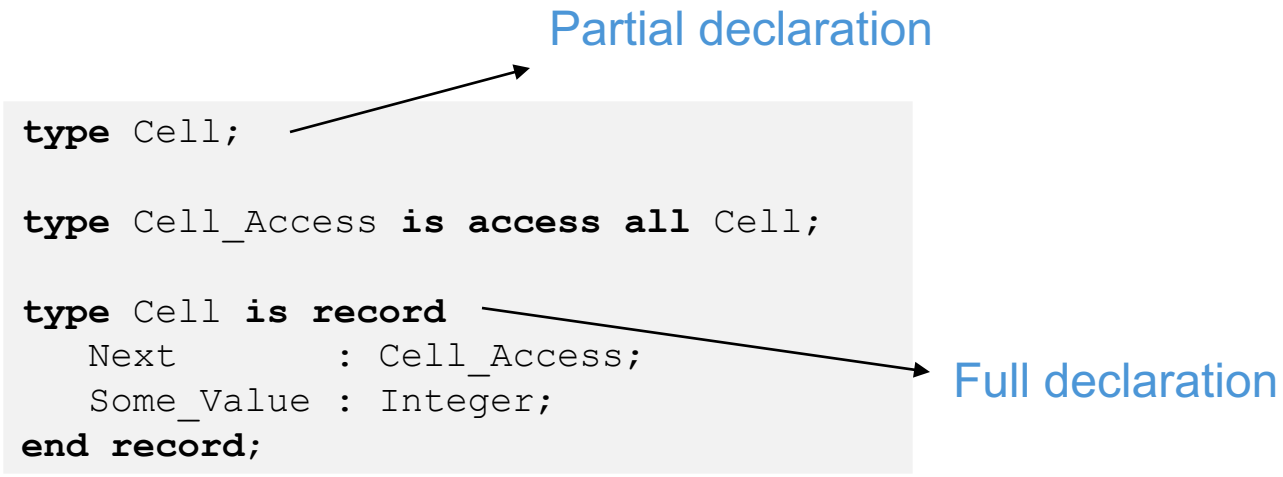
- ***.all*** **does the access dereference**
  - Lets you access the object pointed to by the pointer
- ***.all*** **is optional for**
  - Access on a component of an array
  - Access on a component of a record

Dereference examples

```
type R is record
   F1, F2 : Integer:
end record;

type A_Int is access all Integer;
type A_String is access all String;
type A_R is access all R;

V_Int    : A_Int := new Integer;
V_String : A_String := new String'("abc");
V_R      : A_R := new R;

[…]

V_Int.all := 0;
V_String.all := "cde";
V_String (1) := 'z'; -- similar to V_String.all (1) := 'z';
V_R.all := (0, 0);
V_R.F1 := 1; -- similar to V_R.all.F1 := 1;
```

Using pointers to create recursive structures

- **It is not possible to declare a recursive structure**
- **But there can be an access to the enclosing type**



Referencing the Stack

- By default, object cannot be referenced – and can be optimized out in registers by the compiler
- "aliased" declares an object to be referenced through an access
- " 'Access " gives a reference to the object

```
V : aliased Integer;
A : Int_Access := V'Access;
```

Accessibility Checks

The Accessibility Check Issue

- **Consider the following situation**

```
type Acc is access all Integer;
G : Acc;

procedure P is
   V : aliased Integer;
begin
   G := V'Access;
end P;
```

- **Is the following safe?**

```
G.all := 0;
```

- **Ada forbids the above**

Definition of Nesting Level

- **The "Accessibility Level" of an object is its depth in the subprograms hierarchy**

```
package body P is
   V : aliased Integer; -- Level 0 / Library-Level

   procedure Proc is
      V : aliased Integer; -- Level 1

      procedure Nested is
         V : aliased Integer; -- Level 2
      begin
         null;
      end Nested;
   begin
      null;
   end Nested;
end P;
```

- **A pointer type declared at library level can only point to level 0 objects.**

Access Types and Accessibility Checks

```
package body P is
   type T0 is access all Integer;
   A0 : T0;
   V0 : aliased Integer;

   procedure Proc is
      type T1 is access all Integer;
      A1 : T1;
      V1 : aliased Integer;
   begin
      A0 := V0'Access;
      A0 := V1'Access;
      A0 := V1'Unchecked_Access;

      A1 := V0'Access;
      A1 := V1'Access;
      A1 := T1 (A0);
      A0 := T0 (A1);

      A1 := new Integer;
      A0 := T0 (A1);
   end Proc;
end P;
```

- **To avoid having to face these issues, avoid nested access types**

Getting Around Accessibility Checks

- **Sometimes it is OK to use unsafe accesses to data**
- **“ 'Unchecked_Access ” allows to access to a variable of an incompatible accessibility level**
- **Beware of potential problems!**

```
type Acc is access all Integer;
G : Acc;

procedure P is
   V : aliased Integer;
begin
   G := V'Unchecked_Access;
end P;
```

Quiz

Is this correct? (1/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
type An_Access is access all Integer;

W : Integer;
V : An_Access := W'Access;
```

Is this correct? (1/10)

NO

```
type An_Access is access all Integer;

W : Integer;
V : An_Access := W'Access;
```

W must be aliased to be accessed from an access type.

Is this correct? (2/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
type An_Access is access Integer;

W : aliased Integer;
V : An_Access := W'Access;
```

Is this correct? (2/10)

NO

```
type An_Access is access Integer;

W : aliased Integer;
V : An_Access := W'Access;
```

Access needs to be "all" to point to aliased variables.

It is recommended to always use “access all” for now

Is this correct? (3/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
type An_Access is access all Integer;

procedure Proc is
   W : aliased Integer;
   X : An_Access := W'Access;
begin
   null;
end Proc;
```

Is this correct? (3/10)

NO

```
type An_Access is access all Integer;

procedure Proc is
   W : aliased Integer;
   X : An_Access := W'Access;
begin
   null;
end Proc;
```

X is of accessibility level 0, W of accessibility level one.

Allowing this declaration would permit reference to the object outside of the scope.

Is this correct? (4/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
type R is record
   F1, F2 : Integer;
end record;

type R_Access is access all R;

procedure Proc is
   V : R_Access := new R;
begin
   V.F1 := 0;
   V.all.F2 := 0;
end Proc;
```

Is this correct? (4/10)

YES

```
type R is record
   F1, F2 : Integer;
end record;

type R_Access is access all R;

procedure Proc is
   V : R_Access := new R;
begin
   V.F1 := 0;
   V.all.F2 := 0;
end Proc;
```

Fields of V can be accessed directly or through .all.

However there is a memory leak of the allocated R

Is this correct? (5/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
G : aliased Integer;

procedure Proc is
   type A_Access is access all Integer;
   V : A_Access;
begin
   V := G'Access;
end Proc;
```

Is this correct? (5/10)

YES

```
G : aliased Integer;

procedure Proc is
   type A_Access is access all Integer;
   V : A_Access;
begin
   V := G'Access;
end Proc;
```

A_Access is of level 1. It can point to object of level 0 such as G, its scope is smaller anyway.

Is this correct? (6/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
   type R is record
      F1, F2, F3 : Integer;
   end record;

   type R_Access is access all R;
   type R_Access_Access is access all R_Access;

   V : R_Access_Access;
begin
   V := new R_Access;
   V.all := new R;
   V.F1 := 0;
   V.all.F2 := 0;
   V.all.all.F3 := 0;
```

Is this correct? (6/10)

NO

```
   type R is record
      F1, F2, F3 : Integer;
   end record;

   type R_Access is access all R;
   type R_Access_Access is access all R_Access;

   V : R_Access_Access;
begin
   V := new R_Access;
   V.all := new R;
   V.F1 := 0;
   V.all.F2 := 0;
   V.all.all.F3 := 0;
```

V.F1 doesn't work - only one indirection can be omitted, and V is a pointer to a pointer.

Is this correct? (7/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V1 : A_Access := new Integer;
   V2 : A_Access := V1;
begin
   Free (V1);
   Free (V2);
```

Is this correct? (7/10)

NO

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V1 : A_Access := new Integer;
   V2 : A_Access := V1;
begin
   Free (V1);
   Free (V2);
```

V1 and V2 point to the same object, so the second Free should (hopefully) raise a Storage_Error.

Is this correct? (8/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V : A_Access;
begin
   Free (V);
   V := new Integer;
   Free (V);
   Free (V);
```

Is this correct? (8/10)


```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V : A_Access;
begin
   Free (V);
   V := new Integer;
   Free (V);
   Free (V);
```

V is initialed to null, so the first free doesn't do anything. The second actually frees the object and sets it back to null. The last one, again, calls on null.

Is this correct? (9/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V : A_Access;
   W : aliased Integer;
begin
   V := W'Access;
   Free (V);
```

Is this correct? (9/10)

NO

```
   type A_Access is access all Integer;

   procedure Free is new Ada.Unchecked_Deallocation
      (Integer, A_Access);

   V : A_Access;
   W : aliased Integer;
begin
   V := W'Access;
   Free (V);
```

We're trying to free an object on the stack here, which is not possible.

Is this correct? (10/10)

YES
(click on the check icon)

NO
(click on the error location(s))

```
type A_Access is access all Integer;

type R is record
   V : An_Access;
   W : aliased Integer;
end record;

G : R;

procedure P is
   L : R;
begin
   G.V := G.W'Access;
   L.V := L.W'Access;
end P;
```

Is this correct? (10/10)

NO

```
type An_Access is access all Integer;

type R is record
   V : An_Access;
   W : aliased Integer;
end record;

G : R;

procedure P is
   L : R;
begin
   G.V := G.W'Access;
   L.V := L.W'Access;
end P;
```

L is a local object, its accessibility level is 1.
V is a pointer type expecting an accessibility level of at most 0.

university.adacore.com