intro (5min)

- **who I am**
- **-- SIPB, professional developer for 8 years**
- **why I care**
- **-- lots of experience with c/c++ problems**
- **how I prepared**
- **-- studied the language and the net**

ada the language (1hr)

- syntax
- raw syntax
- numeric operators
- type conversion
- higher order structure
- Tasking is integrated into the language
- common C errors

syntax

- **block structure is marked more clearly/verbosely**
- **; is a terminator not a separator**
- **69 reserved words**
- **types are more sophisticated**
- **arrays have ranges, slices, sliding**
- **type operators**
- **pointers are more controlled:**
- **strongly typed assignment var:=expression;**
- **other type issues**

block structure is marked more clearly/verbosely

```
if ... then ... elsif ... then ... else ... end if;
loop ... exit when ... end loop;
case expr is when value => action; ... end case;
procedure name ... is ... begin ... end name;
for name in reverse 1 .. N loop .. end loop;
while condition loop .. end loop;
null; -- needed *explicitly* in a number of contexts
<<label>>   goto label;
```

case expr is when value => action; ... end case;

- **-- when others => default action;**
- **-- (others must be last, and is ok if none left)**
- **-- when range => action;**
- **-- when val1 | val2 => action;**
- **-- values must be static**
- **-- *all* possible values must be satisfied**

<<label>>   goto label;

-- for machine generated code

-- can't go into a if/case/loop or across arms of if/case

69 reserved words

```
begin ... exception when EX => action; end;
declare ... begin ... end;
varname: type; -- in declarations
cname: constant type := initvalue;
cname: constant := literal number;
```

begin ... exception when EX => action; end;

-- like C++ catch

-- parallel syntax to Ada case...

-- exceptions are in namespace, if you can name it you can raise it

declare ... begin ... end;

-- declarations ordered, like let*

-- safer name:type; syntax

cname: constant := literal number;

-- "names" the number

-- *without* nailing down a type...

- **types are more sophisticated**
  - **type name is (enum, values);**
  - **type newname is new oldname; -- explicit convert only**
  - **constrained types**
  - **type name is record ... end record; -- just like struct**
  - **type name is tagged ...; -- inheritable, extendable**
  - **type newname is new name with ...; -- with null record**
  - **subtype days is Integer range 1..31;**
  - **aggregate construction**

constrained types

```
type name is digits 7;
-- pick a predefined type matching constraint
type name is range -1_000 .. +1_000;
-- constrained integer
-- ranges can be null, can be dynamic
```

```
subtype days is Integer range 1..31;
    -- same ops, fewer values
    -- not a newtype, just a named constraint
    -- most places a range can be used explicitly
```

aggregate construction

- **-- positional (4, Jul, 1776);**
- **-- by name (month=>jul, day=>4, year=>1776);**

arrays have ranges, slices, sliding

- **type name is array (a..b) of type;**
- **type String is array (Positive range <>) of Character;**
- **name(0..2) := (3,2,1);**
- **name(3..5) := "XYZ";**
- **name(4..5) = ('X', 'Y', others => ' ');**
- **array assignment**
- **example matrix type**
- **array of array AOA(i)(j)**
- **multi dimensional array MDA(i,j)**

```
type String is array (Positive range <>) of Character;
    -- <> is "box" giving an "unconstrained" array
    -- useful in arguments, the range supplied later
```

array assignment

between *same* type, not merely conforming

if the type name matches, it's enough that

the *dimensions* match, assignment will "slide".

example matrix type

```
type Mat is
  array (Integer range <>, Integer range <>) of Float
-- must nail *both* or *neither* in subtype
-- can even "deduce" them in place:
U2: constant Mat := ((1.0,0.0),(0.0,1.0));
-- but this uses Integer'first as bound!
-- so use Positive range <> instead...
```

type operators

all discrete types (including float) have

-- name’Length name’First name’Last name’Range

-- name’Succ(value) name’Pred(value)

non-reals also have

-- name’Pos(value)=int name’Val(int)=value

pointers are more controlled:

```
type name_ptr is access name;

var: name_ptr;

var := new name;

var.part  (or var.all.part) var.all
```

strongly typed assignment var:=expression;

- **both sides must have same type**
- **constraints on var must be satisfied**
- **var1,var2 := expression -- evaluates expression twice!**

other type issues

expressions are "dynamic" or "static"

arguments are in, out, in out

functions have a return value, procedures don’t

-- so functions can be used in expressions

raw syntax

- **’x’ is a character**
- **"xyz" is a string of characters**
- **-- "A""B" is (’A’,’"’,’B’)**
- **-- must fit on a single line**
- **-- must not contain control characters**
- **-- case in strings/literals is preserved**
- **numbers**
- **identifiers**
- **comments**
- **1-d array concatenation: "foo" & "bar" = "foobar"**

numbers

35, 1_024, 1e6 (integer!), 3.14159_26536

not 1. or .1

base notation: 2#111# 14#ABC# 2#11.1#e11

-- (base and exp always decimal)

literal numbers are never negative

identifiers

- case insensitive
- iso-latin-1 so accents in identifiers are ok
- identifiers must begin with letters
- max len at least 200 characters, all significant

comments

- **in comments, *anything* is ok**
- **comments begin with -- *only*, like // comments**

1-d array concatenation: "foo" & "bar" = "foobar"

- **-- also takes an element as lhs or rhs (or both)**
- **-- "line 1" & Ada.Characters.Latin_1.LF;**

numeric operators

- **arithmetic:**
- **Integer only operators**
- **(exp) ** if lhs is Integer, rhs is non negative integer**
- **relational operators**
- **boolean operators: (bool x bool -> bool)**
- **precise semantics**
- **precedence:**
- **precedence conflicts forbidden**
- **ranges scalar in, not in range**
- **shortcuts**
- **overloading**

arithmetic:

- **types of left and right must match**
- **result is same type**
- **unary or binary + -**
- **binary ***
- **binary / -- integer / trucates towards 0**
- **abs unary**

Integer only operators

- **rem - remainder (sign of lhs)**
- **mod - modulo (sign of rhs) but see modular types**
- **also: (A+B) mod n = (A mod n)+(B mod n) mod n**

relational operators

- boolean (True,False) return
- lhs/rhs types must match
- = /= < <= > >=

boolean operators: (bool x bool -> bool)

- **not (precedence of abs)**
- **and or xor (equal, but lower than comparators)**
- **-- so can't be mixed without parens to disambiguate**

precise semantics

- **conversion from float to intager always rounds**
- **-- halfway values round away from zero**
- **order of operations is *not* specified.**

precedence:

- **lowest: comparators**
- **binary + -**
- **unary + -**
- **/ * mod rem**
- **highest: ** abs**

precedence conflicts forbidden

a**b**c is forbidden without parens.

(similarly ** with abs)

note that -A**B = -(A**B), not (-A)**B

- ranges scalar in, not in range
  - same precedence as comparators

shortcuts

and then (like C &&), or else (like C ||)

-- these are *not* strictly operators

-- they can’t be overloaded

overloading

- **any operator in "" can be a function name.**
- **can’t change syntax**
- **if = returns boolean, /= is created implicitly**
  - **and can’t be replaced**

type conversion

- **Float(I) *converts* I**
- **Integer’(I) qualifies I**
- **double conversion**

double conversion

```
a: float;
procedure x(r: in out integer);
x(Integer(a)); -- a converted, passed, converted back...
```

higher order structure

- class vs. package
- package just groups things in a namespace
- hierarchical grouping
- a C++ "class" is just a tagged type
- a C++ "template" is like a generic

package just groups things in a namespace

```
package name is
   -- declarations
   type argtype is private;
private
   type argtype is ...; -- real type
end name;
package body name is
    -- definitions
end name;
-- package provides context for scope
```

hierarchical grouping

```
use type;
package subtype; -- type.subtype
package outername renames type.subtype;
```

a C++ "class" is just a tagged type

- **derivation: type der_x is new x with newstuff;**
- **type x is tagged private;**
- **type der_x is new x with private;**
- **functions/procedures can be overridden**
  for more specific types
- **explicit conversion discards or adds:**
  x := (y with extravals);
- **type'Class means "any/all types derived from type"**
  which can be used explicitly in declarations
- **thus, access type'Class is a useful generic pointer**
- **abstract specifies that no objects can be constructed.**

a C++ "template" is like a generic

```
generic
   type Num is digits <>;
   -- can instantiate for any type "like" that
package dostuff is ... end dostuff;
package dostuff_x is new dostuff(x);
```

Tasking is integrated into the language

* tasks are simply separate flows of control...
* protected types as expressions of locking

* tasks are simply separate flows of control...

```
task server is
 entry name(args); -- just like a procedure declaration
end
task body server is
begin
 accept name(args) do
   -- body, just like a procedure
 end
end server;
task client;
task body client is
begin
 server.name(42); -- tightly coupled "rendezvous"
end client;
```

* protected types as expressions of locking

In order to keep objects from getting bit on multiple invocations, "protected" types are protected from simultaneous access:

```
protected type name(args) is
 -- procedure declarations
private
 -- data definitions
end name;
protected body name is
 -- implementations
end name;
```

In this case, it’s not just a package, it’s a complete type, so

```
var: name(42);
```

is used directly.

common C errors

- = vs ==
- [x,y] for [x][y]
- fencepost errors

how to get stuff

- gnat is avaiable for many unix platforms, win95/nt
- tenon MachTen mac port
- compiler (net): ftp.nyu.edu:pub/gnat
- compiler (athena): add gnat; gnatmake ...
- adahome: http://www.adahome.com/
- lecture notes:
  - http://www.mit.edu/people/eichin/ada.html