Ada on the Web
The Web page, Ada Home

```
http://www.adahome.com
```

is by far one of the most helpful Ada Web sites around. It provides resources and links to such resources as compilers, books, Web-based tutorials, and FAQs.

The ADA tutorial at URL http://www.adahome.com/Ammo/cpp2ada.html is recommended.

Ada on acad
The *gnat* compiler is set up on acad. To prepare to use it, add `/usr/bin` to the path string in the file `.login` or `.bashrc` in your home directory. Log out and in again (or source .login) and you're ready to go).

Use file type **adb** for your ada source files. Then, type `gnatmake <file name>` to prepare your ada source file for execution. It will compile, bind and link, with the resulting executable having the same name as `<file name>`.

Introduction to Ada
Ada

- is designed to be a modern, general-purpose language;
- components can be independently written, compiled, and tested;
- supports modern software design methodologies;
- supports encapsulation, data abstraction and information hiding;
- supports parallel processing; and
- supports generic software components enhancing reusability of code.

Packages

The Ada package is the primary mechanism for encapsulation and data abstraction. An Ada package is a collection of related resources such as types, objects, subprograms, tasks, generic units, and other packages. Most packages consist of two parts, a declaration or specification and a body.

The package specification names the resources exported by the package. A package specification has the format (-- begins a one-line comment):

```
package Package_Name is
--  Exportable resources are listed here.
end Package_Name;
```

Later we will look at the specification of the predefined package STANDARD.

The package body contains the code that implements the resources listed in the specification. A package body has the format:

```
package body Package_Name is
--  package body contents are contained here.
end Package_Name;
```

Input/output capability is left to the programmer to define through the use of the package structure. All Ada implementations provide at least the minimal input/output capabilities contained in the package Text_IO.

Note that Ada has different conventions for identifiers, and is case sensitive regarding their use.

Simple Types

The simple types in Ada are the same as those in C/C++, but the names are different in most cases. The four most commonly used simple types are in the box. Note that Ada probably has the strongest typing of any computer language, and the following is illegal:

Integer
Float
Character
Boolean

```
-- declare new type
type INT is new Integer;
a : INT;
b : Integer;

a := b; -- fails.
```

while this succeeds:

```
subtype INT is Integer;
a : INT;
b : Integer;

a := b; -- works.
```

A subtype is simply a way of declaring a subset of an existing type. Therefore, the subtype is compatible with the type it is derived from. On the other hand, the **type** specifier creates a new type, and the compiler does not comprehend the fact that INT is the same as Integer.

Ada has a **string** type, but it can't be declared without a specification of size. Like Pascal, list types often are indexed from 1, not 0.

Note also the differences in syntax:

- In Ada, variables are declared before executable code
- In Ada, the identifier(s) precede the type in the declaration

Operator	C/C++	Ada
Assignment	=	:=
Equality	==	=
NonEquality	!=	/=
Modulus	%	mod
Remainder		rem
AbsoluteValue		abs
Exponentiation		**
Range		..

Table 1. Ada Operators

Operators

The operators used in Ada have some differences from C++. Table 1 provides a comparison between Ada and C++ operators. Of particular note are the differences between assignment and equality operators. Keep this in mind when fixing compiler errors. Further, Ada does not provide special assignment operators such as +=, but does have operators for exponentiation and (sub)range, which C++ doesn't have.

Arrays, Strings

To declare arrays and strings in Ada, a range of indices is given. To get a 20 character string named str, you would declare **str : string(1..20)** . This string is indexed from 1 to 20. You can't use 0..19 to maintain C++ indexing, as a lower bound of 1 or above is required. But, if you want to pass a string to a subroutine (function), you may leave it unconstrained, i.e. you don't have to give a subrange.

Some versions of Ada have a package *Ada.Strings*, that implements unbounded strings.

Arrays are declared using the same form as the string, but while the string declares one object of size equal to the high bound minus the low bound plus one, an array declares that many elements (string really does that too, with char). To declare an array of 10 integers write *List : integer(1..10);* . Another array with the same number of elements is *List2 :integer(-5..4);* . This array can be declared because the lower bound of an array can be specified by the user.

Here's how to declare a type for an array of Integer: *type IntegerArray is array(1..20) of Integer;* .

Further intro will be via examples; here, and on acad.

Ada program **hello_world**

```
1.   ------------------------------------------------------------------------
2.   -- UNIT    : hello_world main program procedure
3.   -- FILE    : hello_world.adb
4.   ------------------------------------------------------------------------
5.
6.   with Text_IO;    -- The with clause makes type, object and
7.                    -- subprogram names in the predefined
8.                    -- package Text_IO available.
9.
10.  use Text_IO;     -- The use clause makes type, object and
11.                   -- procedure names in Text_IO visible.
12.
13.  procedure hello_world is
14.             -- Declarative part of the subprogram
15.             -- normally would go here.  However, this
16.             -- procedure has no variables, etc.
17.
18.  begin      -- Executable statements begin here.
19.
20.     Put_Line ("Hello, World");      -- The Put_Line routine from Text_IO
21.                                     -- prints the message inside the quotes.
22.
23.  end hello_world; -- Every procedure terminates with an
24.                        -- end statement.  The procedure name is
25.                        -- optional, but its use is a good
26.                        -- practice.
```

Ada program hello_world without the use clause:

```
1.   ------------------------------------------------------------------------
2.   -- UNIT      : hello_world_2 main program procedure
3.   -- FILE      : hello_world_2.adb
4.   -- COMPILE   : compada hello_world_2
5.   -- SOURCE    : Ken Shumate, Understanding Ada, page 3
6.   ------------------------------------------------------------------------
7.
8.   with Text_IO;
9.
10.  procedure hello_world_2 is
11.
12.  begin
13.
14.     Text_IO.Put_Line ("Hello, World");  -- Selects the procedure Put_Line
15.                                         -- from the package Text_IO.
16.  end hello_world_2;
```

Ada program doing integer input/output:

```
1.   ------------------------------------------------------------------------
2.   -- UNIT     : program_1 main program procedure
3.   -- FILE     : program_1.adb
4.   -- SOURCE   : Ken Shumate, Understanding Ada, page 7
5.   ------------------------------------------------------------------------
6.
7.
8.
9.   with Text_IO; use Text_IO;
10.
11.  procedure program_1 is
12.     Sum, Value_1, Value_2 : Integer;
13.     package my_integer_IO is new Integer_IO (Integer); use my_integer_IO
14.
15.  begin
16.     Put ("Enter the first integer : "); Get (Value_1);
17.     New_Line;  -- forces subsequent output to be written on a fresh line
18.     Put ("Enter the second integer : "); Get (Value_2);
19.     New_Line;
20.     Sum := Value_1 + Value_2;
21.     Put ("The sum of the numbers is : "); Put (Sum);
22.  end program_1;
```

Ada program doing integer input/output without the use clauses:

```
1.   ------------------------------------------------------------------------
2.   -- UNIT     : program_2 main program procedure
3.   -- FILE     : program_2.adb
4.   -- SOURCE   : Ken Shumate, Understanding Ada, page 7
5.   ------------------------------------------------------------------------
6.
7.   with Text_IO;
8.
9.   procedure program_2 is
10.     Sum, Value_1, Value_2 : Integer;
11.     package my_integer_IO is new Text_IO.Integer_IO (positive);
12.
13.  begin
14.     Text_IO.Put ("Enter the first integer : ");
15.     my_integer_IO.Get (Value_1);
16.     Text_IO.New_line;
17.     Text_IO.Put ("Enter the second integer : ");
18.     my_integer_IO.Get (Value_2);
19.     Text_IO.New_line;
20.     Sum := Value_1 + Value_2;
21.     Text_IO.Put ("The sum of the numbers is : ");
22.     my_integer_IO.Put (Sum, width => 4);
23.     Text_IO.New_line (3);
24.  end program_2;
```