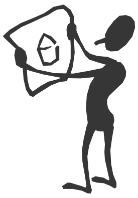
Chapter 4
Data Object and Type

VHDL

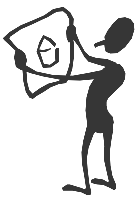
Outline

- Keyword
- Identifiers & Comment
- Data Object
- Data Type
- Scalar Type
- Composite Type
- Pointer Type
- Incomplete Types
- File Type



Reserved Word - Keyword

abs	downto	library	postponed	sri
access	else	linkage	procedure	subtype
after	elsif	literal	process	then
alias	end	loop	pure	to
all	entity	map	range	transport
and	exit	mod	record	type
architecture	file	nand	register	unaffected
array	for	new	reject	units
assert	function	next	rem	until
attribute	generate	nor	report	use
begin	generic	not	return	variable
block	group	null	rol	wait
body	guarded	of	ror	when
buffer	if	on	select	while
bus	impure	open	severity	with
case	in	or	signal	xnor
component	inertial	others	shared	xor
configuration	inout	out	sla	
constant	is	package	sll	
disconnect	label	port	sra	



Identifiers

- **Identifiers = names of things you create**
  - signals, variables, constants
  - architecture names, entity names, component names
  - process names
  - function names, procedure names, etc.
- **Rules**
  - Cannot be reserved words
  - Uppercase and lowercase equivalent
  - Only letters, numbers, and underscore (_)
    - First character is letter
    - First and last character NOT underscore
    - Two underscores in succession illegal

Identifiers Example

- **Which are legal identifiers?**

footer_5	7bits
_load	Execute14_more
doitnow_	Endif
add__subtract	process
don’t_do_it	exor#and
StrongFuzzyLogicDriver	carry/

Extended Identifiers

- An extended identifier is a sequence of characters written between two backslashes. Any of the allowable characters can be used, including characters like., !, @, ',and $. Within an extended identifier, lower-case and upper-case letters are considered to be distinct. Examples of extended identifiers are:
  - \TEST\ -- Differs from the basic identifier TEST.
  - \2FOR$\
  - \process\ -- Distinct from the keyword process.
  - \7400TTL\
  - Two consecutive backslashes represents one backslash \→\\.

Comments

- Comments in a description must be preceded by two consecutive hyphens (--); the comment extends to the end of the line. Comments can appear anywhere within a description. Examples are:
  -- This is a comment; it ends at the end of this line.
  -- To continue a comment onto a second line, a separate
  -- comment line must be started.

- **entity** UART **is end**; --This comment starts after entity declaration.
- Equivalent to // in C & C++.

Objects

- **Objects are things that hold values (containers).**
  - Have a **class** and a **type**
  - May have an explicit initial value (useful for synthesis?)
  - Declared in a package, entity, architecture, or process
  - Visibility limited to region where declared
- **Class determines the kind of operations possible for an object**
- **Type determines the legal values for an object**

- **Classes:**
  - **signal** - value changes as function of time, has a driver; physical wire
  - **variable** - value changes instantly, no concept of time
  - **constant** - value cannot be changed
  - **file** - values accessed from external disk file

Object Declarations

- Syntax
  Class Identifier* : *Type := *InitialValue***;**
- Examples

```
CONSTANT MyNumber  : real      := 3.14159;
SIGNAL   Load_Reg_n: std_logic := 'X';
VARIABLE counter   : bit_vector;

ENTITY ename IS
  Ports
  Declarations  --no variables
END ename;

ARCHITECTURE x OF y IS
  Declarations  --no variables
BEGIN  ...  END x;

PROCESS (a,b)
  Declarations  --no signals
BEGIN  ...  END PROCESS;
```

Signals

- All port are signals.
- Legal only in architecture entity and package .
- Value changes as function of time, has a driver.
- **Physical wire.**
- Store a real data.
- Signal assignments symbol is **<=** for example **x <= a OR b**;
- Signal assignments occurred at the end of the process.
- Changing in the signal value create an event.

```
ARCHITECTURE dataflow OF drive IS
BEGIN
  x <= y;
  y <= z OR a;
  z <= NOT a;
END dataflow;
```


Signal Driver

- Model of 8-input AND gate: x = AND (a_bus(7..0))
- Problem due to scheduling and timing - **x is ALWAYS 0 !!!**
- Why?
- How can it be fixed? (By variable as we will see later)

```
ARCHITECTURE Flaxer OF and8 IS
BEGIN
  anding: PROCESS (a_bus)
  BEGIN
    x <= ‘1’;
    FOR i IN 7 DOWNTO 0 LOOP
      x <= a_bus(I) AND x;
    END LOOP;
  END PROCESS anding;
END Flaxer;
```


Resolution Functions

- What if multiple drivers exist for the same signal?
  - VHDL equivalent of multiple gate outputs wired together
  - Result in hardware if values conflict - weird voltage level, high current flow
  - Result in simulator - unknown logic level ‘X’
  - Why would you create this kind of logic?
- How is this done in VHDL?
  - Multiple concurrent assignment statements
  - Multiple sequential assignment statements <u>in different processes</u>
- Signals with multiple drivers MUST have a special **resolved** type
  - The type has a **resolution function** associated with it that decides the final value: ‘0’, ‘1’, ‘X’, ‘Z’, etc.
  - For example, *std_logic* is a resolved type, while *std_ulogic* is not.
  - What about other types?

Variables

- Legal only in processes (and subprograms)
- Usually used for high-level, algorithmic calculations.
- Easy to write, complex to synthesize.
- Immediate assignments with **:=** for example **x := a OR b**;
- Corrected 8-input AND gate model, using variables:

```
ARCHITECTURE Flaxer OF and8 IS
BEGIN
  anding: PROCESS (a_bus)
    VARIABLE tmp: bit;        <-- New
  BEGIN
    tmp := '1';
    FOR i IN 7 DOWNTO 0 LOOP
      tmp := a_bus(I) AND tmp;
    END LOOP;
    x <= tmp;                 <-- New
  END PROCESS anding;
END Flaxer;
```


Constants

- Legal anywhere.
- The value is assigned to the constant in declaration.
- Usually used for constant value of signal or variable.
- The value can not be changed.

```
ARCHITECTURE Dami OF Coni IS
  CONSTANT RstLevel: bit := '1';
BEGIN
  *
  *
      IF reset = RstLevel THEN
        Q <= "0000";
      ELSE
        Q <= Q + 1;
      END IF;
  *
END Dami;
```

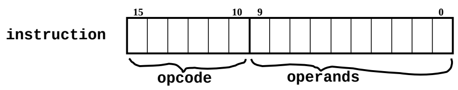

Object Aliases

- Alternative name for an existing object (or part of an object)

ALIAS ***Identifier-Type***: **IS** ***Item-Name***

- May provide better documentation and more readability
- Example:

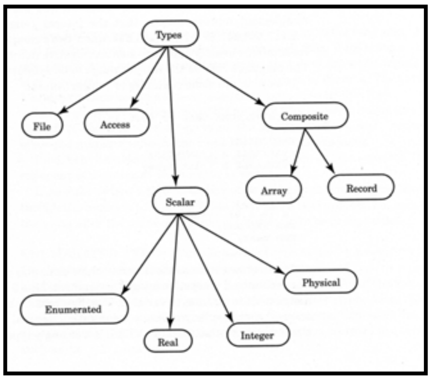
```
SIGNAL instruction: std_logic_vector (15 DOWNTO 0);

ALIAS opcode:   std_logic_vector (5 DOWNTO 0)
                IS instruction(15 DOWNTO 10);

ALIAS operands: std_logic_vector (9 DOWNTO 0)
                IS instruction(9 DOWNTO 0);
```



Types

- The type of an object determines the legal values it may contain
- VHDL is **strongly-typed** language
  - Objects of different base types **cannot** be assigned or compare to one another directly (can use type conversion functions or casting)
- Two major categories of types
  - **Scalar** - holds single value (**enumeration, integer, float, physical**).
  - **Composite** - holds multiple values (**arrays, records**).
  - **Access -** hold pointer to object.
  - **File** - represent a file in the host.
- Types may be predefined or user-defined
  - Predefined types defined in VHDL standards 1076 and 1164
  - User-defined types created by you - very useful

Types



Scalar Types

- **Enumeration Types**
  - List of distinct values an object may hold (similar to enum in C).
  - Predefined type: "boolean", "bit", "character".
- **Integer Types**
  - Set of whole numbers, positive and negative, predefined type **"`integer`"**
  - 32-bit signed values, $-(2^{31}-1)$ to $+(2^{31}-1)$
  - Often use a reduced range for synthesis, e.g.
    `VARIABLE num: integer RANGE -64 TO +64;`
- **Float Types**
  - Floating-point numbers, predefined type **"`real`"**
  - 32-bit single-precision
  - Not for synthesis; hardware too complex
- **Physical Types**
  - Measurement units, predefined type **"`time`"**, (fs, ps, ns, … min, hr)
  - Not meaningful for synthesis

Enumeration Types

- **Enumeration type** = **ordered list of distinct values.**
- The position number of the **leftmost** element is **0**.
- The position number of the next elements is **increment** by **1**.
- The position defines the order (for relational operators).
- **User defined types** - example:

```
TYPE color IS (red, green, blue, black);
SIGNAL MyColor: color; ...
   IF MyColor = red THEN …
   Mycolor <= green;
```

- If the same literal is used in two different enumeration type declarations, the literal is said to be ***overloaded***.

```
TYPE race IS (black, white);
SIGNAL man: race;
man <= black;
```


Standard Enumeration Types - 1076

- <u>**Predefined 1076 types (package standard)**</u>

```
TYPE boolean IS (FALSE, TRUE);
TYPE bit IS ('0', '1');
TYPE character IS (-- 256 ascii code --);
```

nul	soh	stx	etx	eot	enq	ack	bel
bs	ht	lf	vt	ff	cr	so	si
dle	dc1	dc2	dc3	dc4	nak	syn	etb
can	em	sub	esc	fsp	gsp	rsp	usp
' '	'!'	'"'	'#'	'$'	'%'	'&'	'''
'('	')'	'*'	'+'	','	'-'	'.'	'/'
'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'
'8'	'9'	':'	';'	'<'	'='	'>'	'?'
'@'	'A'	'B'	'C'	'D'	'E'	'F'	'G'
'H'	'I'	'J'	'K'	'L'	'M'	'N'	'O'
'P'	'Q'	'R'	'S'	'T'	'U'	'V'	'W'
'X'	'Y'	'Z'	'['	'\'	']'	'^'	'_'
'`'	'a'	'b'	'c'	'd'	'e'	'f'	'g'
'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'
'p'	'q'	'r'	's'	't'	'u'	'v'	'w'
'x'	'y'	'z'	'{'	'|'	'}'	'~'	del

Enumeration Types - 1164

- **Predefined 1164 types**
  - **std_logic**, **std_ulogic**, + their arrays and subtypes
  - All require **library / use** statements before entity declaration:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

- Base type is std_ulogic (unresolved)

```
TYPE std_ulogic IS ('U', -- Uninitialized
                    'X', -- Forcing unknown
                    '0', -- Forcing zero
                    '1', -- Forcing one
                    'Z', -- High impedance (float)
                    'W', -- Weak unknown
                    'L', -- Weak zero
                    'H', -- Weak one
                    '-'  -- Don't care
                   );
SUBTYPE std_logic IS resolved std_ulogic;
```

- Values 'U', 'X', 'W' NOT supported for synthesis
- Literal values are case sensitive; 'z' NOT SAME as 'Z'

Resolved Table of std_logic

Resolved signal can accept multiple values

	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X

Integer Types

- **Integer Types = Set of value fall within a specified integer range.**
- **User defined types** - example:

```
TYPE byte IS RANGE 0 TO 255;
TYPE index IS RANGE MIN TO MAX;
CONSTANT X: byte:= 5;
SIGNAL   Y: index;
```

- The **bounds** of the range for an integer type must be **constant** or ***locally static expression*** (defined at compile time).
- **Integer literals** - value belonging to integer type example:
  - **56349 6E2 0 98_71_28 987128 -1962**
  - **The exponent is power of 10.**
- **Integer can also be written in other base between 2 to 16**
  - ***base # value # E exponent*** (the exponent represent a power of the base).
  - **2#0001_0010_0011# 16#FA# 16#4#E2 ⇒ ($4*16^2 = 1024$)**

Standard Integer Types

- <u>**Predefined 1076 types (package standard)**</u>

```
– TYPE integer IS RANGE implementation_defined;
```
- **32-bit signed values ⇒ $-(2^{31}-1)$ to $+(2^{31}-1)$**
```
– VARIABLE temp: integer;
– VARIABLE num:  integer RANGE -64 TO +64;
– SIGNAL MyChar: integer RANGE -128 TO +127;
– CONSTANT M_LEN: integer:= 100;
```

Floating Point Types

- **Floating Point Types = Set of value fall within a specified real range.**
- **User defined types** - example:

```
TYPE TTL_VOL IS RANGE -0.5 TO 5.5;
TYPE real_data IS RANGE MIN TO MAX;
CONSTANT X: TTL_VOL := 5.0;
SIGNAL   Y: real_data;
```

- The **bounds** of the range for a real type must be **constant** or ***locally static expression*** (defined at compile time).
- **Floating Point literals** - value belonging to F.P. type example:
  - **16.26 0.0 -0.002 3.1415 3.14_15 62.3E-2 5.0E+2**
  - **The exponent is power of 10.**
  - **The literal must include a dot (.)**
- **Floating Point can also be written in other base between 2 to 16**
  - ***base # value # E exponent*** (the exponent represent a power of the base).
  - **2#0110.0100# ⇒ (6.25) 2#1.01#E3 ⇒ (10.0)**

Standard Floating Point Types

- **Predefined 1076 types (package standard)**

```
– TYPE real IS RANGE implementation_defined;
– 32-bit single precision ⇒ -1.000000E38 to +1.000000E38
– VARIABLE temp: real;
– VARIABLE num:  real RANGE 0.0 TO 100.0;
– SIGNAL MyReal: real RANGE -1.0E6 TO +1.0E6 ;
– CONSTANT PI:    real:= 3.1415;
```


Physical Types

- **A physical type** contains values that represent measurement of some physical quantity, like time, length, voltage, or current. Values of this type are expressed as integer multiples of a base unit.
- **User defined types** - example:

```
TYPE current IS RANGE 0 TO 1E9
  UNITS
    nA; -- base unit
    uA  = 1000 nA;
    mA  = 1000 uA;
    Amp = 1000 mA;
  END UNITS;
```

- **Physical literals** - are written as an integer or real literal followed by the unit name. For example:
  - **100 ns 10 V 50 sec 5.6 uA ⇒ (5600) 5.6 nA ⇒ (5)**

Standard Physical Types

- **Predefined 1076 types (package standard)**

```
TYPE time IS RANGE implementation_defined;
  UNITS
      fs;                  -- femtosecond
      ps    = 1000 fs;     -- picosecond
      ns    = 1000 ps;     -- nanosecond
      us    = 1000 ns;     -- microsecond
      ms    = 1000 us;     -- milisecond
      sec   = 1000 ms;     -- second
      min   =   60 sec;    -- minutes
      hr    =   60 min;    -- hours
  END UNITS;
```


Types and Subtypes

- **Subtype is a type with a constraint.**
- ***SUBTYPE NewName IS ExistingType constraint***
  - –For example**: SUBTYPE char IS integer RANGE -128 to 127;**
- **Be careful with type-matching problems**:

```
TYPE    byteT IS INTEGER RANGE 0 to 255; --new type
SUBTYPE byteS IS INTEGER RANGE 0 to 500; --new subtype
SIGNAL  byte2:   INTEGER RANGE 0 to 255;   --integer
SIGNAL  word:    INTEGER RANGE 0 to 65535; --integer
SIGNAL  byte1: byteT;
SIGNAL  byte3: byteS;
byte1 <= byte2;  --Type mismatch!
byte3 <= byte2;  --OK
word  <= byte2;  --OK
```

- The SubTypeObject can always assigned to the the TypeObject.
- The TypeObject can assigned to the the SubTypeObject if the value is in the constraint range.

Composite Types - Array

- **Array Types = multiple elements of same type.**
- **TYPE *ArrayTypeName* IS ARRAY (*Valid RANGE)* OF *ExistingType.***
  - For example**:**

```
TYPE AddrBus IS ARRAY (0  TO  11) OF bit;
SIGNAL   AB1, AB2: AddrBus;
```

- **These are unconstrained array types** - (any number of element possible)**.**
  - **Need constrained arrays for actual use.**
  - For example**:** “box”

```
TYPE  MyStk IS ARRAY (integer RANGE < >) OF bit;
SIGNAL   MyData: MyStk (-127  TO  127);            -- explicit constrained
CONSTANT   UCData: MyStk := (‘1’, ‘0’, ‘0’, ‘1’);  -- implicit constrained
```

- **Element of array can be accessed by specifying the index value .**
  - For example**:**
    - **AB1(5) <= ‘1’;   AB2(0) <= ‘0’;   MyData(-22) <= AB1(3);**

Array - Specials Cases

- **The range of array can be enumeration type.**
  - For example:

```
TYPE EnumRangeType IS  (red, green, blue);
TYPE  ColorArray IS ARRAY ( EnumRangeType ) OF bit;
SIGNAL  CA1: ColorArray;
    CA1(red) <= '1'; CA1(green) <= '0';
```

- **Assignment (or compare) can be made to an entire array, or to an element of an array, or to a slice of an array.**
  - For example:

```
AB1 <=  AB2;    AB2(0) <= '0';    MyData( 0 TO 7 ) <= AB1( 3 to 10);
AB2( 0 TO 3) <= ('1', '0', '0', '1');
```


Array - Predefined Type

- **<u>Predefined 1076 types (package standard).</u>**

```
SUBTYPE natural IS  integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS  integer RANGE 1 TO integer'HIGH;
TYPE  string IS ARRAY ( positive RANGE < > ) OF character;
TYPE  bit_vector IS ARRAY ( natural RANGE < > ) OF bit;
```

- **<u>Predefined 1164 types (package std_logic).</u>**

```
TYPE  std_ulogic_vector IS ARRAY (natural RANGE < > ) OF std_ulogic;
TYPE  std_logic_vector IS ARRAY (natural RANGE < > ) OF std_logic;
```

- **VHDL does not allow a type that is an unconstrained array of an unconstrained array.**
  - **TYPE Mem IS ARRAY (natural RANGE < >) OF std_logic_vector; --Not Allow**
  - **TYPE Bem IS ARRAY (natural RANGE < >) OF std_logic_vector(0 TO 7); --OK**

Value for Array - String Literals

- **String literal = string of characters literals in double quotes (" ").**
  - For example:
    - **"This Is a String Literal"**
    - **"State ""READY"" entered!" --** Two double quote characters represents one.
- **A string literal can be assigned to different types of objects; for example, to a STRING type object or a BIT_VECTOR type object.**
- **The type of a string literal is therefore determined from the context in which it appears.**
  - For example:

```
VARIABLE ErrMessage: string(1 to 19);
ErrMessage := "Fatal ERROR: abort!";
VARIABLE BusValue : bit_vector(0 to 3);
BusValue := "1101";
```

Value for Array - Bit String

- **Bit String literal = A string literal that represents a sequence of bits (values of type bit ⇒ {'0', '1'}).**
- **This sequence of bits, called bit strings, can be represented as either a binary value, an octal value, or a hexadecimal value.**
- **The underscore character can be used in bit string literals for clarity.**
  - For example**:**

```
X"FF0"            -- X for hexadecimal.
B"00_0011_1101"   -- B for binary.
O"327"            -- O for octal.
X"FF_F0_AB"
```

- **The type of a bit string literal is also determined from the context in which it appears.**

Array Assignment

- **There are many different way to assign value to an array object.**

```
VARIABLE  Bus1: bit_vector (0 TO 7) ;
VARIABLE  Bus2: bit_vector (7 DOWNTO 0) ;
  Bus1 := "00110011";                               -- String Literal
  Bus1 := B"0011_0011";                             -- Bit String
  Bus1 := ('0', '0', '1', '1', '0', '0', '1', '1'); -- Positional Associate
  Bus1 := (2 => '1', 5 => '1', OTHERS => '0');      -- Named Associate
  Bus1 := (OTHERS => '0');                          -- All Value set to '0'
```

- **The direction of the range is important.**

```
  Bus2 := ('0', '0', '1', '1', '0', '0', '1', '1'); -- The yellow bit is index 0
  Bus1 := ('0', '0', '1', '1', '0', '0', '1', '1'); -- The yellow bit is index 7
```

- **In Bus1 the MSB is '1' while in Bus2 the MSB is '0'.**

Multi Dimensional Array

- **The language allows for an arbitrary number of dimensions to be associated with an array.**
- **Two-dimensional arrays example**:

```
TYPE Vector8 IS ARRAY (7 DOWNTO 0) OF bit;
TYPE Vector4 IS std_logic_vector(3 DOWNTO 0);
TYPE Table6x2 IS ARRAY (0 TO 5, 1 DOWNTO 0) OF bit;
TYPE Table4x8 IS ARRAY (0 TO 3) OF Vector8;
TYPE Table2x4 IS ARRAY (0 TO 1) OF Vector4;
        -- Object Declaration
CONSTANT Mytable1: table6x2 := ("00", "01", "10",
                                "11", "01", "01");
VARIABLE Mytable2: table2x4 := ("10-Z", "ZX01");
SIGNAL Mytable3: table4x8;
Mytable3(2,1) <= '1';
Mytable3(3)   <= "01010101";
```

Composite Types - Record

- **Record Types = multiple elements of different types.**
- **A record type is analogous to the *record* data type in Pascal and the *struct* declaration in C**.
- **Records Declaration:**

```
TYPE MyRec IS RECORD
  ctl:   std_logic;
  inp:   bit_vector (3 DOWNTO 0);
  outp:  bit_vector (7 DOWNTO 0);
  index: integer RANGE 0 TO 64;
END RECORD;

SIGNAL device1, device2, device3 : MyRec;
SIGNAL final: bit_vector (7 DOWNTO 0);

device1.ctl <= '1';
device2.ctl <= device1.ctl;
device2.inp <= "1011";
final <= device1.outp;
device1 <= device2;
device3 <= ('Z', "1111", "11110000", 32);
```


Access Types - Pointer

- **Access Types = Pointer. (**Usually is not support for synthesis**)**
- **Values belonging to an access type are pointers to a dynamically allocated object of some other type. They are similar to pointers in Pascal and C languages.**
- **Access Declaration:**

```
TYPE Ptr  IS ACCESS MyRec; -- Declare previously
TYPE Fifo IS ARRAY (0 TO 63, 7 DOWNTO 0) OF bit;
TYPE FifoPtr  IS ACCESS Fifo;
```

- **Pointer is an access type whose values are addresses that point to objects.**
- **Every access type may also have the value null, which means that it does not point to any object.**
- **Objects of an access type can only belong to the variable class.**
- **When an object of an access type is declared, the default value of this object is null. For example:**

```
VARIBALE Mod1Ptr, Mod2Ptr: Ptr; -- Default value is null
```


Access Types - Dynamic Allocation

- **Objects to which access types point can be created using *allocators*.**
- **Allocators provide a mechanism to dynamically create objects of a specific type.**
- **NEW is the allocators causes an object of specify type to be created, and the pointer to this object is returned.**
- **The values of the elements of the new object are the default values of each element (the leftmost value of the implied subtype)**.

```
Mod1Ptr := NEW MyRec; -- initialize by default value
Mod2Ptr := NEW MyRec'('Z', "1111", "11110000", 32);
```

- **For every access type, a procedure deallocate is implicitly declared. This procedure, when called, returns the storage occupied by the object to the host environment [like free( ) and delete( )].**

```
deallocate(Mod1Ptr);
```

Access Types - Reference

- **Objects of an access type can be referenced as:**
  - ***obj-ptr*.all: Accesses the entire object pointed to by obj_ptr, where *obj-ptr* is a pointer to an object of any type.**
  - ***array-obj-ptr* (*element-index*): Accesses the specified array element, where *array-obj-ptr* is a pointer to an array object.**
  - ***record-obj-ptr.element-name*: Accesses the specified record element, where *record-obj-ptr* is a pointer to a record object.**

- **Pointers can be assigned to other pointer variables of the same access type.**

```
Mod1Ptr := Mod2Ptr;
```


Incomplete Types

- >>>>>>>>>>>>>>.
- >>>>>>>>>>>>>>

File Types

- >>>>>>>>>>>>>>.
- >>>>>>>>>>>>>>