MODULE 3

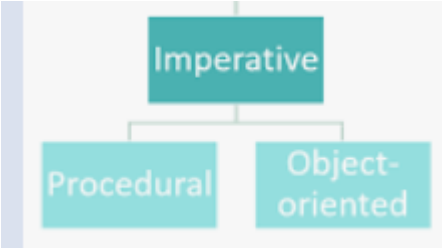
Primitive Data Types

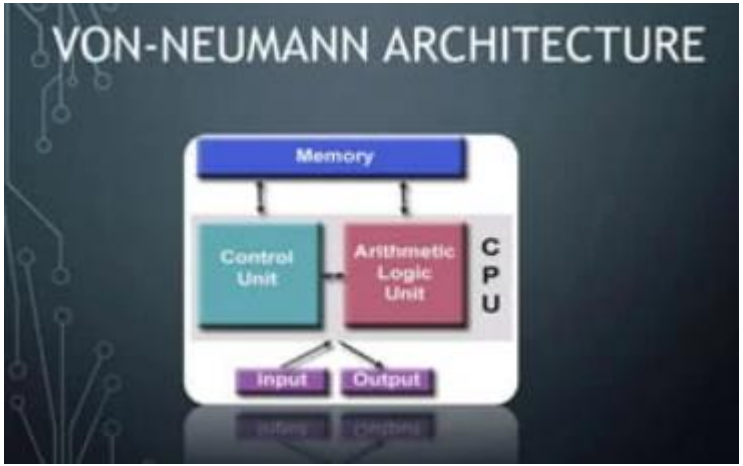
Objectives

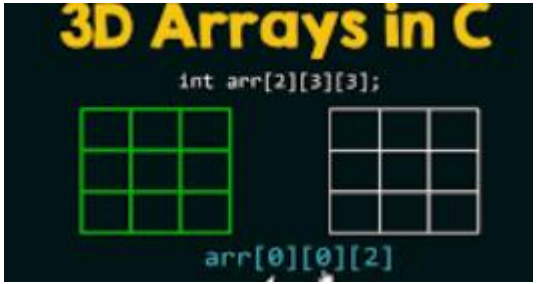
At the end of this module, you should be able to:

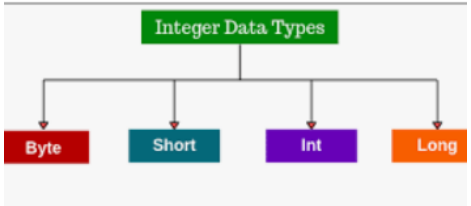
1. Define data objects, values and types;
2. Specify different kinds of primitive data types;
3. Define and give examples of declaration, binding, type checking, type equivalence, type conversion, coercion

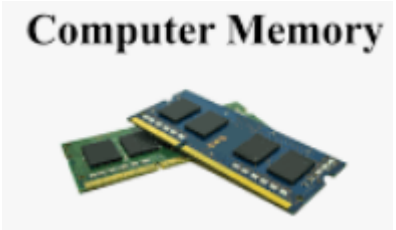
Introduction


Imperative
Procedural
Object-oriented


VON-NEUMANN ARCHITECTURE
Memory
Control Unit
Arithmetic Logic Unit
CPU
Input
Output


3D Arrays in C
int arr[2][3][3];
arr[0][0][2]


Integer Data Types
Byte
Short
Int
Long


Computer Memory

Functional programming languages allow expressions to be named. These named expressions appear like assignments to variable names in imperative languages, but are fundamentally different in that they cannot be changed. So, they are like the named constants of the imperative languages. Pure functional languages do not have variables that are like those of the imperative languages. However, many functional languages do include such variables.

Names

Design Issues

The following are the primary design issues for names:

- Are names case sensitive?
- Are the special words of the language reserved words or keywords?

history note

The earliest programming languages used single-character names. This notation was natural because early programming was primarily mathematical, and mathematicians have long used single-character names for unknowns in their formal notations.

Fortran I broke with the tradition of the single-character name, allowing up to six characters in its names.

Names in Java, C#, and Ada have no length limit, and all characters in them are significant.

In the C-based languages, it has to a large extent been replaced by the so-called camel notation, in which all of the words of a multiple-word name except the first are capitalized, as in myStack.

All variable names in PHP must begin with a dollar sign. In Perl, the special character at the beginning of a variable’s name, $, @, or %, specifies its type (although in a different sense than in other languages). In Ruby, special characters at the beginning of a variable’s name, @ or @@, indicate that the variable is an instance or a class variable, respectively.

For example, the Java method for converting a string to an integer value is parseInt, and spellings such as ParseInt and parseint are not recognized. This is a problem of writability rather than readability, because the need to remember specific case usage makes it more difficult to write correct programs.

Data Objects

Data objects represents containers for data values. Or, they are memory spaces where data values may be stored and retrieved later. In programming, we always keep values either for further computation or for output.

For the purpose of further computation or for output.

2 Types

1. Programmer-defined - e.g. variables and constants
2. System-defined - e.g. maintained by virtual computer

A programmer-defined data object is one that is explicitly created and manipulated by the programmer through declarations and statements in the program.

A system-defined data object on the other hand, is one where the virtual computer set up for housekeeping during program execution and one that is not directly accessible to the programmer.

Data object has attributes, e.g. value type or size

How long can this data object can exist is called their lifetime.

E.g. Create/destroy

At the beginning or dynamically at during execution

Self-Assessment Question 3-1

What is the lifetime of local variables in a Pascal procedure? What is the lifetime of a variable declared in the main program block of Pascal?

Data Values and Value Types

A value is anything that may be evaluated, stored, passed as argument to a procedure, returned by a function, or may be a component of a data structure.

Primitive type is one whose value is atomic and therefore cannot be decomposed. In Pascal, it includes types character, integer, real, boolean, enumerated. Compare this to COBOL with fixed length strings and fixed-point numbers as primitive types.

Composite type is a type whose values are composed or structured from simpler values. E.g. tuples, records, variants, unions, arrays, sets, strings, lists, trees, serial files and many more.

A recursive type is a special case of composite type where the values are of the same type. E.g. List type

Self-Assessment Question 3-2

Define data objects, values, and value types.

Data Types

A data type is a class of data objects together with the set of operations for creating and manipulating them. All programming languages have a set of primitive data types. These primitive data types may be combined to form a composite data type. Newer languages have a facility for programmers to define a composite data type other than the ones provided in the language.

These primitive data types may be combined to form a composite data type.

Primitive Data Types

All languages have primitive data types. Here are some common promitive data types implemented in most languages.

Numeric Data Type

E.g.

short i;

long k;

will allocate four bytes for k and two bytes for i.

In Fortran, real number has an added attribute for PRECISION with two possible values, SINGLE and DOUBLE. A single precision real number X is declared as:

REAL X;

While a double precision real number Y, for example, is declared as:

DOUBLE PRECISION Y;

In ADA and PL/I, where it allow the number of digits in the decimal representation to be specified.

Example:

```
DECLARE PAY FIXED DECIMAL (7,2)
```

Subrange Type

Subrange is introduced to save on storage and to allow better type checking.

For example if we define a variable as having a value in the subrange 1..100,

then, all we need is 8-bit instead of 16-bit or 32-bit of general integer.

Type checking is also facilitated since values falling out of range can easily be detected.

age: 1..20;

age := 130 or age := 0 is illegal

age := age - 1;   however cannot be detected at compile-time.

The PICTURE declaration in COBOL and PL/I

ROOT PICTURE 99999V99

Enumeration Type

For example, a person taching in the university, can either be an instructor, assistant professor, associate professor, or professor. The marital status of a person is one of the following: single, married or widowed.

In Pascal, you can define these as:

rank: (instructor, assistant_professor, associate_professor, professor);

status: (single, married, widowed)

In language where enumeration is not allowed what do you do?

Self-Assessment Question 3-3

What is the basic difference between an enumerated type and a subrange type?

Character Type

Also found in almost all programming language is the character type. The natural extension of this is the character-string data type. Some language like SNOBOL and PL/I allow a string (sequence of characters) to be manipulated as one unit, while others like APL, Pascal and Ada, consider a string as a linear array of characters.

Example in PL/I,

DCL NAME CHAR (25);

While a Pascal declaration for a string with 25 characters is:

name : array[1..25] of char;

But, in PL/I, if you want to contain strings shorter than 25 characters, it should be declared as:

DCL NAME CHAR(25) VARYING;

Self-Assessment 3-4

Enumerate all the primitive data types in C. What is the difference between long and short integer in C?

Declarations

The declaration is that part of a program where the programmer communicates to the language translator information on the numbers and types of data objects needed during program execution.

Pascal declaration,

i : integer;

There are two types of declaration: explicit and implicit declarations.

The sample declaration of Pascal on the previous slide is an example of explicit declaration.

An implicit declaration is found in languages where data objects are created whenever they are about to be used. An example of this is found in BASIC where no explicit declarations are needed.

They are created whenever a value is assigned. One problem with this implicit declaration set up is that the data object stays until program termination.

An explicit declaration serves several purpose both for the programmer and the language translator.

1. Choice of storage representation. The type and size of a variable is known during declaration. They are predetermined. Execution is usually faster compared to when it is created only when needed.
2. Storage management. This information is very useful for managing the storage during run-time. A good strategy is to place all the global variables in one place (global area) and local variables are placed in the stack where they can be reclaimed after the execution of the procedure that defined them.

Generic operations. Through declaration of variables, overloaded operations can easily be supported and efficiently implemented.

For example, the + operator in Pascal is arithmetic addition when given operands of both integer types. The + operator means set union when given operands of both set types. In this case the + operator is overloaded operation.

Type checking. This makes it easy to check whether the operations are supplied with operands of the correct types.

Self-Assessment Question 3-5

What are the reasons why variables are declared?

Binding

A **binding** is an association between an attribute and an entity.

The time at which a binding takes place is called **binding time**.

Bindings can take place at language design time, language imple mentation time, compile time, load time, link time, or run time.

Binding

The binding of a program element to a particular characteristic or property can be viewed as the choice of the property from a set of properties. E.g. consider the Pascal code:

```
var   j : integer;
begin
   j := 0;
   ...
   j := j + 1;
   ...
end
```

For example, the asterisk symbol (*) is usually bound to the multiplication operation at language design time. A data type, such as int in C, is bound to a range of possible values at language implementation time. At compile time, a variable in a Java program is bound to a particular data type.

The occurrence of j in the declaration is an occurrence of binding, i.e, the identifier j is bound to a storage location. The binding time occurs at the time of entry to the procedure during execution.

What was illustrated is that binding occurs whenever a declaration of variables is done. In reality, a data object may participate in several binding occurrences. While the type of a data object does not change during its lifetime, the bindings may change dramatically.

Binding situations:

1. The binding of a data object to one or more value may change during execution. This binding can be modified by the execution of an assignment statement.
2. The binding of a data object to a name may also change during execution. An example of this is when a data object is passed as a parameter and accessible from a procedure using a different name (call by reference)
3. The binding of a data object to one or more data objects of which it is a component. Such a binding is often represented by a pointer value, and may be modified by a change in the pointer.
4. The binding of a data object to a storage location may also change. This occurs when the memory manager of the virtual computer decides to rearrange the data objects in memory (beyond the control of the programmer).

Classes of Binding

1. The first class is binding performed at execution time.
2. Binding performed at translation time. E.g. Pascal code below:
    a. Type UserDefined = 1..100;
    b. Var age: UserDefined;

    The variable age is bound to a particular type only during translation time.

3. Binding performed at language definition time as another class.

4. Finally, we have binding that occurs at implementation time. This is brought about by the differences in hardware where programming languages are implemented. E.g. when the language is implemented in 16-bit machine, the short integer will be allocated in 16-bit as storage. Also this will vary from machine to machine.

Finally, binding times may be classified as early (static or compile-time) and late binding (dynamic or runtime).

Type Checking

Type checking is done to ensure that an operation is provided with the correct types. Consider for example the operation

```
a := b + c;
```

The type of b and c must be integer or real for an addition operation to be valid. The same for a which will hold the resulting value. A compiled language will throw an error otherwise during compiling process. On the other hand, interpreted language will later throw an error during runtime. Which means programmers are given extra advantage since they can catch these errors during compile time.

Self-Assessment Question 3-6

Give at least two difference of dynamic binding from static binding.

Type Equivalence

The issue of type equivalence arises when two data objects are involved in one operation. But there is no problem if the two data objects are identical in types. However, we often encounter operations involving data objects that are not identical but closely related.

Structural equivalence is more lenient but less clear than named equivalence. It defines two types to be compatible if they have the same logical structure.

```
TYPE meter IS NEW integer;
TYPE yard IS NEW integer;
m, n: meter;
y: yard;
```

The assignment m := n; is legal in ADA but not the assignment y := m;. However, the assignment y := m; is perfectly legal in Pascal, Modula-2 and PL/1.

Type Conversion and Coercion

When the operands of an operation have types that are not exactly the same, either a type mismatch error is issued by the compiler or by the virtual computer (depending on when the type check is done) or some form of type conversion is done. Type conversion is an operation that converts a data object of one type and produces the corresponding data object in another type.

Type conversion may be done implicitly (coercion) or explicitly.

Coerce is a way by which a data object of certain type is changed to the correct type. Coercion is usually carried out by the compiler or by the virtual computer. An example of this is the type coercion occurring in the following Pascal expression:

x := n * 1.2;

Here, x is of type real and n is of type integer. The type of n is coerced to be real so that when it is multiplied with another real nurnber, in this case 1.2, we will have a resulting real number that is stored in x. This illustrates integer-to-real conversion being performed implicitly in Pascal. This implicit integer-to-real conversion in Pascal, however, ever, is not permitted in Modula-2.

Explicit type conversion is done using built-in functions provided in the language for the purpose of changing types. For example, in Pascal the function ROUND and TRUNC are built-in functions for converting a real type to an integer type. To illustrate, given i to be an integer type variable and x a real type variable, the following Pascal assignment statement is valid due to explicit type conversion:

i : = TRUNC (x);

A similar method is provided in Ada where for example we can force one of type universal_integer to assume a type integer. To illustrate, consider the literal 3 whose type is universal_ integer and variable i of type integer. The assignment

i := 3;

Is perfectly legal in ADA.

In C, we have the general format: (type) expression

For forcing the type of the resulting expression to have the type as specified in the parenthesis.

Thank You.