User Defined Data type

In programming, data types are used to classify and store different kinds of data. Built-in data types (e.g., int, float, string) are provided by the programming language. However, sometimes built-in types are not sufficient to represent complex real-world entities. User-defined types allow programmers to create their own data types tailored to specific needs

Types of user data types

* Non-composite data type
* Composite data type

**Non-Composite Data Types**

Non-composite data types, also known as primitive or simple data types, are the most basic data types provided by a programming language. They are not composed of other data types and represent a single value. Examples of non-composite user defined data types include:

1. **Enumerated Types (Enums)**: Enums are user-defined data types that consist of a set of named constants. They allow you to define a type with a limited set of possible values. For example, you can define an enum for the days of the week or the suits in a deck of cards.
2. **Pointer Types**: Pointer types are user-defined types that store memory addresses of other variables or objects. They allow you to indirectly access and manipulate data by referencing memory locations.

Non-composite user-defined data types provide a way to create custom types that represent a single value or concept, tailored to the specific needs of the program.

**Composite Data Types**

Composite user-defined data types are custom data types that are composed of one or more other data types, either built-in or user-defined. They allow you to create complex data structures by combining and organizing related data elements. Examples of composite user-defined data types include:

1. **Structures (Structs)**: Structures are user-defined types that group related data elements of different types into a single unit. They allow you to create a custom type that consists of multiple fields, each with its own data type. For example, you can define a struct for a point in 2D space with fields for x and y coordinates.
2. **Classes**: Classes are user-defined types that encapsulate data and behavior into a single unit. They are the foundation of object-oriented programming and allow you to define the structure (fields) and behavior (methods) of objects. Classes can have constructors, properties, and methods to manipulate the data and provide functionality.
3. **Arrays**: Arrays are composite data types that represent an ordered collection of elements of the same data type. They can be user-defined by specifying the size and type of elements they hold. For example, you can define an array of integers or an array of custom objects.
4. **Sets**: Sets are user-defined types that represent an unordered collection of unique elements of the same data type. They can be implemented using various data structures like hash tables or binary search trees.

Composite user-defined data types provide a way to create custom types that are composed of multiple data elements, allowing for more complex and structured data representations.

The distinction between non-composite and composite user-defined data types lies in whether the custom type is composed of other data types or not. Non-composite user-defined types are standalone and represent a single value, while composite user-defined types are built by combining multiple data types into a single unit.

Understanding this distinction helps in designing appropriate data structures and selecting the right user-defined types based on the problem requirements, leading to more organized and efficient code.

**Limitations of Built-in Types**

1. **Lack of Abstraction**
   * Built-in types are general-purpose and do not provide a higher level of abstraction
   * They do not effectively represent real-world objects or concepts
   * Example: Representing a student using built-in types would require multiple variables (e.g., name, age, grade) scattered throughout the code
2. **Lack of Encapsulation**
   * Built-in types do not encapsulate related data and operations into a single unit
   * Data and operations are separate, leading to potential inconsistencies and errors
   * Example: Updating a student's grade requires modifying the grade variable directly, which may lead to invalid states
3. **Lack of Reusability**
   * Code using built-in types for complex entities is often repetitive and hard to reuse
   * Duplicated code leads to maintenance issues and reduces code quality
   * Example: Creating multiple students using built-in types requires duplicating the same code structure for each student

**Benefits of User-Defined Types**

1. **Abstraction**
   * User-defined types provide a higher level of abstraction, representing real-world entities more accurately
   * They allow programmers to define types that match the problem domain
   * Example: Creating a Student type that encapsulates all relevant data (e.g., name, age, grade) into a single entity
2. **Encapsulation**
   * User-defined types encapsulate related data and operations into a single unit
   * Encapsulation ensures data integrity and provides a clear interface for interacting with the type
   * Example: Defining methods like setGrade() and getGrade() within the Student type to control access and maintain valid states
3. **Reusability**
   * User-defined types promote code reuse by encapsulating common functionality
   * They allow the creation of multiple instances of the same type, reducing code duplication
   * Example: Creating multiple Student objects, each with its own data, using the same Student type definition
4. **Modularity and Maintainability**
   * User-defined types improve code modularity by separating concerns and organizing related functionality
   * They make the codebase more maintainable by providing a clear structure and reducing dependencies
   * Example: Modifying the Student type's implementation without affecting code that uses the Student objects
5. **Code Readability and Self-Documentation**
   * User-defined types make the code more readable and self-documenting
   * Type names and method names convey the purpose and behavior of the code
   * Example: Using meaningful names like Student, getName(), and setGrade() makes the code more intuitive and easier to understand

### Enumerated Types

Enumerated types (enums) are user-defined data types that consist of a set of named constants called enumerators. They provide a way to define a variable that can only take on a limited set of predefined values. Example: Defining an enum for days of the week (e.g., DaysOfWeek = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday})

Enums are used when a variable should be restricted to a specific set of values. They improve code readability and maintainability by providing meaningful names for the possible values

* + Example: Declaring a variable of the DaysOfWeek enum type (e.g., today: DaysOfWeek = Monday)

Declaring Enum

TYPE <enum\_name> = [<value1>, <value2>, ..., <valueN>]

* TYPE: Keyword used to define a new data type.
* <enum\_name>: The name you choose for your enumerated type.
* [<value1>, <value2>, ..., <valueN>]: The list of possible values for the enum, separated by commas and enclosed in square brackets.

Example

*Declaring an enum for days of the week:*

TYPE DaysOfWeek = [Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday]

*Declaring an enum for card suits:*

TYPE CardSuit = [Hearts, Diamonds, Clubs, Spades]

Declaring a single variable:

DECLARE Heart: CardSuit

* DECLARE is the keyword used to declare a variable.
* Heart is the name chosen for the variable.
* CardSuit is the enum type assigned to the variable

DECLARE Monday: DaysOfWeek

Assigning value to the enum type

DECLARE today : DAYSOFWEEK

Today <- Monday

Example

DECLARE CardSuit = [Hearts, Diamonds, Clubs, Spades]

DECLARE ChosenSuit: CardSuit <- Diamonds

Explanation

DECLARE CardSuit = [Hearts, Diamonds, Clubs, Spades]

* + This line declares the CardSuit enum type.
  + It defines the possible values that a variable of type CardSuit can hold.
  + The values are enclosed in square brackets and separated by commas.

DECLARE ChosenSuit: CardSuit <- Diamonds

* + This line declares a variable named ChosenSuit of type CardSuit.
  + The DECLARE keyword is used to declare the variable.
  + The variable name ChosenSuit is followed by a colon (:) and the enum type CardSuit.
  + The assignment operator (<-) is used to assign the value Diamonds to the variable ChosenSuit.
  + Diamonds is one of the predefined values of the CardSuit enum type.

After executing this statement, the variable ChosenSuit will hold the value Diamonds, which is one of the valid values defined in the CardSuit enum type.

You can then use the ChosenSuit variable in your code, knowing that it will always hold a valid card suit value. For example:

IF ChosenSuit = Hearts THEN

OUTPUT "The chosen suit is Hearts"

ELSE

OUTPUT "The chosen suit is not Hearts"

ENDIF

In this example, the IF statement checks if the value of ChosenSuit is equal to Hearts. If it is, it outputs "The chosen suit is Hearts". Otherwise, it outputs "The chosen suit is not Hearts".

By using the DECLARE keyword to declare the CardSuit enum type and assign a value to a variable of that type, you ensure that the variable can only hold valid card suit values, making your code more robust and less prone to errors.

In Python, you can declare an enum type using the enum module. Here's how you can declare an enum type in Python:

Import the enum module:

from enum import Enum

Declare the enum type using the Enum class:

class CardSuit(Enum):

HEARTS = 1

DIAMONDS = 2

CLUBS = 3

SPADES = 4

Access the enum members using the enum type and the member names:

print(CardSuit.HEARTS) # Output: CardSuit.HEARTS

print(CardSuit.HEARTS.value) # Output: 1

complete example

from enum import Enum

class CardSuit(Enum):

HEARTS = 1

DIAMONDS = 2

CLUBS = 3

SPADES = 4

# Declare a variable of the CardSuit enum type

chosen\_suit = CardSuit.DIAMONDS

# Check the value of the enum variable

if chosen\_suit == CardSuit.HEARTS:

print("The chosen suit is Hearts")

else:

print("The chosen suit is not Hearts")

# Iterate over the enum members

for suit in CardSuit:

print(suit)

Output

The chosen suit is not Hearts

CardSuit.HEARTS

CardSuit.DIAMONDS

CardSuit.CLUBS

CardSuit.SPADES

* The CardSuit enum type is declared with four members: HEARTS, DIAMONDS, CLUBS, and SPADES.
* The variable chosen\_suit is declared and assigned the value CardSuit.DIAMONDS.
* The if statement checks if chosen\_suit is equal to CardSuit.HEARTS and prints the appropriate message.
* The for loop iterates over all the members of the CardSuit enum type and prints each member.

### Pointer Types

Pointer types are used to store memory addresses of other variables or objects. It contains the memory address of another variable or object, allowing indirect access to that data. Example: Declaring a pointer to an integer (e.g., var intPtr: ^Integer)

Pointers are used for efficient memory management, dynamic memory allocation, and accessing data structures. They allow passing variables by reference, enabling modifications to the original data. Example: Allocating memory dynamically for an integer and accessing its value through a pointer (e.g., intPtr := new Integer; intPtr^ := 42)

Declaring pointer type

TYPE <pointer\_type\_name> = ^<data\_type>

* TYPE: Keyword used to define a new data type.
* <pointer\_type\_name>: The name you choose for your pointer type.
* ^: The caret symbol indicates that it is a pointer type.
* <data\_type>: The data type that the pointer will point to.

Example

TYPE IntPtr = ^INTEGER Declaring a pointer type to a integer

TYPE CharPtr = ^CHAR Declaring a pointer type to a character

TYPE StudentRecord = RECORD

name: STRING

age: INTEGER

grade: REAL

ENDRECORD

TYPE StudentPtr = ^StudentRecord

* IntPtr is a pointer type that can point to an integer value.
* CharPtr is a pointer type that can point to a character value.
* StudentRecord is a user-defined record type that represents a student with fields for name, age, and grade.
* StudentPtr is a pointer type that can point to a StudentRecord value.

Once you have declared a pointer type, you can declare variables of that pointer type using the DECLARE keyword

DECLARE ptr1: IntPtr

DECLARE ptr2: CharPtr

DECLARE studentPtr: StudentPtr