PASCAL - RECORDS

Pascal arrays allow you to define type of variables that can hold several data items of the same kind but a record is another user-defined data type available in Pascal which allows you to combine data items of different kinds.

Records consist of different fields. Suppose you want to keep track of your books in a library, you might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Record

To define a record type, you may use the type declaration statement. The record type is defined as −

```
type
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

Here is the way you would declare the Book record −

```
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

The record variables are defined in the usual way as

```
var
   r1, r2, ... : record-name;
```

Alternatively, you can directly define a record type variable as −

```
var
Books : record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

Accessing Fields of a Record

To access any field of a record, we use the member access operator . . The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of structure −

```
program exRecords;
```

```
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;

var
   Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)

begin
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;

   (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;

   (* print Book1 info *)
   writeln ('Book 1 title : ', Book1.title);
   writeln('Book 1 author : ', Book1.author);
   writeln( 'Book 1 subject : ', Book1.subject);
   writeln( 'Book 1 book_id : ', Book1.book_id);
   writeln;

   (* print Book2 info *)
   writeln ('Book 2 title : ', Book2.title);
   writeln('Book 2 author : ', Book2.author);
   writeln( 'Book 2 subject : ', Book2.subject);
   writeln( 'Book 2 book_id : ', Book2.book_id);
end.
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Records as Subprogram Arguments

You can pass a record as a subprogram argument in very similar way as you pass any other variable or pointer. You would access the record fields in the similar way as you have accessed in the above example −

```
program exRecords;
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;

var
   Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)
```

```
(* procedure declaration *)
procedure printBook( var book: Books );

begin
   (* print Book info *)
   writeln ('Book  title : ', book.title);
   writeln('Book  author : ', book.author);
   writeln( 'Book  subject : ', book.subject);
   writeln( 'Book book_id : ', book.book_id);
end;

begin
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;

   (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;

   (* print Book1 info *)
   printbook(Book1);
   writeln;

   (* print Book2 info *)
   printbook(Book2);
end.
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Pointers to Records

You can define pointers to records in very similar way as you define pointer to any other variable as follows −

```
type
record-ptr = ^ record-name;
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

Now, you can store the address of a record type variable in the above-defined pointer variable. To declare a variable of the created pointer type, you use the var keyword −

```
var
   r1, r2, ... : record-ptr;
```

Before using these pointers, you must create storage for a record-name type variable, which will be manipulated by these pointers.

```
new(r1);
new(r2);
```

To access the members of a record using a pointer to that record, you must use the ^. operator as follows −

```
r1^.feild1 := value1;
r1^.feild2 := value2;
...
r1^fieldn := valuen;
```

Finally, don't forget to dispose the used storage, when it is no longer in use −

```
dispose(r1);
dispose(r2);
```

Let us re-write the first example using a pointer to the Books record. Hope this will be easy for you to understand the concept −

```
program exRecords;
type
BooksPtr = ^ Books;
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;

var
  (* Declare Book1 and Book2 of pointer type that refers to Book type *)
   Book1, Book2: BooksPtr;

begin
   new(Book1);
   new(book2);

   (* book 1 specification *)
   Book1^.title  := 'C Programming';
   Book1^.author := 'Nuha Ali ';
   Book1^.subject := 'C Programming Tutorial';
   Book1^.book_id := 6495407;

   (* book 2 specification *)
   Book2^.title := 'Telecom Billing';
   Book2^.author := 'Zara Ali';
   Book2^.subject := 'Telecom Billing Tutorial';
   Book2^.book_id := 6495700;

   (* print Book1 info *)
   writeln ('Book 1 title : ', Book1^.title);
   writeln('Book 1 author : ', Book1^.author);
   writeln( 'Book 1 subject : ', Book1^.subject);
   writeln( 'Book 1 book_id : ', Book1^.book_id);

   (* print Book2 info *)
   writeln ('Book 2 title : ', Book2^.title);
   writeln('Book 2 author : ', Book2^.author);
   writeln( 'Book 2 subject : ', Book2^.subject);
   writeln( 'Book 2 book_id : ', Book2^.book_id);

   dispose(Book1);
   dispose(Book2);
end.
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

The With Statement

We have discussed that the members of a record can be accessed using the member access operator . . This way the name of the record variable has to be written every time. The **With** statement provides an alternative way to do that.

Look at the following code snippet taken from our first example −

```
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;
```

The same assignment could be written using the **With** statement as −

```
(* book 1 specification *)
With Book1 do
begin
   title  := 'C Programming';
   author := 'Nuha Ali ';
   subject := 'C Programming Tutorial';
   book_id := 6495407;
end;
```