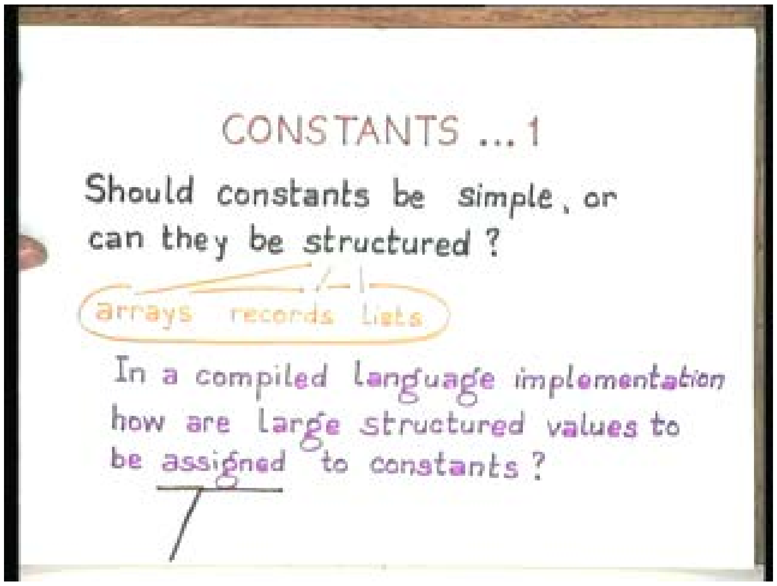
**Principles of Programming Languages**
**Prof: S. Arun Kumar**
**Department of Computer Science and Engineering**
**Indian Institute of Technology**
**Delhi**
**Lecture no 21**
**Lecture Title: Structured Data**

Welcome to lecture 21. Last time we looked at various storage-allocation strategies for simple data. Let me just briefly summarize it. Firstly, if we had constants I said that there were lots of important questions of policy that one should answer before it gets down to implementation. Should you allow structured constants? Should you allow simple constants? What happens if you take various decisions concerning these? Should you insist on compile time evaluation of a constant value? Should it be done at runtime? If so should you do it just before program execution or should you allow it to be more flexible?

(Refer Slide Time: 00:44)



## CONSTANTS ... 1

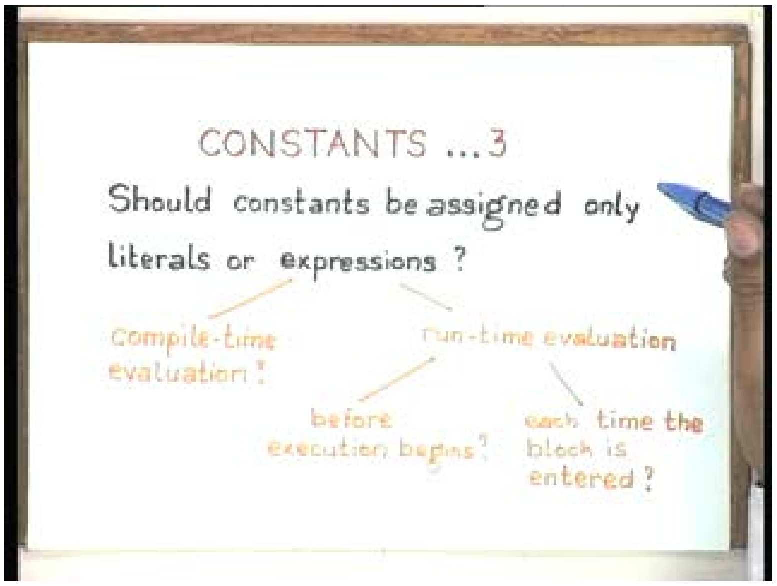
Should constants be simple, or can they be structured?

arrays records lists

In a compiled language implementation how are large structured values to be assigned to constants?

Each time the block is entered it is a constant only for its particular life time and not a constant for the entire program. Should constants just be names of some literals? Should they be allowed expression values? If you allow expression values then would you insist on a compile time evaluation or a run time evaluation?

(Refer Slide Time: 01:36)



CONSTANTS ...3

Should constants be assigned only literals or expressions ?

compile-time evaluation ?

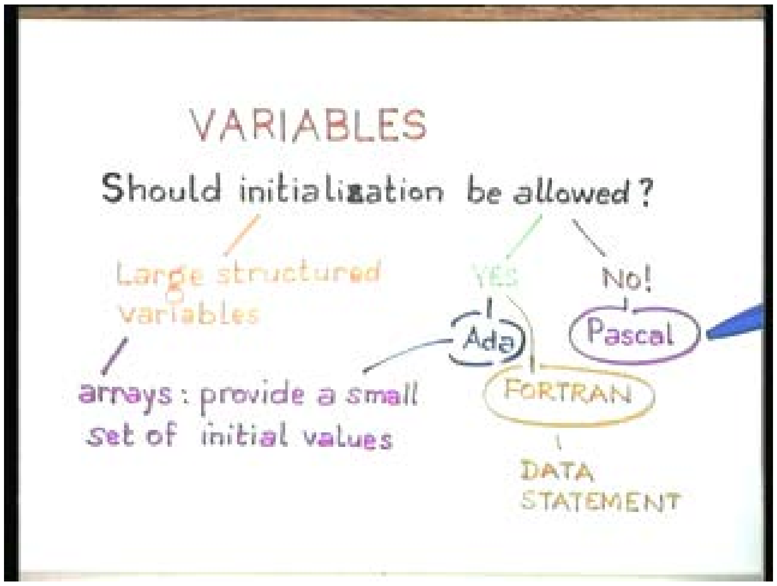
run-time evaluation

before execution begins ?

each time the block is entered ?

If it is a run time evaluation would it be just before program execution begins or should the constant be declared each time the block is entered? As far as variables are concerned should you allow initialization? Should you insist on initialization always? What happens to initialization of large structured variables? What kinds of languages allow what kinds of initialization for variables? Finally we also classified variables in various kinds which are automatic in the sense that they are created at block entry and destroyed at block exit.

(Refer Slide Time: 02:12)



VARIABLES

Should initialization be allowed ?

Large structured variables

arrays : provide a small set of initial values

YES

Ada

FORTRAN

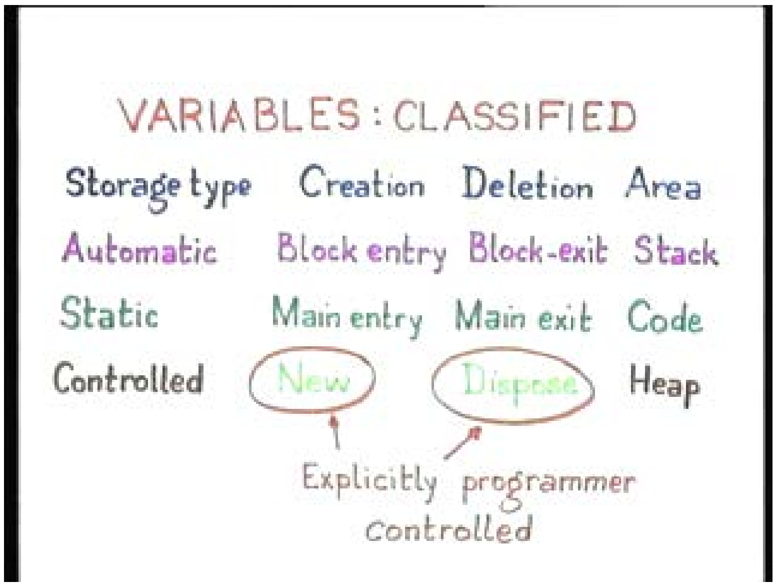
DATA STATEMENT

No!

Pascal

The own variables or the static variables which are created during programming and at entry to the main program have a life time extending right through the execution of the main program, the area in which it would be most natural to allocate these variables. Then of course there are programmer controlled variables, which are explicitly programmer controlled and are usually allocated on a heap.

(Refer Slide Time: 02:43)



## VARIABLES : CLASSIFIED

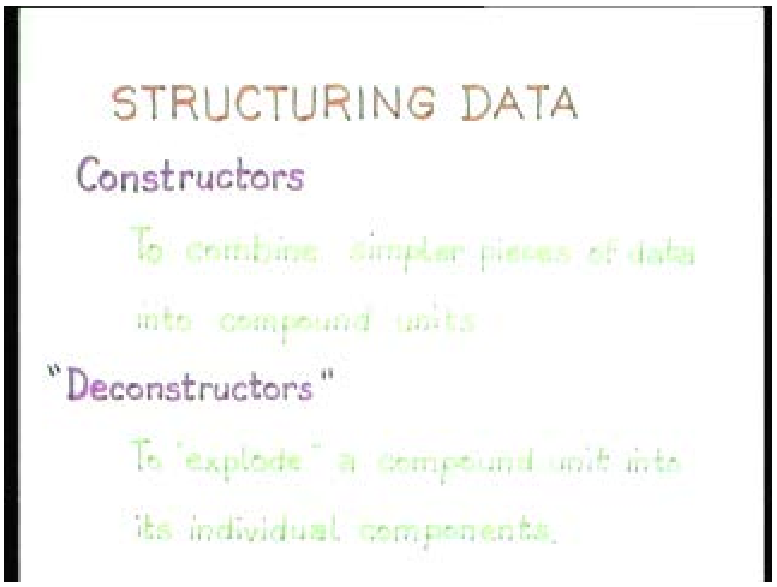
| Storage type | Creation | Deletion | Area |
|---|---|---|---|
| Automatic | Block entry | Block-exit | Stack |
| Static | Main entry | Main exit | Code |
| Controlled | New | Dispose | Heap |

Explicitly programmer controlled

They have a life time which is different from their scope in the sense that they are explicitly created or destroyed by the programmer. We will look at structured data in some greater detail. Whatever we said previously holds for simple variables and simple constants. When I talk about simple variables or simple constants I mean that the underlying virtual machine already supports those types as data types. There are predefined operations, predefined sets of values and so, they are simple in that respect. If you had an underlying virtual machine which only dealt with matrices then that would be a simple data type for that machine but in most general purpose languages the simple data types usually are scalars of the form integers, reals, constants, strings, Booleans etc.

We are talking of structuring simple data into more compound data. It means that you have to somehow combine simpler pieces of scalar data in the view of the virtual machine on which you are implementing into compound units to be regarded as a single data item. In any kind of structuring operation there are constructors and what I have called, but which no other programming language text calls, ‘deconstructors’.
When you put together pieces of data and structure them into large units then you also require methods of exploding that large unit and obtaining the simpler components.

(Refer Slide Time: 04:43)



STRUCTURING DATA

Constructors

To combine simpler pieces of data into compound units

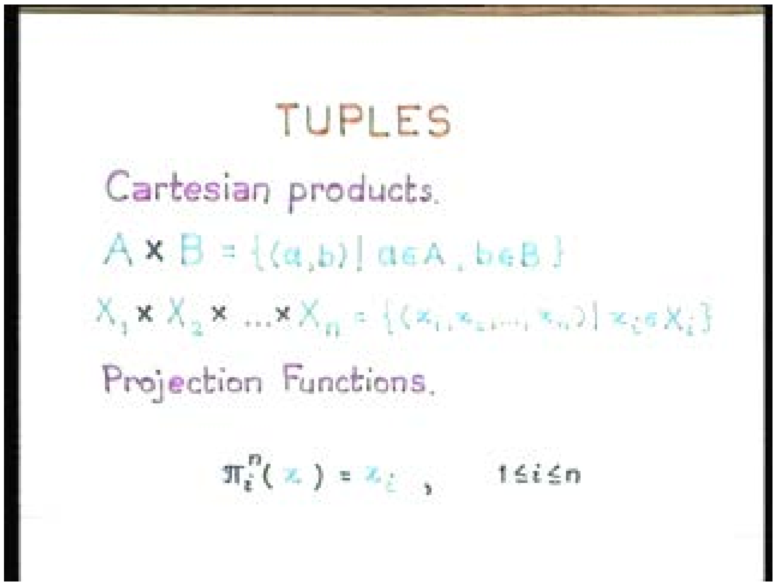
"Deconstructors"

To "explode" a compound unit into its individual components.

I am calling them deconstructors but in the case of particular structuring operations they have particular names. So, I would not call it destructor because that means completely destroying the data but deconstructor is that you split it up into its components again. You take some simpler pieces of data, combine them together but later you might want to access the individual components for some reason may be to construct new structured data. You would like to deconstruct a compound unit into its individual components.

What are the most common structuring mechanisms that mathematics provides and how does one model those structuring concepts in our programming? Let us start with the simplest possible structuring mechanism in mathematics and that is of tuples. The constructor in this case is just a Cartesian product. You can take Cartesian products of two sets for example and you get ordered pairs; you get n tuples by taking Cartesian products and Cartesian product of n sets and you get n tuples of this form. Given any tuple what you require are corresponding projection functions. Given a tuple the 'ith' component of that tuple should be extractable. Remember that all these individual sets may be of different types meaning they may all be really different sets. What you are looking for are forms of projections and projection functions. Projection functions are the deconstructors for a Cartesian product.

(Refer Slide Time: 8:10)



TUPLES

Cartesian products.

$A \times B = \{(a,b) \mid a \in A, b \in B\}$

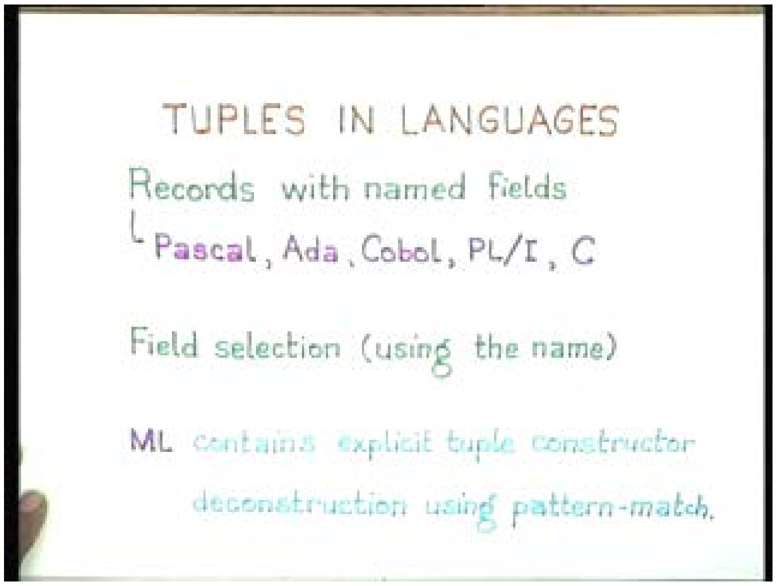
$X_1 \times X_2 \times \ldots \times X_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in X_i\}$

Projection Functions.

$\pi_i^n(x) = x_i \quad , \quad 1 \leq i \leq n$

Please note that deconstructor is really my name. It really is not available in any other programming language text and deconstructor is actually a philosophical word dealing with some modern concepts in linguistic philosophy. But I think it is a good name to coin so we will use it for this purpose. Projection functions are what you require for exploding a Cartesian product and extracting individual components. Let us look at how these constructors and deconstructors are provided in some languages.

(Refer Slide Time: 09:28)



TUPLES IN LANGUAGES

Records with named fields

└ Pascal, Ada, Cobol, PL/I, C

Field selection (using the name)

ML contains explicit tuple constructor

deconstruction using pattern-match.

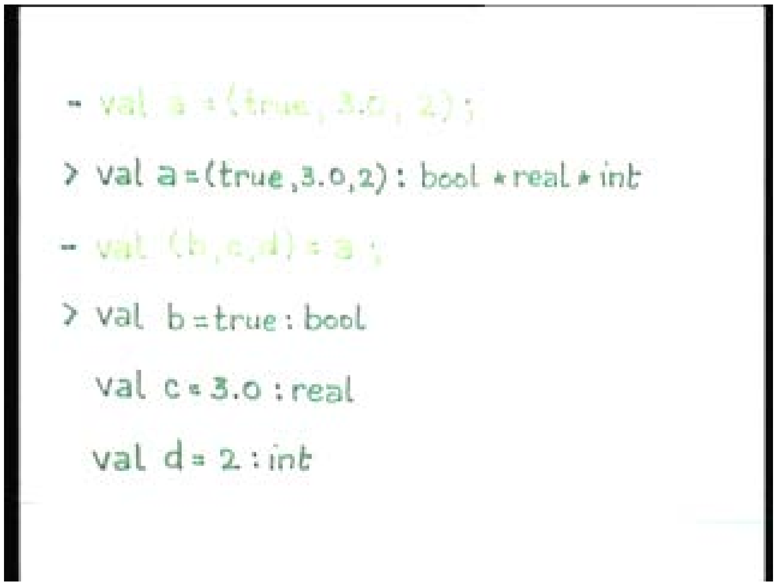
In most imperative languages you have records with named fields as the only tuple formation construct and the fact that you have got named fields.

This means that you use a field selection operation which uses the name. A field selection in a record is the major operation to explode or to extract an individual component of a record.

In ML of course there is an explicit tuple construction mechanism and an explicit deconstruction which is through a pattern matching rule. Since ML is quite clear about the kinds of patterns a tuple must possess, you could go through a typical ML session like this in which you construct a tuple A, which has these values and the ML type inferencing mechanism automatically gives this a type which is defined by this Cartesian product. Of course the individual patterns actually make it clear what domains are involved in the Cartesian product and in what order.

The most common way of deconstructing is to give a declaration where for example, B, C and D are names which are not pre-declared. It is not that while constructing this tuple you gave these field names. These field names are absolutely new but the pattern matching facility of ML clearly indicates that if there is some new name and you are declaring this in this fashion then you have to do an appropriate mapping of what might be ML variables to their appropriate values.

(Refer Slide Time: 12:03)



```
- val a = (true, 3.0, 2);
> val a=(true,3.0,2): bool * real * int
- val (b,c,d) = a ;
> val  b=true: bool
  val  c=3.0 :real
  val  d= 2 : int
```

You get a response which actually is something like this and so the deconstruction is very often done through pure pattern matching and this is not true just of the tuples but it is true of any data type in ML. For any kind of structured data in ML you can actually do deconstruction through such a pattern matching mechanism. Whether they are ML records or records of records to any depth; records of tuples of lists etc you can do deconstruction just by using the pattern matching facility.

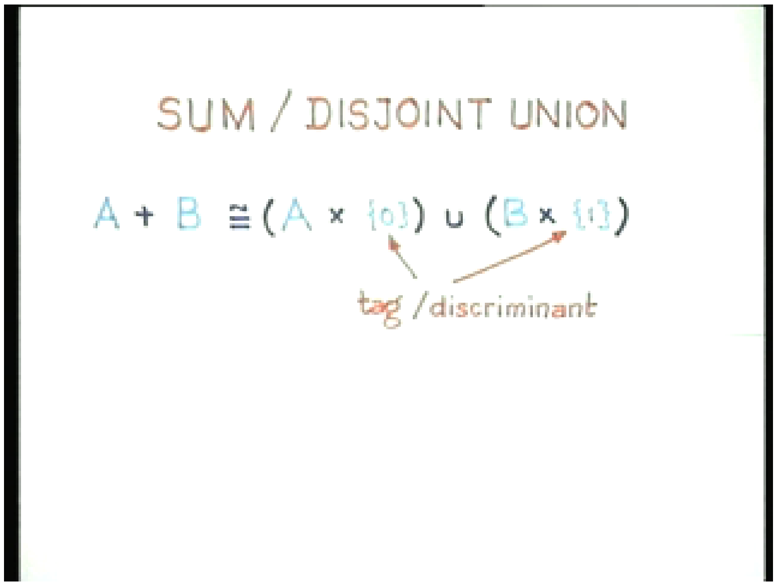
In most other languages while ML has taken a very strict mathematical view towards data types, the constructions are such that they model an elementary mathematical text including what kinds of constructions and deconstruction mechanisms should be provided and how they should be implemented whereas most languages actually have not considered this problem of providing a separate construction or deconstruction mechanism for many mathematical operations. They have usually confused the abstract operation with a representational equality for example; the Pascal record is one such case but most commonly are their kinds of mechanisms used in Lisp and Scheme.

A Cartesian product is clearly a different operation from a sequence-forming operation. The fact that Cartesian products are isomorphic to certain subsets of Lisp formation is a different matter altogether. But it is fundamentally a different operation whereas most of the older functional languages mainly Lisp and Scheme which is just a daughter of Lisp with a simpler structure, only use list construction as a main operation and list operations themselves as the deconstructors. An ordered pair in a Lisp or Scheme is no different from a list of two elements.

An ordered triplet is no different from a list of three elements whereas logically they are two different things. A list is meant to model a sequence and not a tuple. However, they are isomorphic and since lists are a more powerful construction operation and you can program tuples through lists it is assumed that it is not necessary to have a fresh data type called tuples. In fact most of these languages actually do this. In the past they have confused what constitutes an abstract construction or a deconstruction operation from the fact that it is easily implement-able and they have confused these two issues.

The other important operation is what might be called the sum or disjoint union or a co-product. The Cartesian product is a product and this is a co-product in some category of domains. It is what I might call a sum. You have the sum of two sets and what I said loosely was that this is really like maintaining the identities of the elements in the individual sets. These two sets may not be disjoint but you have to somehow maintain the identities of the elements drawn from these two sets. So, it is natural to have a tag or a discriminant which will clearly give an identity to the elements in this union.

(Refer Slide Time: 16:30)



## SUM / DISJOINT UNION

$$A + B \cong (A \times \{0\}) \cup (B \times \{1\})$$

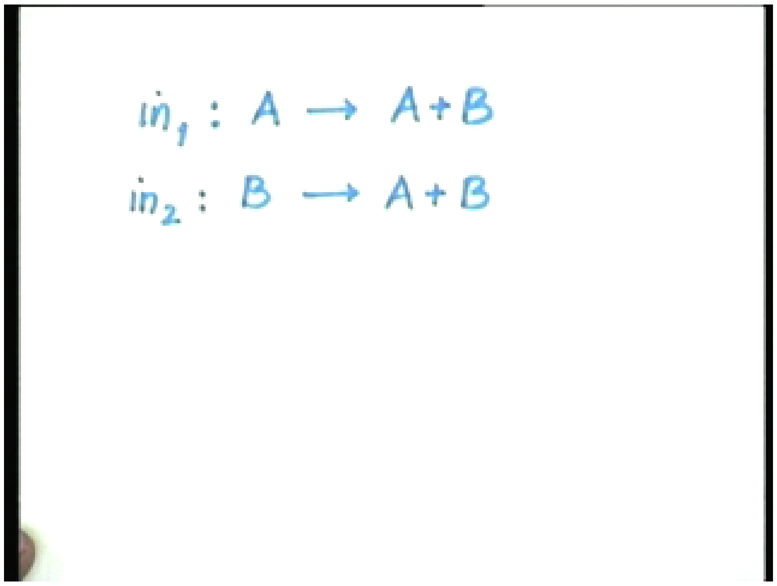
tag / discriminant

When we are talking about a sum or a disjoint union we are really talking about two injection functions which are of this form.

in1: A −> $A + B$

in2: B→ A $+ B$

(Refer Slide Time: 17:05)

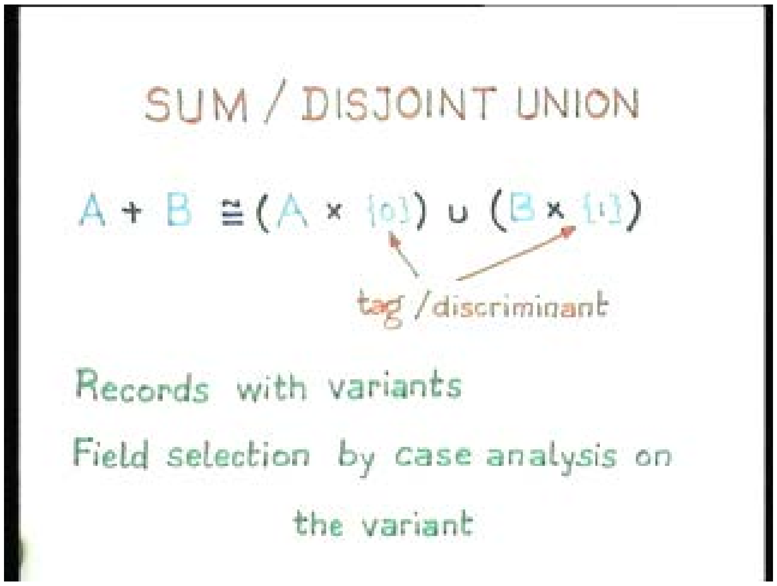


$$in_1 : A \longrightarrow A+B$$
$$in_2 : B \longrightarrow A+B$$

Actually these two injection functions are really what constitute the constructors for the disjoint union operation. If you look at it as a construction which is obtained through injection functions it is not necessarily the same as having a tag or a colour to distinguish

between the elements. This is only a representational mechanism; it is not a logical mechanism. Logical operation is just A + B.
In my past lectures whenever I have used the disjoint union, I may have used equality but here I am saying that it is not really equality. It is an isomorphism which means that the two sides are not exactly identical. There is some logical information in this '+' which is made explicit through a tag or a discriminant and this is really representational rather than logical. However, what has happened is that just as in my past lectures I have actually identified $(A \times \{0\})$ with $(B \times \{1\})$. Most languages have also identified these two and they use a tag in some form or the other.

(Refer Slide Time: 19:00)



SUM / DISJOINT UNION

$A + B \cong (A \times \{0\}) \cup (B \times \{1\})$

tag / discriminant

Records with variants

Field selection by case analysis on the variant

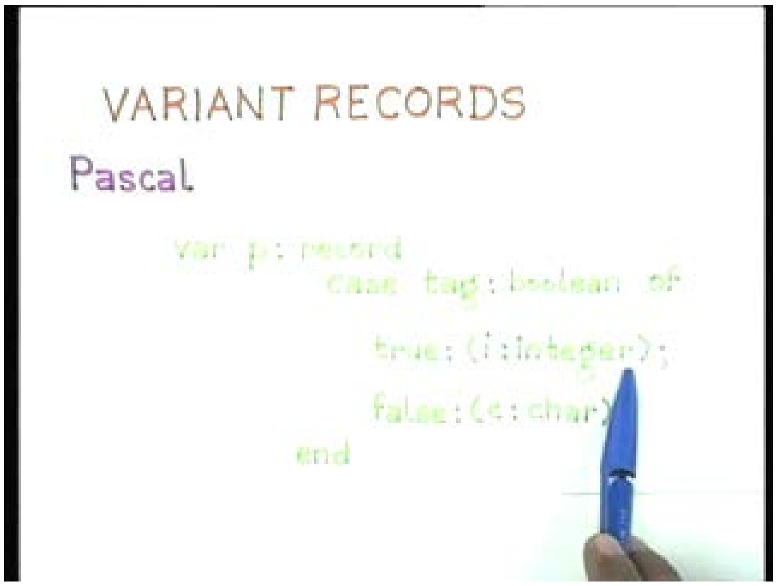
Most languages actually have records with variant fields and in order to identify individual fields rather they take the identity of the individual elements in this co-product $A + B$ . They have used this tag and the operations that come with the appropriate tag.
You do a case analysis on the variant. This is used most commonly in all languages starting after Algol 60. PL 1 and COBOL have a stuck mechanism which is similar to the records of Pascal. The first language to use the variant mechanism is PL 1, then Algol 68 and then Pascal.

Most languages actually have a form of records with variants and the variant field is a tag. The tag field is used to disambiguate to provide an identity for where a certain element comes from. However, let us look at the Pascal variant records.
There is a certain insecurity which actually was exploited even by Niklaus Wirth in his own original Pascal compiler. But however even though it was very useful for his compiler, the construct came under tremendous criticism because of this reason.

I can have a record declaration where the variant is a part of a larger record declaration which means that you are taking some Cartesian products initially and then you are taking a disjoint union of two different kinds of Cartesian products.

That is really how you should look at it but let us usually look at the variant in isolation. There is a tag filed which may be a Boolean and if the tag is true then what it has is an underlying integer and if the tag is false it has a character. In this particular declaration you are just taking a disjoint sum of integers and characters. What is there within the parenthesis could be more complicated. It could be another Cartesian product or some such thing. This declaration is a variable and since it is a record and you use field selection and in the normal idea of designing such a record you can do a case analysis on the tag and do appropriate operations probably such as integer operations or character operations.

(Refer Slide Time: 21:34)



## VARIANT RECORDS

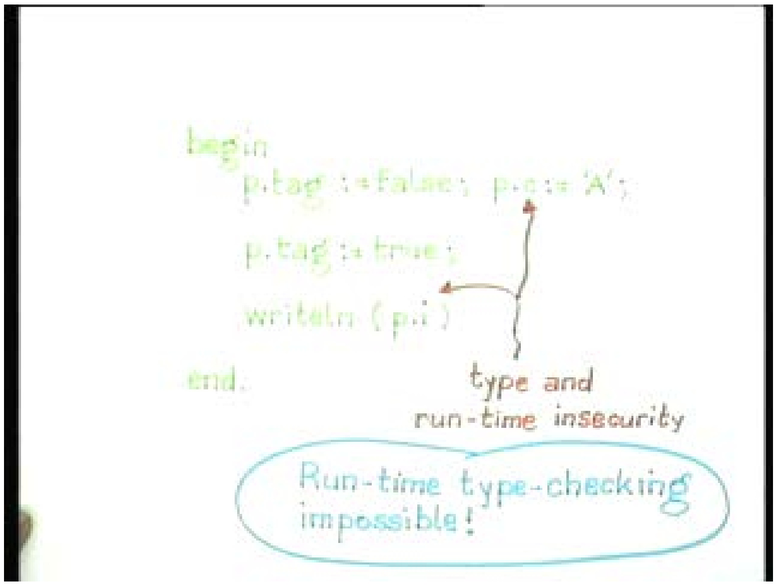
Pascal

```
var p : record
            case tag:boolean of
               true:(i:integer);
               false:(c:char)
        end
```

Remember that when you take a disjoint union of two sets the injection functions are such that  once the identity of the individual components of a set are known the operations of that data type become applicable. You import also the operations of that data type. For example; if you are taking the disjoint union of integers and characters and you know that there is an element which actually came from the integers then you could for example do addition and subtraction. You could take two different records whose parent component is integer and you could add those values.

If the characters are there you could compare the two characters and take some successor or predecessor of the character. But the point is that if you have two different records P and Q with the same declaration and one has a tag which is true and the other has a tag which is false then theoretically, you cannot add the integer component of one to the character component of the other. The operations remain distinct. However, since the record is just a variable and its individual fields are also just variables one could actually assign the tag value and then one could change the value of the tag and then look at an operation from the other data type. You started of with P being from the characters and you assigned it a character and then you changed the tag. So, P became part of the integers and now you can write out the integer value. This for example leads to type and runtime insecurities.

(Refer Slide Time: 25:44)



```
begin
    p.tag := false;  p.c := 'A';
    p.tag := true;
    writeln (p.i)
end.
```

type and
run-time insecurity

Run-time type-checking
impossible!

The pride of Pascal implementation is that every type is compile- time determinable. With the variant records you have something that is not compile-time determinable and that is not even really run time dependable and in the sense that you cannot introduce run time checks because you do not know how the tag is going to vary. After all I could change this to some complicated Boolean expression. So, it has a schizophrenic type which keeps changing.

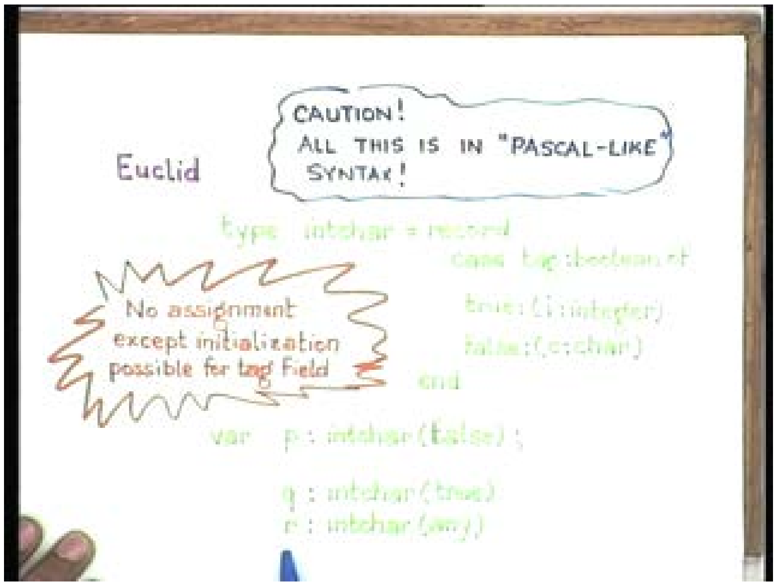
The actual value does not change of the individual components but there is a side effect on their type of the individual component by changing the value of the tag field.
This is the reason this construct came under severe criticism because it leads to firstly, an abuse of the type structure of the language and then it makes runtime type checking really impossible. After all there are other ways of doing the same thing and they should be used. You should not abuse a flexibility that is provided.

Many languages actually avoided this problem in various ways. One is the language of Euclid. Let me first say that I will always use Pascal or ML type syntax and I do not swear by that syntax. I do not know the syntax of Euclid or of the many other languages that I would be discussing and I will write them in a sort of Pascal like fashion. The whole idea is that you should worry about the semantics rather than the syntax. This syntax is definitely not Euclid compilable. What I have used is a Pascal like syntax here.

In that PASCAL declaration you could have actually defined that variant record as a separate type and then declared that variable P as being of that type. You can give that type a name and then declare that variable P as of that type. That could have been done. But the point is that it does not solve the problem of those runtime insecurities. EUCLID actually parameterized the tag in the declaration itself. If you declared some variable to be of this type then since this is a variant record type you had to give an initialization for that tag and the compile time check was introduced to ensure that there was no way that the tag was changed in the program.

The identity of the disjoint union was preserved by putting an initialization for the tag field in the declaration of a variable itself. So, you could not mix two variables very easily. You could however define two different variables of the form p and q like this. So, p comes from the integer component of the disjoint union and q comes from the character component of the disjoint union. The properties of the injection function are rigidly maintained but very often you require temporary variables.

(Refer Slide Time: 28:30)



Euclid

CAUTION!
ALL THIS IS IN "PASCAL-LIKE"
SYNTAX!

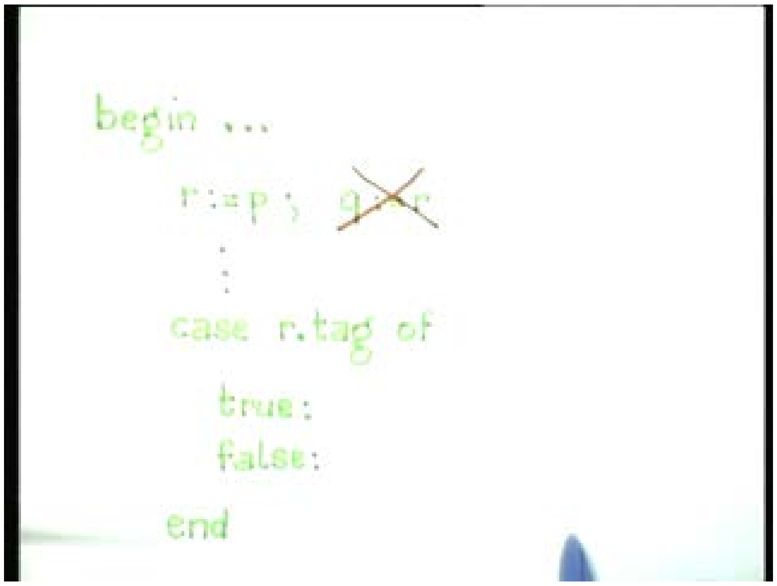
```
type intchar = record
                  case tag:boolean of
                   true:(i:integer)
                   false:(c:char)
                end
      var p : intchar(false);

          q : intchar(true)
          r : intchar(any)
```

No assignment
except initialization
possible for tag Field

You could declare some variable like r in which there was a new reserved word called 'any' and it meant that you could assign any type to this. The natural question is, in the main program can I assign a value to p, assign that p to r and assign that r to q? Thereby I would have moved from one component of the disjoint union to the other. So, Euclid banned all forms of such assignments. If you are going to use r as a temporary variable you could assign p to r but you could never assign r to q. However, once r has been declared to be of the same type as p, you could assign r to some other variable say s which is of the same type as p. There is a runtime type checking. Based on the tag value you could do appropriate assignments.

(Refer Slide Time: 30:33)



begin ...
    r:=p; ~~q:=r~~
      :
    case r.tag of
      true:
      false:
    end

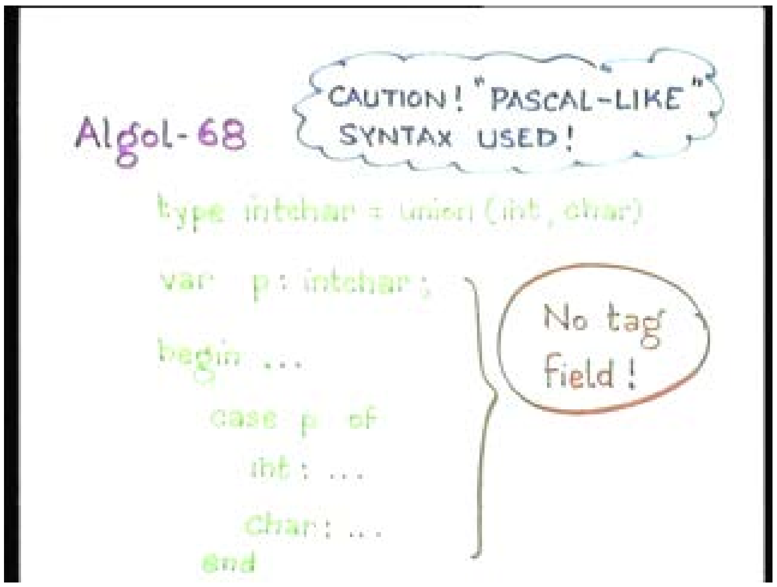
If this portion were inside a loop then occasionally depending on the value of the variables that you are using, r might be either of the type integers or it might be of the type character and you could do a case analysis. But you could never make such an assignment. Note that in some sense this r is of a super type compared to either of the p or q. By assigning p to r you are assigning a p which is of narrower type to a variable of a wider type. That is called widening.

You could do widening but you could not do narrowing which means taking a value of the super type and assigning it to a narrower type. You could not do this arbitrarily; it was very difficult to do this. There were some predefined tags but it also complicates matters a lot when you want to actually do explicit type cohesion. But their single purpose in life was to avoid the pit falls of the variant records in Pascal and so they took this view.

The language Algol-68 actually took a more pragmatic view. Let me first mention that Algol-68 has got no relation whatsoever to Algol 60 and it is not like FORTRAN 4, FORTRAN 77 and FORTRAN 90. Algol-68 is a completely new language whose main design issues were not only to specify the context-free syntax. Algol 60 was the first language which used a BNF notation to specify the context-free syntax and the idea in Algol 68 was not only that you should specify context-free syntax; you should also specify context-sensitive syntax.

They had grammar rules which were so complicated that I doubt if anybody other than the language designers have actually written substantial programs in this language.
But Algol 68 had a very interesting method which was just that they actually gave a union of types in this fashion. You could actually define a union like this and do a case analysis. There was no tag field and therefore there was no possibility of abusing the tag field.

(Refer Slide Time: 33:12)


Algol-68

CAUTION! "PASCAL-LIKE" SYNTAX USED!

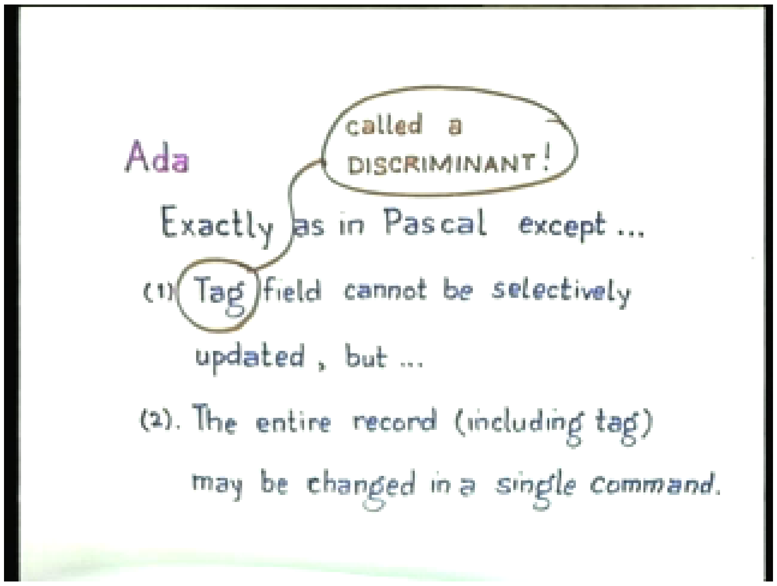
```
type intchar = union (int, char)
var  p : intchar;
begin ...
   case p of
     int : ...
     char : ...
   end
```

No tag field!

You could look at any particular variable of this co-product type and you could do a case analysis based on whether it comes from here or it comes from here. There was absolutely no tag field and therefore there were no tag field assignments; there was no way you could change the tag field assignments, there was no question of widening or narrowing, they thought they had put everything in watertight compartments and things could be compile-time checked. But you could do assignments between various variables of the type ‘int char’. But along with that information there is the actual value of the assignment; the parent type that came from injection function which was also carried through.

You had to always do a case analysis on finding what the injection function used in order to do whatever manipulations you want. More modern languages like Ada for some reason just used exactly what Pascal has done in spite of knowing about all the insecurities of the Pascal variant record. The only difference was that Ada gave it a four syllable name called a discriminant which sounds more grandiose. But in order to avoid the insecurities they said that any time you are trying to change the tag field you cannot selectively update the tag field. You have to change the entire record that is you have to reassign the entire record. So, in this way they allowed for a flexible cohesion of types but then that cohesion meant that the programmer by having to change the entire record including the tag knew exactly what he was doing.

(Refer Slide Time: 36:00)



Ada

called a DISCRIMINANT!

Exactly as in Pascal except ...

(1) Tag field cannot be selectively updated, but ...

(2). The entire record (including tag) may be changed in a single command.

So, you do the changing of the entire record in a block but Ada also allowed various kinds of compound initializations.
Since ADA allowed mechanisms and even syntactical mechanisms for doing initializations of aggregate objects or compound objects you could change the entire record also in a single command through various syntactic means by the use of various kinds of brackets.

If you had records within a record so the components of a record would be written in parenthesis very much like the ML tuples and if they were records within records then you would have more and more nested parentheses inside and so a large record could be changed in a single assignment command by using an aggregate. The large record including the tag could be changed in a single command by using an appropriate aggregate but then that only meant that the programmer knew explicitly that he was changing the tag field and he was doing the change in the tag field at the same time that he was changing the other components. They are not being done in a selective or distributed fashion.

The code for changing the tag field and changing other areas of the record were not distributed in various places within that program. It was all done in one single command.
So, it localized the variant record problem of Pascal which they found the most acceptable to them. There is a co-product construct also in ML but that is at the level of data types. Let us see what a typical ML co-product would look like. If you look at a disjoint union it is not just the sum of two sets. It is actually a sum of two different data types.

If you have integers and characters and if you are taking the disjoint union of integers and characters then you are also importing for the appropriate components the appropriate operations available with integers or the appropriate operations with characters. You are actually defining not just a new set but a new set with a new set of operations where each operation comes from either of the two injection functions.

If you look at this construction of an int char data type, it is not just a construction of a new set but it also has summation, multiplication, ordinal numbers of characters, getting a character from an integer and it has all these operations except that all these operations are conditional and undefined.
For example; ordinal number of an integer or adding two integers is not available in the character domain. It is conditional and so you are actually defining a new data type of this form. By data type we mean a set equipped with its operations. Integers are not just a set but they are data type, which have associated operations with them. You could define a data type like this. My ML syntax is also quite informal so I do not guarantee that all this is compilable.

You actually define the injection functions in ML as;

```
int char = in1 of int 1
           in2 of char
```

Each time you would carry this injection function itself as the tag for each element. Whenever you are assigning a value to this data type you would actually write something like this in1 5 let us say. The injection function itself acts as a tag and they have two distinct tags. Since it is a functional language everything has a constant value. So, you actually have two different tags for this data type. If you did an n fold summation you had n different data types because the injection functions were always tagged on as patterns with the component and the injection functions ensured that you had type compatibility whenever you performed operations.

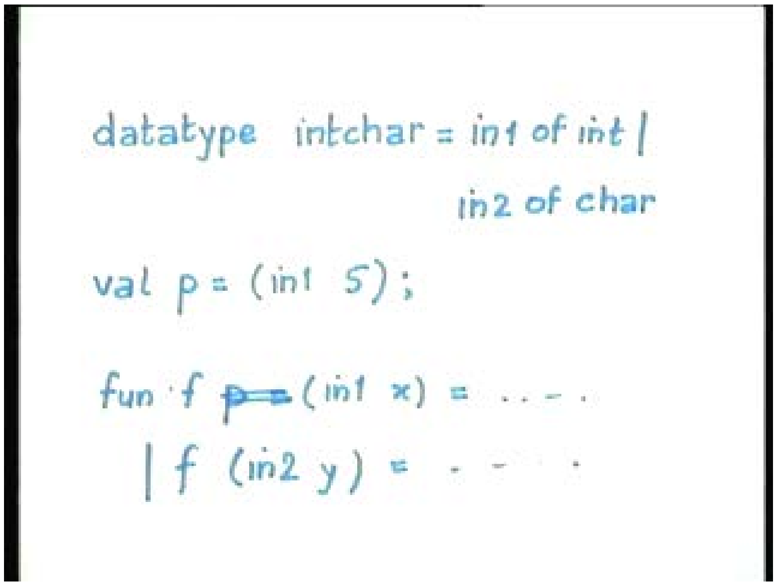
You could not do strange cohesions unless you were explicitly conscious of it and unless you took the ordinal value of some character you could not treat it is an integer. ML actually follows the mathematical definition of a co-product or a summation in a very strict fashion. Lastly, we have to look at storage allocation issues and also that given an arbitrary variable of this type you could do a case analysis. You could have a function some,

$funf\,(\text{int}\,x) = .....$

$1\text{f}\;(\text{in2y}) = .....$

You could do the standard case analysis that is available in ML through the pattern matching that is also available to do the case analysis for an arbitrary variable of this summation type.

(Refer Slide Time: 42:35)

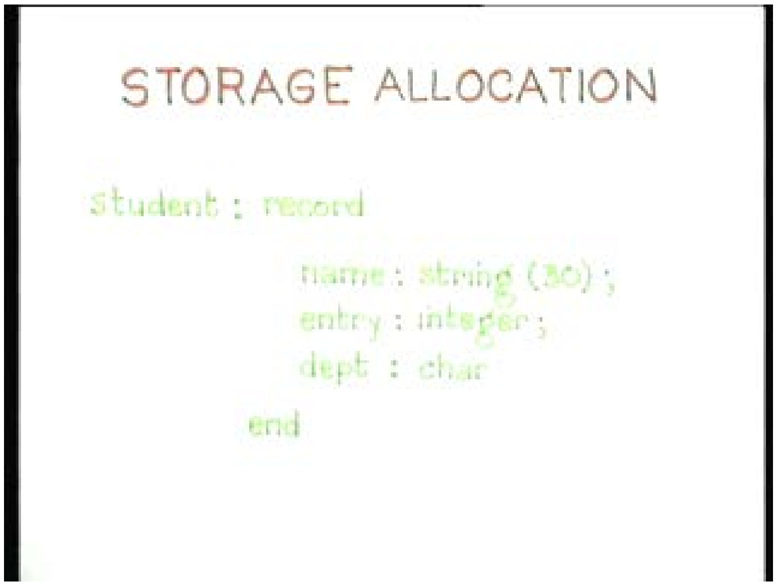


datatype intchar = in1 of int |
in2 of char

val p = (in1 5);

fun f ~~p =~~ (in1 x) = . . . .
| f (in2 y) = . . . .

(Refer Slide Time: 43:20)

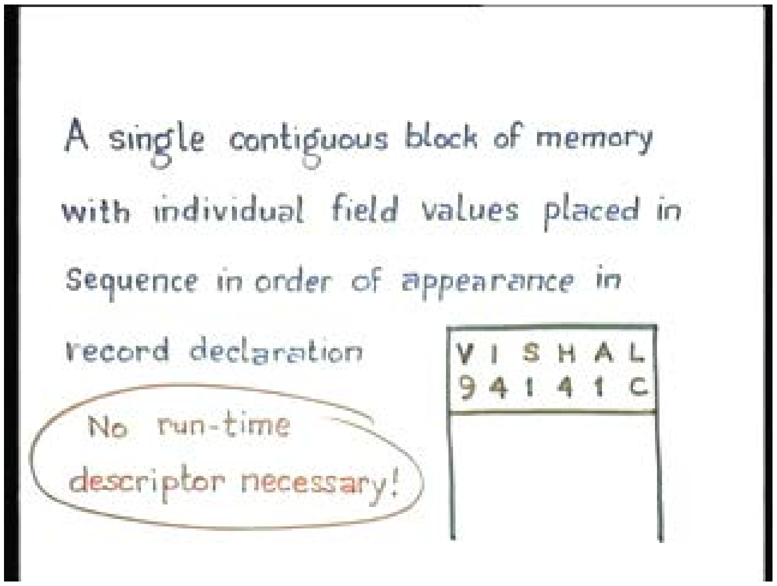


## STORAGE ALLOCATION

```
Student : record
              name : string (30) ;
              entry : integer ;
              dept : char
          end
```

Finally we have to look at storage allocation issues. Let us first look at ordinary records without variants. By records here I do not mean ML records but in general Pascal like records or PL 1 and COBOL like structures. You know how much space you require for each of these components and you allocate a contiguous block of memory equivalent to the sum of the space that is required for each of them at runtime. You just allocate it all contiguously and you just place the individual fields in contiguous locations and in the order of appearance in the record declaration which also brings in another issue. It is just another of these various insecurities in this imperative languages which have records.

(Refer Slide Time: 44:09)

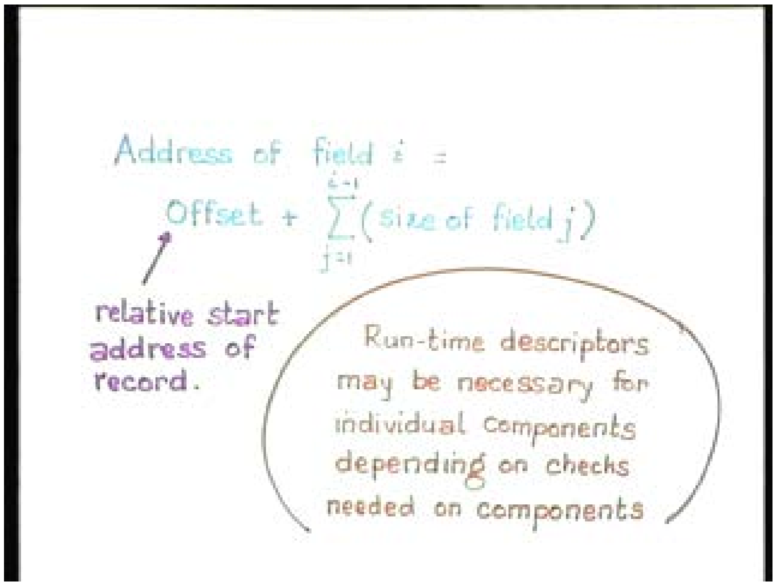

A single contiguous block of memory with individual field values placed in Sequence in order of appearance in record declaration

V I S H A L
9 4 1 4 1 C
No run-time descriptor necessary!

If you do this then you do not really require any runtime descriptor. If it is a variant record in Pascal then you do not really know what kind of runtime descriptor you can keep for it. The address of any particular field which you require for deconstruction is just some offset plus some of the sizes of the individual fields up to that and since the order of appearance in the declaration is important, which is the order in which they are also stored, you can just take the size of individual fields and contiguously allocate it. You can randomly access an individual field of a record by a simple compile time calculation. Only the offset is going to be known at runtime. Part of the offset consists of a relative start of the block in the runtime stack.

The other is the absolute position of where that block is going to start in the runtime environment and that depends on its life time.
It depends on the number of procedures, calls or number of blocks under it etc.
The offset consists of an absolute component which is not known except at runtime and a fixed relative component which is known at compile time.
The size of each of these fields is known at compile time. This calculation of the fixed offset from the current base of activation record to the appropriate field is all compile time determinable. So, you can do random access quite easily with such a structure.

(Refer Slide Time: 46:15)


Address of field $i$ =

$$\text{Offset} + \sum_{j=1}^{i-1} (\text{size of field } j)$$

relative start address of record.

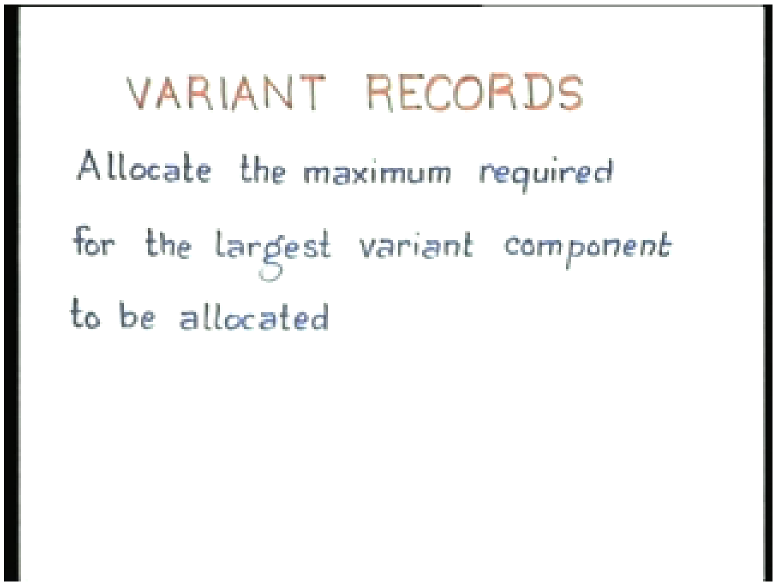
Run-time descriptors may be necessary for individual components depending on checks needed on components

Since a record is a structured data type using individual components you might require some runtime descriptors in order to do various kinds of checking.
After all for example, you might have a record whose one component is an array with certain fixed bounds then you have to ensure that the array index does not go out of bounds. The runtime descriptors are usually necessary for arrays.

For the record as a whole you are not going to have any runtime descriptors. You are just going to have run time descriptors as appropriate for the individual components. In the case of a variant record you really have nothing to do except allocate some maximum value computed over all possible variants of the record. You just look at all possible variants of the record and find out which one requires the maximum amount of storage.

The fixed part of the record anyway is going to use only that much which is one of the reasons why Pascal insists that the tag field and the variant component should always be the last declaration in a record if it is present so that the fixed parts of the record anyway have fixed memory allocation and the variant part could have variable amounts of memory allocation.

(Refer Slide Time: 48:12)



## VARIANT RECORDS

Allocate the maximum required for the largest variant component to be allocated

But normally, most Pascal implementations would just take the maximum over all possible variants that you have and just allocate that much space. If you are looking at smaller variants, they just occupy a space that is a part of this larger space that you have allocated and the initial position of that larger space is all that they will occupy.
That is how these records structures are implemented and that is how they also allow random access to the fields of the record.