## Record and Structure Issues

- A record or struct is a collection of *fields*.
  Storage must be allocated for the structure and the layout of the fields in the structure must be computed.
- Compilers symbol table must be organized so that references to record fields can be resolved.
  Most languages require fully specified record field references: R.A[I].B.C[J].D.
  PL/I allows partially specified field references and floating subscripts , e.g. R[I].B.D[J] as long as the reference is unambiguous.
  *This is terrible language design*.
- Storage for records and structures can be *packed* or *unpacked*. Data items in a structure can be aligned on their natural hardware boundaries or they can be *unaligned*. Packing usually implies unaligned data. Access to unaligned data is usually much slower than access to aligned data.



## Record and Structure Issues

- It must usually be possible to access any field of a structure given only the base address of the structure and a *compatible* declaration for the structure.
- There are several possible definitions for *compatible declaration*
  - *Strict Equivalence* The declaration used to access the structure must be the one that was used to declare the structure. (Pascal, Turing).
  - *Left to Right Equivalence* The declaration used to access the structure needs to match the structure declaration only up to the field being accessed. (C, PI/I)
    *Implies*: Must map structure fields in left-to-right order.
    Position of field can not depend on following fields.
  - *Major Minor Equivalence* Access to any embedded substructure in a structure must be possible given only the address of the substructure and a declaration for the substructure. (most languages)
    *Implies:* Must use same algorithm for structures and contained substructures
    Position of fields in a substructure can only depend on the substructure.



## Storage Mapping for Records and Structures

- For most languages, the fields of a structure are laid out in memory in left to right order. *Internal fill* is introduced between scalar items as required so that scalar items are properly aligned on their natural hardware boundaries.
- Most structure storage mapping algorithms try to achieve
  - Proper storage alignment for all scalar data items.
  - Economical use of space, i.e. minimize internal fill.
  - Use *no* internal mapping information (embedded pointers)
  - Allow access to field given only base address and compatible declaration.
- If a structure is packed (data items are unaligned) then the fields in a record are layed out in order with no intervening fill

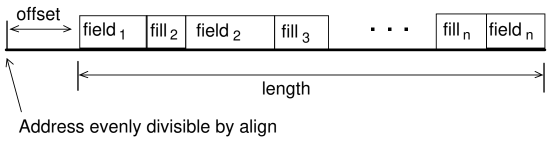


## Storage Mapping and Alignment

- There are several possible algorithms for laying out fields in a record. They differ in the sophistication of their fill minimization strategies.
- Definitions for storage alignment
  $address_i$ is the relative address (in bits) of $field_i$ in the structure
  $align_i$ is the alignment constraint (in bits) for $field_i$
  $align_i$ depends on the type of $field_i$.
  $size_i$ is the size (in bits) of $field_i$
  $fill_i$ is the amount of internal fill introduced in front of $field_i$
  so that it is properly aligned.
- The normal alignment constraint for structure fields is
  $offset_i \; = \; address_i \;\; mod \;\; align_i$
  Where $offset_i$ is the offset from an address evenly divisible by $align_i$ to the start of $field_i$. $offset_i = 0$ for scalar data items.

## Structure Storage Mapping Algorithms

- A storage mapping algorithm for a structure with $n$ fields computes
  - $align$ the alignment factor for the entire structure
  - $offset$ the offset for the entire structure.
  - $fill_i$ the amount of fill in front of $field_i,\ 2 \leq i \leq n$
  - $length$ the total size of the structure.
    - $length = length_1 + \sum_{i=2}^{n}(length_i\ +\ fill_i\ )$
- Storage mapping memory layout




- The *mod* function used for storage mapping algorithms is defined as:
  - For $a > 0, \quad 0 \leq x\ mod\ a < a$
  - For $x < 0, \quad x\ mod\ a\ \ =\ \ a - |x|\ mod\ a$



## Structure Mapping Algorithm 1

- A simple algorithm sometimes used for Cobol.
  Structure is an ordered collection of fields. Substructure is ignored.
  $length_i$ and $align_i$ determined only by $field_i$.
- Algorithm:

  $align = max_{i=1}^{n}\ align_i$
  $offset = 0$
  $length = length_1$
  **for**( i = 2 ; i <= $n$ ; i++ ) $\{$
  $\quad fill_i = -length\ mod\ align_i$
  $\quad length = length + fill_i + length_i$
  $\}$

- Algorithm 1 adds just enough fill in front of each field to make the field properly aligned in memory.
- Works for left-to-right equivalence, but not major-minor equivalence.



## Storage Mapping Algorithm 2

- Map structures inside-out, i.e. substructures first, one level at a time.
  Uses the freedom to adjust $align$ and $offset$ to save space.
  Aligns the field with the most severe alignment constraint only after packing the previous fields as tightly as possible.
- Algorithm 2:

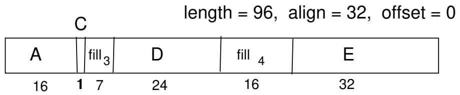
  $align = align_1 \qquad offset = offset_1 \qquad length = length_1$
  **for**( i = 2 ; i <= $n$ ; i++ ) $\{$
  $\quad$ **if**( $align \geq align_i$ )
  $\quad\quad fill_i = (\ offset_i - offset - length\ )\ mod\ align_i$
  $\quad$ **else** $\{$
  $\quad\quad fill_i = (\ offset_i - offset - length\ )\ mod\ align$
  $\quad\quad offset = (\ offset_i - fill_i - length\ )\ mod\ align_i$
  $\quad \}$
  $\quad align = max(\ align\ ,\ align_i\ )$
  $\quad length = length + fill_i + length_i$
  $\}$



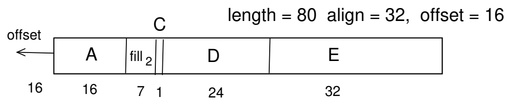
## Structure Mapping Example

```
DECLARE
    1 X ,
        2 A  CHAR( 2 ) ,          /*  align = 8,  length = 16 */
        2 B  ,
            3 C BIT( 1 ) ,          /* align = 1, length = 1 */
            3 D CHAR( 3 ) ,      /* align = 8, length = 24 */
            3 E FIXED BINARY( 31, 0 ) ;
                                   /* align = 32, length = 32 */
```

### Algorithm 1




### Algorithm 2



## Algorithm 2 - Substructure First

| | i | align | offset | length | $align_i$ | $offset_i$ | $length_i$ | $fill_i$ |
|---|---|---|---|---|---|---|---|---|
| C | 1 | | | | 1 | 0 | 1 | |
| before loop | | 1 | 0 | 1 | | | | |
| D | 2 | | | | 8 | 0 | 24 | 0 |
| $fill_2$ = ( 0 - 0 - 1 ) mod 1 = -1 mod 1 = 0 | | | | | | | | |
| offset = ( 0 - 0 - 1 ) mod 8 = -1 mod 8 = 8 - 1 mod 8 = 7 | | | | | | | | |
| | | 8 | 7 | 25 | | | | |
| E | 3 | | | | 32 | 0 | 32 | 0 |
| else case | | | | | | | | |
| $fill_3$ = ( 0 - 7 - 25 ) mod 8 = -32 mod 8 = 0 | | | | | | | | |
| offset = ( - 0 - 0 - 25 ) mod 32 = -25 mod 32 = 32 - 25 mod 32 = 7 | | | | | | | | |
| | | 32 | 7 | 57 | | | | |

So, the inner structure B has Length 57, Alignment 32, Offset 7



## Algorithm 2 - Outer Structure

| | i | align | offset | length | $align_i$ | $offset_i$ | $length_i$ | $fill_i$ |
|---|---|---|---|---|---|---|---|---|
| A | 1 | | | | 8 | 0 | 16 | |
| before loop | | 8 | 0 | 16 | | | | |
| B | 2 | | | | 32 | 7 | 57 | 7 |
| else case | | | | | | | | |
| $fill_2$ = ( 7 - 0 - 16 ) mod 8 = -9 mod 8 = 8 - 9 mod 8 = 7 | | | | | | | | |
| offset = ( 7 - 7 - 16 ) mod 32 = -16 mod 32 = 32 - 16 mod 32 = 16 | | | | | | | | |
| | | 32 | 16 | 80 | | | | |



## Checking Structure Mapping

Goal of algorithm: to guarantee that all atomic data items are properly aligned. Checking:

- Assume that the address given by *start_of_structure - structure_offset* is properly aligned according to *structure_alignment*

  i.e., in the example below, the address 16 bits before the start of the structure should be evenly divisible by 32.
- Find the relative location of each atomic field in the structure by adding up the offset, the lengths of all fields that precede it and the length of all fill that precedes it.
- Check that this relative offset is evenly divisible by the alignment constraint for the atomic item.

Examples:

| Field | Relative location | Alignment constraint |
|---|---|---|
| A | 16 | 16 mod 16 = 0 |
| D | 16 + 16 + 7 + 1 = 24 | 24 mod 8 = 0 |
| E | 16 + 16 + 7 + 1 + 24 = 64 | 64 mod 32 = 0 |



## Other Record/Structure Issues

- Can implementation reorder the fields in a record or structure to minimize internal fill?
- Storage mapping for unions and variant records.
- Rubber objects (e.g. arrays of dynamic size) in records.
- Value of fill and record comparisons.
- Initialization of record variables, initialization in record *type* declarations.

## Dynamic Arrays

- In many languages, the lower and/or upper bounds of an array in
  $typeA[\ L_1 : U_1\ ,\ L_2 : U_2\ ,\ \ldots\ L_n : U_n\ ]$
  can be run time expressions, the size of the array is determined when these expressions are evaluated at run time.
- for dynamic arrays, compiler must emit run time code to
  - Validate bounds at run time. $\forall i L_i \leq U_i$
  - Allocate storage for the array and save the $base$ address of the array.
  - Save a table of mapping information that will be used to calculate array subscripts.
  - Usually save $L_i$ and $mult_i$. $U_i$ is only needed for array subscript range checking.
    If all $L_i$ are compile time constants then only $mult_i$ is required.
    This is why Turing requires the lower bounds of all arrays to be compile time constants.
- If the language allows subarrays (e.g. PL/I, APL) then saving $mult_i$ instead of $stride_i$ makes subarray computation much easier.



## Sub Arrays

- If $typeA[\ L_1 : U_1\ ,\ L_2 : U_2\ ,; L_3 : U_3\ ]$ is a 3 dimensional array, then it has the subarrays:

| | | | |
|---|---|---|---|
| A[I, J, K] | A[ I, *, * ] | A[ *, J, * ] | A[ *, *, K ] |
| A[ *, J, K ] | A[[ I, *, K] | A[ I, J, * ] | A[ *, *, * ] |

- For an array reference $A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]$
  Assume each array subscript $E_i$ is an expression or *. Calculate

$$newBase = base + \sum_{i=1}^{n}(if E_i = *\ then\ 0\ else\ (E_i - L_i) * mult_i)$$

  For each $E_i$ which is *, copy table entry for that dimension to new table.
- Dimension of new sub array is number of subscripts where $E_i$ = *



## Array and Subarray Example

- The array declaration **int** A[ 1:4, 0:3, -10:10, 1:12 ] has the table

| | | |
|---|---|---|
| base | 4 | 16128 |
| 1 | 4 | 4032 |
| 0 | 3 | 1008 |
| -10 | 10 | 48 |
| 1 | 12 | 4 |

- The table for the subarray A[ *, 2, *, 6 ] is

| | | |
|---|---|---|
| base+2036 | 2 | 48384 |
| 1 | 4 | 4032 |
| -10 | 10 | 48 |

- Note that the subarray is not compact (logically consecutive array elements are not in contiguous memory locations).

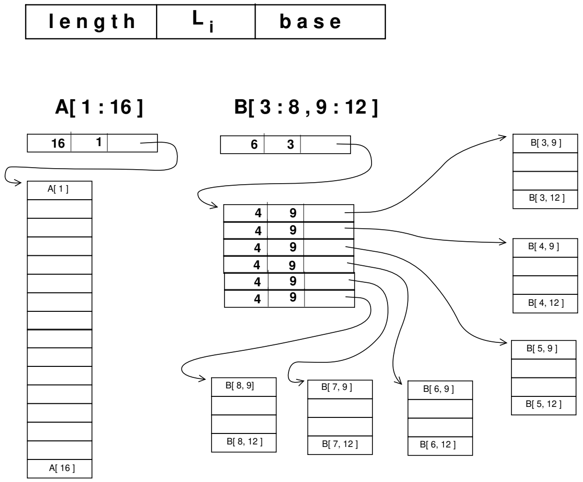


## Descriptor based Array Access

- An alternative approach to array storage mapping is to consider only all arrays as one dimensional and to use a runtime data structure called a *array descriptor* or *dope vector* for array access.
- This approach costs some runtime storage for the descriptors, but it generalizes nicely to arrays of arbitrary dimensions and allows right subarrays.
- Advantages: Allows easy array bounds checking, compatible with virtual memory mapping of array rows.
- Disadvantages: without copying, can only do right most subarrays, i.e. A[ I, J, *, * ].
  Array access cost is not symmetric w.r.t rows and columns.
- Space required for descriptors is $1 + \sum_{i=2}^{n} \prod_{j=2}^{i} stride_j$

## Array Descriptor Example

**Descriptor**



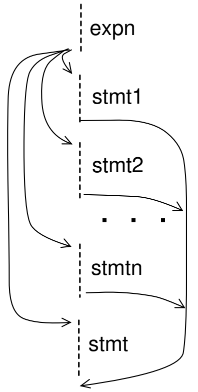



food for slide eater



## Translating Switch and Case Statements

```
switch   ( expn ) {
  case    L1  :  statement
  case    L2  :  statement
           .  .  .
  case    Ln  :  statement
  default     :  statement
}
```






## Switch/Case Translation Issues

- Use different implementation strategies depending on label set
  - *Dense Set* Use branch table indexed by expression.
  - *Sparse Set, Small* Map statement into a chain of if statements.
  - *Sparse Set, Large* Generate an internal hash table.
  - *Non Indexable* Generate an internal hash table.
- Switch/case can be compiled in one pass by recording information on each case and generating a branch table after all cases have been seen.
- Need stack or table of labels, and statement start addresses.
- Must be able to handle nested case statements.
- Should test expression limits to avoid wild jumps.

## Switch Processing

- Each case in a switch statement is associated at least one (and perhaps several) case labels.
- For each case, need to keep track of label value and start of corresponding statement.
- Example that follows assumes compact and dense case label set and builds a branch table implementation.
- For convenience, branch table is built *after* the switch statement.
- Use branch generation and patching mechanisms similar to those used for **if** and **for** statements. Assume a list of (label, statement start , break statement addresses ) triples. Utility functions:
  - Create one list node $makeCaseNode(label, start, breaks)$
  - Append node to list $append(list, node)$
  - Emit branching table $emitTable(list)$



## Translating switch Statements - labels and breaks

$switchCase \;=\; caseLabels \;\; statement$

$\quad node_{list}$ =

$\quad\quad$ makeCaseNode( $caseLabels_{list}, statement_{start}, statement_{breaks}$ )

$caseLabels \;=\; caseLabel$

$\quad node_{list} = caseLabel_{list}$

$caseLabels \;=\; caseLabels \;\; caseLabel$

$\quad node_{list}$ = append( $caseLabels_{list}, caseLabel_{list}$ )

$caseLabel \;=$ **case** $label$ **:**

$\quad node_{list}$ = makeNode($label_{value}$ , unused , unused )

$statement \;=$ **break**

$\quad node_{breaks} = nextcode_{true}$

$\quad$ emit( branch, true, ?, unused )



## Translating switch Statements - control

$statement \;=$ **switch** $switchHead \;\{\; switchList \;\}$

$\quad node_{start} = switchHead_{start}$

$\quad$ patch( $switchHead_{true}, nextcode$ )

$\quad$ emitTable( $switchList_{list}$ )

$\quad$ patch( $switchList_{breaks}, nextcode$)

$switchList \;=\; switchCase$

$\quad node_{list} = switchCase_{list}$

$switchList \;=\; switchList \;\; switchCase$

$\quad node_{list}$ = append( $switchList_{list}, switchCase_{list}$ )

$switchHead \;=\; (\;\; expn \;\; )$

$\quad node_{start} = expn_{start}$

$\quad node_{true} = nextcode$

$\quad$ emit( subsaddr, ?, $expn_{result}, R_i$ )

$\quad$ emit( branch, true, $R_i$, unused )



## switch Example

| | | | | | |
|---|---|---|---|---|---|
| switch( I + 1 ) { | 1 | ( add | , I | , =1 | , $R_1$ ) |
| | 2 | ( subsadr | , $T_{13}$ | , $R_1$ | , $R_2$ ) |
| | 3 | ( branch | , true | , $R_2$ | , ) |
| case 1 : J += 2 ; | 4 | ( add | , J | , =2 | , $R_3$ ) |
| | 5 | ( assign | , $R_3$ | , | , J ) |
| break ; | 6 | ( branch | , true | , $T_{17}$ | , ) |
| case 3: J = J * 4 ; | 7 | ( mult | , J | , =4 | , $R_4$ ) |
| | 8 | ( assign | , $R_4$ | , | , J ) |
| break ; | 9 | ( branch | , true | , $T_{17}$ | , ) |
| case 2 : | | | | | |
| case 4 ; J = J - I ; | 10 | ( sub | , J | , I | , $R_5$ ) |
| | 11 | ( assign | , $R_5$ | , | , J ) |
| break ; | 12 | ( branch | , true | , $T_{17}$ | , ) |
| } | 13 | ( branch | , true | , $T_4$ | , ) |
| | 14 | ( branch | , true | , $T_{10}$ | , ) |
| | 15 | ( branch | , true | , $T_7$ | , ) |
| | 16 | ( branch | , true | , $T_{10}$ | , ) |