During IR generation, do we make use of target machine specifics?
- Addressing modes
- Specific, weird instructions
- Size of data

No

Isolate machine dependencies in final code generation

To retarget the compiler...

Replace the final code generation

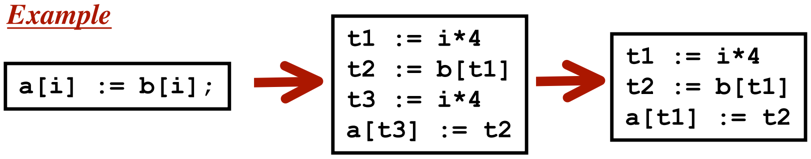
Don’t need to modify... IR code generation
Optimization phase

Yes

IR code generator creates more specific code

The optimizer can improve this code

Example



```
a[i] := b[i];
```

```
t1  := i*4
t2  := b[t1]
t3  := i*4
a[t3]  := t2
```

```
t1  := i*4
t2  := b[t1]
a[t1]  := t2
```

Our Approach to Variables

For each routine (and main body)
- Run through the stmts and generate IR quads
  - Also compute “**maxArgNumber**”
    - When generating IR code for “foo”

```
... bar(aaa, bbb, ccc, ddd) ...
        1    2    3    4
```

- Assign offset to our variables

```
nextOffset := -4
for each VarDecl p do
        p.offset := nextOffset
        nextOffset := nextOffset - 4
endFor
```

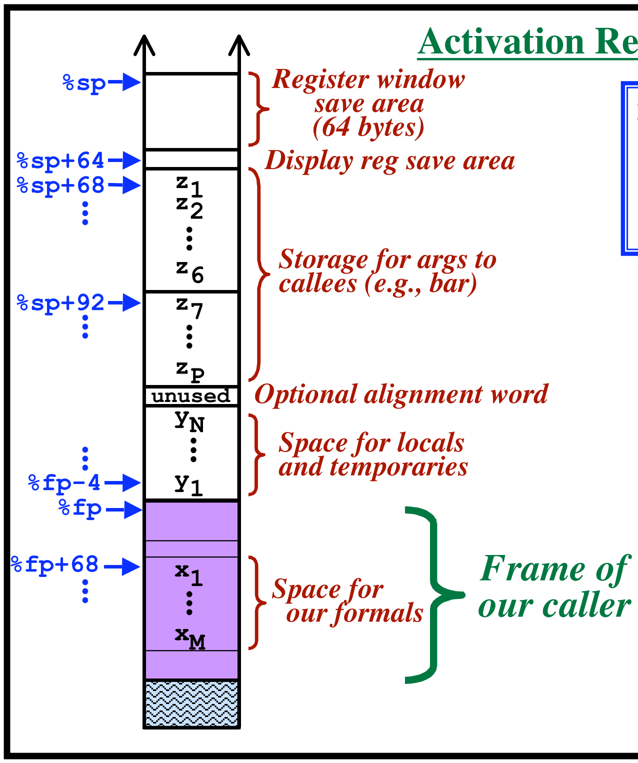
- Assign offset to our formals

```
nextFormal := 68
for each Formal f do
        f.offset := nextFormal
        nextFormal := nextFormal + 4
endFor
```

- Compute “**sizeOfFrame**”

```
body.sizeOfFrame := ...
```

Activation Record Layout

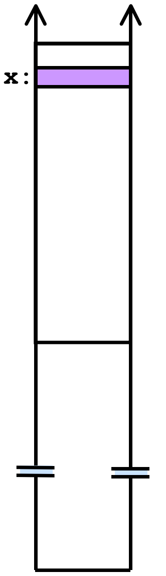


```
procedure foo (x1,x2,...xM)
  var y1,y2,...yN;
  begin
    ... bar(z1,z2,...zP)...
  end;
```

Let “P” be the *MaxArgNumber*, for all routines that “foo” calls

Records (in C-like languages)

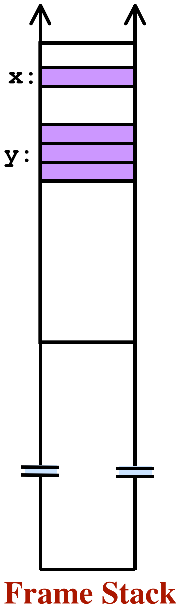
```
var x: integer;
```



Frame Stack

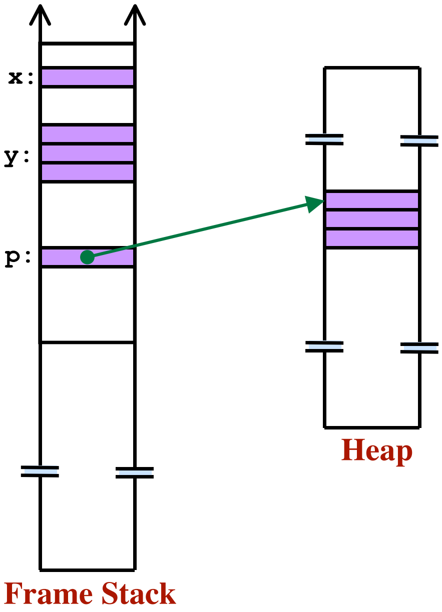
Records (in C-like languages)

```
var x: integer;

type T is record
            f1: ...;
            f2: ...;
            f3: ...;
          end;
var y: T;
```



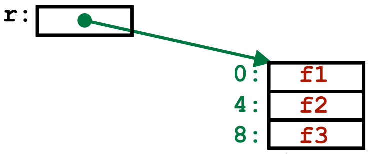
Records (in C-like languages)

```
var x: integer;

type T is record
            f1: ...;
            f2: ...;
            f3: ...;
          end;
var y: T;

var p: ptr to T;
```



Records in PCAT

```
type T is record
             f1: ...;
             f2: ...;
             f3: ...;
           end;
var r: T;
... := r.f2;
r.f3 := ...;
```



We will allocate all records and arrays on the heap

IR Code for reading a field "... := r.f2"

```
t8  := r+4
t9  := *t8
...  := t9
```

genExpr → t9

IR Code for setting a field "r.f3 := ..."

```
t11  := r+8
```

genLValue → t11

```
t12  := ...
```

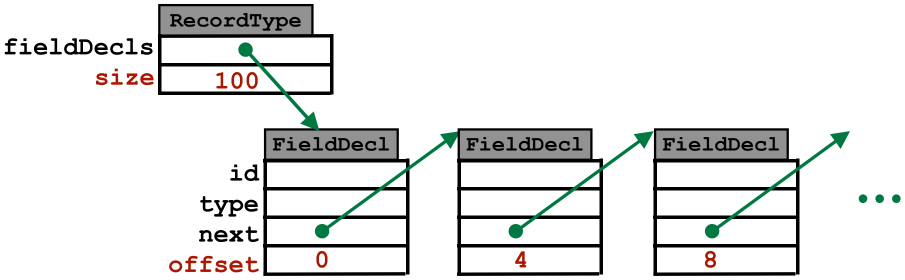
genExpr → t12

```
*t11  := t12
```

Our Approach

When walking the AST...
    we will visit all nodes...
        whenever we see a "RecordType" (in genRecordType)...

- Walk the list of fields.
  Fill in the offsets.
- Set the "size" of the entire record (in bytes).
- Must recursively handle nested record types.



Translating Assignment Statements

Source:

```
x := y + 3;
```

Translation:

```
t1 := &x
t2 := ...
*t1 := t2
```

genLValue → t1

genExpr → t2

Most general approach.
Handles complex L-Values

```
a[ foo(...) ] := y + 3;
```

Problem:

Difficult to optimize this:

```
t := &x
...
*t := ...
```

Solution:

In **genAssignStmt** watch for special case:
L-Value is a simple variable

```
t2 := ...
x := t2
```

This is easier to optimize (see optimization phase)

Goal: Reduce temporary usage!

Goal: Reduce Temp Usage

$E_0 \rightarrow E_1 * E_2$ Generates:

```
t5 := ...E1...
t6 := ...E2...
t3 := t5 * t6
```

Note: t5 and t6 are never used again.
Idea: Recycle!

`newTemp() → temp`

Create a new temp and return it.

`recycle (temp)`

Maintain a collection of unused temporaries
(Each routine will start with an empty collection)

recycle ()
Add the temp to collection.

newTemp()
Get a temp from the collection.
Create a new temp only when necessary.

Source:

```
x := ((a*b) + (c*d)) - (e*f);
```

Before:

```
t1  := a * b
t2  := c * d
t3  := t1 + t2
t4  := e * f
t5  := t3 - t4
x  := t5
```

Compiler Code:

```
   ...
   tx = genExpr();
   ty = genExpr();
   recycle (tx);
   recycle (ty);
   tz = newTemp();
   IR.add(tz,tx,ty);
   ...
```

When to recycle?

In expressions, every temp is used exactly once

So call recycle when the temp is used as an operand.

Source:

```
x := ((a*b) + (c*d)) - (e*f);
```

Before:

```
t1  := a * b
t2  := c * d
t3  := t1 + t2
t4  := e * f
t5  := t3 - t4
x  := t5
```

Compiler Code:

```
   ...
   tx = genExpr();
   ty = genExpr();
   recycle (tx);
   recycle (ty);
   tz = newTemp();
   IR.add(tz,tx,ty);
   ...
```

When to recycle?

In expressions, every temp is used exactly once

So call recycle when the temp is used as an operand.

With Recycling:

```
t1  := a * b
t2  := c * d
t1  := t1 + t2
t2  := e * f
t1  := t1 - t2
x  := t1
```

Recycling Bins

PCAT

Only one kind of temp (4 bytes long)

Other compilers:

Many kind of temps...

byte, word, double

Approach:

Will have a "bin" (i.e., collection) for each kind of temp.

`recycleByteTemp (temp)`

Returns the temp to the recycling bin for "byte" temps

`newByteTemp () → temp`

Check the "byte" recycling bin before creating

```
recycleWordTemp (temp)
newWordTemp () → temp

recycleDoubleTemp (temp)
newDoubleTemp () → temp
```

Arrays in PCAT

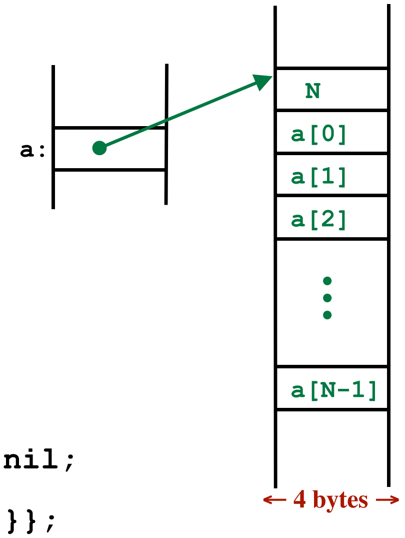
Array of N elements

`a[0], a[1], ... a[N-1]`

All arrays will be stored in the heap.

Will be stored in a block of N+1 words.

The first word will contain the number of elements in the array.



At runtime, we must check for...

"Index out-of-bounds" error

"Uninitialized array" error

```
var a: array of real := nil;
...
a := {{ 1000 of 123.456 }};

... a[i] ...
```

Generating Code to Access "a[i]"

Source:

```
...a[expr]...
```

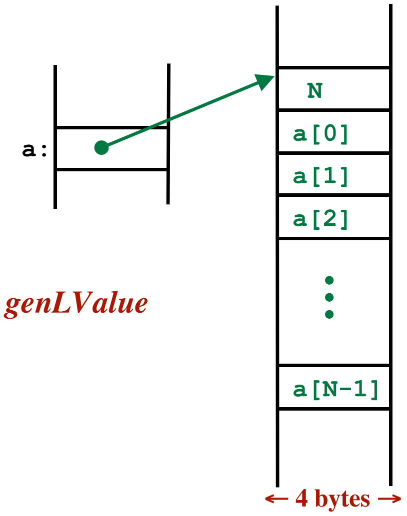
Code Generated:

```
t1 := &a
t2 := *t1
if t2 = 0 goto null_ptr_error
t3 := ...expr...
if t3 < 0 goto bounds_error
t4 := *t2
if t3 >= t4 goto bounds_error
t5 := t3 * 4
t6 := t5 + t2
t7 := t6 + 4
```

genLValue → t1

genExpr → t3

genLValue → t7



Now t7 contains the address of the word in question

Array Representation

How is an array stored in memory?
Where is A[i] stored?

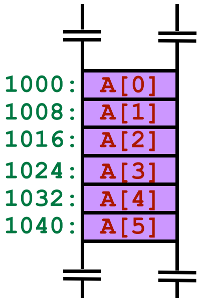
Array Representation

How is an array stored in memory?
Where is A[i] stored?

Assumptions:

- Array starts at A[0]
- No other information (e.g., “size”) stored in the array
- No indirection, no pointers

Example
w = 8 bytes
base = 1000



Array Representation

How is an array stored in memory?
Where is A[i] stored?

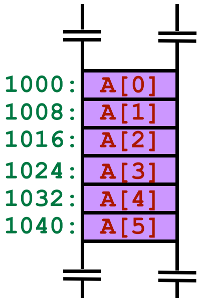
Assumptions:

- Array starts at A[0]
- No other information (e.g., “size”) stored in the array
- No indirection, no pointers

Let:
w = width (in bytes) of each element
base = address of 1st byte of the array

The address of A[i]
base + (i * w)

Example
w = 8 bytes
base = 1000



Array Representation

How is an array stored in memory?
Where is A[i] stored?

Assumptions:

- Array starts at A[0]
- No other information (e.g., “size”) stored in the array
- No indirection, no pointers

Let: **w** = width (in bytes) of each element
base = address of 1st byte of the array

The address of A[i]

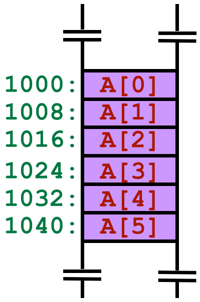
base + (i * w)

Example: A[3]

1000 + (3*8) = 1024

Example

w = 8 bytes
base = 1000



Array Representation

Assumption: Array can start anywhere

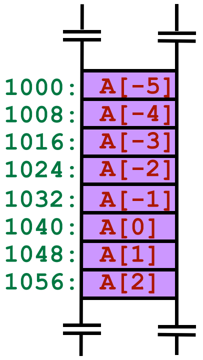
A[-5], A[-4], ..., A[0], ...
B[6], B[7], ...

Array Representation

Assumption: Array can start anywhere
A[-5], A[-4], ..., A[0], ...
B[6], B[7], ...

Let:
w = width of elements
base = starting address of the array
low = smallest legal index (e.g., -5)

Example
w = 8 bytes
base = 1000
low = –5



Array Representation

Assumption: Array can start anywhere
A[-5], A[-4], ..., A[0], ...
B[6], B[7], ...

Let:
w = width of elements
base = starting address of the array
low = smallest legal index (e.g., -5)

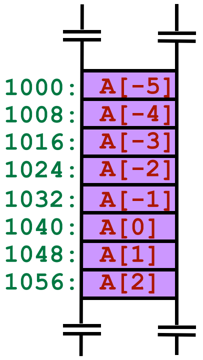
The address of A[i]

Before:
base + (i * w)

Now:
base + ((i - low) * w)

Example: A[2]
1000 + ((2 - (-5)) * 8) = 1056
= 2 * 8 + (1000 - (-5 * 8)) = 16 + 1040 = 1056

Example
w = 8 bytes
base = 1000
low = –5



The Zero-Normalized Base

Address of A[i]:

base + ((i – low) * w)

Rewriting:

i*w + (base – (low * w))

If “base” and “low” are known at compile-time
we can precompute this constant:
base – (low * w)

The Zero-Normalized Base

Address of A[i]:

base + ((i – low) * w)

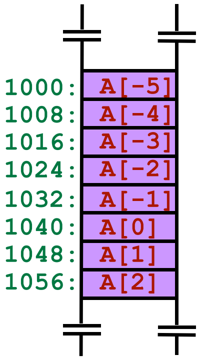
Rewriting:

i*w + (base – (low * w))

If “base” and “low” are known at compile-time
we can precompute this constant:
base – (low * w)
1000 – (-5 * 8)
= 1040

Example

w = 8 bytes
base = 1000
low = –5



The Zero-Normalized Base

Address of A[i]:

base + ((i – low) * w)

Rewriting:

i*w + (base – (low * w))

If "base" and "low" are known at compile-time
we can precompute this constant:
base – (low * w)
1000 – (-5 * 8)
= 1040

This is the address of A[0].

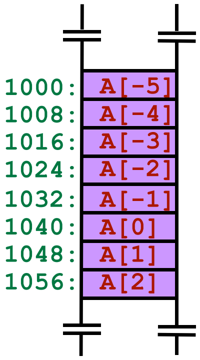
The "***zero-normalized base***"

Address of A[i]:

i*w + *constant*

Example

w = 8 bytes
base = 1000
low = –5



1000: A[-5]
1008: A[-4]
1016: A[-3]
1024: A[-2]
1032: A[-1]
1040: A[0]
1048: A[1]
1056: A[2]

The Zero-Normalized Base

Address of A[i]:

base + ((i – low) * w)

Rewriting:

i*w + (base – (low * w))

If "base" and "low" are known at compile-time
we can precompute this constant:
base – (low * w)
1000 – (-5 * 8)
= 1040

This is the address of A[0].

The "***zero-normalized base***"

Address of A[i]:

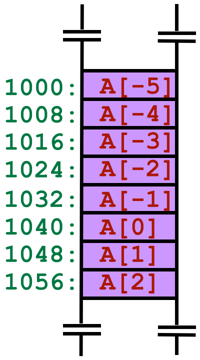
i*w + *constant*

Address of A[2]:

2*8 + 1040
= 1056

Example

w = 8 bytes
base = 1000
low = –5



1000: A[-5]
1008: A[-4]
1016: A[-3]
1024: A[-2]
1032: A[-1]
1040: A[0]
1048: A[1]
1056: A[2]

The Zero-Normalized Base

The "zero-normalized base" works,
even if array does NOT contain A[0].

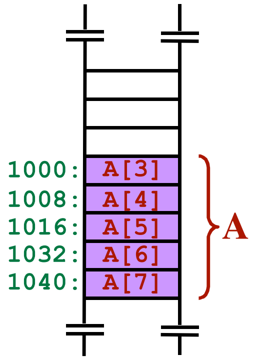
The Zero-Normalized Base

The "zero-normalized base" works,
even if array does NOT contain A[0].

Example:

low = 3
A[3] .. A[7]

Example
w = 8 bytes
base = 1000
low = 3



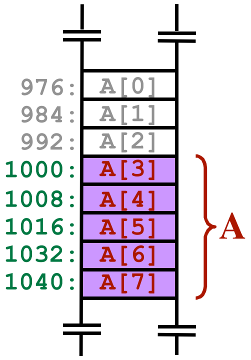
The Zero-Normalized Base

The "zero-normalized base" works,
even if array does NOT contain A[0].

Example:

low = 3
A[3] .. A[7]

Example
w = 8 bytes
base = 1000
low = 3



The Zero-Normalized Base

The "zero-normalized base" works,
even if array does NOT contain A[0].

Example:

low = 3
A[3] .. A[7]

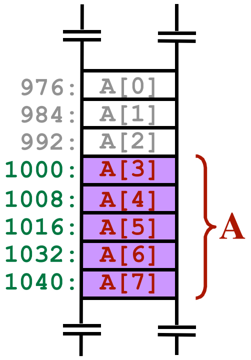
Address of A[i]:

i*w + (base – (low * w))
i*w + *constant*

Zero-Normalized Base:

base – (low * w)
1000 – (3 * 8)
= 976

Example
w = 8 bytes
base = 1000
low = 3



The Zero-Normalized Base

The "zero-normalized base" works,
even if array does NOT contain A[0].

Example:

low = 3
A[3] .. A[7]

Address of A[i]:

i*w + (base – (low * w))
i*w + *constant*

Zero-Normalized Base:

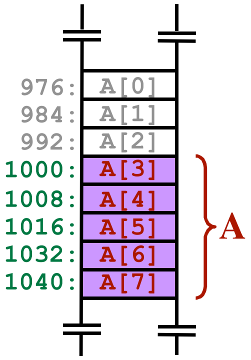
base – (low * w)
1000 – (3 * 8)
= 976

Address of A[5]:

i*w + *constant*
5*8 + 976
=1016

Example

w = 8 bytes
base = 1000
low = 3



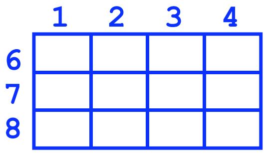
Multi-Dimensional Arrays

Two Dimensional Arrays:

```
var A: array [6..8,1..4] of double;
```

Rows **Columns**

```
...A[i,j]...
```



Three Dimensional Arrays:

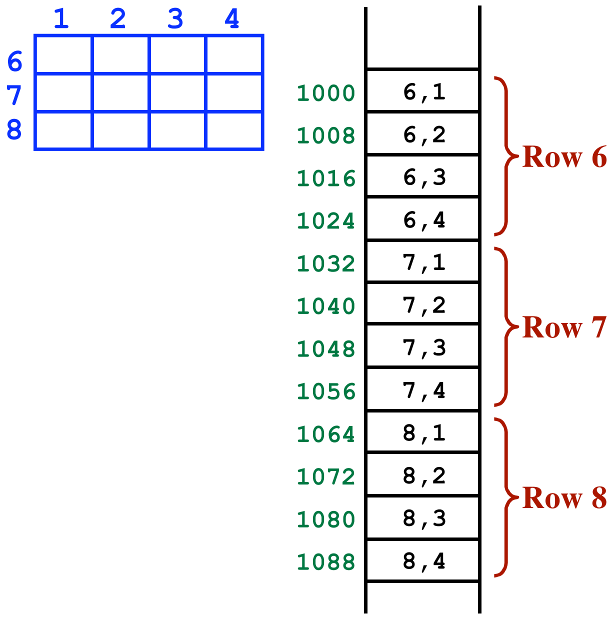
```
var B: array [6..8,1..4,0..9] of double;

...B[i,j,k]...
```

Multi-Dimensional Arrays...

```
var C: array [6..8,1..4, ••• ,0..9] of ...;
```

How do we place the array elements in memory?

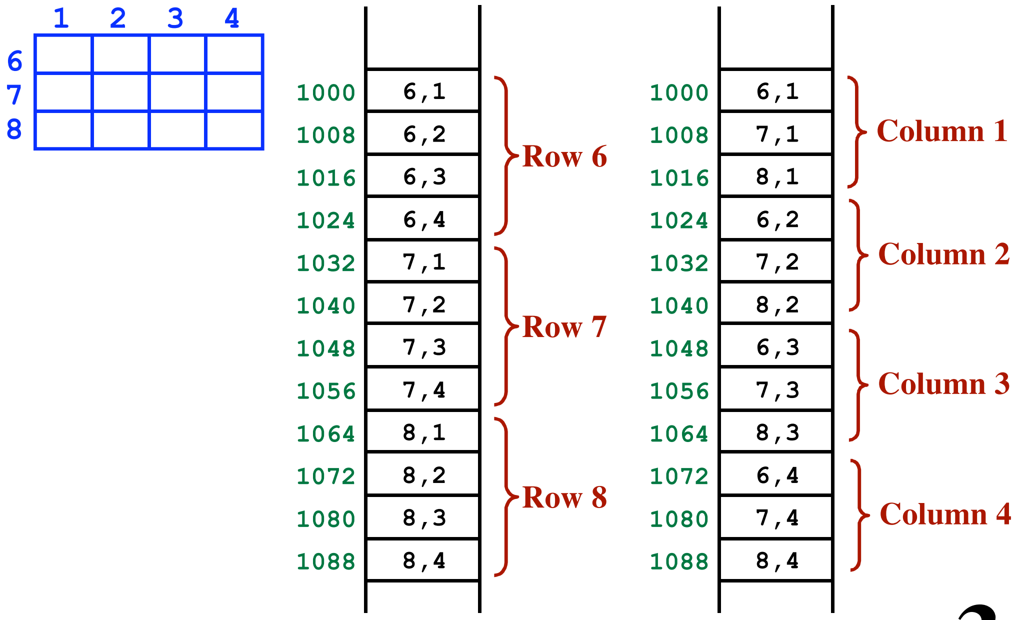
Row-Major Order



1000	6,1	Row 6
1008	6,2	
1016	6,3	
1024	6,4	
1032	7,1	Row 7
1040	7,2	
1048	7,3	
1056	7,4	
1064	8,1	Row 8
1072	8,2	
1080	8,3	
1088	8,4	

Row-Major Order

Column-Major Order

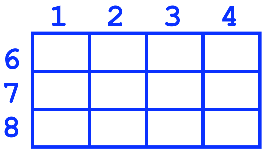


1000	6,1	Row 6
1008	6,2	
1016	6,3	
1024	6,4	
1032	7,1	Row 7
1040	7,2	
1048	7,3	
1056	7,4	
1064	8,1	Row 8
1072	8,2	
1080	8,3	
1088	8,4	

1000	6,1	Column 1
1008	7,1	
1016	8,1	
1024	6,2	Column 2
1032	7,2	
1040	8,2	
1048	6,3	Column 3
1056	7,3	
1064	8,3	
1072	6,4	Column 4
1080	7,4	
1088	8,4	

Row-Major Order

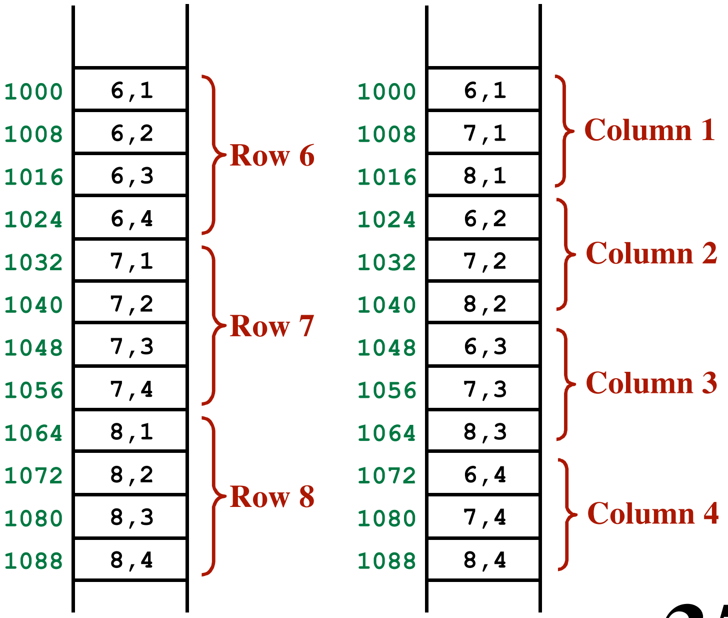
Column-Major Order



Row-Major is most common.
Like decimal numbers
(Last digit varies fastest)

3,687
3,688
3,689
3,690
3,691

<u>"Row-Major Order"</u>
This idea extends to higher dimensions.



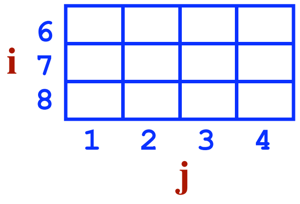
Address	Row-Major Order	
1000	6,1	Row 6
1008	6,2	
1016	6,3	
1024	6,4	
1032	7,1	Row 7
1040	7,2	
1048	7,3	
1056	7,4	
1064	8,1	Row 8
1072	8,2	
1080	8,3	
1088	8,4	

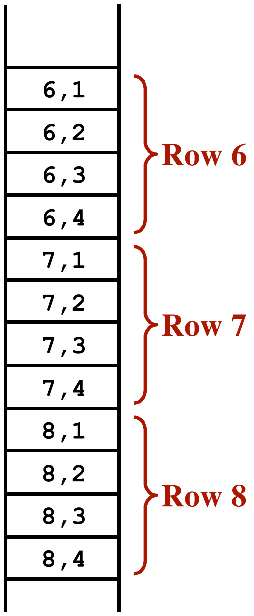
Address	Column-Major Order	
1000	6,1	Column 1
1008	7,1	
1016	8,1	
1024	6,2	Column 2
1032	7,2	
1040	8,2	
1048	6,3	Column 3
1056	7,3	
1064	8,3	
1072	6,4	Column 4
1080	7,4	
1088	8,4	

---

3 Dimensional Arrays, Row-Major Order

`A[6..7, 1..4, 3..7]`

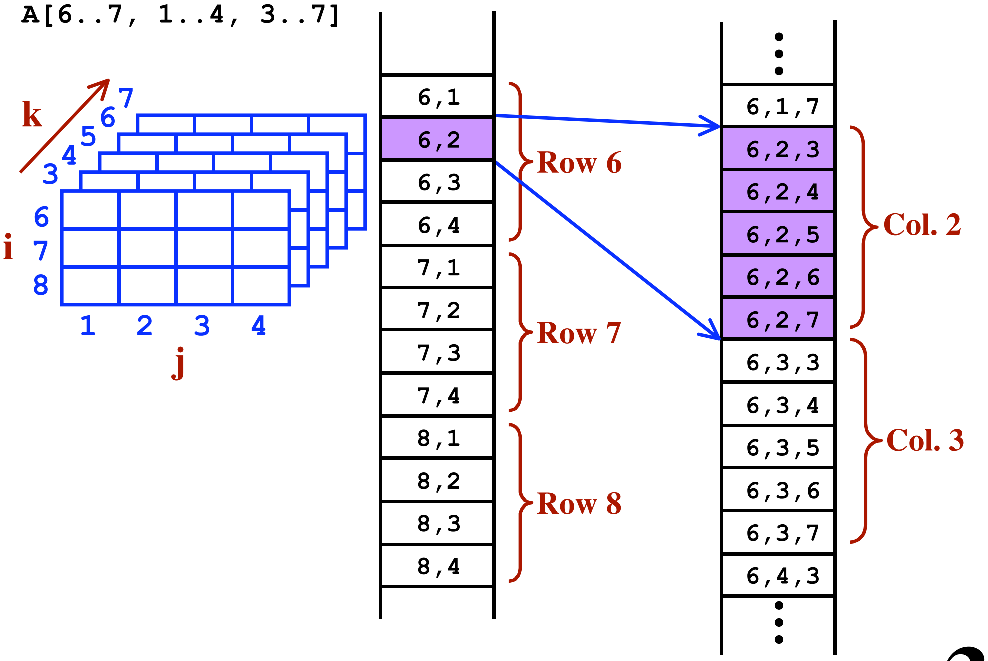




6,1	Row 6
6,2	
6,3	
6,4	
7,1	Row 7
7,2	
7,3	
7,4	
8,1	Row 8
8,2	
8,3	
8,4	

3 Dimensional Arrays, Row-Major Order

`A[6..7, 1..4, 3..7]`



Where is A[i] Stored?

Assumption: Two-Dimensions, Row-Major Order

`var A: array [0..8, 0..4] of real;`

Let: w = width of elements 8
base = starting address of the array 1000

$high_1$ = highest row index `A[0..8, 0..4]`

$high_2$ = highest column index `A[0..8, 0..4]`

Compute:

N_1 = number of rows
= $high_1$ + 1 `8 +1 = 9`

N_2 = number of columns
= $high_2$ + 1 `4 +1 = 5`
= size of each row!

A[i,j]:

base + (i * N_2 + j) * w

Where is A[i] Stored?

Assumption: Two-Dimensions, Row-Major Order

`var A: array [6..8, 1..4] of real;`

Let:

w = width of elements	8
base = starting address of the array	1000
$\textbf{low}_1$ = lowest row index	`A[6..8, 1..4]`
$\textbf{high}_1$ = highest row index	`A[6..8, 1..4]`
$\textbf{low}_2$ = lowest column index	`A[6..8, 1..4]`
$\textbf{high}_2$ = highest column index	`A[6..8, 1..4]`

Compute:

N_1 = number of rows

= $\text{high}_1 - \text{low}_1 + 1$ `8-6+1 = 3`

N_2 = number of columns

= $\text{high}_2 - \text{low}_2 + 1$ `4-1+1 = 4`

= size of each row!

A[i,j]:

$$\text{base} + ((i - \text{low}_1) * N_2 + (j - \text{low}_2)) * w$$

(Repeating...)

A[i,j] is stored at:

$$\text{base} + ((i - \text{low}_1) * N_2 + (j - \text{low}_2)) * w$$

6 operations

Can we compute any of this at compile-time?

(Repeating...)

A[i,j] is stored at:

$$base + ((i - low_1) * N_2 + (j - low_2)) * w$$

6 operations

Can we compute any of this at compile-time?

Rewriting

$$((i * N_2) + j) * w + (base - ((low_1 * N_2) + low_2) * w)$$

- Assume the array bounds are fixed at compile-time.
- Assume the base address is known at compile-time

$$((i * N_2) + j) * w + \underbrace{(base - ((low_1 * N_2) + low_2) * w)}$$

4 operations

The “Zero-Normalized Base”

The address of A[0,0]

Compile-time constant:
Pre-compute it!!!

(Repeating...)

A[i,j] is stored at:

$$base + ((i - low_1) * N_2 + (j - low_2)) * w$$

A[0,0] is stored at:

$$base + ((0 - low_1) * N_2 + (0 - low_2)) * w$$

Rewriting:

$$base + ((-low_1 * N_2) + (-low_2)) * w$$

$$base - ((low_1 * N_2) + low_2) * w$$

$$((i * N_2) + j) * w + \underbrace{(base - ((low_1 * N_2) + low_2) * w)}$$

Compile-time constant:
Pre-compute it!!!

Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: $A\ [\ 0..high_1,\ 0..high_2,\ \ldots,\ 0..high_K]$

Number of dimensions = K

`A[i,j]` is stored at:

$\text{base} + (\ i * N_2 + j\ ) * w$

The general case: $A\ [low_1..high_1,\ low_2..high_2,\ \ldots,\ low_K..high_K]$

`A[i,j]`:

$\text{base} + (\ (i - low_1) * N_2 + (j - low_2)\ ) * w$

Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: $A\ [\ 0..high_1,\ 0..high_2,\ \ldots,\ 0..high_K]$

Number of dimensions = K

`A[i,j]` is stored at:

$\text{base} + (\ i * N_2 + j\ ) * w$

$A[i_1,i_2,i_3]$:

$\text{base} + (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ ) * w$

The general case: $A\ [low_1..high_1,\ low_2..high_2,\ \ldots,\ low_K..high_K]$

`A[i,j]`:

$\text{base} + (\ (i - low_1) * N_2 + (j - low_2)\ ) * w$

$A[i_1,i_2,i_3]$:

$\text{base} + (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3)\ ) * w$

Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: $A\ [\ 0..high_1,\ 0..high_2,\ \ldots,\ 0..high_K]$

Number of dimensions = K

$A[i,j]$ is stored at:

$base + (\ i * N_2 + j\ ) * w$

$A[i_1,i_2,i_3]$:

$base + (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ ) * w$

$A[i_1,i_2,i_3,i_4]$:

$base + (\ (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ ) * N_4 + i_4\ ) * w$

The general case: $A\ [low_1..high_1,\ low_2..high_2,\ \ldots,\ low_K..high_K]$

$A[i,j]$:

$base + (\ (i - low_1) * N_2 + (j - low_2)\ ) * w$

$A[i_1,i_2,i_3]$:

$base + (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3\ )\ ) * w$

$A[i_1,i_2,i_3,i_4]$:

$base + (\ (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3\ )\ ) * N_4 + (i_4 - low_4\ )\ ) * w$

Accessing Multi-Dimensional Arrays

Assume all indexes start at zero: $A\ [\ 0..high_1,\ 0..high_2,\ \ldots,\ 0..high_K]$

Number of dimensions = K

$A[i,j]$ is stored at:

$base + (\ i * N_2 + j\ ) * w$

$A[i_1,i_2,i_3]$:

$base + (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ ) * w$

$A[i_1,i_2,i_3,i_4]$:

$base + (\ (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ ) * N_4 + i_4\ ) * w$

$A[i_1,i_2,i_3,\ldots i_K]$:

$base + (\ \ldots\ (\ (\ i_1 * N_2 + i_2\ ) * N_3 + i_3\ )\ \ldots * N_K + i_K\ ) * w$

The general case: $A\ [low_1..high_1,\ low_2..high_2,\ \ldots,\ low_K..high_K]$

$A[i,j]$:

$base + (\ (i - low_1) * N_2 + (j - low_2)\ ) * w$

$A[i_1,i_2,i_3]$:

$base + (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3\ )\ ) * w$

$A[i_1,i_2,i_3,i_4]$:

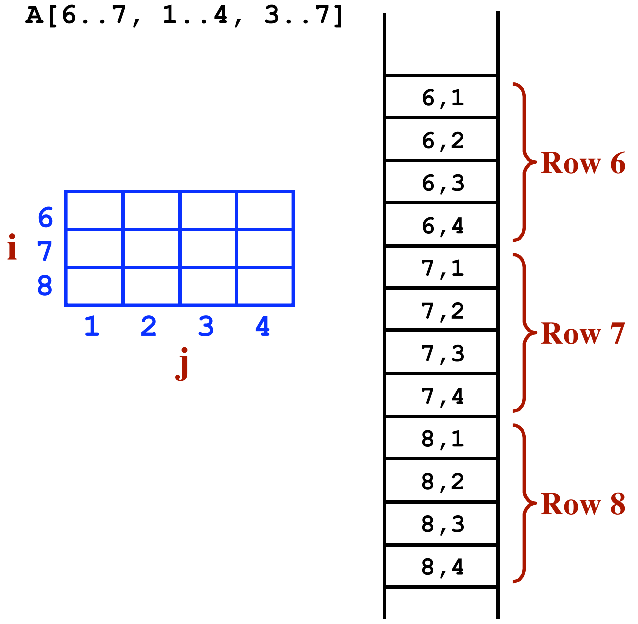
$base + (\ (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3\ )\ ) * N_4 + (i_4 - low_4\ )\ ) * w$

$A[i_1,i_2,\ldots i_K]$:

$base + (\ \ldots\ (\ (\ (i_1 - low_1) * N_2 + (i_2 - low_2)\ ) * N_3 + (i_3 - low_3\ )\ )\ \ldots * N_K + (i_K - low_K\ )\ ) * w$

3 Dimensional Arrays

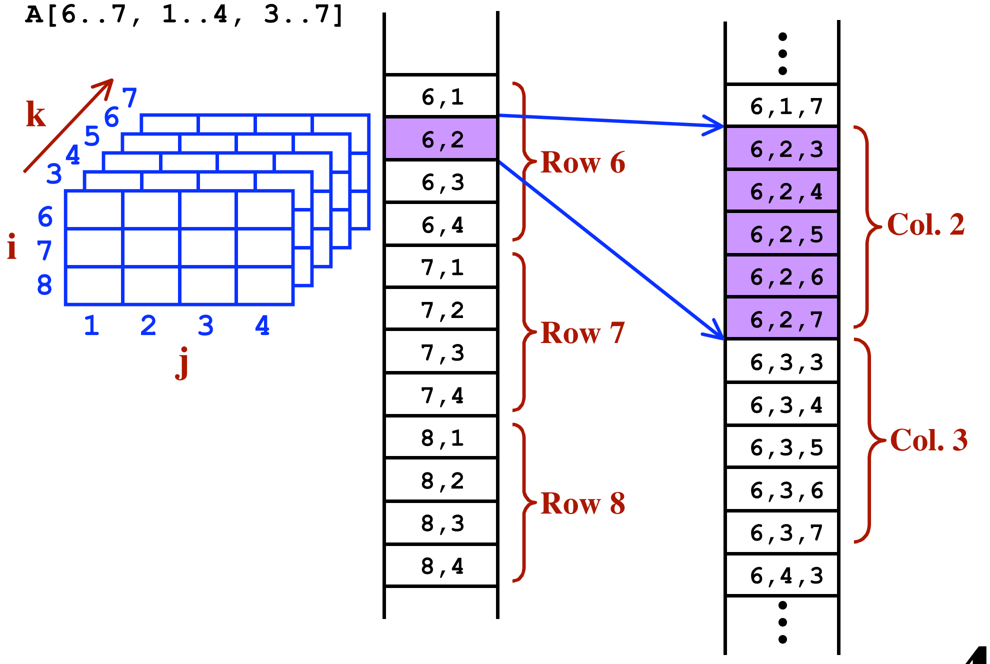
```
A[6..7, 1..4, 3..7]
```



---

3 Dimensional Arrays

```
A[6..7, 1..4, 3..7]
```



Precomputing the Zero-Normalized Base

$A[i_1,i_2,...i_K]$ is stored at:

$$\text{base} + (\ldots (((i_1 - \text{low}_1) * N_2 + (i_2 - \text{low}_2)) * N_3 + (i_3 - \text{low}_3)) \ \ldots$$
$$* N_K + (i_K - \text{low}_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((\text{low}_1 * N_2 + \text{low}_2) * N_3 + \text{low}_3) \ \ldots * N_K + \text{low}_K) * w$$

Precomputing the Zero-Normalized Base

$A[i_1,i_2,...i_K]$ is stored at:

$$\text{base} + (\ldots (((i_1 - \text{low}_1) * N_2 + (i_2 - \text{low}_2)) * N_3 + (i_3 - \text{low}_3)) \ \ldots$$
$$* N_K + (i_K - \text{low}_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((\text{low}_1 * N_2 + \text{low}_2) * N_3 + \text{low}_3) \ \ldots * N_K + \text{low}_K) * w$$

Performing the computation at runtime...

i_1

Precomputing the Zero-Normalized Base

`A[i1,i2,...iK]` is stored at:

$$\text{base} + (\ldots (((i_1 - low_1) * N_2 + (i_2 - low_2)) * N_3 + (i_3 - low_3)) \ldots * N_K + (i_K - low_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((low_1 * N_2 + low_2) * N_3 + low_3) \ldots * N_K + low_K) * w$$

Performing the computation at runtime...

i_1
$i_1 * N_2 + i_2$

Precomputing the Zero-Normalized Base

`A[i1,i2,...iK]` is stored at:

$$\text{base} + (\ldots (((i_1 - low_1) * N_2 + (i_2 - low_2)) * N_3 + (i_3 - low_3)) \ldots * N_K + (i_K - low_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((low_1 * N_2 + low_2) * N_3 + low_3) \ldots * N_K + low_K) * w$$

Performing the computation at runtime...

i_1
$i_1 * N_2 + i_2$
$(i_1 * N_2 + i_2) * N_3 + i_3$

Precomputing the Zero-Normalized Base

$A[i_1,i_2,...i_K]$ **is stored at:**

$$\text{base} + (\ldots (((i_1 - low_1) * N_2 + (i_2 - low_2)) * N_3 + (i_3 - low_3)) \ \ldots$$
$$* N_K + (i_K - low_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((low_1 * N_2 + low_2) * N_3 + low_3) \ \ldots * N_K + low_K) * w$$

Performing the computation at runtime...

$$i_1$$
$$i_1*N_2 + i_2$$
$$(i_1*N_2 + i_2)*N_3 + i_3$$
$$...((i_1*N_2 + i_2)*N_3 + i_3)... *N_K + i_K$$

Precomputing the Zero-Normalized Base

$A[i_1,i_2,...i_K]$ **is stored at:**

$$\text{base} + (\ldots (((i_1 - low_1) * N_2 + (i_2 - low_2)) * N_3 + (i_3 - low_3)) \ \ldots$$
$$* N_K + (i_K - low_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((low_1 * N_2 + low_2) * N_3 + low_3) \ \ldots * N_K + low_K) * w$$

Performing the computation at runtime...

$$i_1$$
$$i_1*N_2 + i_2$$
$$(i_1*N_2 + i_2)*N_3 + i_3$$
$$...((i_1*N_2 + i_2)*N_3 + i_3)... *N_K + i_K$$
$$(...((i_1*N_2 + i_2)*N_3 + i_3)... *N_K + i_K)* w$$

Precomputing the Zero-Normalized Base

$A[i_1, i_2, \ldots i_K]$ is stored at:

$$\text{base} + (\ldots (((i_1 - \text{low}_1) * N_2 + (i_2 - \text{low}_2)) * N_3 + (i_3 - \text{low}_3)) \ldots * N_K + (i_K - \text{low}_K)) * w$$

Factoring out the constant...

$$(\ldots ((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K) * w$$
$$+ \text{base} - (\ldots ((\text{low}_1 * N_2 + \text{low}_2) * N_3 + \text{low}_3) \ldots * N_K + \text{low}_K) * w$$

Performing the computation at runtime...

$$i_1$$
$$i_1 * N_2 + i_2$$
$$(i_1 * N_2 + i_2) * N_3 + i_3$$
$$\ldots((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K$$
$$(\ldots((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K) * w$$
$$(\ldots((i_1 * N_2 + i_2) * N_3 + i_3) \ldots * N_K + i_K) * w + \text{constant}$$

Checking Array Limits

1-Dimensional

- Check i before computation of address

  $\text{low} \le i \le \text{high}$

- Perform the address computation

  $p = \text{base} + (i - \text{low}) * w$

  Check that address is within array

  $\text{base} \le p < (\text{base} + \text{sizeInBytes})$

Checking Array Limits

1-Dimensional

- Check i before computation of address
  $low \le i \le high$
- Perform the address computation
  $p = base + (i - low) * w$
  Check that address is within array
  $base \le p < (base + sizeInBytes)$

Multi-Dimensional

- Check each index individually
  a[i,j,k] $low_1 \le i \le high_1$
  $low_2 \le j \le high_2$
  $low_3 \le k \le high_3$
- Perform the address computation
  p = base + ...
  Check that address is within array
  $base \le p < (base + sizeInBytes)$

Checking Array Limits

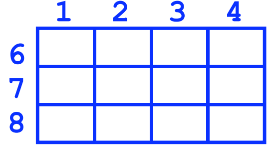
1-Dimensional

- Check i before computation of address
  $low \le i \le high$
- Perform the address computation
  $p = base + (i - low) * w$
  Check that address is within array
  $base \le p < (base + sizeInBytes)$

Multi-Dimensional

- Check each index individually
  a[i,j,k] $low_1 \le i \le high_1$
  $low_2 \le j \le high_2$
  $low_3 \le k \le high_3$
- Perform the address computation
  p = base + ...
  Check that address is within array
  $base \le p < (base + sizeInBytes)$
  Faster, but flawed!

Faster, but flawed



Example: `A[6,10]`
Not in array
Perform address calculation
$base + ((i - low_1)*N_2+(j - low_2))*w$
$base + ((6-6)*4+(10-1))*w$
$= base + (9)*w$
$base + ((8-6)*4+(2-1))*w$
$= base + (9)*w$
The access is still within the array!

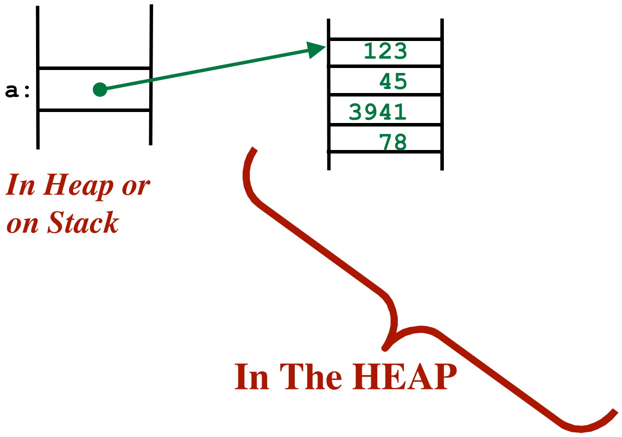
Arrays in PCAT

Always 1 dimensional.
Always start at zero.
Multi-dimensional arrays?

Arrays in PCAT

Always 1 dimensional.
Always start at zero.
Multi-dimensional arrays?

```
var a: array of                         integer;
... a[5]        ...
...   (a[5])          ...
```



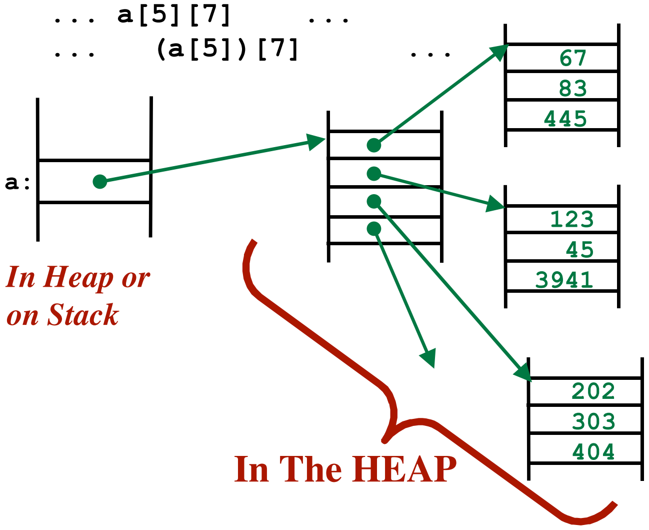
In Heap or on Stack

In The HEAP

Arrays in PCAT

Always 1 dimensional.
Always start at zero.
Multi-dimensional arrays?

```
var a: array of array of          integer;
... a[5][7]     ...
...    (a[5])[7]       ...
```



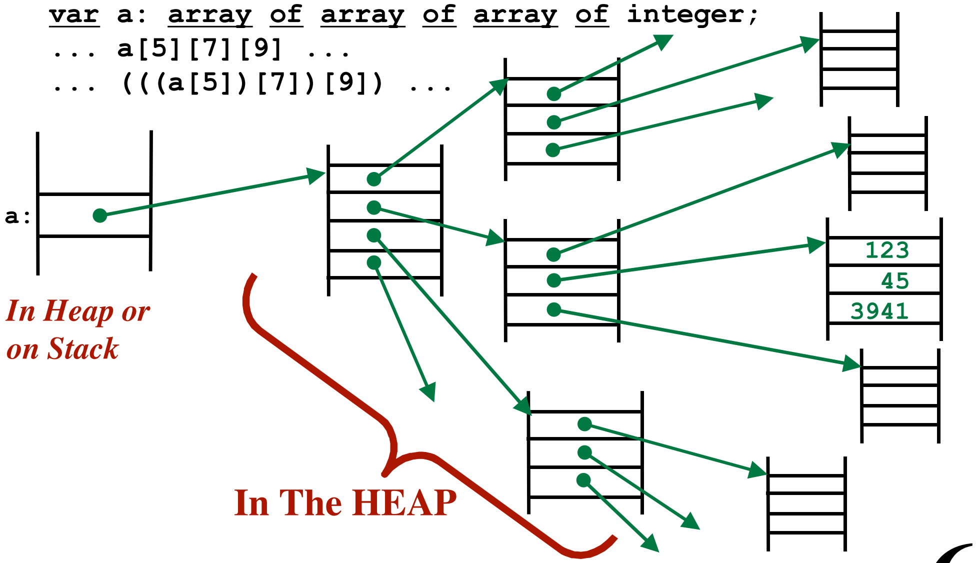
In Heap or on Stack

In The HEAP

Arrays in PCAT

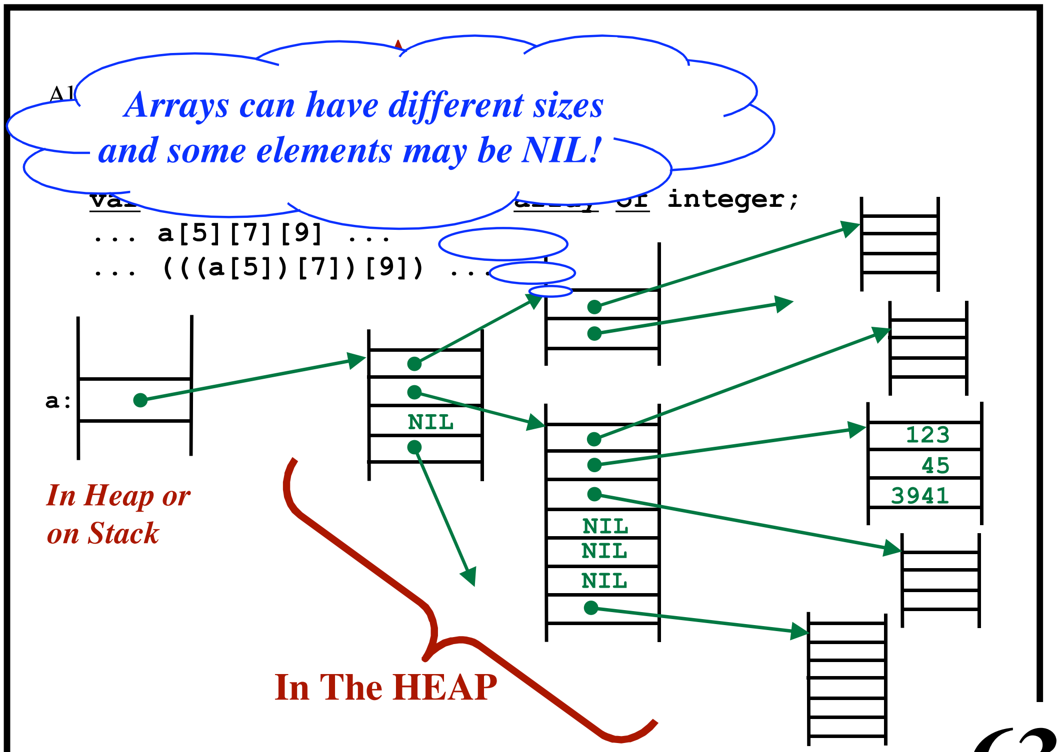
Always 1 dimensional.
Always start at zero.
Multi-dimensional arrays?

```
var a: array of array of array of integer;
... a[5][7][9]  ...
... (((a[5])[7])[9]) ...
```



In Heap or on Stack

In The HEAP



Arrays can have different sizes and some elements may be NIL!

```
Al
var                      array of integer;
... a[5][7][9] ...
... (((a[5])[7])[9]) ...
```

In Heap or on Stack

In The HEAP

AKA: "Case Statements"

Switch Statements

```
switch expr
   case value$_1$: Stmt-List$_1$
   case value$_2$: Stmt-List$_2$
   ...
   case value$_N$: Stmt-List$_N$
   default: Stmt-List$_{N+1}$
endSwitch
```

In C/C++/Java

Stmt-List$_i$ will fall through to Stmt-List$_{i+1}$

Must use "break"

Switch Statements

switch expr
  case value_1: Stmt-List_1
  case value_2: Stmt-List_2
  . . .
  case value_N: Stmt-List_N
  default: Stmt-List_{N+1}
endSwitch

AKA: "Case Statements"

"Switch Expression"

"Case Arms"
"Case Clauses"

"Default Case" is optional
(If missing, fall through)

In C/C++/Java
  Stmt-List_i will fall through to Stmt-List_{i+1}
    Must use "break"

The "value"s must be constants (i.e., statically known)
  "*Statically Executable*"
  "*Statically Evaluatable*"
  "*Statically Computable*"

```
static final int MAX = 100;
...
  case 43*(17+MAX): Stmt-List_i
```

Switch Statements

Three Implementation Techniques:

(1) Sequence of N expilicit tests

(2) Precompute a table of N entries
Generate code to quickly search this table

(3) Direct Jump Table
Generate a vector of N addresses
Use the switch value as an offset into this table
Execute a "Jump-Indirect" through the table

(1) Sequence of N Tests

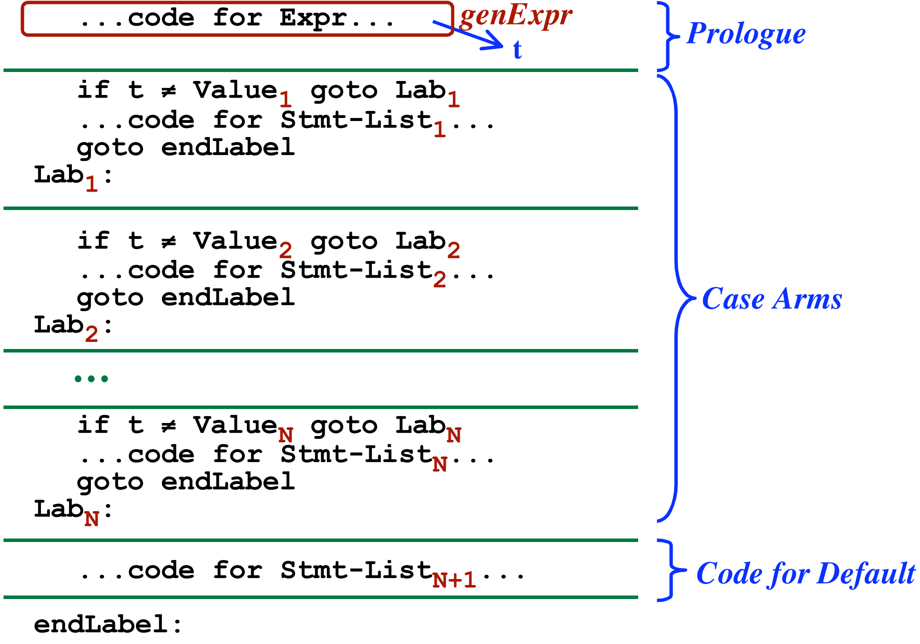
Treat the switch statement exactly like a sequence if-then-else statements.

```
switch expr
  case value_1: Stmt-List_1
  case value_2: Stmt-List_2
  ...
  case value_N: Stmt-List_N
  default: Stmt-List_N+1
endSwitch
```

```
t := expr
if t = value_1 then
  Stmt-List_1
elseIf t = value_2 then
  Stmt-List_2
...
elseIf t = value_N then
  Stmt-List_N
else
  Stmt-List_N+1
endIf
```

---

(1) Sequence of N Tests



```
   ...code for Expr...
   if t ≠ Value_1 goto Lab_1
   ...code for Stmt-List_1...
   goto endLabel
Lab_1:
   if t ≠ Value_2 goto Lab_2
   ...code for Stmt-List_2...
   goto endLabel
Lab_2:
   ...
   if t ≠ Value_N goto Lab_N
   ...code for Stmt-List_N...
   goto endLabel
Lab_N:
   ...code for Stmt-List_N+1...
endLabel:
```

(1) Sequence of N Tests

```
   ...code for Expr...
```

genExpr → t

Prologue

```
   if t ≠ Value1 goto Lab1
   ...code for Stmt-List1...
   goto endLabel
Lab1:

   if t ≠ Value2 goto Lab2
   ...code for Stmt-List2...
   goto endLabel
Lab2:

   •••

   if t ≠ ValueN goto LabN
   ...code for Stmt-ListN...
   goto endLabel
LabN:
```

Ca

This code is easy to generate.
But, it is difficult to recognize it as a “switch statement.”
We want to do that during the Optimization phase

```
   ...code for Stmt-ListN+1...
```

Code for Default

```
endLabel:
```

---

(1) Sequence of N Tests

```
   ...code for Expr...
```

genExpr → t

```
   if t = Value1 goto Lab1
   if t = Value2 goto Lab2
   ...
   if t = ValueN goto LabN
   goto LabN+1
```

Prologue

```
Lab1:
   ...code for Stmt-List1...
   goto endLabel
Lab2:
   ...code for Stmt-List2...
   goto endLabel
   •••
LabN:
   ...code for Stmt-ListN...
   goto endLabel
```

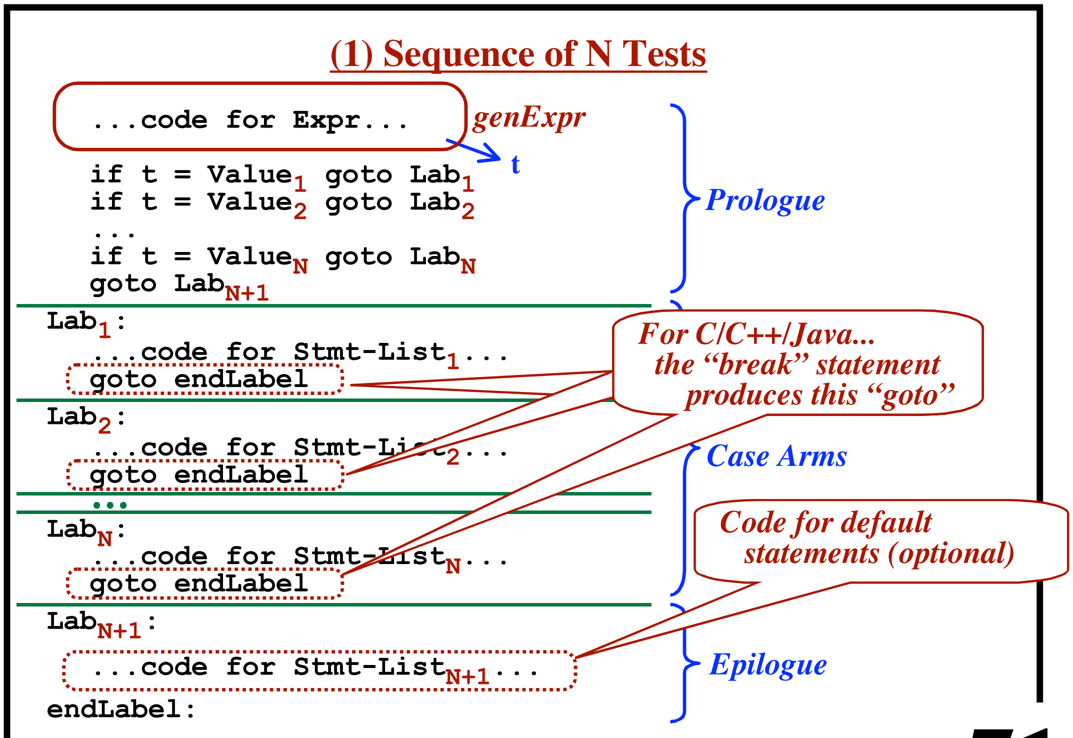
Case Arms

```
LabN+1:
   ...code for Stmt-ListN+1...
endLabel:
```

Epilogue

Code for default statements (optional)

(1) Sequence of N Tests



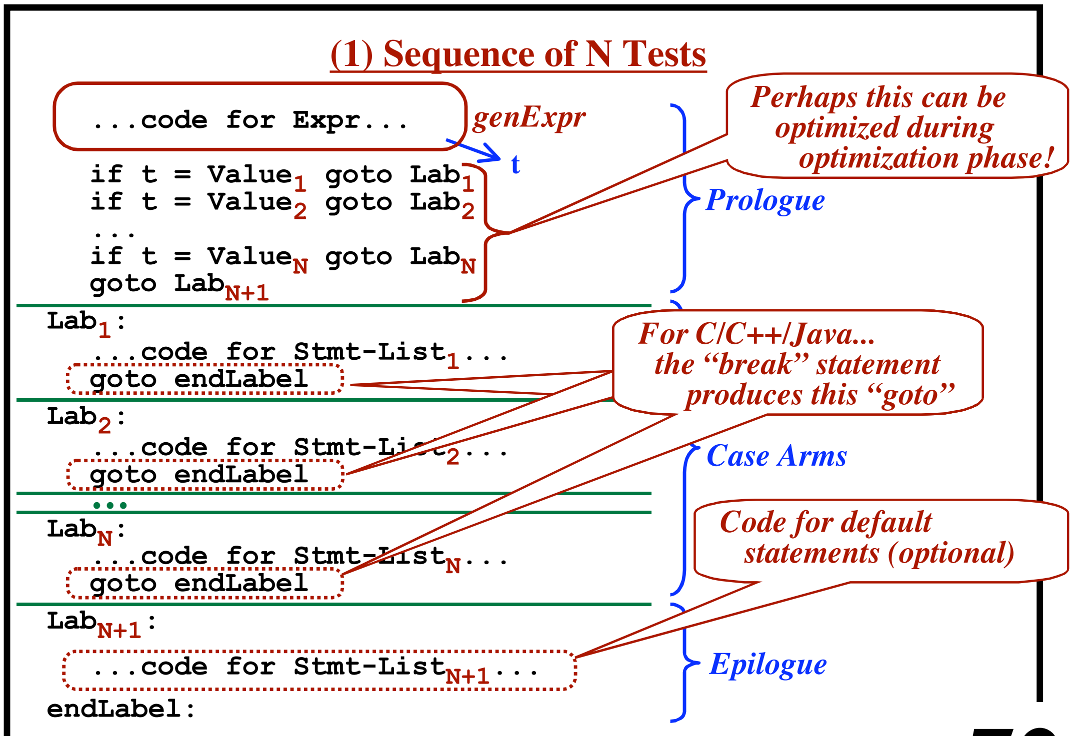
```
   ...code for Expr...
    if t = Value1 goto Lab1
    if t = Value2 goto Lab2
    ...
    if t = ValueN goto LabN
    goto LabN+1
Lab1:
   ...code for Stmt-List1...
    goto endLabel
Lab2:
   ...code for Stmt-List2...
    goto endLabel
    •••
LabN:
   ...code for Stmt-ListN...
    goto endLabel
LabN+1:
   ...code for Stmt-ListN+1...
endLabel:
```

For C/C++/Java... the "break" statement produces this "goto"

Code for default statements (optional)

---

(1) Sequence of N Tests



```
   ...code for Expr...
    if t = Value1 goto Lab1
    if t = Value2 goto Lab2
    ...
    if t = ValueN goto LabN
    goto LabN+1
Lab1:
   ...code for Stmt-List1...
    goto endLabel
Lab2:
   ...code for Stmt-List2...
    goto endLabel
    •••
LabN:
   ...code for Stmt-ListN...
    goto endLabel
LabN+1:
   ...code for Stmt-ListN+1...
endLabel:
```

Perhaps this can be optimized during optimization phase!

For C/C++/Java... the "break" statement produces this "goto"

Code for default statements (optional)

(1) Sequence of N Tests

To generate code for a Switch statement...

Prologue

```
    Create a new label "endLabel"
    Generate code to evaluate the expr into temporary "t"
    Run through all case arms
        Compute $\text{Value}_i$
        Create a new label "$\text{Lab}_i$"
        Generate "if t=$\text{Value}_i$ goto $\text{Lab}_i$"
        Remember each label
    Generate "goto $\text{Lab}_{N+1}$"
```

Run through all case arms a second time...

```
    To generate code for
        case $\text{Value}_i$: $\text{Stmt-List}_i$
    Generate label "$\text{Lab}_i$:"
    Generate the code for $\text{Stmt-List}_i$
    Generate label "goto endLabel"
```

Epilogue

```
    Generate label "$\text{Lab}_{N+1}$:"
    Generate code for default statements (optional)
    Generate label "endLabel:"
```

Previously:

```
if t = $\text{Value}_1$ goto $\text{Lab}_1$
if t = $\text{Value}_2$ goto $\text{Lab}_2$
...
if t = $\text{Value}_N$ goto $\text{Lab}_N$
goto $\text{Lab}_{N+1}$
```

Ideas for IR Instructions:

```
switch  t,$\text{Lab}_{N+1}$
case    $\text{Value}_1$,$\text{Lab}_1$
case    $\text{Value}_2$,$\text{Lab}_2$
...
case    $\text{Value}_N$,$\text{Lab}_N$
```

Maybe we can generate fast code to search for the right case...

(2) Precompute a Table and Search It

Approach:
    Build a table in static storage
        Each entry contains a Value and a Label
    Generate code to search the table
        Upon finding the matching value
            The code will jump to the stored label

VTAB	LTAB
Value$_1$	Lab$_1$
Value$_2$	Lab$_2$
⋮	⋮
Value$_N$	Lab$_N$

Implementation of the table

```
Table:
       .word   7903
       .word   Lab_43
       .word   4067
       .word   Lab_44
         •••
       .word   8989
       .word   Lab_45
```

Code to do a linear search

```
    p := &Table
Loop:
    if Table[p] = t
      goto *(Table[p+4])
    p := p + 8
    goto Loop
```

---

(2) Precompute a Table and Search It

Dealing with the default case?
    The Switch expression value does not match any “Case Clause”

Idea: Use a *“Sentinel”*

- Add one extra entry to the table.
    Use the label of the default code.
    Will fill in value at runtime.
- Generate code to store the value of “t” into the last entry.
    During the search...
        If no values match, we’ll match the last entry!

VTAB	LTAB
Value$_1$	Lab$_1$
Value$_2$	Lab$_2$
⋮	⋮
Value$_N$	Lab$_N$
0	Lab$_{N+1}$

Use a Hash Table

Linear search is slow... Use a hash-based search!

At compile-time

We know the number of values.
Determine the optimal hash table size.
Determine the hash function.
Build the table (pre-compile it)

Generate code to:

- Compute the switch expression
- Compute Hash(expr)
- Search the table until...
  - Match is found
  - Null entry is found
- Jump indirect through the table

Hash Search Example

Source:

```
switch x+17
  case 2004: Stmt-List1
  case 5006: Stmt-List2
  case 4003: Stmt-List3
  case 7009: Stmt-List4
  case 6006: Stmt-List5
  case 3001: Stmt-List6
  default: Stmt-ListN+1
endSwitch
```

Hash Search Example

Source:

```
switch x+17
   case 2004: Stmt-List1
   case 5006: Stmt-List2
   case 4003: Stmt-List3
   case 7009: Stmt-List4
   case 6006: Stmt-List5
   case 3001: Stmt-List6
   default: Stmt-ListN+1
endSwitch
```

Number of cases: 6
Hash Table size: 10
Hash Function: hash(v) = v mod 10

2004 → 4
5006 → 6
4003 → 3
7009 → 9
6006 → 6
3001 → 1

Hash Search Example

Source:

```
switch x+17
   case 2004: Stmt-List1
   case 5006: Stmt-List2
   case 4003: Stmt-List3
   case 7009: Stmt-List4
   case 6006: Stmt-List5
   case 3001: Stmt-List6
   default: Stmt-ListN+1
endSwitch
```

Build Table

	VTAB	LTAB
0	0	NULL
1	3001	Lab$_6$
2	0	NULL
3	4003	Lab$_3$
4	2004	Lab$_1$
5	0	NULL
6	5006	Lab$_2$
7	6006	Lab$_5$
8	0	NULL
9	7009	Lab$_4$

Number of cases: 6
Hash Table size: 10
Hash Function: hash(v) = v mod 10

2004 → 4
5006 → 6
4003 → 3
7009 → 9
6006 → 6
3001 → 1

Hash Search Example

Source:

switch x+17
  case 2004: Stmt-List$_1$
  case 5006: Stmt-List$_2$
  case 4003: Stmt-List$_3$
  case 7009: Stmt-List$_4$
  case 6006: Stmt-List$_5$
  case 3001: Stmt-List$_6$
  default: Stmt-List$_{N+1}$
endSwitch

Number of cases: 6
Hash Table size: 10
Hash Function: hash(v) = v mod 10

2004 → 4
5006 → 6
4003 → 3
7009 → 9
6006 → 6
3001 → 1

Build Table

	VTAB	LTAB
0	0	NULL
1	3001	Lab$_6$
2	0	NULL
3	4003	Lab$_3$
4	2004	Lab$_1$
5	0	NULL
6	5006	Lab$_2$
7	6006	Lab$_5$
8	0	NULL
9	7009	Lab$_4$

At runtime...

Compute `t := x+17`
Compute Hash(`t`)
Search the table
  If VTAB[p] matches... Jump-Indirect
  If LTAB[p] = null, jump to default Lab$_{N+1}$

---

(3) Direct-Jump Table

switch x+17
  case 4: Stmt-List$_1$
  case 6: Stmt-List$_2$
  case 3: Stmt-List$_3$
  case 9: Stmt-List$_4$
  case 7: Stmt-List$_5$
  case 2: Stmt-List$_6$
  default: Stmt-List$_{N+1}$
endSwitch

Determine the range of values
Build a table this size.
  Each table entry will contain a label

	LTAB
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

(3) Direct-Jump Table

```
switch x+17
   case 4: Stmt-List$_1$
   case 6: Stmt-List$_2$
   case 3: Stmt-List$_3$
   case 9: Stmt-List$_4$
   case 7: Stmt-List$_5$
   case 2: Stmt-List$_6$
   default: Stmt-List$_{N+1}$
endSwitch
```

	LTAB
0	
1	
2	Lab$_6$
3	Lab$_3$
4	Lab$_1$
5	
6	Lab$_2$
7	Lab$_5$
8	
9	Lab$_4$

Determine the range of values
Build a table this size.
    Each table entry will contain a label

---

(3) Direct-Jump Table

```
switch x+17
   case 4: Stmt-List$_1$
   case 6: Stmt-List$_2$
   case 3: Stmt-List$_3$
   case 9: Stmt-List$_4$
   case 7: Stmt-List$_5$
   case 2: Stmt-List$_6$
   default: Stmt-List$_{N+1}$
endSwitch
```

	LTAB
0	Lab$_{N+1}$
1	Lab$_{N+1}$
2	Lab$_6$
3	Lab$_3$
4	Lab$_1$
5	Lab$_{N+1}$
6	Lab$_2$
7	Lab$_5$
8	Lab$_{N+1}$
9	Lab$_4$

Determine the range of values
Build a table this size.
    Each table entry will contain a label
Unused table entries?
    Fill with **Lab**$_{N+1}$

(3) Direct-Jump Table

switch x+17
  case 4: Stmt-List$_1$
  case 6: Stmt-List$_2$
  case 3: Stmt-List$_3$
  case 9: Stmt-List$_4$
  case 7: Stmt-List$_5$
  case 2: Stmt-List$_6$
  default: Stmt-List$_{N+1}$
endSwitch

	LTAB
0	Lab$_{N+1}$
1	Lab$_{N+1}$
2	Lab$_6$
3	Lab$_3$
4	Lab$_1$
5	Lab$_{N+1}$
6	Lab$_2$
7	Lab$_5$
8	Lab$_{N+1}$
9	Lab$_4$

Determine the range of values
Build a table this size.
  Each table entry will contain a label
Unused table entries?
  Fill with **Lab**$_{N+1}$
Generate code to...
  Compute the switch expression
  Use t as an index into the table
  Perform Indirect-Jump through the table

(3) Direct-Jump Table

switch x+17
  case 4: Stmt-List$_1$
  case 6: Stmt-List$_2$
  case 3: Stmt-List$_3$
  case 9: Stmt-List$_4$
  case 7: Stmt-List$_5$
  case 2: Stmt-List$_6$
  default: Stmt-List$_{N+1}$
endSwitch

	LTAB
0	Lab$_{N+1}$
1	Lab$_{N+1}$
2	Lab$_6$
3	Lab$_3$
4	Lab$_1$
5	Lab$_{N+1}$
6	Lab$_2$
7	Lab$_5$
8	Lab$_{N+1}$
9	Lab$_4$

Determine the range of values
Build a table this size.
  Each table entry will contain a label
Unused table entries?
  Fill with **Lab**$_{N+1}$
Generate code to...
  Compute the switch expression
  Use t as an index into the table
  Perform Indirect-Jump through the table

This approach only works when the range of values is “small”.
  Otherwise, LTAB is too large.
NOTE: The values can be “shifted”
  35,002 .. 35,009 ⇒ 0..7

Switch Table Implementation - Recap

(1) Sequence of Explicit Tests

(2) Table plus Search

- Linear Search
- Hash-based search
- Other search (e.g., Binary Search)

(3) Direct Jump Table

- Very Fast!
  - ...but, can only use if range is small

Which method is best?

Which method(s) does "gcc" use?
...Under what circumstances?

Nice to know how smart/dumb your compiler is...
...so you can write efficient code!