Structure References

Most programming languages provide a mechanism to aggregate data together into a **structure**.

Example In C, we could use the following structure to create lists of integers.

```
struct node {
   int value;
   struct node *next;
};
struct node NILNode = { 0, (struct node*) 0 };
struct node *NIL = &NILNode;
```

The introduction of structures and pointers creates two distinct problems for the compiler: **anonymous values** and **structure layout**.

Understanding Structure Layouts

To emit code for structure references, the compiler needs to know

- **starting address** of the structure instance
- **offset** and **length** of each structure element

The compiler can build a separate **table of structure layouts** to maintain this information.

- textual name for each structure element
- its offset within the structure
- its source-language data type

Understanding Structure Layouts

Structure Layout Table

Name	**Length**	**1^{st} Element**
`node`	8	•
...	...	•

Structure Element Table

Name	**Length**	**Offset**	**Type**	**Next**
`node.value`	4	0	`int`	•
`node.next`	4	4	`struct node *`	•
...	...	...	...	...

Understanding Structure Layouts

Note Entries in the **structure element table** (shown on the previous slide) use fully qualified names to avoid conflicts due to reuse of a name in several distinct structures.

With this information, the compiler can easily generate code for structure references.

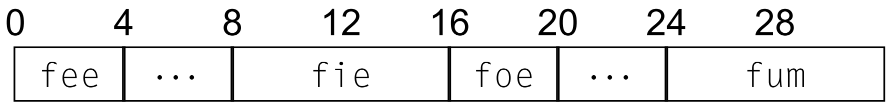
Example The compiler might translate the reference `p1->next`, for a pointer to node `p1`, into the following ILOC code.

$$
\begin{array}{llcll}
\texttt{loadI} & \texttt{4} & \Rightarrow & \texttt{r}_1 & \ldots\text{offset of } \texttt{next} \\
\texttt{loadAO} & \texttt{r}_{\texttt{p1}}\texttt{, r}_1 & \Rightarrow & \texttt{r}_2 & \ldots\text{value of } \texttt{p1->next}
\end{array}
$$

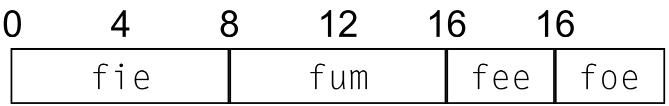
Understanding Structure Layouts

In laying out a structure and assigning offsets to its elements, the compiler must obey the **alignment rules** of the target architecture.

```
struct example {
   int fee;
   double fie;
   int foe;
   double fum;
};
```



Elements in Declaration Order



Elements Ordered by Alignment

Note Whether the compiler can reorder structure elements in memory arbitrarily depends whether the source language definition exposes the layout of a structure to the user.

Array of Structures

Many programming languages allow the user to declare an **array of structures**. The compiler has two options how to layout such data in memory.

1. a structure-valued array with multiple copies of the structure layout
2. a structure composed of elements that are arrays

Note The second option is only possible if the programmer cannot take the address of a structure-valued element of an array.

Depending on how the surrounding code accesses the data, these two strategies may have strikingly **different performance** on a system with cache memory.

Unions and Runtime Tags

Many languages support structures with **multiple**, **data-dependent** interpretations.

Unions and variants present one additional complication

- the possibility exists that element names are **not unique**
- compiler must resolve each reference to a unique offset and type in the runtime object

```
struct n1 {
    int kind;
    int value;
};
```

```
struct n2 {
    int kind;
    float value;
};
```

```
union one {
    struct n1 inode;
    struct n2 fnode;
} u1;
```

This problem has a **linguistic solution**: the programming language can force the programmer to make the reference unambiguous.

Unions and Runtime Tags

Example To reference an integer value, the programmer specifies `u1.inode.value`, whereas to reference a floating-point value, the programmer specifies `u1.fnode.value`.

The **fully qualified name** resolves any ambiguity.

The same mechanism is used to disambiguate references to implicitly defined structure-valued union elements.

Example Union `two` has the same properties as `one`.

Again, the programmer has to specify a fully qualified name such as `u2.inode.value` to reference an integer value and `u2.fnode.value` to reference a floating-point value.

```
union two {
   struct {
      int kind;
      int value;
   } inode;
   struct {
      int kind;
      float value;
   } fnode;
} u2;
```

Unions and Runtime Tags

As an alternative to linguistic solutions, some systems rely on **runtime discrimination**.

- each variant in the union has a field (tag) that distinguishes it from all other variants
- compiler emits code to check the value of the tag field and handle object accordingly

```
type shapeKind = (square, rectangle, circle);
     shape = record
                centerx : integer;
                centery : integer;
                case kind : shapeKind of
                   square : (side : integer);
                   rectangle : (length, height : integer);
                   circle : (radius : integer);
             end;
```

Pointers and Anonymous Values

A C program creates an instance of a structure in one of two ways

- declare a structure instance, *e.g.*, `NILNode` in the earlier example
- allocate a structure instance explicitly

Example For a variable declared as a pointer to `node`, the allocation looks as follows.

```
fee = (struct node*) malloc(sizeof(node));
```

The only access to this new node is through the pointer `fee`. Thus, we think of it as an **anonymous value**, since it has no permanent name.

Because the only name for an anonymous value is a pointer, the compiler cannot easily determine if two pointer references specify the **same** memory location.

Pointers and Anonymous Values

Example Consider the following code fragment.

```
1  p1 = (node*) malloc(sizeof(node));
2  p2 = (node*) malloc(sizeof(node));
3  if (...) {
4     p3 = p1;
5  } else {
6     p3 = p2;
7  }
8  p1->value = ...;
9  p3->value = ...;
10 ... = p1->value;
```

The first two lines create anonymous `nodes`.

Line 8 writes through `p1`, while line 9 writes through `p3`.

On line 9, `p3` can refer to either the node allocated in line 1 or in line 2.

Finally, line 10 references `p1->value`. The compiler cannot easily determine at this point which value is used.

The uncertainty introduced by pointers **prevents** the compiler from keeping values used in pointer-based references in registers.

Pointers and Anonymous Values

Anonymous objects further complicate the problem because they introduce an **unbounded set** of objects to track.

A similar effect occurs with arrays: the compiler would need to perform in-depth analysis of array subscripts to determine whether two array references overlap.

While challenging, the problem of disambiguating array references is easier than the problem of disambiguating pointer references.

Analysis to disambiguate pointer references and array references is a **major** source of potential improvement in program performance.

- interprocedural data-flow analysis to discover all objects each pointer can point to
- data-dependence analysis to understand the patterns of array references

Control-Flow Constructs

As the compiler generates code, it can build up **basic blocks** by simply aggregating consecutive, unlabeled, non-control-flow operations.

To tie a set of blocks together so that they form a procedure, the compiler must insert code that implements the **control-flow operations** of the source program.

- build a **control-flow graph** and use it for analysis, optimization, and code generation
- code for control-flow constructs resides **at or near the end** of each basic block

While many different syntactic conventions have been used to express control flow, the number of underlying concepts is **small**.

Conditional Execution

Most programming languages provide some version of an `if-then-else` construct.

```
if expr
   then statement1
   else statement2
statement3
```

The compiler needs to generate code that

- evaluates *expr* and branches to *statement*$_1$ or *statement*$_2$, accordingly
- implements the two statements and at the end jumps to *statement*$_3$

As we already saw on Slide 355, the compiler has many options for implementing `if-then-else` constructs.

Conditional Execution

With trivial `then` and `else` parts the primary consideration for the compiler is matching the expression evaluation to the underlying hardware.

As the `then` and `else` parts grow, the importance of efficient execution inside the `then` and `else` parts begins to **outweigh** the cost of executing the controlling expression.

Example On a machine that supports **predicated execution**, using predicates for large blocks in the `then` and `else` parts can waste execution cycles.

With large blocks of code under both the `then` and `else` parts, the cost of unexecuted instructions may outweigh the overhead of using a conditional branch.

Unit 1	Unit 2
comparison $\Rightarrow$ r_1	
(r_1) op_1	$(\neg r_1)$ op_{11}
(r_1) op_2	$(\neg r_1)$ op_{12}
(r_1) op_3	$(\neg r_1)$ op_{13}
(r_1) op_4	$(\neg r_1)$ op_{14}
(r_1) op_5	$(\neg r_1)$ op_{15}
(r_1) op_6	$(\neg r_1)$ op_{16}
(r_1) op_7	$(\neg r_1)$ op_{17}
(r_1) op_8	$(\neg r_1)$ op_{18}
(r_1) op_9	$(\neg r_1)$ op_{19}
(r_1) op_{10}	$(\neg r_1)$ op_{20}

	Unit 1	Unit 2
	compare and branch	
L_1:	op_1	op_2
	op_3	op_4
	op_5	op_6
	op_7	op_8
	op_9	op_{10}
	jumpI $\rightarrow$ L_3	
L_2:	op_{11}	op_{12}
	op_{13}	op_{14}
	op_{15}	op_{16}
	op_{17}	op_{18}
	op_{19}	op_{20}
	jumpI $\rightarrow$ L_3	
L_3:	nop	

Conditional Execution

Choosing between branching and predication to implement an `if-then-else` requires some care. Several issues need to be considered.

1. **Expected frequency of execution**: if one path executes significantly more often, techniques that speed up its execution (*e.g.*, branch prediction, speculative execution, and instruction reordering) may produce faster code
2. **Uneven amounts of code**: if one path contains many more instructions, this may weigh against predication or for a combination of predication and branching
3. **Control flow inside the construct**: if either path contains nontrivial control flow, in particular nested `if`-statements, then predication may be a poor choice

Conditional Execution

Predication in the Real World

Nvidia's CUDA parallel computing platform 32 threads are executed together as a **warp**. All threads in a warp execute the **same** instruction at the **same** time.

In order to support **different** threads doing **different** things, CUDA has predicated instructions that are executed only if a logical flag is `true`.

```
if (x < 0.0)
   r = -1.0;
else
   r = sqrt(x);
```

```
     p = (x < 0.0);
 p:  r = -1.0;
!p:  r = sqrt(x);
```

This is called **warp divergence**. The performance of the code generated by the `nvcc` compiler is impacted by the same issues as outlined on the previous slide.

Loops and Iteration

Most programming languages include `loop` constructs to perform iteration.

The first FORTRAN compiler introduced the `do` loop to perform iteration. Today, loops are found in many forms. For the most part, they have a similar structure.

Example Consider the C `for` loop, which has three controlling expressions.

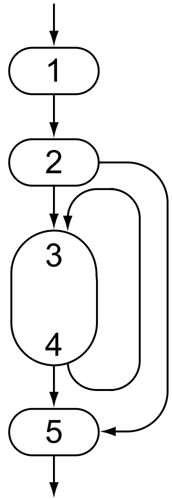
- e_1 performs initialization
- e_2 evaluates to a Boolean and governs execution of the loop
- e_3 executes at the end of each iteration and, potentially, updates the values used in e_2

```
for (e1; e2; e3) {
    loop body
}
```

Loops and Iteration

The following schema shows how the compiler might lay out the code.

```
for (e1; e2; e3) {
    loop body
}
```



Step	Action
1	evaluate e_1
2	if ($\neg e_2$) then goto 5
3	*loop body*
4	evaluate e_3 if (e_2) then goto 5
5	*code following loop*

We will use this basic schema to explain the implementation of several kinds of loops.

Loops and Iteration

If the loop body consists of a single basic block, *i.e.*, contains no other control flow, then the loop that results from this schema has **an initial branch** plus **one branch per iteration**.

The compiler can hide the latency of this branch in several ways

- **branch hint prefix**: mark the branch in Step 4 as likely to be taken
- **delay slot**: move instructions from loop body into the branch delay slot(s)
- **loop unrolling**: if number of iterations is predictable, combine multiple iterations

Delay Slot

Some computer architectures feature **delay slots** to mask the latency of time-consuming instructions, such as loads and branches. A delay slot is an instruction that gets executed without the effects of a preceding instruction.

Loops and Iteration

Branch Prediction in Intel x86-64

In the past, Intel's ISA supported static branch prediction prefixes (`2EH` and `3EH`), a feature that has been abandoned since.

Modern Intel CPUs use a Branch Target Buffer (BTB) to manage branching history dynamically. If this information is not available, the CPU uses a simple heuristic:

- conditional forward branches are predicted as **not being taken**
- conditional backward branches are predicted as **being taken**

This heuristic fits well with the basic schema that we will use to implement loops.

For Loops

To map a `for` loop into code, the compiler follows the general schema from before.

```
for (i = 1; i <= 100; i++) {
    loop body
}
next statement
```

	`loadI`	`1`	$\Rightarrow$	r_i	…Step 1
	`loadI`	`100`	$\Rightarrow$	r_1	…Step 2
	`cmp_GT`	r_i, r_1	$\Rightarrow$	r_2	
	`cbr`	r_2	$\rightarrow$	L_1, L_2	
L_1:	*loop body*				…Step 3
	`addI`	r_i, 1	$\Rightarrow$	r_i	…Step 4
	`cmp_LE`	r_i, r_1	$\Rightarrow$	r_3	
	`cbr`	r_3	$\rightarrow$	L_1, L_2	
L_2:	*next statement*				…Step 5

The compiler can also shape the loop to have only **one copy** of the test. In this form, Step 4 evaluates e_3 and then jumps to Step 2, *i.e.*, replace `cmp_LE` and `cbr` with `jumpI`.

For Loops

Overall, this form of the loop is **one operation smaller** than the two-test form.

However, it creates a **two-block loop** for even the simplest loops, and it **lengthens the path** through the loop by at least one operation.

	`loadI`	`1`	$\Rightarrow$	r_i	...Step 1
	`loadI`	`100`	$\Rightarrow$	r_1	...Step 2
L_1:	`cmp_GT`	r_i, r_1	$\Rightarrow$	r_2	
	`cbr`	r_2	$\rightarrow$	L_1, L_2	
	loop body				...Step 3
	`addI`	r_i, `1`	$\Rightarrow$	r_i	...Step 4
	`jumpI`		$\rightarrow$	L_1	
L_2:	*next statement*				...Step 5

This more compact loop form is only appropriate if **code size** is a serious consideration.

While Loops

A `while` loop can also be implemented with the loop schema. Since it has no initialization, the code is even more compact.

```
while (x < y) {
    loop body
}
next statement
```

	`cmp_LT`	r_x, r_y	$\Rightarrow$	r_1	...Step 2
	`cbr`	r_1	$\rightarrow$	L_1, L_2	
L_1:	*loop body*				...Step 3
	`cmp_LT`	r_x, r_y	$\Rightarrow$	r_2	...Step 4
	`cbr`	r_2	$\rightarrow$	L_1, L_2	
L_2:	*next statement*				...Step 5

Replicating the test in Step 4 creates the possibility of a loop with a **single basic block**.

Until Loops

An `until` loop iterates as long as the controlling expression is `false`.

It checks the controlling expression **after** each iteration. Thus, it always enters the loop and performs at least one iteration and produces a particularly simple loop structure.

```
{
    loop body
} until (x < y)
next statement
```

L_1:	*loop body*				…Step 3
	`cmp_LT`	r_x, r_y	$\Rightarrow$	r_1	…Step 4
	`cbr`	r_1	$\rightarrow$	L_1, L_2	
L_2:	*next statement*				…Step 5

Note The `do` loop known from C, C++, and Java is similar to the `until` loop with the difference that it iterates as long as the controlling expression is `true`.

Break and Exit

Several languages provide a **structured way** to exit a control-flow construct.

In a loop, `break` transfers control to the first statement following the loop. For nested loops, a `break` typically exits the innermost loop.

Some languages, *e.g.*, Ada and Java, use labels to control which enclosing constructs will exited by the `break` statement.

C also uses `break` in `switch` statements to transfer control to the statement that follows the `switch` statement.

All of these actions have simple implementations

- each loop and each case statement ends with a label for the statement that follows it
- a `break` can be implemented as an immediate jump to that label

Skip and Continue

Some languages include a `skip` or `continue` statement that jumps to the next iteration of a loop.

This construct can be implemented in two ways

- **immediate jump** to the code that reevaluates the controlling expression, tests its value, and branches accordingly
- **insert a copy** of the evaluation, test, and branch at the point where the skip occurs

Case Statements

Many programming languages include some variant of a case statement.

The basic strategy is straightforward

1. evaluate the controlling expression
2. branch to the selected case
3. execute the code for that case

Steps 1 and 3 are well understood. The complex part of case-statement implementation lies in choosing an efficient method to locate the designated case.

No single method works well for all case statements. We examine three strategies: a **linear search**, a **binary search**, and a **computed address**.

Linear Search

The simplest way to locate the appropriate case is to treat the case statement as the specification for a nested set of `if-then-else` statements.

```
switch(e) {
    case 0:    block_0;
               break;
    case 1:    block_1;
               break;
    case 3:    block_3;
               break;
    default:   block_d;
               break;
}
```

```
t_1 ← e
if (t_1 = 0)
    then block_0
    else if (t_1 = 1)
        then block_1
        else if (t_1 = 3)
            then block_3
            else block_d
```

The compiler should order cases according to estimated execution frequency.

Binary Search

As the number of cases rises, the efficiency of linear search becomes a problem.

If the compiler can impose an order on the case labels, it can use **binary search** to obtain a logarithmic search rather than a linear one.

- build a compact ordered table of case labels and corresponding branch labels
- use binary search to discover a matching case label, or the absence of a match
- finally, either branch to the corresponding label or to the default case.

Binary Search

```
switch(e) {
    case 0:     block0;
                break;
    case 15:    block15;
                break;
    case 23:    block23;
                break;
    ...
    case 99:    block99;
                break;
    default:    blockd;
                break;
}
```

```
t1 ← e
down ← 0          ...lower bound
up ← 10           ...upper bound + 1
while (down + 1 < up) {
    middle ← (up + down) ÷ 2
    if (Value[middle] ≤ t1)
        then down ← middle
        else up ← middle
}
if (Value[down] = t1)
    then jump to Label[down]
    else jump to LBd
```

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

Computing the Address Directly

If the case labels form a compact set, the compiler can do better than binary search.

In this case, the compiler can build a compact vector, or **jump table**, that contains the block labels, and find the appropriate label by index into the table.

- for a dense label set, this scheme generates compact and efficient code
- the cost is small and constant: a brief calculation, a memory reference, and a `jump`
- if a few holes exist in the label set, they can be filled with the label for the default case

Computing the Address Directly

```
switch(e) {
   case 0:    block0;
              break;
   case 1:    block1;
              break;
   case 2:    block2;
              break;
   ...
   case 9:    block9;
              break;
   default:   blockd;
              break;
}
```

$t_1 \leftarrow e$
if $(t_1 < 0$ or $t_1 > 9)$
    then *jump to* LB_d
    else
        $t_2 \leftarrow @Table + t_1 \times 4$
        $t_3 \leftarrow memory(t_2)$
        jump to t_3

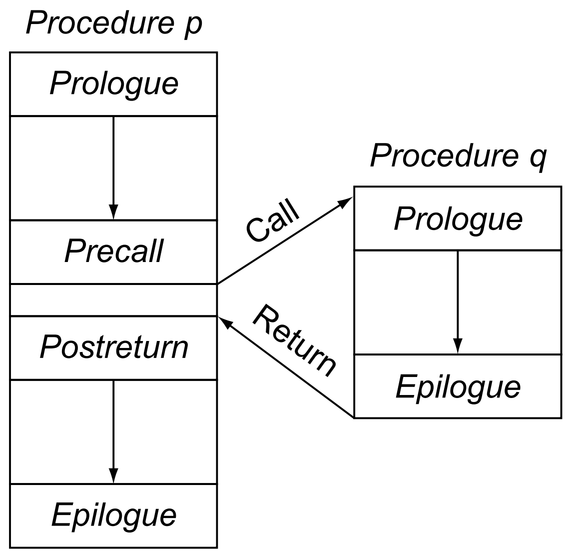
Label
LB_0
LB_1
LB_2
LB_3
LB_4
LB_5
LB_6
LB_7
LB_8
LB_9

Procedure Calls

The implementation of procedure calls is, for the most part, straightforward.

Recall: a procedure call consists of
- a **precall** and a **postreturn** sequence in the caller
- a **prologue** and an **epilogue** sequence in the callee

In the following, we focus on issues that affect the compiler's ability to generate **efficient**, **compact**, and **consistent** code for procedure calls.



As a general rule, moving operations from the precall and postreturn sequences into the prologue and epilogue sequences should **reduce the overall size** of the final code.

Evaluating Actual Parameters

When it builds the precall sequence, the compiler must emit code to evaluate the actual parameters to the call. The compiler treats each actual parameter as an expression.

- **call-by-value parameters**: evaluate the expression and store its value in location designated for that parameter, *i.e.*, a register or callee's AR
- **call-by-reference parameter**: evaluate the parameter to an address and store it in the location designated for that parameter

Note If a call-by-reference parameter has no storage location, then the compiler may need to allocate space for that value so that it has an address to pass to the callee.

The compiler should use a consistent **evaluation order** for parameters, *i.e.*, either left to right or right to left, as evaluating parameters might have side effects.

Saving and Restoring Registers

As both the cost of memory operations and the number of registers have risen, the cost of **saving and restoring registers** has increased to the point that it needs careful attention.

1. **Using multi-register memory operations**: many ISAs support doubleword and quadword load and store operations for adjacent registers.
2. **Using a library routine**: replace the sequence of individual memory operations with a call to a compiler-supplied save or restore routine
3. **Combining responsibilities**: caller and callee pass a value back and forth that specifies which registers each must save