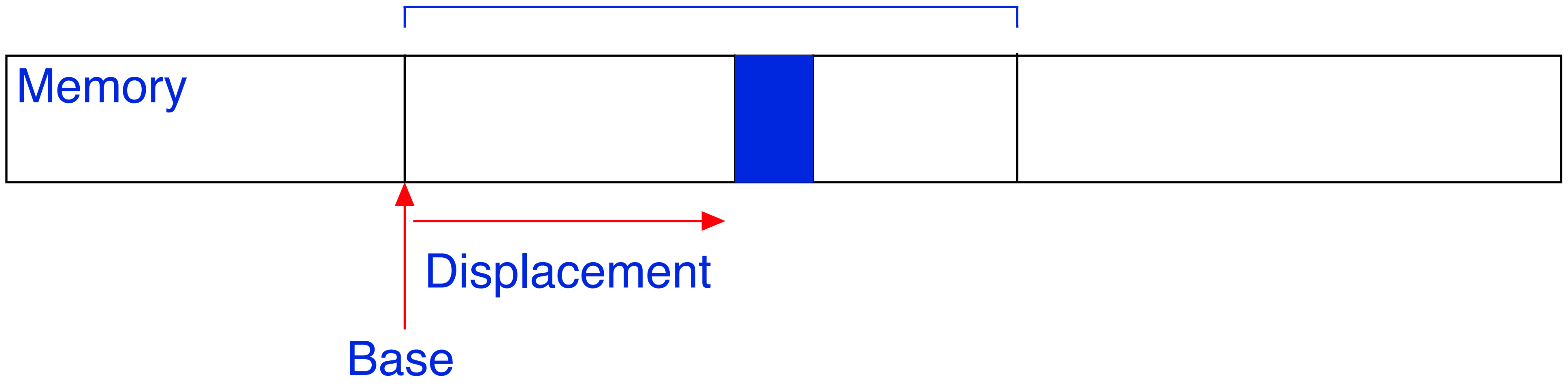
Storage Allocation

- The usual method of storage allocation is based around locating each variable using a *base* and a *displacement*
- When thought of in the abstract machine, the base represents a *lexical level*, and the displacement is the *ordinal number*
- When thought of on the target machine, the base is usually a *pointer register*, and the displacement is a *memory offset*
- The Storage Allocator's job is to map one to the other
- The base-displacement model has been formalized as *data descriptors*



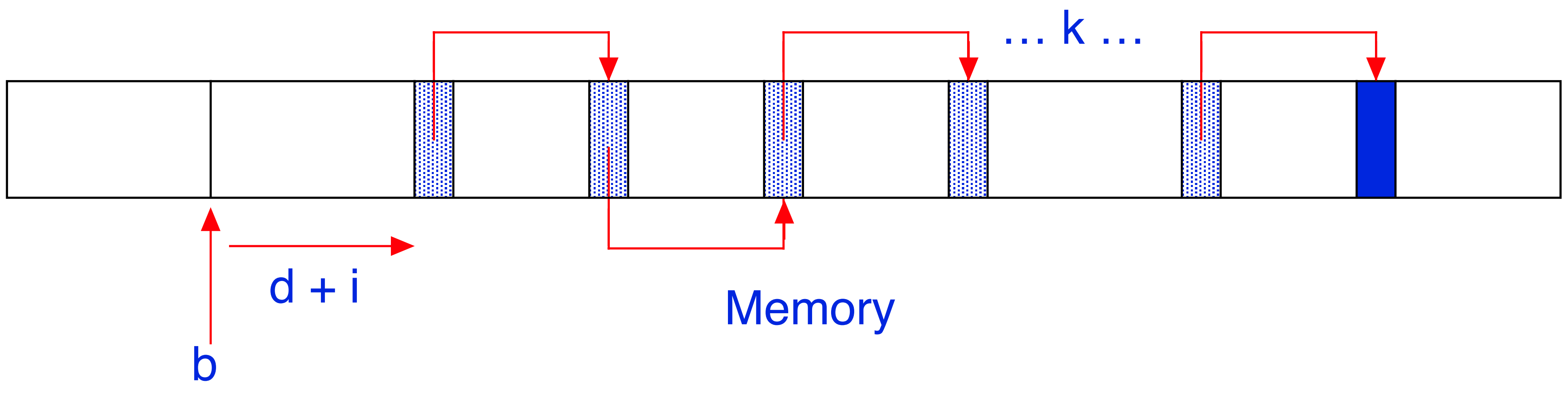
Data Descriptors

- *Data descriptors* provide an abstract representation of the addressing required by most modern languages - this permits much of the code generation phase to be independent of the source language

- The data items used in the program are compiled by the Storage Allocator to data descriptors and the Code Generator then translates the data descriptors to the target architecture

- A *data descriptor* consists of a *base* (b), a *displacement* (d), an *index* (i) and a number of indirections (k), in the form:

$@^{k}\texttt{b.d.i}$



Data Descriptors

$$@^{k}b.d.i$$

- The *base* (b) is a *register*, *null* (i.e. no base) or a *lexical level* - if it is a lexical level, the lexical level has a data descriptor of its own that gives the value of the base (i.e. its Display entry)

- The *displacement* (d) is a positive or negative *compile time constant*

- The *index* (i) is an *index register* or *null*

- The *level of indirection* (k) is a compile time constant that tells the number of times the value computed from the base, displacement and index must be dereferenced

- The sum of the base, displacement and index is called the *numeral* of the data descriptor

$$\mathtt{numeral}(@^{k}b.d.i) = b + d + i$$

Data Descriptors $@^k b.d.i$

- The *value* represented by the data described by the data descriptor is:

If $k = 0$, then the value of the data descriptor is the numeral of the data descriptor

```
value(@⁰b.d.i) = numeral(@⁰ b.d.i) = b + d + i
```

If $k = 1$, then the value of the data descriptor is the value of the memory location indexed by the numeral

```
value(@¹b.d.i) = Memory[numeral(@⁰b.d.i]
               = Memory[b + d + i]
```

If $k = 2$, then the value is the value of the memory location indexed by the value of the memory location indexed by the numeral

```
value (@²b.d.i)
        = Memory[Memory[numeral(@⁰ b.d.i)]]
        = Memory[Memory[b + d + i]]
```

... and so on

Data Descriptor Examples

- By convention, zero indirections and *null* bases and indexes are omitted when writing data descriptors, for example:

Full form	Usual form	Meaning
`@`^{0} `null.5.null`	`5`	`constant 5`
`@`^{0} `r3.0.null`	`r3`	`value of r3`
`@`^{1} `r2.5.null`	`@r2.5`	`Mem[r2+5]`
`@`^{2} `r1.44.null`	`@@r1.44`	`Mem[Mem[r1+44]]`
`@`^{1} `r1.0.r3`	`@r1.0.r3`	`Mem[r1+r3]`

Variables and Data Descriptors

- Data descriptors can be used to represent (LL,ON) addressing of variables in a programming language

```
var w: int                          @LL0.0
var x: boolean                      @LL0.2

procedure Q (
    a: int                          @LL1.0
    var b: boolean)                 @@LL1.2          <– note @@
  const c := 42                     @⁰null.42 = 42
  type R:
    record
      f:real                        @LL2.0
      g : boolean                   @LL2.4
    end record
  var y:r                           @LL1.4
  register var Z: int               @⁰r3.0 = r3    <– not std
  …                                                    Pascal
end Q
```

Target Architectures and Data Descriptors

- Data descriptors can also be used to describe the addressing modes of target computers

PDP-11	Addressing mode	Data Descriptor	
immediate	\$d	d	
absolute	d	@d	*(note: error in text)*
register	r_n	r_n	
register deferred	*r	@r	
index mode	d(r)	@r.d	
index deferred	*d(r)	@@r.d	

IBM/360	Addressing mode	Data Descriptor	
immediate	d	d	
register	r_n	r_n	
storage	d(b)	@b.d	*(b is a base reg)*
indexed	d(b,i)	@b.d.i	

Target Architectures and Data Descriptors

- Few languages (or compilers that implement them) require more than two levels of indirection, and virtually no target architectures support more (some don't support even that many), so we normally limit our data descriptors to *k* <= 2

- In the rare case where more than two levels of indirection are needed, (or an offset is needed in the middle of an indirection chain) a temporary register is used to reduce to two levels or less

- Example:

```
@@@b.d.i = @(@@b.d.i)
         = @t          where t = value(@@b.d.i)
```

- So in practice, to implement 3 levels of indirec tion, we generate extra *addressing code* to move the value of *@@b.d.i* into a temporary register *t*, and replace the data descriptor with *@t*

Addressing Modes Table

- For purposes of data and addressing, we can characterize the target machine by the set of data descriptors it supports

	Machine Addressing Mode			
Data Descriptor	IBM 360	PDP-11	Sparc	
d	d	$d	d	
@d	d(r0)	*$d	-	
@@d	-	-	-	
b	b	b	b	
@b	0(b)	*b	[b+0]	
@@b	-	*0(b)	-	
b.d	-	-	-	
@b.d	d(b)	d(b)	[b+d]	
@@b.d	-	-	-	
@b.d.i	d(b,i)	-	[b+i]	if (d = 0)
@@b.d.i	-	-	-	

Missing Addressing Modes

- What happens if a data descriptor is needed for the data structures of a programming language, but the target machine does not have a corresponding addressing mode?

- Missing addressing modes must be simulated by emitting code sequences (*addressing code*) to transform the data descriptor to a data descriptor that can be directly represented as an addressing mode of the machine.

- Example:
  The Sun Sparc does not have any mode for @@b.d, but Pascal needs that mode for reference parameters (i.e., parameter is address of argument variable in memory), so generate:

```
mov   b,t                  where t is a free register
add   t,d,t
```

  then use @t as the new data descriptor for the parameter

Storage Allocation - Size and Alignment

- In order to implement Storage Allocation of variables on a target machine, we need two properties for each data type:

  - its *size* (in bytes or other memory units), and

  - its *alignment* (what addressing multiple it must start on)

- *Size* constraints come from the *language*, which may specify the range of values required for a type, and the *machine*, which may limit the instructions that can be used to certain sizes of operands (e.g, 32-bit, 64-bit, etc.)

- *Alignment* is a restriction on the multiple of bytes or memory units that the address of variables of a data type must be aligned on - for example, the address of an integer on most machines must be divisible by 4, since integers must be aligned on a 4 byte boundary to be fetched from memory in one piece

- Alignment restrictions may also come from the language, but (more likely) they come from the hardware structure of the machine

Storage Allocation - Size and Alignment

- We begin by enumerating the size and alignment restrictions of each *primitive* type of the language, and then infer the size and alignment of structures such as arrays and records from the sizes and alignments of their components

- We can therefore summarize everything we need to know about a machine for storage allocation in a simple table of the sizes and alignments of the primitive types of the language

	PDP11		**Sparc**	
Type	Size	Align	Size	Align
`char`	1	1	1	1
`boolean`	1	1	1	1
`integer`	2	2	4	4
`real`	4	2	8	4
`pointer`	2	2	4	4

Alignment Costs

- Some architectures do not place any restrictions on alignment - for example, the Motorola *MC68000* has no alignment restrictions (any type can be aligned on any byte in memory)

- However - there is almost always a performance penalty for accessing unaligned data - for example, the *MC68000* accesses integers twice as fast if they are aligned on a two byte boundary

- RISC architectures, such as the *Sun Sparc*, require data types to be aligned, and will crash with a hardware error if they are not

- Adjacent types in records may involve wasted space ("holes") - for example if a record has two adjacent fields, one a byte, the second a four byte aligned integer, then three bytes are wasted

- In an array of these records, 3 bytes *per element* would be wasted, which can amount to a lot in a large array

- Depending on the language, the compiler may or may not be permitted to reorder record fields to minimize wasted space (e.g., Pascal yes, C no)

Storage Allocation Algorithm

- Given the sizes and alignments of the primitive types, we can use one simple recursive algorithm to do all storage allocation for any machine
- Variables in a language can only be: primitive types, records and arrays, and combinations of these
- If we treat each lexical level (procedure scope, class or module body, etc.) as a record, then we only need allocation for these three and we're done

Primitive Types:

```
size := SizeTable[primitiveType]
alignment := AlignmentTable[primitiveType]
```

Arrays (of anything):

```
size := (upper bound - lower bound + 1)
             * size(element type)
alignment := alignment(element type)
```

Storage Allocation Algorithm (cont'd)

Records:

```
sizeSoFar := 0
alignmentSoFar := 0

for each field
   displacement(field) :=
      roundup(sizeSoFar, alignment(field))
   sizeSoFar :=
      displacement(field) + size(field)
   alignmentSoFar :=
      max(alignmentSoFar, alignment(field))
end for

alignment := alignmentSoFar
size := roundup(sizeSoFar, alignment)
```

where *roundup(x,y)* rounds up *x* to a multiple of *y*

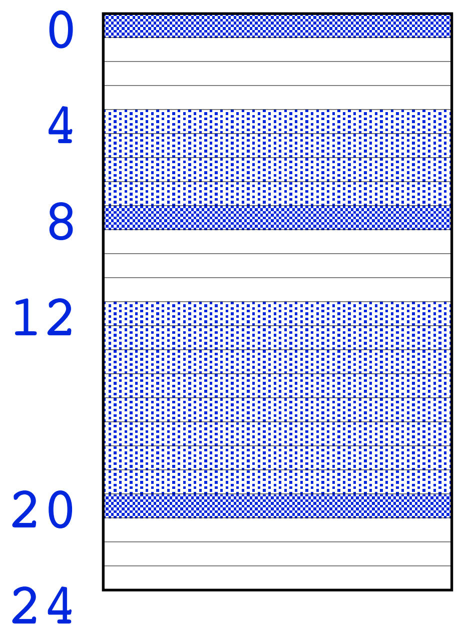
- This algorithm (including the primitive and array cases on the previous slide), applied recursively, handles all storage allocation for any program

Storage Allocation Algorithm Example

```
var R:
    record
        d : char
        x : int
        c : char
        f : real
        e : char
    end record
```

Type	Sparc Size	Sparc Align
char	1	1
boolean	1	1
integer	4	4
real	8	4
pointer	4	4

Field	Size	Align	Disp	SizeSoFar	AlignSoFar
d	1	1	0	1	1
x	4	4	4	8	4
c	1	1	8	9	4
f	8	4	12	20	4
e	1	1	20	21	4
			Final	24	4



Summary

- Part of the job of the back end of a compiler is understanding how to allocate and access data in the memory of the target machine

- Storage of data in memory is normally considered using a *base - displacement* model, which subsumes both LL,ON addressing and the addressing on real machines

- *Data descriptors* are a general formal notation for base-displacement addressing

- The addressing capabilities of a machine can be characterized as the set of data descriptors for which the machine has corresponding *addressing modes* - these parameterize the code generator

- Storage allocation for any machine can be achieved starting with a simple table of *sizes* and *alignments* for the primitive types of the language - a general machine independent algorithm handles all storage allocation from there

Next

- Begin Code Generation!