14 *Processing Data Structure Declarations and References*

14.1 Type Declarations

As we learned in Chapters 8, 9 and 10, type declarations provide much of the information that drives semantic processing and type checking. These declarations also provide information vital for the translation phase of compilation.

Every data object must be allocated space in memory at run-time. Hence it is necessary to determine the space requirements for each kind of data object, whether predefined or user-defined.

We will assume that the symbol table entry associated with each type or class name has a field (an attribute) called size. For predefined types size is determined by the specification of the programming language. For example, in Java an `int` is assigned a size of 4 bytes (one word), a `double` is assigned a size of 8 bytes (a double word), etc. Some predefined types, like strings, have their size determined by the content of the string. Hence in C or C++ the string `"cat"` will require 4 bytes (including the string terminator). In Java, where strings are implemented as `String` objects, `"cat"` will require at least 3 bytes, and probably more (to store the `length` field as well as type and allocation information for the object).

As discussed below, classes, structures and arrays have a size determined by the number and types of their components. In all cases, allocation of a value of type `T` means allocation of a block of memory whose size is T.size.

An architecture also imposes certain “natural sizes” to data values that are computed. These sizes are related to the size of registers or stack values. Thus the value of `A+1` will normally be computed as a 32 bit value even if `A` is declared to be a `byte` or `short`. When a value computed in a natural size is stored, truncation may occur.

14.2 Static and Dynamic Allocation

Identifiers can have either global or local scope. A global identifier is visible throughout the entire text of a program. The object it denotes must be allocated throughout the entire execution of a program (since it may be accessed at any time).

In contrast, a local identifier is visible only within a restricted portion of a program, typically a subprogram, block, structure or class. The object a local identifier denotes need only be allocated when its identifier is visible.

We normally distinguish two modes of allocation for run-time variables (and constants)—**static** and **dynamic** allocation. A local variable that exists only during the execution of a subprogram or method can be dynamically allocated when a call is made. Upon return, the variable ceases to exist and its space can be deallocated.

In contrast, a globally visible object exists for the entire execution of a program; it is allocated space statically by the compiler prior to execution.

In C and C++ global variables are declared outside the body of any subprogram. In Java, global variables occur as static members of classes. Static local variables in C and C++ are also given static allocation—only one memory allocation is made for the whole execution of the program.

C, C++ and Java also provide for dynamic allocation of **values** on a run-time heap. Class objects, structures and arrays may be allocated in the heap, but all such values must be accessed in a program through a local or global variable (or constant).

What happens at run-time when a declaration of local or global is executed? First and foremost, a memory location must be allocated. The **address** of this location will be used whenever the variable is accessed.

The details of how a memory location is allocated is machine-dependent, but the general approach is fairly standard. Let’s consider globals first. In C and C++ (and many other related languages like Pascal, Ada and Fortran), a block of run-time memory is created to hold all globals (variables, constants, addresses, etc.). This block is extended as global declarations are processed by the compiler, using the size attribute of the variable being declared. For example, a compiler might create a label `globals`, addressing an initially empty block of memory. As global declarations are processed, they are given addresses relative to globals. Thus the declarations

```
int a;
char b[10];
float c;
```

would cause `a` to be assigned an address at `globals+0` and `b` to assigned an address at `globals+4`. We might expect `c` to receive an address equivalent to `globals+14`, but on many machines **alignment** of data must be considered. A `float` is usually one word long, and word-length values often must be placed at memory address that are a multiple of 4. Hence `c` may well be placed at an address of `globals+16`, with 2 bytes lost to guarantee proper alignment.

In Java, global variables appear as static members of a program's classes. These members are allocated space within a class when it is loaded. No explicit addresses are used (for portability and security reasons). Rather, a static member is read using a `getstatic` bytecode and written using a `putstatic` bytecode. For example, the globals `a`, `b` and `c`, used above, might be declared as

```
class C {
   static int a;
   static int b[] = new int[10];
   static float f;
}
```

The assignment `C.a = 10;` would generate

```
bipush 10
putstatic C/a I
```

The notation `C/a` resolves to a reference to field `a` of class `C`. The symbol `I` is a **typecode** indicating that field `a` is an `int`. The JVM loader and interpreter translate this reference into the memory location assigned to field `a` in `C`'s class file.

Similarly, the value of `C.f` (a `float` with a typecode of `F`) is pushed onto the top of the run-time stack using

```
getstatic C/f F
```

Locals are dynamically created when a function, procedure or method is called. They reside within the frame created as part of the call (see Section 11.2). Locals are assigned an offset within a frame and are addressed using that offset. Hence if local variable `i` is assigned an offset of 12, it may be addressed as frame_pointer+12. On most machines the frame is accessed using a dedicated frame register or using the stack top. Hence if `$fp` denotes the frame register, `i` can be accessed as `$fp+12`. This expression corresponds to the **indexed** addressing mode found on most architectures. Hence a local may be read or written using a single load or store instruction.

On the JVM, locals are assigned indices within a method's frame. Thus local `i` might be assigned index 7. This allows locals to be accessed using this index into the current frame. For example `i` (assuming it is an integer) can be loaded onto the run-time stack using

```
iload 7
```

Similarly, the current stack-top can be stored into `i` using

```
istore 7
```

Corresponding load and store instructions exist for all the primitive type of Java (`fload` to load a float, `astore` to store an object reference, etc.).

14.2.1 Initialization of Variables

Variables may be initialized as part of their declaration. Indeed, Java mandates a default initialization if no user-specified initialization is provided.

Initialization of globals can be done when a program or method is loaded into memory. That is, when the memory block assigned to global variables is created, a bit pattern corresponding to the appropriate initial values is loaded into the memory locations. This is an easy thing to do—memory locations used to hold program code are already initialized to the bit patterns corresponding to the program's machine-level instructions. For more complex initializations that involve variables or expressions, assignment statements are generated at the very beginning of the main program.

Initialization of locals is done primarily through assignment statements generated at the top of the subprogram's body. Thus the declaration

```
int limit = 100;
```

would generate a store of 100 into the location assigned to `limit` prior to execution of the subprogram's body. More complex initializations are handled in the same way, evaluating the initial value and then storing it into the appropriate location within the frame.

14.2.2 Constant Declarations

We have two choices in translating declarations of constant (final) values. We can treat a constant declaration just like a variable declaration initialized to the value of the constant. After initialization, the location may be read, but is never written (semantic checking enforces the rule that constants may not be redefined after declaration).

For constants whose value is determined at run-time, we have little choice but to translate the constant just as we would a variable. For example given

```
const int limit = findMax(dataArray);
```

we must evaluate the call to `findMax` and store its result in a memory location assigned to `limit`. Whenever `limit` is read, the value stored in the memory location is used.

Most constants are declared to be **manifest constants** whose value is evident at compile-time. For these declarations we can treat the constant as a synonym for its defining value. This means we can generate code that directly accesses the defining value, bypassing any references to memory. Thus given the Java declaration

```
final int dozen = 12;
```

we can substitute 12 for references to `dozen`. Note however that for more complex constant values it may still be useful to assign a memory location initialized to the defining value. Given

```
final double pi = 3.14159265358979323846264338327950288 41971;
```

there is little merit in accessing the value of `pi` in literal form; it is easier to allocate a double word and initialize it once to the appropriate value. Note too that local manifest constants can be allocated globally. This is a useful thing to do since the overhead of creating and initializing a location need be paid only once, at the very start of execution.

14.3 Classes and Structures

One of the most important data structures in modern programming language is the class (or structure or record). In Java, all data and program declarations appear within classes. In C and C++, structures and classes are widely used.

What must be done to allocate and use a class object or structure? Basically, classes and structures are just containers that hold a collection of fields. Even classes, which may contain method declarations, really contain only fields. The bodies of methods, though logically defined within a class, actually appear as part of the translated executable code of a program.

When we process a class or structure declaration, we allocate space for each of the fields that is declared. Each field is given an offset within the class or structure, and the overall size necessary to contain all the fields is determined. This process is very similar to the assignment of offsets for locals that we discussed above.

Let us return to our earlier example of variables `a`, `b` and `c`, now encapsulated within a C-style structure:

```
struct {
    int a;
    char b[10];
    float f;
} S
```

As each field declaration is processed, it is assigned an offset, starting at 0. Offsets are adjusted upward, if necessary, to meet alignment rules. The overall size of all fields assigned so far is recorded. In this example we have

Field	Size	Offset
a	4	0
b	10	4
c	4	16

The size of `S` is just `c`'s offset plus its size. `S` also has an **alignment requirement** that corresponds to a's alignment requirement. That is, when objects of type `S` are allocated (globally, locally or in the heap), they must be word-aligned because field `a` requires it.

In Java, field offsets and alignment are handled by the JVM and class file structure. Within a class file for `C`, each of its fields is named, along with its type. An overall size requirement is computed from these field specifications.

To access a field within a structure or class, we need to compute the address of the field. The formula is simple

address(A.b) = address(A)+Offset(b).

That is, the address of each field in a class object or structure is just the starting address of the object plus the field's offset within the object. With static allocation no explicit addition is needed at run-time. If the object has a static address `Adr`, and `b`'s offset has value `f`, then `Adr+f`, which is a valid address expression, can be used to access `A.b`.

In Java, the computation of a field address is incorporated into the `getfield` and `putfield` bytecodes. An object reference is first pushed onto the stack (this is the object's address). The instruction

```
getfield C/f F
```

determines the offset of field `f` (a `float` with a typecode of `F`) in class `C`, and then adds this offset to the object address on the stack. The data value at the resulting address is then pushed onto the stack. Similarly,

```
putfield C/f F
```

fetches the object address and data value at the top of the stack. It adds `f`'s offset in class `C`, adds it to the object address, and stores the data value (a `float`) at the field address just computed.

For local class objects or structures, allocated in a frame, we use the same formula. The address of the object or struct is represented as frame_pointer+ObjectOffset. The field offset is added to this: frame_pointer+ObjectOffset+FieldOffset. Since the object and field offsets are both constants, they can be added together to form the usual address form for locals, frame_pointer+Offset. As usual, this address reduces to a simple indexed address within an instruction.

In Java, class objects that are locals are accessed via object references that are stored within a frame. Such references are loaded and stored using the `aload` and `astore` bytecodes. Fields are accessed using `getfield` and `putfield`.

Heap-allocated class objects or structures are accessed through a pointer or object reference. In Java all objects **must** be heap-allocated. In C, C++ and related languages, class objects and structures may be dynamically created using `malloc` or `new`. In these cases, access is much the same as in Java—a pointer to the object is loaded into a register, and a field in the object is accessed as register+offset (again the familiar indexed addressing mode).

14.3.1 Variant Records and Unions

A number of varieties of classes and structures are found in programming languages. We'll discuss two of the most important of these—**variant records** and **unions**.

Variant records were introduced in Pascal. They are also found in other programming languages, including Ada. A variant record is a record (that is, a structure) with a special field called a **tag**. The tag, an enumeration or integer value, chooses among a number of mutually exclusive groups of fields called **variants**. That is, a variant record contains a set of fixed fields plus additional fields that may be allocated depending on the value of the tag. Consider the following example, using Pascal's variant record notation.

```
shape = record
```

```
            area : real;
            case kind : shapeKind of
               square : (side : real);
               circle :  (radius : real)
          end;
```

This definition creates a variant record named `shape`. The field `area` is always present (since all shapes have an area). Two kinds of shapes are possible—squares and circles. If `kind` is set to `square`, field `side` is allocated. If `kind` is set to `circle`, field `radius` is allocated. Otherwise, no additional fields are allocated.

Since fields in different variants are mutually exclusive, we don't allocate space for them as we do for ordinary fields. Rather, the variants **overlay** each other. In our example, `side` and `radius` share the same location in the `shape` record, since only one is "active" in any record.

When we allocate offsets for fields in a variant record, we start the first field of each variant at the same offset, just past the space allocated for the tag. For shape we would have

Field	Size	Offset
`area`	4	0
`kind`	1	4
`side`	4	8
`radius`	4	8

The size assigned to `shape` (with proper alignment) is 12 bytes—enough space for `area`, `kind` and either `side` or `radius`.

As with ordinary structures and records, we address fields be adding their offset to the record's starting address. Fields within a variant are accessible only if the tag is properly set. This can be checked by first comparing the tag field with its expected value before allowing access to a variant field. Thus if we wished to access field `radius`, we would first check that `kind` contained the bit pattern assigned to `circle`. (Since this check can slow execution, it is often disabled, just as array bound checking is often disabled.)

Java contains no variant records, but achieves the same capabilities by using subclasses. A class representing all the fields except the tag and variants is first created. Then a number of subclasses are created, one for each desired variant. No explicit tag is needed—the name assigned to each subclass acts like a tag. For example, for shapes we would define.

```
class Shape {
         float area;
}
class Square extends Shape {
         float side;
}
```

```
class Circle extends Shape {
        float radius;
}
```

Ordinary JVM bytecodes for fields (`getfield` and `putfield`) provide access and security. Before a field is accessed, its legality (existence and type) are checked. Hence it is impossible to access a field in the "wrong" subclass. If necessary, we can check which subclass (that is, which kind of shape) we have using the `instanceof` operator.

Pascal also allows variant records without tags; these are equivalent to the union types found in C and C++. In these constructs fields overlay each other, with each field having an offset of 0. As one might expect, such constructs are quite error-prone, since it is often unclear which field is the one to use. In the interests of security Java does not provide a union type.

14.4 Arrays

14.4.1 Static One-Dimensional Arrays

One of the most fundamental data structures in programming languages is the array. The simplest kind of array is one that has a single index and static (constant) bounds. An example (in C or C++) is

```
int a[100];
```

Essentially an array is allocated as a block of N identical data objects, where N is determined by the declared size of the array. Hence in the above example 100 consecutive integers are allocated.

An array, like all other data structures, has a size and an alignment requirement. An array's size is easily computed as

size(array) = NumberOfElements * size(Element).

If the bounds of an array are included within it (as is the case for Java), the array's size must be increased accordingly.

An array's alignment restriction is that of its components. Thus an integer array must be word-aligned if integers must be word-aligned.

Sometimes padding is used to guarantee alignment of **all** elements. For example, given the C declaration

```
struct s {int a; char b;} ar[100];
```

each element (a struct) is padded to a size of 8 bytes. This is necessary to guarantee that `ar[i].a` is always word-aligned.

When arrays are assigned, size information is used to determine how many bytes to copy. A series of load store instructions can be used, or a copy loop, depending on the size of the array.

In C, C++ and Java, all arrays are **zero-based** (the first element of an array is always at position 0). This rule leads to a very simple formula for the address of an array element:

address(A[i]) = address(A) + i * size(Element)

For example, using the declaration of `ar` as an array of struct `s` given above,
address(ar[5]) = address(ar) + 5*size(s) = address(ar) + 5*8 = address(ar)+40.
Computing the address of a field within an array of structures is easy too. Recall that
address(struct.field) = address(struct) + offset(field).
Thus address(struct[i].field) =
address(struct[i]) + offset(field) = address(struct) + i * size(struct) + offset(field).
For example, address(ar[5].b) =
address(ar[5]) + offset(b) = address(ar)+40+4 = address(ar)+44.

In Java, arrays are allocated as objects; all the elements of the array are allocated within the object. Exactly how the objects are allocated is unspecified by Java's definition, but a sequential contiguous allocation, just like C and C++, is the most natural and efficient implementation.

Java hides explicit addresses within the JVM. Hence to index an array it is not necessary to compute the address of a particular element. Instead, special array indexing instructions are provided. First, an array object is created and assigned to a field or local variable. For example,

```
int ar[] = new int[100];
```

In Java, an array assignment just copies a reference to an array object; no array values are duplicated. To create a copy of an array, the `clone` method is used. Thus `ar1 = ar2.clone()` gives `ar1` a newly created array object, initially identical to `ar2`.

To load an array element (onto the stack top), we first push a reference to the array object and the value of the index. Then an "array load" instruction is executed. There is actually a family of array load instructions, depending on the type of the array element being loaded: `iaload` (integer array element), `faload` (floating array element), `daload` (double array element), etc. The first letter of the opcode denotes the element type, and the suffix "`aload`" denotes an array load.

For example, to load the value of `ar[5]`, assuming `ar` is a local array given a frame index of 3, we would generate

```
aload 3   ; Load reference to array ar
iconst_5 ; Push 5 onto stack
iaload   ; Push value of ar[5] onto the stack
```

Storing into an array element in Java is similar. We first push three values: a reference to an array object, the array index, and the value to be stored. The array store instruction is of the form `Xastore`, where X is one of `i`, `l`, `f`, `d`, `a`, `b`, `c`, or `s` (depending on the type to be stored). Hence to implement `ar[4] = 0` we would generate

```
load 3    ; Load reference to array ar
iconst_4 ; Push 4 onto stack
iconst_0 ; Push 0 onto stack
iastore  ; Store 0 into ar[4]
```

Array Bounds Checking. An array reference is legal only if the index used is in bounds. References outside the bounds of an array are undefined and dangerous, as data unrelated to the array may be read or written. Java, with its attention to security, checks that an array index is in range when an array load or array store instruction is executed. An illegal index forces an `ArrayIndexOutOfBounds-Exception`. Since the size of an array object is stored within the object, checking the validity of an index is easy, though it does slow access to arrays.

In C and C++ indices out of bounds are also illegal. Most compilers do not implement bounds checking, and hence program errors involving access beyond array bounds are common.

Why is bounds checking so often ignored? Certainly speed is an issue. Checking an index involves two checks (lower and upper bounds) and each check involves several instructions (to load a bound, compare it with the index, and conditionally branch to an error routine). Using unsigned arithmetic, bounds checking can be reduced to a single comparison (since a negative index, considered unsigned, looks like a *very* large value). Using the techniques of Chapter 16, redundant bounds checks can often be optimized away. Still, array indexing is a very common operation, and bounds checking adds a real cost (though buggy programs are costly too!).

A less obvious impediment to bounds checking in C and C++ is the fact that array names are often treated as equivalent to a pointer to the array's first element. That is, an `int[]` and a `*int` are often considered the same. When an array pointer is used to index an array, we don't know what the upper bound of the array is. Moreover, many C and C++ programs intentionally violate array bounds rules, initializing a pointer one position before the start of an array or allowing a pointer to progress one position past the end of an array.

We can support array bounds checking by including a "size" parameter with every array passed as a parameter and every pointer that steps through an array. This size value serves as an upper bound, indicating the extent to access allowed. Nevertheless, it is clear that bounds checking is a difficult issue in languages where difference between pointers and array addresses is blurred.

Array parameters often require information beyond a pointer to the array's data values. This includes information on the array's size (to implement assignment) and information on the array's bounds (to allow subscript checking). An array descriptor (sometimes called a **dope vector**), containing this information, can be passed for array parameters instead of just a data pointer.

Non-zero Lower Bounds. In C, C++ and Java, arrays always have a lower bound of zero. This simplifies array indexing. Still, a single fixed lower bound can lead to clumsy code sequences. Consider an array indexed by years. Having learned not to represent years as just two digits, we'd prefer to use a full four digit year as an index. Assuming we really want to use years of the twentieth and twenty-first centuries, an array starting at 0 is very clumsy. So is explicitly subtracting 1900 from each index before using it.

Other languages, like Pascal and Ada, have already solved this problem. An array of the form `A[low..high]` may be declared, where all indices in the range `low`, ..., `high` are allowed. With this array form, we can easily declare an array index by four digit years: `data[1900..2010]`.

With a non-zero lower bound, our formula for the size of an array must be generalized a bit:

size(array) = (UpperBound - LowerBound + 1) * size(Element).

How much does this generalization complicate array indexing? Actually, surprisingly little. If we take the Java approach, we just include the lower bound as part of the array object we allocate. If we compute an element address in the code we generate, the address formula introduced above needs to be changed a little:

address(A[i]) = address(A) + (i - low) * size(Element)

We subtract the array's lower bound (low) before we multiply by the element size. Now it is clear why a lower bound of zero simplifies indexing—a subtraction of zero from the array index can be skipped. But the above formula can be rearranged to:

address(A[i]) = address(A) + (i * size(Element)) - (low * size(Element)) =
address(A) - (low * size(Element)) + (i * size(Element))

We now note that low and size(Element) are normally compile-time constants, so the expression (low * size(Element)) can be reduced to a single value, bias. So now we have

address(A[i]) = address(A) - bias + (i * size(Element))

The address of an array is normally a static address (a global) or an offset relative to a frame pointer (a local). In either case, the bias value can be folded into the array's address, forming a new address or frame offset reduced by the value of `bias`.

For example, if we declare an array `int data[1900..2010]`, and assign `data` an address of 10000, we get a `bias` value of 1900*size(int) = 7600. In computing the address of `data[i]` we compute 2400+i*4. This is exactly the same form that we used for zero-based arrays.

Even if we allocate arrays in the heap (using `new` or `malloc`), we can use the same approach. Rather than storing a pointer to the array's first element, we store a pointer to its zero-th element (which is what subtracting bias from the array address gives us). We do need to be careful when we assign such arrays though; we must copy data from the first valid position in the array. Still, indexing is far more common than copying, so this approach is still a good one.

Dynamic and Flex Arrays. Some languages, including Algol 60, Ada and Java support **dynamic arrays** whose bounds and size are determined at run-time. When the scope of a dynamic array is entered, its bounds and size are evaluated and fixed. Space for the array is then allocated. The bounds of a dynamic array may include parameters, variables and expressions. For example, in C extended to allow dynamic arrays, procedure `P` we might include the declaration

```
int examScore[numOfStudents()];
```

Because the size of a dynamic array isn't known at compile-time, we can't allocate space for it globally or in a frame. Instead, we must allocate space either on the stack (just past the current frame) or in the heap (Java does this). A pointer to the location of the array is stored in the scope in which the array is declared. The size of the array (and perhaps its bounds) are also stored. Using our above example, we would allocate space for `examScores` as shown in Figure 14.1. Within `P`'s frame we allocate space for a pointer to `examScore`'s values as well as it size.

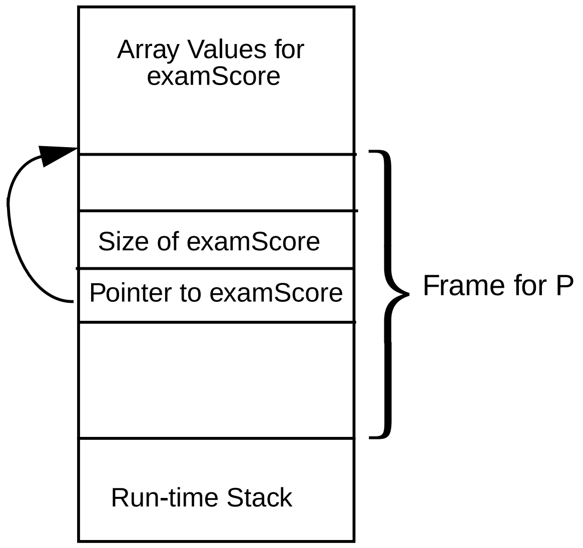

Figure 14.1 Allocation of a Dynamic Array

Accessing a dynamic array requires an extra step. First the location of the array is loaded from a global address or from an offset within a frame. Then, as usual, an offset within the array is computed and added to the array's starting location.

A variant of the dynamic array is the **flex array**, which can expand its bounds during execution. (The Java `Vector` class implements a flex array.) When a flex array is created, it is given a default size. If during execution an index beyond the array's current size is used, the array is expanded to make the index legal. Since the ultimate size of a flex array isn't known when the array is initially allocated, we store the flex array in the heap, and point to it. Whenever a flex array is indexed, the array's current size is checked. If it is too small, another larger array is allocated, values from the old allocation are copied to the new allocation, and the array's pointer is reset to point to the new allocation.

When a dynamic or flex array is passed as a parameter, it is necessary to pass an array descriptor that includes a pointer to the array's data values as well as information on the array's bounds and size. This information is needed to support indexing and assignment.

14.4.2 Multidimensional Arrays

In most programming languages multidimensional arrays may be treated as arrays of arrays. In Java, for example, the declaration

```
    int matrix[][] = new int[5][10];
```

first assigns to `matrix` an array object containing five references to integer arrays. Then, in sequence, five integer arrays (each of size ten) are created, and assigned to the array matrix references (Figure 14.2).

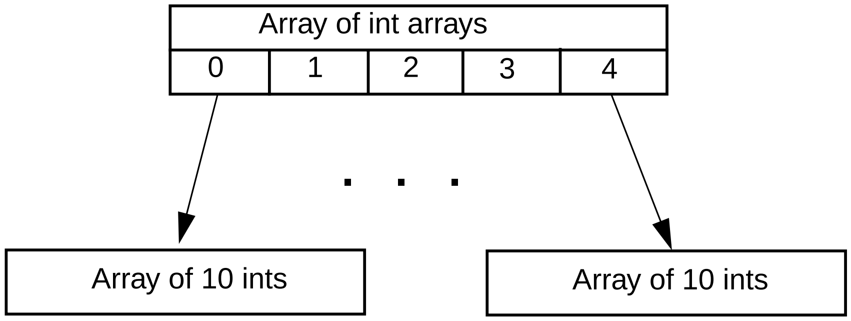

Figure 14.2 A Multidimensional Array in Java

Other languages, like C and C++, allocate one block of memory, sufficient to contain all the elements of the array. The array is arranged in **row-major** form, with values in each row contiguous and individual rows placed sequentially (Figure 14.3). In row-major form, multidimensional arrays really are arrays of arrays, since in an array reference like `A[i][j]`, the first index (`i`) selects the i-th row, and the second index (`j`) chooses an element within the selected row.

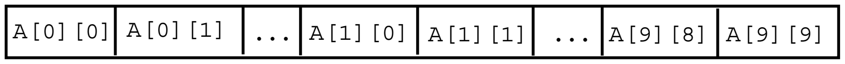

Figure 14.3 Array `A[10][10]` Allocated in Row-Major Order

An alternative to row-major allocation is **column-major** allocation, which is used in Fortran and related languages. In column-major order values in individual columns are contiguous, and columns are placed adjacent to each other (Figure 14.4). Again, the whole array is allocated as a single block of memory.

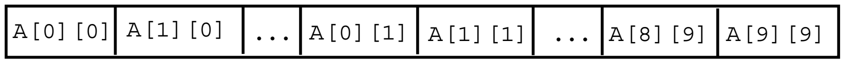

Figure 14.4 Array `A[10][10]` Allocated in Column-Major Order

How are elements of multidimensional arrays accessed? For arrays allocated in row-major order (the most common allocation choice), we can exploit the fact that multidimensional arrays can be treated as arrays of arrays. In particular, to compute the address of `A[i][j]`, we first compute the address of `A[i]`, treating `A` as a one dimensional array of values that happen to be arrays. Once we have the address of `A[i]`, we then compute the address of `X[j]`, where `X` is the starting address of `A[i]`.

Let's look at the actual computations needed. Assume array `A` is declared to be an n by m array (e.g., it is declared as `T A[n][m]`, where `T` is the type of the array's elements).

We now know that

address(A[i][j]) = address(X[j]) where X = address(A[i]).
address(A[i]) = address(A) + i * size(T) * m.

Now

address(X[j]) = address(X) + j * size(T).

Putting these together,

address(A[i][j]) = address(A) + i * size(T) * m + j * size(T) =
address(A) + (i * m + j) * size(T).

Computation of the address of elements in a column-major array is a bit more involved, but we can make a useful observation. First, recall that **transposing** an array involves interchanging its rows and columns. That is, the first column of the array becomes the first row of the transposed array, the second column becomes the second row and so on (see Figure 14.5).

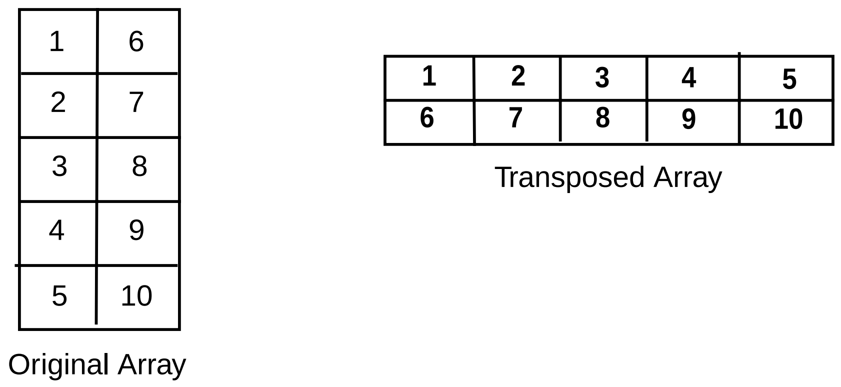

Figure 14.5 An Example of Array Transposition

Now observe that a column-major ordering of elements in an array corresponds to a row-major ordering in the transposition of that array. Allocate an n by m array, `A`, in column-major order and consider any element `A[i][j]`. Now transpose `A` into `AT`, an m by n array, and allocate it in row-major order. Element `AT[j][i]` will always be the same as `A[i][j]`.

What this means is that we have a clever way of computing the address of A[i][j] in a column-major array. We simply compute the address of AT[j][i], where AT is treated as a row-major array with the same address as A, but with interchanged row and column sizes (A[n][m] becomes AT[m]n]).

As an example, refer to Figure 14.5. The array on the left represents a 5 by 2 array of integers; in column-major order, the array's elements appear in the order 1 to 10. Similarly, the array on the right represents a 2 by 5 array of integers; in row-major order, the array's elements appear in the order 1 to 10. It is easy to see that a value in position [i][j] in the left array always corresponds to the value at [j]i] in the right (transposed) array.

In Java, indexing multidimensional arrays is almost exactly the same as indexing one dimensional arrays. The same JVM bytecodes are used. The only difference is that now more than one index value must be pushed. For example, if A is a two dimensional integer array (at frame offset 3), to obtain the value of A[1][2] we would generate

```
aload 3   ; Load reference to array A
iconst_1  ; Push 1 onto stack
iconst_2  ; Push 2 onto stack
iaload    ; Push value of A[1][2] onto the stack
```

The JVM automatically checks that the right number of index values are pushed, and that each is in range.

14.5 Implementing Other Types

In this section we'll consider implementation issues for a number of other types that appear in programming languages.

Enumerations. A number of programming languages, including C, C++, Pascal and Ada, allow users to define a new data type by enumerating its values. For example, in C we might have

```
enum color { red, blue, green } myColor;
```

An enumeration declaration involves both semantics and code generation. A new type is declared (e.g. color). Constants of the new type are also declared (red, blue and green). Finally, a variable of the new type (myColor) is declared.

To implement an enumeration type we first have to make values of the enumeration available at run-time. We assign internal encoding to the enumeration's constant values. Typically these are integer values, assigned in order, starting at zero or one. Thus we might treat red as the value 1, blue as the value 2 and green as the value 3. (Leaving zero unassigned makes it easy to distinguish uninitialized variables, which often are set to all zeroes). In C and C++, value assignment starts at zero.

C and C++ allow users to set the internal representation for selected enumeration values; compiler selected values must not interfere with those chosen by a programmer. We can allocate initialized memory locations corresponding to each enumeration value, or we can store the encoding in the symbol table and fill them directly into instructions as immediate operands.

Once all the enumeration values are declared, the size required for the enumeration is chosen. This is usually the smallest "natural" memory unit that can accommodate all the possible enumeration values. Thus we might choose to allocate a byte for variables of type `color`. We could actually use as few as two bits, but the cost of additional instructions to insert or extract just two bits from memory makes byte-level allocation more reasonable.

A variable declared to be an enumeration is implemented just like other scalar variables. Memory is allocated using the size computed for the enumeration type. Values are manipulated using the encoding chosen for the enumeration values.

Thus variable `myColor` is implemented as if it were declared to be of type `byte` (or `char`). The assignment `myColor = blue` is implemented by storing 2 (the encoding of `blue`) into the byte location assigned to `myColor`.

Subranges. Pascal and Ada include a useful variant of enumerated and integer types—the subrange. A selected range of values from an existing type may be chosen to form a new type. For example

```
type date = 1..31;
```

A subrange declaration allows a more precise definition of the range of values a type will include. It also allows smaller memory allocations if only a limited range of values is possible.

To implement subranges, we first decide on the memory size required. This is normally the smallest "natural" unit of allocation sufficient to store all the possible values. Thus a byte would be appropriate for variables of type `date`. For subranges involving large values, larger memory sizes may be needed, even if the range of values is limited. Given the declaration

```
type year = 1900..2010;
```

we might well allocate a half word, even though only 111 distinct values are possible. Doing this makes loading and storing values simpler, even if data sizes are larger than absolutely necessary.

In manipulating subrange values, we may want to enforce the subrange's declared bounds. Thus whenever a value of type `year` is stored, we can generate checks to verify that the value is in the range 1900 to 2010. Note that these checks are very similar to those used to enforce array bounds.

Ada allows the bounds of a subrange to be expressions involving parameters and variables. When range bounds aren't known in advance, a maximum size memory allocation is made (full or double word). A run-time descriptor is created and initialized to the range bounds, once they are evaluated. This descriptor is used to implement run-time range checking.

Files. Most programming languages provide one or more file types. File types are potentially complex to implement since they must interact with the computer's operating system. For the most part, a compiler must allocate a "file descriptor" for each active file. The format of the file descriptor is system-specific; it is determined by the rules and conventions of the operating system in use.

The compiler must fill in selected fields from the file's declaration (name of the file, its access mode, the size of data elements, etc.). Calls to system routines to open and close files may be needed when the scope containing a file is entered or exited. Otherwise, all the standard file manipulation operations—read, write, test for end-of-file, etc., simply translate to calls to corresponding system utilities.

In Java, where all files are simply instances of predefined classes, file manipulation is no different than access to any other data object.

Pointers and Objects. We discussed the implementation of pointers and object references in Section 12.7. In Java, all data except scalars are implemented as objects and accessed through object references (pointers). Other languages routinely use a similar approach for complex data objects. Hence, as we learned above, dynamic and flex arrays are allocated on the stack or heap, and accessed (invisibly) through pointers. Similarly, strings are often implemented by allocating the text of the string in the heap, and keeping a pointer to the current text value in the heap.

In Java, all objects are referenced through pointers, so the difference between a pointer and the object it references is obscured. In languages like C and C++ that have both explicit and implicit pointers, the difference between pointer and object can be crucial.

A good example of this is the difference between array and structure assignment in C and C++. The assignment `struct1 = struct2` is implemented as a memory copy; all the fields within `struct2` are copied into `struct1`. However, the assignment `array1 = array2` means something very different. `array2` represents the *address* of `array2`, and assigning it to `array1` (a constant address) is illegal.

Whenever we blur the difference between a pointer and the object it references, similar ambiguities arise. Does `ptr1 = ptr2` mean copy a pointer or copy the object pointed to? In implementing languages it is important to be clear on exactly when evaluation yields a pointer and when it yields an object.

Exercises

1. Local variables are normally allocated within a frame, providing for automatic allocation and deallocation when a frame is pushed and popped. Under what circumstance must a local variable be dynamically allocated?
   Are there any advantages to allocating a local variable statically (i.e., giving it a single fixed address)? Under what circumstances is static allocation for a local permissible?
2. Consider the following C/C++ structure declarations
   ```
   struct {int a; float b; int c[10];} s;
   struct {int a; float b; } t[10];
   ```
   Choose your favorite computer architecture. Show the code that would be generated for `s.c[5]` assuming `s` is statically allocated at address 1000. What code would be generated for `t[3].b` if `t` is allocated within a frame at offset 200?
3. Consider the following Java class declaration
   ```
   class C1 {int a; float b; int c[]; C1 d[];};
   ```
   Assume that `v` is a local reference to `C1`, with an index of 4 within its frame. Show the JVM code that would be generated for `v.a`, `v.c[5]`, `v.d[2].b` and `v.d[2].c[4]`.
4. Explain how the following three declaration would be translated in C, assuming that they are globally declared (outside all subprogram bodies).
   ```
   const int ten = 10;
   const float pi = 3.14159;
   const int limit = get_limit();
   ```
   How would the above three declaration be translated if they are locals declarations (within a subprogram body)?
5. Assume that in C we have the declaration `int a[5][10][20]`, where `a` is allocated at address 1000. What is the address of `a[i][j][k]` assuming `a` is allocated in row-major order? What is the address of `a[i][j][k]` assuming `a` is allocated in column-major order?
6. Most programming languages (including Pascal, Ada, C, and C++) allocate global aggregates (records, arrays, structs and classes) statically, while local aggregates are allocated within a frame. Java, on the other hand, allocates all aggregates in the heap. Access to them is via object references allocated statically or within a frame.
   Is it less efficient to access an aggregate in Java because of its mandatory heap allocation? Are there any advantages to forcing all aggregates to be uniformly allocated in the heap?
7. In Java, subscript validity checking is mandatory. Explain what changes would be needed in C or C++ (your choice) to implement subscript validity checking. Be sure to address the fact that pointers are routinely used to access array elements. Thus you should be able to checks array accesses that are done

through pointers, including pointers that have been incremented or decremented.

8. Assume we add a new option to C++ array that are heap-allocated, the **flex** option. A flex array is automatically expanded in size if an index beyond the array's current upper limit is accessed. Thus we might see

```
   ar = new flex int[10]; ar[20] = 10;
```

   The assignment to position 20 in `ar` forces an expansion of `ar's` heap allocation. Explain what changes would be needed in array accessing to implement flex arrays. What should happen if an array position beyond an array's current upper limit is read rather than written?

9. Fortran library subprograms are often called from other programming languages. Fortran assumes that multi-dimensional arrays are stored in column-major order; most other languages assume row-major order. What must be done if a C program (which uses row-major order) passes a multi-dimensional array to a Fortran subprogram. What if a Java method, which stores multi-dimensional arrays as arrays of array object references, passes such an array to a Fortran subprogram?

10. Recall that offsets within a record or struct must sometimes be adjusted upward due to alignment restrictions. Thus in the following two C structs, `S1` requires 6 bytes whereas `S2` requires only 4 bytes.

```
   struct {              struct {
      char  c1;             char  c1;
      short s;              char  c2;
      char  c2;             short s;
   } S1;                 } S2;
```

   Assume we have a list of the fields in a record or struct. Each is characterized by its size and alignment restriction 2^a. (A field with an alignment restriction 2^a must be assigned an offset that is a multiple of 2^a).

   Give an algorithm that determines an ordering of fields that minimizes the overall size of a record or struct while maintaining all alignment restrictions. How does the execution time of your algorithm (as measured in number of execution steps) grow as the number of fields increases?