8

VARIABLES IN A HIGH-LEVEL LANGUAGE

This chapter will explore the low-level implementation of variables found in high-level languages. Although assembly language programmers usually have a good feel for the connection between variables and memory locations, high-level languages add sufficient abstraction to obscure this relationship. This chapter will cover the following topics:

- The runtime memory organization typical for most compilers
- How the compiler breaks up memory into different sections and how the compiler places variables into each of those sections
- The attributes that differentiate variables from other objects
- The difference between static, automatic, and dynamic variables
- How compilers organize automatic variables in a stack frame
- The primitive data types that hardware provides for variables
- How machine instructions encode the address of a variable

When you finish reading this chapter, you should have a good understanding of how to declare variables in your program to use the least amount of memory and produce fast-running code.

8.1 Runtime Memory Organization

An operating system like Linux or Windows puts different types of data into different areas (*sections* or *segments*) of main memory. Although it is possible to control the memory organization by running a linker and specifying various command-line parameters, by default Windows loads a typical program into memory using the organization appearing in Figure 8-1 (Linux is similar, although it rearranges some of the sections).

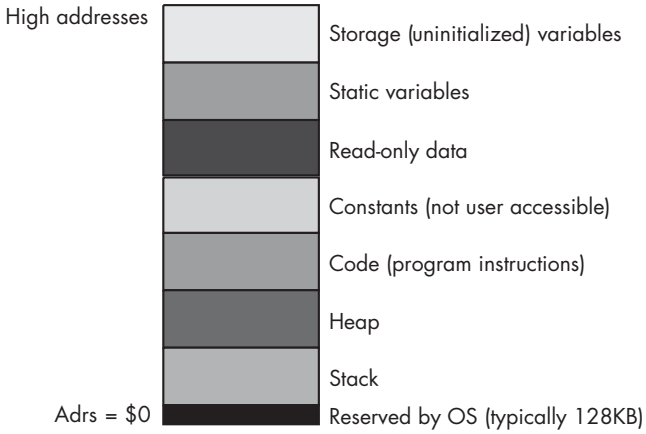

Figure 8-1: Typical runtime memory organization for Windows

The operating system reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at the lowest addresses in memory. One reason the OS reserves this space is to help detect NULL pointer references. Programmers often initialize pointers with NULL (zero) to indicate that the pointer is not valid. Should you attempt to access memory location zero under such an operating system, the OS will generate a *general protection fault* to indicate that you've accessed an invalid memory location.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the code section, the constant section, the initialized static-object section, and the uninitialized data section. Each of these memory sections corresponds to some type of data you can create in your programs.

Most of the time, a given application can live with the default layouts chosen for these sections by the compiler and linker/loader. In some cases, however, knowing the memory layout can allow you to develop shorter programs. For example, because the code section is usually read-only, it might

be possible to combine the code, constant, and read-only data sections into a single section, thereby saving any padding space that the compiler/linker may place between these sections. Although for large applications this is probably insignificant, for small programs it can have a big impact on the size of the executable.

The following sections discuss each of these sections in detail.

8.1.1 The Code, Constant, and Read-Only Sections

The code section in memory contains the machine instructions for a program. Your compiler translates each statement you write into a sequence of one or more byte values (machine instruction opcodes). The CPU interprets these opcode values during program execution.

Most compilers also attach a program's read-only data and *constant pool* (constant table) sections to the code section because, like the code instructions, the read-only data is already write-protected. However, it is perfectly possible under Windows, Linux, and many other operating systems to create a separate section in the executable file and mark it as read-only. As a result, some compilers do support a separate *read-only* data section, and some compilers even create a different section (the constant pool) for the constants that the compiler emits. Such sections contain initialized data, tables, and other objects that the program should not change during program execution.

Many compilers will generate multiple code sections and leave it up to the linker to combine those sections into a single code segment prior to execution. To understand why a compiler might do this, consider the following short Pascal code fragment:

```
    if( SomeBooleanExpression ) then begin

        << Some code that executes 99.9% of the time >>

    end
    else begin

        << Some code that executes 0.1% of the time >>

    end;
```

Without worrying about how it does so, assume that the compiler can figure out that the `then` section of this `if` statement executes far more often than the `else` section. An assembly programmer, wanting to write the fastest possible code, might encode this sequence as follows:

```
    << evaluate Boolean expression, leave True/False in EAX >>
    test( eax, eax );
    jz exprWasFalse;
    << Some code that executes 99.9% of the time >>
rtnLabel:
    << Code normally following the last END in the
               Pascal example >>
```

```
        .
        .
        .
// somewhere else in the code, not in the direct execution path
// of the above:

exprWasFalse:
    << Some code that executes 0.1% of the time >>

    jmp rtnLabel;
```

This assembly code might seem a bit convoluted, but keep in mind that any control transfer instruction is probably going to consume a lot of time because of pipelined operation on modern CPUs (see *Write Great Code, Volume 1*, for the details). Code that executes without branching (or that *falls straight through*) executes the fastest. In the previous example, the common case falls straight through 99.9 percent of the time. The rare case winds up executing two branches (one to transfer to the `else` section and one to return back to the normal control flow). But because this code rarely executes, it can afford to take longer to execute.

Many compilers use a little trick to move sections of code around like this in the machine code they generate—they simply emit the code in a sequential fashion, but they place the `else` code in a separate section. Here's some MASM code that demonstrates this principle in action:

```
    << evaluate Boolean expression, leave True/False in EAX >>
    test eax, eax
    jz exprWasFalse
    << Some code that executes 99.9% of the time >>
alternateCode segment

    << Some code that executes 0.1% of the time >>

    jmp rtnLabel;
alternateCode ends

rtnLabel:
    << Code normally following the last END in the Pascal example >>
```

Even though the `else` section code appears to immediately follow the `then` section's code, placing it in a different segment tells the assembler/linker to move this code and combine it with other code in the `alternateCode` segment. This little trick, which relies upon the assembler or linker to do the code movement, can simplify HLL compilers. GCC, for example, uses this trick to move code around in the assembly language file it emits. As a result, you'll see this trick being used on occasion. Therefore, expect some compilers to produce multiple code segments.

8.1.2 The Static Variables Section

Many languages provide the ability to initialize a global variable during the compilation phase. For example, in the C/C++ language, you could use statements like the following to provide initial values for these static objects:

```
    static int i = 10;
    static char ch[] = ( 'a', 'b', 'c', 'd' };
```

In C/C++ and other languages, the compiler will place these initial values in the executable file. When you execute the application, the operating system will load the portion of the executable file that contains these static variables into memory so that the values appear at the addresses associated those variables. Therefore, when the program first begins execution, `i` and `ch` will magically have these values bound to them.

The static section is often called the `DATA` or `_DATA` segment in the assembly listings that most compilers produce. As an example, consider the following C code fragment and the TASM assembly code that the Borland C++ compiler produces for it:

```
#include <stdlib.h>
#include <stdio.h>

static char *c = NULL;
static int i = 0;
static int j = 1;
static double array[4] = {0.0, 1.0, 2.0, 3.0};

int main( void )
{

      .
      .
      .
```

And here's the assembly code emitted by the Borland C++ compiler for the declarations in this C example:

```
_DATA   segment dword public use32 'DATA'
        align   4
_c      label   dword
        dd      0
        align   4
_i      label   dword
        dd      0
        align   4
_j      label   dword
        dd      1
        align   4
```

```
_array  label   qword
        db        0,0,0,0,0,0,0,0
        db        0,0,0,0,0,0,240,63
        db        0,0,0,0,0,0,0,64
        db        0,0,0,0,0,0,8,64
_DATA   ends
```

As you can see in this example, Borland's C++ compiler places these variables in the `_DATA` segment.

8.1.3 The BSS Section

Most operating systems will *zero out* memory prior to program execution. Therefore, if an initial value of zero is suitable, you don't need to waste any disk space with the static object's initial value. Generally, however, compilers treat uninitialized variables in a static section as though you've initialized them with zero, thereby consuming disk space. Some operating systems provide another section type, the *BSS section*, to avoid this waste of disk space.

The BSS section is where compilers typically put static objects that don't have an explicit initial value. BSS stands for *block started by a symbol,* and it is an old assembly language term describing a pseudo-opcode you would use to allocate storage for an uninitialized static array. In modern operating systems like Windows and Linux, the OS allows the compiler/linker to put all uninitialized variables into a BSS section that simply tells the OS how many bytes to set aside for that section. When the operating system loads the program into memory, it reserves sufficient memory for all the objects in the BSS section and fills this range of memory with zeros. It is important to note that the BSS section in the executable file doesn't contain any actual data. For this reason, programs that declare large uninitialized static arrays in a BSS section will consume less disk space. The following is the C/C++ example from the previous section, modified to remove the initializers so that the compiler will place the variables in the BSS section:

```
#include <stdlib.h>
#include <stdio.h>

static char *c;
static int i;
static int j;
static double array[4];

int main( void )
{
      .
      .
      .
```

Here is the Borland C++ output:

```
_BSS    segment dword public use32 'BSS'
        align   4
_c      label   dword
        db      4       dup(?)
        align   4
_i      label   dword
        db      4       dup(?)
        align   4
_j      label   dword
        db      4       dup(?)
        align   4
_array  label   qword
        db      32      dup(?)
_BSS    ends
```

Not all compilers use a BSS section. Many Microsoft languages and linkers, for example, simply combine the uninitialized objects with the static/data section and explicitly give them an initial value of zero. Although Microsoft claims that this scheme is faster, it certainly makes executable files larger if your code has large, uninitialized arrays (because each byte of the array winds up in the executable file, something that would not happen if the compiler were to place the array in a BSS section). Note, however, that this is a default condition and you can change this by setting the appropriate linker flags.

8.1.4 The Stack Section

The *stack* is a data structure that expands and contracts in response to procedure invocations and returns, among other things. At runtime, the system places all automatic variables (nonstatic local variables), subroutine parameters, temporary values, and other objects in the stack section of memory in a special data structure called the *activation record* (the activation record is aptly named because the system creates an activation record when a subroutine first begins execution and deallocates the activation record when the subroutine returns to its caller). Therefore, the stack section in memory is very busy.

Many CPUs implement the stack using a special-purpose register called the *stack pointer*. Other CPUs (particularly RISC) don't provide an explicit stack pointer and, instead, use a general-purpose register for this purpose. If a CPU provides an explicit stack pointer register, we say that the CPU supports a hardware stack; if a program uses a general-purpose register for this purpose, then we say that the CPU uses a software-implemented stack. The 80x86 is a good example of a CPU that provides a hardware stack—the PowerPC family is a good example of a CPU family that implements the stack in software (most PowerPC programs use R1 as the stack pointer register). Systems that provide hardware stacks can generally manipulate data on

the stack using fewer instructions than systems that implement the stack in software. On the other hand, RISC CPU designers who've chosen to use a software stack implementation feel that the presence of a hardware stack actually slows down all instructions the CPU executes. In theory, you could argue that the RISC designers are right; in practice, the 80x86 family includes some of the fastest CPUs around, providing ample proof that having a hardware stack doesn't necessarily mean you'll wind up with a slow CPU.

8.1.5 The Heap Section and Dynamic Memory Allocation

Although simple programs may only need static and automatic variables, sophisticated programs need the ability to allocate and deallocate storage dynamically (at runtime) under program control. In the C and High-Level Assembler (HLA) languages, you would use the `malloc` and `free` functions for this purpose. C++ provides the `new` and `delete` operators. Pascal uses `new` and `dispose`. Other languages provide comparable routines. These memory-allocation routines share a few things in common:

- They let the programmer request how many bytes of storage to allocate.
- They return a *pointer* to the newly allocated storage (that is, the address of that storage).
- They provide a facility for returning the storage space to the system once it is no longer needed so the system can reuse it in a future allocation call.

Dynamic memory allocation takes place in a section of memory known as the *heap*. Generally, an application refers to data on the heap using pointer variables, either implicitly or explicitly; some languages, like Java, implicitly use pointers behind the programmer's back. As such, these objects in heap memory are usually referred to as *anonymous variables* because they are referred to by their memory address (via pointers) rather than by a name.

The OS and application create the heap section in memory after the program begins execution; the heap is never a part of the executable file. Generally, the operating system and language runtime libraries maintain the heap for an application. Despite the variations in memory management implementations, it's still a good idea for you to have a basic idea of how heap allocation and deallocation operate because an inappropriate use of the heap management facilities will have a very negative impact on the performance of your applications.

8.2 What Is a Variable?

If you consider the word *variable*, it should be obvious that it describes something that *varies*. But exactly what is it that varies? To most programmers the answer will seem obvious: it's the value that can vary during program execution. In fact, there are several things that can vary, so before attempting to describe what a variable is, it is probably a good idea to discuss some attributes that variables (and other objects) may possess. To do this, I must first define *attribute*, *binding*, *static objects*, *dynamic objects*, *scope*, and *lifetime*.

8.2.1 Attributes

An *attribute* is some feature that is associated with an object. For example, common attributes of a variable include that variable's name, its memory address, its runtime value, a data type associated with that value, and the size (in bytes) of that variable. Different objects may have different sets of attributes. For example, a *data type* is an object that possesses attributes such as a name and size, but it won't usually have a value or memory location associated with it. A *constant* can have attributes such as a value and a data type, but it does not have a memory location and it might not have a name (for example, if it is a literal constant). A *variable* may possess all of these attributes. Indeed, the attribute list usually determines whether an object is a constant, data type, variable, or something else.

8.2.2 Binding

Binding is the process of associating an attribute with an object. For example, when a value is assigned to a variable, the value is *bound* to that variable at the point of the assignment. The value remains bound to the variable until some other value is bound to it (via another assignment operation). Likewise, if you allocate memory for a variable while the program is running, the variable is *bound* to the memory address at that point. The variable and address are bound until you associate a different address with the variable. Binding needn't occur at runtime. For example, values are bound to constant objects during compilation, and such bindings cannot change while the program is running. Similarly, some variables can have their address bound to them at compile time, and the memory address cannot change during program execution.

8.2.3 Static Objects

Static objects have an attribute bound to them prior to the execution of the application. Constants are good examples of static objects; they have the same value bound to them throughout the execution of the application. Global (program-level) variables in programming languages like Pascal, C/C++, and Ada are also examples of static objects because they have the same memory address bound to them throughout the program's lifetime. The system binds attributes to a static object before the program begins execution (usually during compilation or during the linking phase, though it is possible to bind values even earlier).

8.2.4 Dynamic Objects

Dynamic objects have some attribute bound to them during program execution. The program may choose to change that attribute (*dynamically*) while the program is running. Dynamic attributes usually cannot be determined at compile time. Examples of dynamic attributes include values bound to variables at runtime and memory addresses bound to certain variables at runtime (e.g., via a *malloc* or other memory allocation function call).

8.2.5 Scope

The *scope* of an identifier is that section of the program where the identifier's name is bound to the object. Because names in most compiled languages exist only during compilation, scope is usually a static attribute (although in some languages it is possible for scope to be a dynamic attribute). By controlling where a name is bound to an object, it is possible to reuse that name elsewhere in the program.

Most modern imperative programming languages (e.g., C/C++/C#, Java, Pascal, and Ada) support the concept of *local* and *global* variables. A local variable's name is bound to a particular object only within a given section of a program (for example, within a particular function). Outside the scope of that object, the name can be bound to a different object. This allows a global and a local object to share the same name without any ambiguity. This may seem potentially confusing, but being able to reuse variable names like `i` or `j` throughout a project can spare the programmer from having to dream up equally meaningless unique variable names for loop indexes and other uses in the program. The scope of the object's declaration determines where the name applies to a given object.

In interpretive languages, where the interpreter maintains the identifier names during program execution, scope can be a dynamic attribute. For example, in various versions of the BASIC programming language, the `dim` statement is an executable statement. Prior to the execution of `dim`, the name you define might have a completely different meaning than it does after executing `dim`. SNOBOL4 is another language that supports dynamic scope. Generally, most programming languages avoid dynamic scope because using it can result in difficult-to-understand programs—but the fact that most languages avoid dynamic scope doesn't mean it doesn't exist.

In general, scope can apply to any attribute, not just names. In this book, however, I'll only use the term scope to describe where a name is associated with a given variable.

8.2.6 Lifetime

The *lifetime* of an attribute extends from the point when you first bind an attribute to an object to the point you break that bond, perhaps by binding a different attribute to the object. If the program associates some attribute with an object and never breaks that bond, the lifetime of the attribute is from the point of association to the point the program terminates. For example, the lifetime of a variable is from the time you first allocate memory for the variable to the moment you deallocate that variable's storage. As a program binds static objects prior to execution (and such attributes do not change during program execution), the lifetime of a static object extends from when the program begins execution to when the application terminates.

8.2.7 So What Is a Variable?

A variable is an object that can have a value bound to it dynamically. That is, the program can change the variable's value attribute at runtime. Note the operative word *can*. It is only necessary for the program to be able to change a variable's value at runtime; it doesn't *have* to bind multiple values in order to consider the object a variable.

Dynamic binding of a value to an object is the defining attribute of a variable, though other attributes may be dynamic or static. For example, the memory address of a variable can be statically bound to the variable at compile time or dynamically bound at runtime. Likewise, variables in some languages have dynamic types that change during program execution, while other variables have static types that remain fixed over the execution of a given program. Only the binding of the value determines whether the object is a variable or something else (such as a constant).

8.3 Variable Storage

Values must be stored in and retrieved from memory. To do this, a compiler must bind a variable to one or more memory locations. The variable's type determines the amount of storage it requires. Character variables may require as little as a single byte of storage, while large arrays or records can require thousands or millions of bytes of storage. To associate a variable with some memory, a compiler (or runtime system) binds the address of that memory location to that variable. When a variable requires two or more memory locations, the system will usually bind the address of the first memory location to the variable and assume that the contiguous locations following that address are also bound to the variable at runtime.

Three types of bindings are possible between variables and memory locations: static binding, pseudo-static (automatic) binding, and dynamic binding. Variables are generally classified as *static*, *automatic*, or *dynamic* based upon how the variable is bound to its memory location.

8.3.1 Static Binding and Static Variables

Static binding occurs prior to runtime, at one of four possible times: at language-design time, at compile time, at link time, or when the system loads the application into memory (but prior to execution). Binding at language design time is not all that common, but it does occur in some languages (especially assembly languages). Binding at compile time is common in assemblers and compilers that directly produce executable code. Binding at link time is fairly common (for example, some Windows compilers do this). Binding at load time, when the operating system copies the executable into memory, is probably the most common for static variables.

8.3.1.1 Binding at Language-Design Time

An address can be assigned at language-design time when a language designer associates a language-defined variable with a specific hardware address (for example, an I/O device or a special kind of memory), and that address never changes in any program. Such objects are common in embedded systems and rarely found in applications on general-purpose computer systems. For example, on an 8051 microcontroller, many C compilers and assemblers automatically associate certain names with fixed locations in the 128 bytes of data space found on the CPU. CPU register references in assembly language are good example of variables bound to some location at language-design time.

8.3.1.2 Binding at Compile Time

An address can be assigned at compile-time when the compiler knows the memory region where it can place static variables at runtime. Generally, such compilers generate absolute machine code that must be loaded at a specific address in memory prior to execution. Most modern compilers generate relocatable code and, therefore, don't fall into this category. Nevertheless, lower-end compilers, high-speed student compilers, and compilers for embedded systems often use this binding technique.

8.3.1.3 Binding at Link Time

Certain linkers (and related tools) have the ability to link together various relocatable object modules of an application and create an absolute load module. So while the compiler produces relocatable code, the linker binds memory addresses to the variables (and machine instructions). Usually, the programmer specifies (via command-line parameters or a linker script file) the base address of all the static variables in the program; the linker will bind the static variables to consecutive addresses starting at the base address. Programmers who are placing their applications in ROM memory (such as a BIOS ROM for a PC) often employ this scheme.

8.3.1.4 Binding at Load Time

The most common form of static binding occurs at load time. Executable formats such as Microsoft's PE/COFF and Linux's ELF usually contain relocation information embedded in the executable file. The operating system, when it loads the application into memory, will decide where to place the block of static variable objects and will then patch all the addresses within instructions that reference those static objects. This allows the loader (for example, the operating system) to assign a different address to a static object each time it loads it into memory.

8.3.1.5 Static Variable Binding

A *static variable* is one that has a memory address bound to it prior to program execution. Static variables enjoy a couple of advantages over other variable types. Because the compiler knows the address of the variable prior to runtime, the compiler can often use an *absolute addressing mode* or some

other simple addressing mode to access that variable. Static variable access is often more efficient than other variable accesses because no additional setup is needed to access a static variable.[1]

Another feature of static variables is that they retain any value bound to them until you explicitly bind another value or until the program terminates. This means that static variables retain values while other events (such as procedure activation and deactivation) occur. Different threads in a multithreaded application can also share data using static variables.

Static variables also have a few disadvantages worth mentioning. First of all, because the lifetime of a static variable matches that of the program, static variables consume memory the entire time the program is running. This is true even if the program no longer requires the value held by the static object. Another disadvantage to static variables (particularly when using the absolute addressing mode) is that the entire absolute address must usually be encoded as part of the instruction, making the instruction much larger. Indeed, on most RISC processors an absolute addressing mode isn't even available because you cannot encode an absolute address in a single instruction.

Another disadvantage to using static variables is that code that uses static objects is not *reentrant* (meaning two threads or processes can be concurrently executing the same code sequence); more effort is required to use that code in a multithreaded environment (where two copies of a section of code could be executing simultaneously, both accessing the same static object). However, multithreaded operation introduces a lot of complexity that I don't want to get into here, so I'll ignore this issue for now. See any good textbook on operating system design or concurrent programming for more details concerning the use of static objects. *Foundations of Multithreaded, Parallel, and Distributed Programming* by Gregory R. Andrews (Addison-Wesley, 1999) is a good place to start.

The following example demonstrates the use of static variables in a C program and shows the 80x86 code that the Borland C++ compiler generates to access those variables:

```
#include <stdio.h>

static int i = 5;
static int j = 6;

int main( int argc, char **argv )
{

    i = j + 3;
    j = i + 2;
    printf( "%d %d", i, j );
    return 0;
```

[1] At least, on an 80x86 CPU or some other CPU that supports absolute addresses. Some RISC processors do not support absolute addressing, so the program must set up a "static frame pointer" or "global frame register" when the program first begins execution, but this only has to be done once, so we can ignore the performance issues associated with this.

```
}


; Following are the memory declarations
; for the 'i' and 'j' variables. Note that
; these are declared in the global '_DATA'
; section.

_DATA   segment dword public use32 'DATA'
        align   4
_i      label   dword
        dd      5
        align   4
_j      label   dword
        dd      6
_DATA   ends

_TEXT   segment dword public use32 'CODE'
_main   proc    near
?live1@0:
   ;
   ;    int main( int argc, char **argv )
   ;
        push      ebp
        mov       ebp,esp
   ;
   ;    {
   ;
   ;            i = j + 3;
   ;
@1:
        ; Load the EAX register with the
        ; current value of the global _j
        ; variable using the displacement-only
        ; addressing mode, add three to the
        ; value, and store into '_i':

        mov       eax,dword ptr [_j]
        add       eax,3
        mov       dword ptr [_i],eax
   ;
   ;            j = i + 2;
   ;
        ; Load the EDX register with the
        ; current value of the '_i' global
        ; variable using the displacement-
        ; only addressing mode, add two to
        ; this value, and store into
        ; '_j':

        mov       edx,dword ptr [_i]
        add       edx,2
        mov       dword ptr [_j],edx
```

```
   ;
   ;           printf( "%d %d", i, j );
   ;
        push      dword ptr [_j]
        push      dword ptr [_i]
        push      offset s@
        call      _printf
        add       esp,12
   ;
   ;           return 0;
   ;
        ; xor eax, eax sets the main function
        ; return value to zero.

        xor       eax,eax
   ;
   ;    }
   ;
@3:
@2:
        pop       ebp
        ret
_main   endp
_TEXT   ends
_DATA   segment dword public use32 'DATA'

; s@ is a string used by the printf function:

s@      label   byte
        ;       s@+0:
        db      "%d %d",0
        align   4
_DATA   ends
```

As the comments point out, the assembly language code the compiler emits uses the displacement-only addressing mode to access all the static variables.

8.3.2 Pseudo-Static Binding and Automatic Variables

Automatic variables have an address bound to them when a procedure or other block of code begins execution. The program releases that storage when the block or procedure completes execution. Such objects are called *automatic variables* because the runtime code automatically allocates and deallocates storage for them, as needed.

In most programming languages, automatic variables use a combination of static and dynamic binding known as *pseudo-static binding*. The compiler assigns an offset from a base address to a variable name during compilation. At runtime the offset always remains fixed, but the base address can vary. For example, a procedure or function allocates storage for a block of local variables and then accesses the local variables at fixed offsets from the start

of that block of storage. Although the compiler cannot determine the final memory address of the variable at runtime, it can select an offset that never changes during program execution, hence the name *pseudo-static*.

Some programming languages use the term *local variables* in place of automatic variables. A local variable is one whose name is statically bound to a given procedure or block (that is, the scope of the name is limited to that procedure or block of code). Therefore, *local* is a static attribute in this context. It's easy to see why the terms *local variable* and *automatic variable* are often confused. In some programming languages, such as Pascal, local variables are always automatic variables and vice versa. Nonetheless, always keep in mind that the *local* attribute is a static attribute, while the *automatic* attribute is a dynamic one.

Automatic variables have a couple of important advantages. First, they only consume storage while the procedure or block containing them is executing. This allows multiple blocks and procedures to share the same pool of memory for their automatic variable needs. Although some extra code must execute in order to manage automatic variables (in a memory structure known as an *activation record*), this only requires a few machine instructions on most CPUs and only has to be done once for each procedure/block entry and exit. While in certain circumstances, the cost can be significant, the extra time and space needed to set up and tear down the activation record is usually inconsequential. Another advantage of automatic variables is that they often use a base-plus-offset addressing mode, where the base of the activation record is kept in a register and the offsets into the activation record are small (often 256 bytes or fewer). Therefore, CPUs don't have to encode a full 32-bit or 64-bit address as part of the machine instruction—just an 8-bit (or other small) displacement, yielding shorter instructions. It's also worth noting that automatic variables are "thread-safe," and code that uses automatic variables can be reentrant. This is because each thread maintains its own stack space (or similar data structure) where compilers maintain automatic variables; therefore, each thread will have its own copy of any automatic variables the program uses.

Automatic variables do have some disadvantages. If you want to initialize an automatic variable, you have to use machine instructions to do so. You cannot initialize an automatic variable, as you can static variables, when the program loads into memory. Also, any values maintained in automatic variables are lost whenever you exit the block or procedure containing them. As noted in the previous paragraph, automatic variables require a small amount of overhead; some machine instructions must execute in order to build and destroy the activation record containing those variables.

Here's a short C example that uses automatic variables and the 80x86 assembly code that the Microsoft Visual C++ compiler produces for it:

```
#include <stdio.h>


int main( int argc, char **argv )
{
```

```
    int i;
    int j;

    j = 1;
    i = j + 3;
    j = i + 2;
    printf( "%d %d", i, j );
    return 0;
}
```

Assembly code emitted for the previous C code:

```
; Data emitted for the string constant
; in the printf function call:

_DATA   SEGMENT
$SG790  DB      '%d %d', 00H
_DATA   ENDS


PUBLIC  _main
EXTRN   _printf:NEAR
; Function compile flags: /Ods

_TEXT   SEGMENT
_j$ = -8
_i$ = -4
_argc$ = 8
_argv$ = 12
_main   PROC NEAR
; File g:\t.c
; Line 7
;
; Build the "activation record" that
; holds the automatic (local) variables:

        push    ebp
        mov     ebp, esp
        push    ecx ; Storage for _i on stack
        push    ecx ; Storage for _j on stack

; Line 13 // j = 1;

        mov     DWORD PTR _j$[ebp], 1

; Line 14 // i = j + 3;

        mov     eax, DWORD PTR _j$[ebp]
        add     eax, 3
        mov     DWORD PTR _i$[ebp], eax

; Line 15 // j = i + 2;
```

```
        mov     eax, DWORD PTR _i$[ebp]
        inc     eax
        inc     eax
        mov     DWORD PTR _j$[ebp], eax

; Line 16 // printf function call

        push    DWORD PTR _j$[ebp]
        push    DWORD PTR _i$[ebp]
        push    OFFSET FLAT:$SG790
        call    _printf
        add     esp, 12    ; 0000000cH

; Line 17 // Return zero as function result.

        xor     eax, eax

; Line 18 // Deallocates activation record

        leave

; Returns from main.

        ret     0
_main   ENDP
_TEXT   ENDS
```

Note that when accessing automatic variables, the assembly code uses a base-plus-displacement addressing mode (for example, `_j$[ebp]`). This addressing mode is often shorter than the displacement-only addressing mode that static variables use (assuming, of course, that the offset to the automatic object is within 127 bytes of the base address held in EBP).

8.3.3 Dynamic Binding and Dynamic Variables

A *dynamic variable* is one that has storage bound to it at runtime. In some languages, the application programmer is completely responsible for binding addresses to dynamic objects; in other languages, the runtime system automatically allocates and deallocates storage for a dynamic variable.

Dynamic variables are generally those allocated on the heap via a memory allocation function such as `malloc` or `new`. The compiler has no way of determining the runtime address of a dynamic object. Therefore, the program must always refer to a dynamic object indirectly by using a pointer.

The big advantage to dynamic variables is that the application controls their lifetimes. Dynamic variables consume storage only as long as necessary, and the runtime system can reclaim that storage when the variable no longer requires it. Unlike automatic variables, the lifetime of a dynamic variable is not tied to the lifetime of some other object, such as a procedure or code block entry and exit. Memory is bound to a dynamic variable at the point

the variable first needs it, and the memory can be released at the point the variable no longer needs it.[2] For variables that require considerable storage, dynamic allocation can make efficient use of memory as dynamically allocated variables hold onto the memory only as long as necessary.

Another advantage to dynamic variables is that most code references dynamic objects using a pointer. If that pointer value is already sitting in a CPU register, the program can usually reference that data using a short machine instruction, requiring no extra bits to encode an offset or address.

Dynamic variables have several disadvantages. First, usually some storage overhead is necessary to maintain dynamic variables. Static and automatic objects usually don't require extra storage associated with each such variable appearing in a program; the runtime system, on the other hand, often requires some number of bytes to keep track of each dynamic variable present in the system. This overhead ranges anywhere from 4 or 8 bytes to many dozens of bytes (in an extreme case) and keeps track of things like the current memory address of the object, the size of the object, and its type. If you're allocating small objects, like integers or characters, the amount of storage required for bookkeeping purposes could exceed the storage that the actual data requires. Also, most languages reference dynamic objects using pointer variables; as such, some additional storage is required by the pointer variable above and beyond the actual storage for the dynamic data.

Another problem with dynamic variables is performance. Because dynamic data is usually found in memory, the CPU has to access memory (which is slower than cached memory) on nearly every dynamic variable access.[3] Even worse, accessing dynamic data often requires two memory accesses—one to fetch the pointer's value and one to fetch the dynamic data, indirectly through the pointer. Another problem is that managing the *heap*, the place where the runtime system keeps the dynamic data, can also be expensive. Whenever an application requests storage for a dynamic object, the runtime system has to search for a contiguous block of free memory large enough to satisfy the request. This search operation can be expensive, depending on the organization of the runtime heap (which affects the amount of overhead storage associated with each dynamic variable). Furthermore, when releasing a dynamic object, the runtime system may need to execute some code in order to make that storage available for use by other dynamic objects. These runtime heap allocation and deallocation operations are usually far more expensive than allocating and deallocating a block of automatic variables during procedure entry/exit.

Another problem with dynamic variables that should be considered here is that some languages (e.g., Pascal and C/C++) require the application programmer to explicitly allocate and deallocate storage for dynamic variables. Because the allocation and deallocation is not automatic, defects can creep into the code because of errors made by the application programmer. This is

[2] In practice, many runtime systems will not bother breaking the address binding until the system actually needs the storage for another purpose, but this issue is not important here.

[3] Some compilers are smart enough to keep some dynamic data in registers, avoiding memory in certain cases, but in many cases the runtime code will have to access main memory when referencing dynamic data.

why languages such as C# attempt to handle dynamic allocation automatically for the programmer, even though this can be more expensive (slower). Here's a short example in C that demonstrates the kind of code that the Microsoft Visual C++ compiler will generate in order to access dynamic objects allocated with `malloc`.

```
#include <stdlib.h>
#include <stdio.h>


int main( int argc, char **argv )
{

    int *i;
    int *j;


    i = malloc( sizeof( int ) );
    j = malloc( sizeof( int ) );
    *i = 1;
    *j = 2;
    printf( "%d %d", *i, *j );
    free( i );
    free( j );
    return 0;
}
```

Here's the machine code the compiler generates, including manually inserted comments that describe the extra work needed to access dynamically allocated objects:

```
_DATA   SEGMENT
$SG1139 DB      '%d %d', 00H
_DATA   ENDS
PUBLIC  _main
EXTRN   _free:NEAR
EXTRN   _malloc:NEAR
EXTRN   _printf:NEAR
; Function compile flags: /Ods
_TEXT   SEGMENT
_j$ = -8
_i$ = -4
_argc$ = 8
_argv$ = 12
_main   PROC NEAR
; File g:\t.c

; Line 8 // Construct the activation record

        push    ebp
        mov     ebp, esp
        push    ecx ; Allocates storge for
        push    ecx ; _i and _j.
```

```
; Line 14
; Call malloc and store the returned
; pointer value into the _i variable:

        push    4
        call    _malloc
        pop     ecx
        mov     DWORD PTR _i$[ebp], eax

; Line 15
; Call malloc and store the returned
; pointer value into the _j variable:

        push    4
        call    _malloc
        pop     ecx
        mov     DWORD PTR _j$[ebp], eax

; Line 16
; Store 1 into the dynamic variable pointed
; at by _i. Note that this requires two
; instructions.

        mov     eax, DWORD PTR _i$[ebp]
        mov     DWORD PTR [eax], 1

; Line 17
; Store 2 into the dynamic variable pointed
; at by _j. This also requires two instructions.

        mov     eax, DWORD PTR _j$[ebp]
        mov     DWORD PTR [eax], 2

; Line 18
; Call printf to print the dynamic variables'
; values:

        mov     eax, DWORD PTR _j$[ebp]
        push    DWORD PTR [eax]
        mov     eax, DWORD PTR _i$[ebp]
        push    DWORD PTR [eax]
        push    OFFSET FLAT:$SG1139
        call    _printf
        add     esp, 12

; Free the two variables
;
; Line 19
        push    DWORD PTR _i$[ebp]
        call    _free
        pop     ecx
; Line 20
        push    DWORD PTR _j$[ebp]
        call    _free
```

```
        pop     ecx


; Line 21
; Return a function result of zero:

        xor     eax, eax

; Line 22
; Deallocate the activation record and
; return from main.

        leave
        ret     0
_main   ENDP
_TEXT   ENDS
END
```

As you can see, a lot of extra work is needed to access dynamically allocated variables via a pointer.

8.4 Common Primitive Data Types

Computer data always has a data type attribute that describes how the program interprets that data. The data type also determines the size (in bytes) of the data in memory. Data types can be divided into two classes: those that the CPU can hold in a CPU register and operate upon directly and those that are composed of the following smaller data types. I'll use the term *primitive data type* to describe atomic objects upon which the CPU may operate directly, and I'll use the term *composite data types* to describe those aggregate objects made up of smaller, primitive data types. In the following sections we'll review (from *Volume 1*) the primitive data types found on most modern CPUs, and in the next chapter I'll begin discussing composite data types.

8.4.1 Integer Variables

Most programming languages provide some mechanism for storing integer values in memory variables. In general, a programming language uses either unsigned binary representation, two's-complement representation, or binary-coded decimal representation (or a combination of these) to represent integer values.

Perhaps the most fundamental property of an integer variable in a programming language is the number of bits allocated to represent that integer value. In most modern programming languages, the number of bits used to represent an integer value is usually 8, 16, 32, 64, or some other power of 2. Many languages only provide a single size for representing integers, but some languages let you select from one of several different sizes. You choose the size based on the range of values you want to represent, the amount of memory you want the variable to consume, and the performance of arithmetic operations involving that value. Table 8-1 lists some common sizes and ranges for various signed, unsigned, and decimal integer variables.

Table 8-1: Common Integer Sizes and Their Ranges

Size, in Bits	Representation	Unsigned Range
8	Unsigned	0..255
	Signed	–128..+127
	Decimal	0..99
16	Unsigned	0..65,535
	Signed	–32768..+32,767
	Decimal	0..9999
32	Unsigned	0..4,294,967,295
	Signed	–2,147,483,648..+2,147,483,647
	Decimal	0..99999999
64	Unsigned	0..18,446,744,073,709,551,615
	Signed	–9,223,372,036,854,775,808..+9,223,372,036,854,775,807
	Decimal	0..9999999999999999
128	Unsigned	0..340,282,366,920,938,463,463,374,607,431,768,211,455
	Signed	–170,141,183,460,469,231,731,687,303,715,884,105,728 .. +170,141,183,460,469,231,731,687,303,715,884,105,727
	Decimal	0..99,999,999,999,999,999,999,999,999,999,999,999

Not all languages will support all of these different sizes (indeed, to support all of these different sizes in the same program, you would probably have to use assembly language). As noted earlier, some languages provide only a single size, which is usually the processor's native integer size (that is, the size of a CPU general-purpose integer register).

Languages that do provide multiple integer sizes often don't give you an explicit choice of sizes from which to choose. For example, the C programming language provides up to four different integer sizes: `char` (which is always 1 byte), `short`, `int`, and `long`. With the exception of the `char` type, C does not specify the sizes of these integer types other than to state that short integers are less than or equal to `int` objects in size, and `int` objects are less than or equal to `long` integers in size. (In fact, all three could be the same size.) C programs that depend on integers being a certain size may fail when compiled with different compilers that don't use the same sizes as the first compiler.

While it may seem inconvenient that various programming languages avoid providing an exact specification of the size of an integer variable in the language definition, keep in mind that this ambiguity is intentional. When one declares an "integer" variable in a given programming language, the language leaves it up to the compiler's implementer to choose the *best* size for that integer, based on performance and other considerations. The definition of "best" may change based on the CPU for which the compiler generates code. For example, a compiler for a 16-bit processor may choose to implement 16-bit integers because the CPU processes them most efficiently. A compiler for a 32-bit processor, however, may choose to implement 32-bit integers (for the same reason). Languages that specify the exact size of various

integer formats (such as Java) can suffer as processor technology marches along and it becomes more efficient to process larger data objects. For example, when the world switched from 16-bit processors to 32-bit processors in general-purpose computer systems, it was actually faster to do 32-bit arithmetic on most of the newer processors. Therefore, compiler writers redefined "integer" to mean "32-bit integer" in order to maximize the performance of programs employing integer arithmetic.

Some programming languages provide support for unsigned integer variables as well as signed integers. At first glance, it might seem that the whole purpose behind supporting unsigned integers is to provide twice the number of positive values when negative values aren't required. In fact, there are many other reasons why great programmers might choose unsigned over signed integers when writing efficient code.

On some CPUs, unsigned integer multiplication and division are faster than their signed counterparts. Comparing values within the range $0..n$ can be done more efficiently using unsigned integers rather than signed integer (requiring only a single comparison against n in the unsigned case); this is especially important when checking bounds of array indices when the array's element indexes begin at zero.

Many programming languages will allow you to include variables of different sizes within the same arithmetic expression. The compiler will automatically sign-extend or zero-extend operands to the larger size within an expression as needed to compute the final result. The problem with this automatic conversion is that it hides the fact that extra work is required when processing the expression, and the expressions themselves don't explicitly show this. An assignment statement such as

```
    x = y + z - t;
```

could be a short sequence of machine instructions if the operands are all the same size, or it could require some additional instructions if the operands have different sizes. For example, consider the following C code:

```
#include <stdio.h>

static char c;
static short s;
static long l;

static long a;
static long b;
static long d;

int main( int argc, char **argv )
{

    l = l + s + c;
    printf( "%ld %ld %ld", l, s, c );

    a = a + b + d;
```

```
    printf( "%ld %ld %ld", a, b, d );

    return 0;
}
```

Compiled with the Borland C++ compiler, you get the following two assembly language sequences for the two assignment statements:

```
;           l = l + s + c;
;
@1:
        movsx     eax,word ptr [_s]
        add       eax,dword ptr [_l]
        movsx     edx,byte ptr [_c]
        add       eax,edx
        mov       dword ptr [_l],eax

;           a = a + b + d;
;
        mov       edx,dword ptr [_a]
        add       edx,dword ptr [_b]
        add       edx,dword ptr [_d]
        mov       dword ptr [_a],edx
```

As you can see, the statement that operates on variables whose sizes are all the same uses fewer instructions than the one that mixes operand sizes in the expression.

Another thing to note, when using different-sized integers in an expression, is that not all CPUs support all operand sizes as efficiently. While it should be fairly obvious that using an integer size that is larger than the CPU's general-purpose integer registers will produce inefficient code, it might not be quite as obvious that using *smaller* integer values can be inefficient as well. Many RISC CPUs only work on operands that are exactly the same size as the general-purpose registers. Smaller operands must first be zero-extended or sign-extended to the size of a general-purpose register prior to any calculations involving those values. Even on CISC processors, such as the 80x86, that have hardware support for different sizes of integers, using certain sizes can be more expensive. For example, under 32-bit operating systems, instructions that manipulate 16-bit operands require an extra *opcode prefix byte* and are, therefore, larger than instructions that operate on 8-bit or 32-bit operands.

8.4.2 Floating-Point/Real Variables

Like integers, many HLLs provide multiple floating-point variable sizes. Most languages provide at least two different sizes, a 32-bit single-precision floating-point format and a 64-bit double-precision floating-point format, based on the IEEE 754 floating-point standard. A few languages provide 80-bit floating-point variables, based on Intel's 80-bit extended-precision floating-point format, but such usage is becoming rare.

Different floating-point formats trade off space and performance for precision. Calculations involving smaller floating-point formats are usually quicker than calculations involving the larger formats. However, you give up precision to achieve improved performance and size savings (see *Write Great Code, Volume 1,* Chapter 4 for details).

As with expressions involving integer arithmetic, you should avoid mixing different-sized floating-point operands in an expression. The CPU (or FPU) must convert all floating-point values to the same format before using them. This can involve additional instructions (consuming more memory) and additional time. Therefore, you should try to use the same floating-point types throughout an expression, wherever possible.

Conversion between integer and floating-point formats is another expensive operation you should avoid. Modern HLLs attempt to keep variables' values in registers as much as possible. Unfortunately, on most modern CPUs it is impossible to move data between the integer and floating-point registers without first copying that data to memory (which is expensive, because memory access is slow compared with register access). Furthermore, conversion between integer and floating-point numbers often involves several specialized instructions. All of this consumes time and memory. Whenever possible, avoid these conversions.

8.4.3 Character Variables

Standard character data in most modern HLLs consumes one byte per character. On CPUs that support byte addressing, such as the Intel 80x86 processor, a compiler can reserve a single byte of storage for each character variable and efficiently access that character variable in memory. Some RISC CPUs, however, cannot access data in memory except in 32-bit chunks (or some other size other than 8 bits).

For CPUs that cannot address individual bytes in memory, HLL compilers usually reserve 32 bits for a character variable and only use the LO byte of that double-word variable for the character data. Because few programs have a large number of scalar character variables,[4] the amount of space wasted is hardly an issue in most systems. However, if you have an unpacked array of characters, the wasted space can become significant. I'll return to this issue in Chapter 9.

Modern programming languages support the Unicode character set. Unicode characters require 2 bytes of memory to hold the character's data value. On CPUs that support byte or word addressing, HLL compilers generally reserve only 2 bytes for a Unicode character variable. On CPUs that cannot efficiently access objects smaller than 32 bits, HLL compilers usually reserve 32 bits and use only the LO 16 bits for the Unicode character data.

Lately, because 16 bits cannot encode a sufficient number of characters to represent all the world's different alphabets and symbol sets, applications have begun using multibyte character sets such as UTF-8. These encode individual characters using a variable-length string of 1 to 5 characters (see Chapter 10).

[4] *Scalar*, in this context, means "not an array of characters."

8.4.4 Boolean Variables

A Boolean variable requires only a single bit to represent the two values *True* or *False*. HLLs will usually reserve the smallest amount of memory possible for such variables (a byte on machines that support byte addressing, and a larger amount of memory on those CPUs that can only address words or double words).

Although most HLL compilers usually reserve the smallest amount of addressable memory possible for a Boolean variable, this isn't always the case. Some languages (like FORTRAN) allow you to create multibyte Boolean variables (for example, the FORTRAN LOGICAL*4 data type).

Some languages (C for example) don't support an explicit Boolean data type. They use an integer data type to represent Boolean values. In such languages, you get to choose the size of your Boolean variables by choosing the size of the integer you use to hold the Boolean value. For example, in a typical 32-bit implementation of the C/C++ languages, you can define 1-byte, 2-byte, or 4-byte Boolean values as shown here:[5]

C Integer Data Type	Size of Boolean Object
char	1 byte
short int	2 bytes
long int	4 bytes

Some languages, under certain circumstances, will use only a single bit of storage for a Boolean variable when that variable is a field of a record or an element of an array. I'll return to this discussion in Chapter 9 when considering composite data structures.

8.5 Variable Addresses and High-level Languages

The organization, class, and type of variables in your programs can affect the efficiency of the code that a compiler produces. Additionally, issues like the order of declaration, the size of the object, and the placement of the object in memory can have a big impact on the running time of your programs. In this section, I'll describe how you can organize your variable declarations to produce efficient code.

As for immediate constants encoded in machine instructions, many CPUs provide specialized addressing modes that access memory more efficiently than other, more general, addressing modes. Just as you can reduce the size and improve the speed of your programs by carefully selecting the constants you use, you can make your programs more efficient by carefully choosing how you declare variables. But whereas with constants you are primarily concerned with their values, with variables you must consider the address in memory where the compiler places those variables.

[5] Assuming, of course, that your C/C++ compiler uses 16-bit integers for short integers and 32-bit integers for long integers.

The 80x86 is a typical example of a CISC processor that provides multiple address sizes. When running on a modern 32-bit operating system like Linux or Windows, the 80x86 CPU supports three address sizes: 0-bit, 8-bit, and 32-bit. The 80x86 uses 0-bit displacements for register-indirect addressing modes. I'll ignore the 0-bit displacement addressing for the time being because 80x86 compilers generally don't use this particular addressing mode to access variables you explicitly declare in your code. The 8-bit and 32-bit displacement addressing modes are the more interesting ones for the current discussion.

8.5.1 Storage Allocation for Global and Static Variables

The 32-bit displacement is, perhaps, the easiest to understand. Variables you declare in your program, which the compiler allocates in memory rather than in a register, have to appear somewhere in memory. On most 32-bit processors, the address bus is 32 bits wide, so it takes a 32-bit address to access a variable at an arbitrary location in memory. An instruction that encodes this 32-bit address as part of the instruction can access any memory variable. The 80x86 provides the *displacement-only* addressing mode whose effective address is exactly the 32-bit constant embedded in the instruction.

A problem with 32-bit addresses (one that gets even worse as we move to 64-bit processors with a 64-bit address) is that the address winds up consuming the largest portion of the instruction's encoding. Certain forms of the displacement-only addressing mode on the 80x86, for example, have a 1-byte opcode and a 4-byte address. Therefore, 80 percent of the instruction's size is consumed by the address. On typical RISC processors, the situation is even worse. Because the instructions are uniformly 32 bits long on a typical RISC CPU, you cannot encode a 32-bit address as part of the instruction. In order to access a variable at an arbitrary 32-bit address in memory, you need to load the 32-bit address of that variable into a register and then use the register indirect addressing mode to access the memory variable. This could require three 32-bit instructions as Figure 8-2 demonstrates; that's expensive in terms of both speed and space.

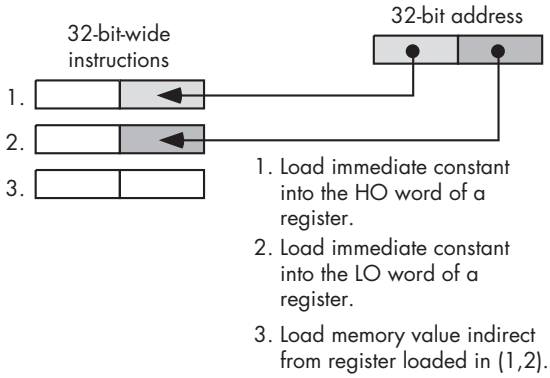

Figure 8-2: RISC CPU access of an absolute address

Because RISC CPUs don't run horribly slower than CISC processors, it should be obvious that compilers rarely generate code this bad. In reality, programs running on RISC CPUs often keep base addresses to blocks of objects in registers, so they can efficiently access variables in those blocks using short offsets from the base register. But how do compilers deal with arbitrary addresses in memory?

8.5.2 Using Automatic Variables to Reduce Offset Sizes

One way to avoid large instruction sizes with large displacements is to use an addressing mode with a smaller displacement. The 80x86, for example, provides an 8-bit displacement form for the base-plus-indexed addressing mode. This form allows you to access data at an offset of –128 through +127 bytes around a base address contained in a 32-bit register. RISC processors have similar features, although the number of displacement bits is usually larger (16 bits), allowing a greater range of addresses.

By pointing a 32-bit register at some base address in memory and placing your variables near that base address, you can use the shorter forms of these instructions so your program will be smaller and will run more quickly. Obviously, this isn't too difficult if you're working in assembly language and you have direct access to the CPU's registers. However, if you're working in an HLL, you may not have direct access to the CPU's registers and even if you did, you probably couldn't convince the compiler to allocate your variables at convenient addresses. How do you take advantage of this small-displacement addressing mode in your HLL programs? The answer is that you don't explicitly specify the use of this addressing mode, the compiler does it for you automatically.

Consider the following trivial function in Pascal:

```
function trivial( i:integer; j:integer ):integer;
var
    k:integer;
begin

    k := i + j;
    trivial := k;

end;
```

Upon entry into this function, the compiled code constructs an *activation record* (sometimes called a *stack frame*). An activation record is a data structure in memory where the system keeps the local data associated with a function or procedure. The activation record includes parameter data, automatic variables, the return address, temporary variables that the compiler allocates, and machine-state information (for example, saved register values). The runtime system allocates storage for an activation record on the fly and, in fact, two different calls to the procedure or function may place the activation record at different addresses in memory. In order to access the data in an activation record, most HLLs point a register (usually called the *frame pointer*) at the activation record, and then the procedure or function references

automatic variables and parameters at some offset from this frame pointer. Unless you have many automatic variables and parameters or your local variables[6] and parameters are quite large, these variables generally appear in memory at an offset that is near the base address. This means that the CPU can use a small offset when referencing variables near the base address held in the frame pointer. In the Pascal example given earlier, parameters `i` and `j` and the local variable `k` would most likely be within a few bytes of the frame pointer's address, so the compiler can encode these instructions using a small displacement rather than a large displacement. If your compiler allocates local variables and parameters in an activation record, all you have to do is arrange your variables in the activation record so that they appear near the base address of the activation record. But how do you do that?

Construction of an activation record begins in the code that calls a procedure. The caller places the parameter data (if any) in the activation record. Then the execution of an assembly language `call` instruction adds the return address to the activation record. At this point, construction of the activation record continues within the procedure itself. The procedure copies the register values and other important state information and then makes room in the activation record for local variables. The procedure must also update the frame-pointer register (e.g., EBP on the 80x86) so that it points at the base address of the activation record.

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```
procedure ARDemo( i:uns32; j:int32; k:dword ); @nodisplay;
var
    a:int32;
    r:real32;
    c:char;
    b:boolean;
    w:word;
begin ARDemo;
    .
    .
    .
end ARDemo;
```

Whenever an HLA program calls this `ARDemo` procedure, it builds the activation record by pushing the data for the parameters onto the stack. The calling code for this procedure will push the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the `i` parameter, then pushes the value for the `j` parameter, and finally pushes the data for the `k` parameter. After pushing the parameters, the program calls the `ARDemo` procedure. Immediately upon entry into the `ARDemo` procedure, the stack contains these four items arranged as shown in Figure 8-3, assuming the stack grows from high memory addresses to low memory addresses (as it does on most processors).

[6] Remember, in Pascal local variables are always automatic variables, so this discussion will use the two terms interchangeably.

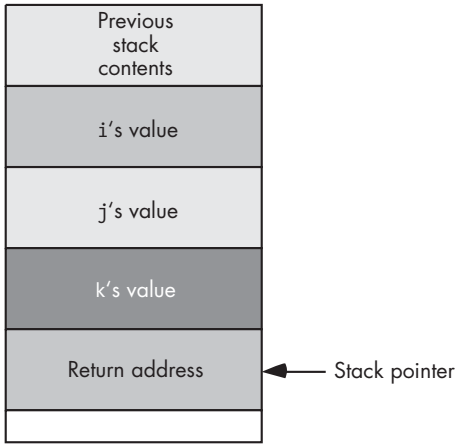

Figure 8-3: Stack organization immediately upon entry into ARDemo

The first few instructions in `ARDemo` will push the current value of the frame-pointer register (e.g., EBP on the 80x86) onto the stack and then copy the value of stack pointer (ESP on the 80x86) into the frame-pointer register. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 8-4 on the 80x86 CPU.

To access objects in the activation record you must use offsets from the frame-pointer register (EBP in Figure 8-4) to the desired object.

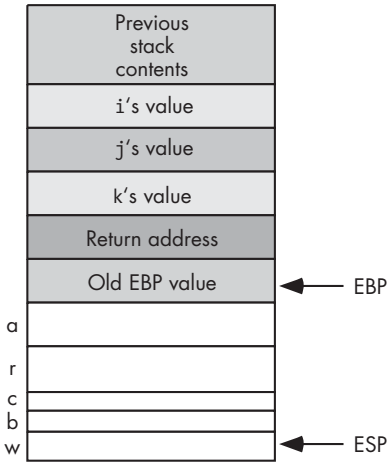

Figure 8-4: Activation record for ARDemo

The two items of immediate interest are the parameters and the local variables. You can access the parameters at positive offsets from the frame-pointer register; you can access the local variables at negative offsets from the frame-pointer register, as Figure 8-5 shows.

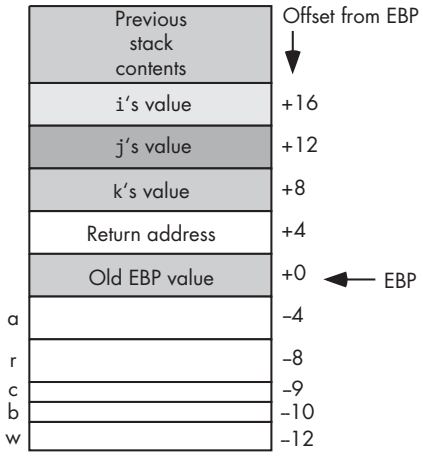

Figure 8-5: Offsets of objects in the ARDemo activation record on the 80x86

Intel specifically reserves the EBP (extended base pointer) to point at the base of the activation record. Therefore, compilers will typically use this register as the frame-pointer register when allocating activation records on the stack. Some compilers attempt to use the 80x86 ESP (stack pointer) register as the pointer to the activation record because this reduces the number of instructions in the program. Whether the compiler uses EBP, ESP, or some other register, the bottom line is that the compiler typically points some register at the activation record, and most of the local variables and parameters are near the base address of the activation record. That is the important issue for the discussion that follows.

As you can see in Figure 8-5, all the local variables and parameters in the `ARDemo` procedure are within 127 bytes of the frame-pointer register (EBP). This means that on the 80x86 CPU, an instruction that references one of these variables or parameters will be able to encode the offset from EBP using a single byte. Because of the way the program builds the activation record, parameters will appear at positive offsets from the frame-pointer register, and local variables will appear at negative offsets from the frame-pointer register.

For procedures that have only a few parameters and local variables, the CPU will be able to access all parameters and local variables using a small offset (that is, 8 bits on the 80x86, 16 bits on various RISC processors). Consider, however, the following C/C++ function:

```
int BigLocals( int i, int j );
{
    int array[256];
    int k;
        .
        .
        .
}
```

The activation record for this function appears in Figure 8-6. One difference you'll notice between this activation record and the ones for the Pascal and HLA functions is that C pushes its parameters on the stack in the reverse order (that is, it pushes the last parameter first, and it pushes the first parameter last). This difference, however, does not impact our discussion.

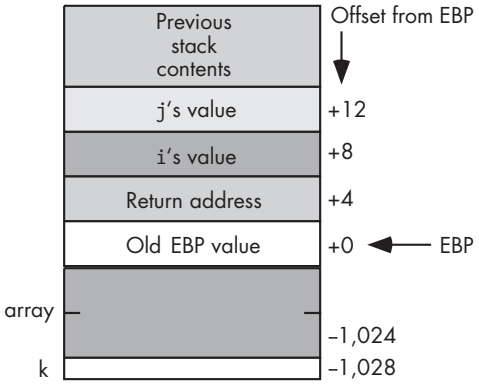

Figure 8-6: Activation record for `BigLocals` function

The important thing to note in Figure 8-6 is that the local variables `array` and `k` have large negative offsets. With offsets of –1,024 and –1,028 (assuming an integer is 32 bits), the displacements from EBP to `array` and `k` are well outside the range that the compiler can encode into a single byte on the 80x86. Therefore, the compiler will have no choice but to encode these displacements using a 32-bit value. Of course, this will make accessing these local variables in the function quite a bit more expensive.

Nothing can be done about the `array` variable in this example (no matter where you put it, the offset to the base address of the array will be at least 1,024 bytes from the activation record's base address). However, consider the activation record appearing in Figure 8-7.

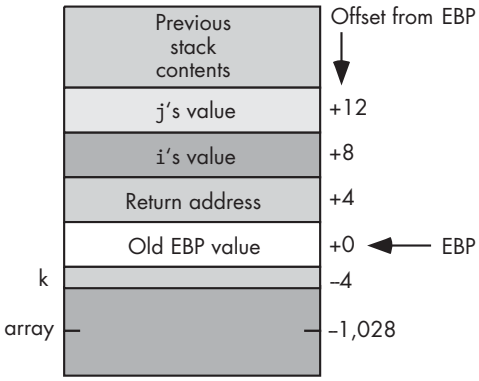

Figure 8-7: Another possible activation record layout for the `BigLocals` function

In this figure, the compiler has rearranged the local variables in the activation record. Although it will still take a 32-bit displacement to access the `array` variable, accessing `k` now uses an 8-bit displacement (on the 80x86) because `k`'s offset is –4. You can produce these offsets with the following code:

```
int BigLocals( int i, int j );
{
    int k;
    int array[256];
        .
        .
        .
}
```

In theory, this isn't a terribly difficult optimization for a compiler to do (rearranging the order of the variables in the activation record), so you'd expect the compiler to make this modification for you so that it can access as many local variables as possible using small displacements. In practice, not all compilers actually do this optimization for various technical and practical reasons (specifically, it can break some poorly written code that makes assumptions about the placement of variables in the activation record).

If you want to ensure that the maximum number of local variables in your procedure have the smallest possible displacements, the solution is trivial: declare all your 1-byte variables first, your 2-byte variables second, your 4-byte variables next, and so on up to the largest local variable in your function. Generally, though, you're probably more interested in reducing the size of the maximum number of instructions in your function rather than reducing the size of the offsets required by the maximum number of variables in your function. For example, if you have 128 1-byte variables and you declare these variables first, you'll only need a single byte displacement if you access them. However, if you never access these variables, the fact that they have a 1-byte displacement rather than a 4-byte displacement saves you nothing. The only time you save any space is when you actually access that variable's value in memory via some machine instruction that is using a 1-byte displacement rather than a 4-byte displacement. Therefore, to reduce your function's object code size, you want to maximize the number of instructions that use a small displacement. If you refer to a 100-byte array far more often than any other variable in your function, you're probably better off declaring that array first, even if it only leaves 28 bytes of storage (on the 80x86) for other variables that will use the shorter displacement.

RISC processors typically use a 16-bit offset to access fields of the activation record. Therefore, you have more latitude with your declarations when using a RISC chip (which is good, because when you do exceed the 16-bit limitation, accessing a local variable gets *really expensive*). Unless you're declaring one or more arrays that consume more than 32,768 bytes (combined), the typical compiler for a RISC chip is going to generate decent code.

This same argument applies to parameters as well as local variables. However, it's rare to find code passing a large data structure (by value) to a function because of the expense involved.

8.5.3 Storage Allocation for Intermediate Variables

Intermediate variables are those that are local to one procedure/function but global to another. You'll find intermediate variables in block-structured languages like Pascal/Delphi/Kylix, Ada, Modula-2, and HLA that support nested procedures. Consider the following example program in Pascal:

```
program nestedProcedures;
var
    globalVariable: integer;

    procedure procOne;
    var
        intermediateVariable: integer;

        procedure procTwo;
        var
            localVariable:integer;
        begin

            localVariable := intermediateVariable +
                                    globalVariable;
                .
                .
                .
        end; (* procTwo *)

    begin (* procOne *)
        .
        .
        .
    end; (* procOne *)

begin (* main program *)
        .
        .
        .
end. (* main program*)
```

As you can see in this code fragment, nested procedures can access variables found in the main program (that is, global variables) as well as variables found in procedures containing the nested procedure (that is, the intermediate variables). As you've seen, local variable access is inexpensive compared to global variable access (because you always have to use a larger offset to access global objects within a procedure). Intermediate variable access, as is done in the `procTwo` procedure, is expensive. The difference between local and global variable accesses is the size of the offset/displacement coded into the instruction—with local variables typically using a shorter offset than is possible for global objects. Intermediate accesses, on the other hand, typically require several machine instructions. This makes the instruction sequence that accesses an intermediate variable several times slower and several times larger than accessing a local (or even global) variable.

The problem with using intermediate variables is that the compiler must maintain either a linked list of activation records or a table of pointers to the activation records (this table is called the *display*) in order to reference intermediate objects. To access an intermediate variable, the `procTwo` procedure must either follow a chain of links (there would be only one link in this example) or it would have to do a table lookup in order to get a pointer to `procOne`'s activation record. Worse still, maintaining the display of this linked list of pointers isn't exactly cheap. The work needed to maintain these objects has to be done on every procedure/function entry and exit, even when the procedure or function doesn't access any intermediate variables on a particular call. Although there are, arguably, some software engineering benefits to using intermediate variables (having to do with information hiding) versus a global variable, keep in mind that access to intermediate objects is expensive.

8.5.4 Storage Allocation for Dynamic Variables and Pointers

Pointer access in an HLL provides another opportunity for optimization in your code. Pointers can be expensive to use but, under certain circumstances, they can actually make your programs more efficient by reducing displacement sizes.

A pointer is simply a memory variable whose value is the address of some other memory object (therefore, pointers are the same size as an address on the machine). Because most modern CPUs only support indirection via a machine register, indirectly accessing an object is typically a two-step process: First the code has to load the value of the pointer variable into a register and then the program has to refer (indirectly) to the object through that register.

Consider the following C/C++ code fragment and the corresponding HLA assembly code:

```
int *pi;
    .
    .
    .
i = *pi;    // Assume pi is initialized with a
            //  reasonable address at this point.
```

And here is the corresponding 80x86/HLA assembly code:

```
pi: pointer to int32;
    .
    .
    .
mov( pi, ebx );     // Again, assume pi has
mov( [ebx], eax );  //  been properly initialized
mov( eax, i );
```

Had pi been a regular variable rather than pointer object, this code could have dispensed with the mov([ebx], eax); instruction. Therefore, the use of this pointer variable has both increased the size of the program and reduced the execution speed by inserting an extra instruction into the code sequence that the compiler generates.

Note that if you indirectly refer to an object several times in close succession, then the compiler may be able to reuse the pointer value it has loaded into the register, thus amortizing the cost of the extra instruction across several different instructions. Consider the following C/C++ code sequence and the corresponding HLA code. Here is the C/C++ source code:

```
int *pi;
    .
    .   // Assume code in this area
    .   //  initializes pi appropriately.
    .
*pi = i;
*pi = *pi + 2;
*pi = *pi + *pi;
printf( "pi = %d\n", *pi );
```

Here's the corresponding 80x86/HLA code:

```
pi: pointer to int32;
    .
    . // Assume code in this area
    . //  initializes pi appropriately.
    .
// Extra instruction that we need to initialize EBX

mov( pi, ebx );

mov( i, eax );
mov( eax, [ebx] );  // This code can clearly be optimized;
mov( [ebx], eax );  //  we'll ignore that fact for the
add( 2, eax );      //  sake of the discussion here.
mov( eax, [ebx] );
mov( [ebx], eax );
add( [ebx], eax );
mov( eax [ebx] );
stdout.put( "pi = ", (type int32 [ebx]), nl );
```

Note that this code loads the actual pointer value into EBX only once. From that point forward the code will simply use the pointer value contained in EBX to reference the object at which pi is pointing. Of course, any compiler that can do this optimization can probably eliminate five redundant memory loads and stores from this assembly language sequence, but I'll assume that they aren't redundant for the time being. The first thing about this code you should note is that it didn't have to reload EBX with the value

of pi every time it wanted to access the object at which pi points. Therefore, we only have one instruction of overhead (mov(pi, ebx);) amortized across six of these instructions. That's not too bad at all.

Indeed, a good argument could be made that this code is more optimal than accessing a local or global variable directly. An instruction of the form

```
mov( [ebx], eax );
```

uses a 0-bit displacement encoded into the instruction. Therefore, this move instruction is only 2 bytes long rather than 3, 5, or even 6 bytes long. If pi is a local variable, then it's quite possible that the original instruction that copies pi into EBX is only 3 bytes long (a 2-byte opcode and a 1-byte displacement). Because instructions of the form mov([ebx], eax); are only 2 bytes long, it only takes three instructions to "break even" on the byte count using indirection rather than an 8-bit displacement. After the third instruction that references whatever pi points at, the code involving the pointer is actually shorter.

You can even use indirection to provide efficient access to a block of global variables. As noted earlier, the compiler generally cannot determine the address of a global object while it is compiling your program. Therefore, it has to assume the worst case and allow for the largest possible displacement/offset when generating machine code to access a global variable. Of course, you've just seen that you can reduce the size of the displacement value from 32 bits down to 0 bits by using a pointer to the object rather than accessing the object directly. Therefore, you could take the address of the global object (with the C/C++ & operator, for example) and then use indirection to access the variable. The problem with this approach is that it requires a register (a precious commodity on any processor, but especially on the 80x86 that has only six general-purpose registers to utilize). If you access the same variable many times in rapid succession, then this 0-bit displacement trick can make your code more efficient. However, it's somewhat rare to access the same variable a large number of times in a short sequence of code without also needing to access several other variables. Therefore, the compiler may have to flush the pointer from the register and reload the pointer value later (thereby reducing the efficiency of this approach). If you're working on a RISC chip with many registers, you can probably employ this trick to your advantage. On a processor with a limited number of registers, you won't be able to employ this trick as often.

8.5.5 Using Records/Structures to Reduce Instruction Offset Sizes

There is a trick that you can use to gain access to several variables with a single pointer: put all those variables into a structure, and then use the address of the structure. By accessing the fields of the structure via the pointer, you can get away with using smaller instructions to access the objects. This works almost exactly as you've seen for activation records (indeed, activation records are, literally, records that the program references indirectly via the *frame-pointer register*). About the only difference between accessing objects

indirectly in a user-defined record/structure and accessing objects in the activation record is that most compilers won't let you refer to fields in a user structure/record using negative offsets. Therefore, you're limited to about half the number of bytes that are normally accessible in an activation record. For example, on the 80x86 you can access the object at offset zero from a pointer using a 0-bit displacement and objects at offsets 1..+127 using a single byte displacement. Consider the following C/C++ example that uses this trick:

```
typedef struct vars
{
    int i;
    int j;
    char *s;
    char name[20];
    short t;
};

static vars v;
vars *pv = &v;  // Initialize pv with the address of v.
        .
        .
        .
    pv->i = 0;
    pv->j = 5;
    pv->s = &pv->name;
    pv->t = 0;
    strcpy( pv->name, "Write Great Code!" );
        .
        .
        .
```

A well-designed compiler will load the value of `pv` into a register exactly once for this code fragment. Because all the fields of the `vars` structure are within 127 bytes of the base address of the structure in memory, an 80x86 compiler can emit a sequence of instructions that require only 1-byte offsets, even though the `v` variable itself is a static/global object. Note, by the way, that the first field in the `vars` structure is special. Because this is at offset zero in the structure, this allows the use of a 0-bit displacement when accessing this field. Therefore, it's a good idea to put your most-often-referenced field first in a structure if you're going to refer to that structure indirectly.

Using indirection in your code does come at a cost. On a limited-register CPU such as the 80x86, using this trick will tie up a register for some period and that may, effectively cause the compiler to generate worse code. If the compiler must constantly reload the register with the address of the structure in memory, you can watch the savings that this trick buys you evaporate rather quickly. When using this trick, you should look at the assembly code the compiler generates and verify that you're actually saving something. Tricks such as using pointers to structures vary in effectiveness across different processors (and different compilers for the same processor). Therefore, it's a really good idea to look at the code generated by your compiler when using a trick such as this in order to make sure that your trick is actually saving you something rather than costing you something.

8.5.6 Register Variables

While on the subject of registers, it's worthwhile to point out one other 0-bit displacement way to access variables in your programs. You can also access your variables by keeping them in machine registers. Machine registers are always the most efficient place to keep variables and parameters. Unfortunately, only in assembly language and, to a limited extent, C/C++, do you have any control over whether the compiler should keep a variable or parameter in a register. In some respects, this is not bad. Good compilers do a much better job of register allocation than the casual programmer does. However, an expert programmer can do a better job of register allocation than a compiler because the expert programmer understands the data the program will be processing and the frequency of access to a particular memory location. (And of course, the expert programmer can first look at what the compiler is doing, whereas the compiler doesn't have the benefit of first looking at what the expert programmer has done.)

Some languages, such as Delphi and Kylix, provide limited support for programmer-directed register allocation. In particular, the Delphi/Kylix compilers provide a compiler option that you can use to tell the compiler to pass the first three (ordinal) parameters for a function or procedure in the EAX, EDX, and ECX registers. This is known as the *fastcall calling convention* and several C/C++ compilers support it as well (e.g., Borland's C++ and C++Builder compilers).

In Delphi/Kylix and certain other languages, control of the fastcall parameter passing convention is the only control you get. The C/C++ language, however, provides the `register` keyword, a storage specifier (much like the `const`, `static`, and `auto` keywords) that tells the compiler that the programmer expects to use the variable frequently and the compiler should attempt to keep the variable in a register. Note that the compiler can choose to ignore the `register` keyword (in which case the compiler reserves variable storage using automatic allocation). Many compilers ignore the `register` keyword altogether because the compiler's authors feel that they can do a better job of register allocation than any programmer (a somewhat arrogant assumption). Of course, on some register-starved machines such as the 80x86, there are so few registers to work with that it might not even be possible to allocate a variable to a register throughout the execution of some function. Nevertheless, some compilers do respect the programmer's wishes and *will* allocate a few variables in registers if you request that they do so.

Most RISC compilers reserve several registers for passing parameters and several registers for local variables. Therefore, it's a good idea (if possible) to place the parameters you access most frequently first in the parameter declaration because they're probably the ones that the compiler would allocate in a register.[7] The same is true for local variable declarations. Always declare frequently used local variables first because many compilers may allocate those (ordinal) variables in registers.

[7] Many optimizing compilers are smart enough to choose which variables they keep in registers based on how the program uses those variables.

One problem with compiler register allocation is that it is static. That is, the compiler determines which variables to place in registers based on an analysis of your source code during compilation, not during runtime. Compilers often make assumptions (that are usually correct) like "this function references variable xyz far more often than any other variable, so it's a good candidate for a register variable." Indeed, by placing the variable in a register, the compiler will certainly reduce the *size* of the program. However, it could also be the case that all those references to xyz sit in code that rarely, if ever, executes. Although the compiler might save some space (by emitting smaller instructions to access registers rather than memory), the code won't run appreciably faster. After all, if the code rarely or never executes, then making that code run faster does not contribute much to the execution time of the program. On the other hand, it's also quite possible to bury a single reference to some variable in a deeply nested loop that executes many times. With only one reference in the entire function, the compiler's optimizer may overlook the fact that the executing program references the variable frequently. Although compilers have gotten smarter about handling variables inside loops, the fact is that no compiler can predict how many times an arbitrary loop will execute at runtime. Human beings are much better at predicting this sort of behavior (or, at least, measuring it with a profiler); therefore, humans are the best ones to make better decisions concerning variable allocation in registers.

8.6 Variable Alignment in Memory

On many processors (particularly RISC), there is another efficiency concern you must take into consideration. Many modern processors will not let you access data at an arbitrary address in memory. Instead, all accesses must take place on some native boundary (usually 4 bytes) that the CPU supports. Even when a CISC processor allows memory accesses at arbitrary byte boundaries, it's often more efficient to access primitive objects (bytes, words, and double words) on a boundary that is a multiple of the object's size (see Figure 8-8).

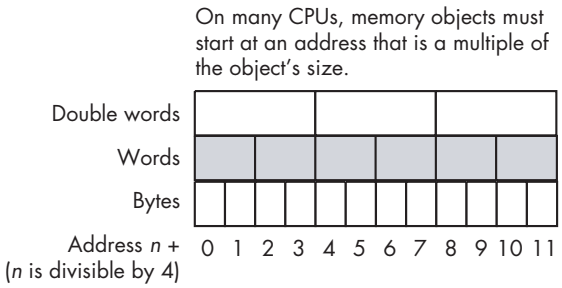

Figure 8-8: Variable alignment in memory

If the CPU supports unaligned accesses—that is, if the CPU allows you to access a memory object on a boundary that is not a multiple of the object's primitive size—then it should be possible to pack the variables into the

activation record. This way, you would obtain the maximum number of variables having a short offset. However, because unaligned accesses are sometimes slower than aligned accesses, many optimizing compilers will insert *padding bytes* into the activation record in order to ensure that all variables are aligned on a reasonable boundary for their native size (see Figure 8-9). This trades off slightly better performance for a slightly larger program.

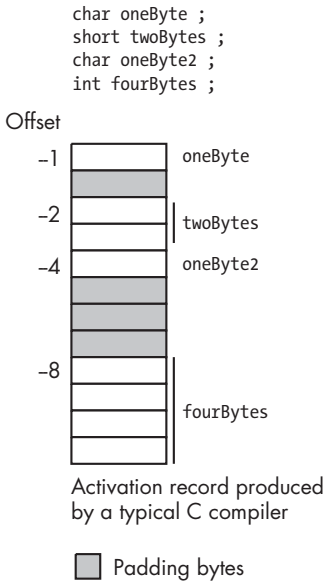

Figure 8-9: Padding bytes in an activation record

However, if you put all your double-word declarations first, your word declarations second, your byte declarations third, and your array/structure declarations last, you can improve both the speed and size of your code. The compiler will usually ensure that the first local variable you declare appears at a reasonable boundary (typically a double-word boundary). By declaring all your double-word variables first, you ensure that all such variables appear at an address that is a multiple of 4 (because compilers usually allocate adjacent variables in your declarations in adjacent locations in memory). The first word-sized object you declare will also appear at an address that is a multiple of 4, and that means its address is also a multiple of 2 (which is best for word accesses). By declaring all your word variables together, you ensure that each word variable appears at an address that is a multiple of 2. On processors that allow byte access to memory, the placement of the byte variables (with respect to efficiently accessing the byte data) is irrelevant. By declaring all your local byte variables last in a procedure or function, you generally ensure that such declarations do not impact the performance of the double-word and word variables you also use in the function. Figure 8-10 shows what a typical activation record will look like if you declare your variables as in the following function.

```
int someFunction( void )
{
    int d1;   // Assume ints are 32-bit objects
    int d2;
    int d3;
    short w1; // Assume shorts are 16-bit objects
    short w2;
    char b1;  // Assume chars are 8-bit objects
    char b2;
    char b3;
        .
        .
        .
} // end someFunction
```

Note in Figure 8-10 how all the double-word variables (`d1`, `d2`, and `d3`) begin at addresses that are multiples of 4 (–4, –8, and –12). Also, notice how all the word-sized variables (`w1` and `w2`) begin at addresses that are multiples of 2 (–14 and –16). The byte variables (b1, b2, and b3) begin at arbitrary addresses in memory (both even and odd addresses).

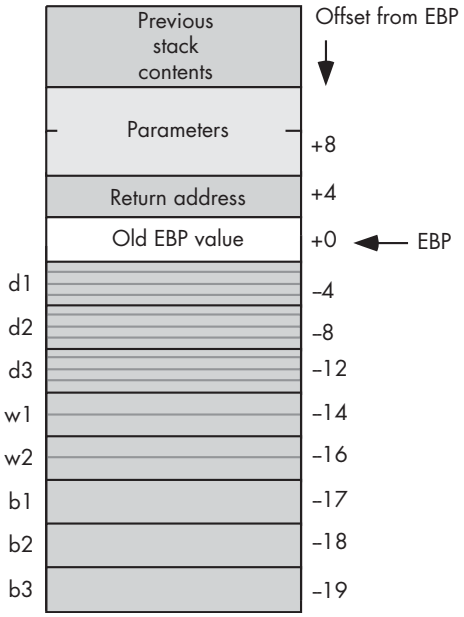

Figure 8-10: Aligned variables in an activation record

Now consider the following function that has arbitrary (unordered) variable declarations and the corresponding activation record (appearing in Figure 8-11):

```
int someFunction2( void )
{
```

```
    char b1;  // Assume chars are 8-bit objects
    int d1;   // Assume ints are 32-bit objects
    short w1; // Assume shorts are 16-bit objects
    int d2;
    short w2;
    char b2;
    int d3;
    char b3;
        .
        .
        .
} // end someFunction2
```

As you can see in Figure 8-11, every variable except the byte variables appear at an address that is inappropriate for the object. On processors that allow memory accesses at arbitrary addresses, it may take more time to access a variable that is not aligned on an appropriate boundary.

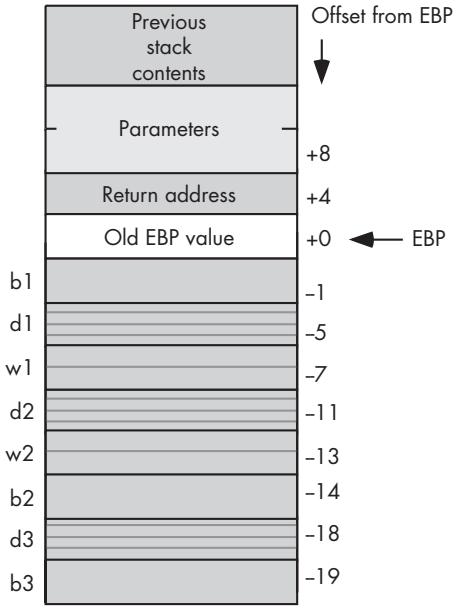

Figure 8-11: Unaligned variables in an activation record

Some processors do not allow a program to access an object at an unaligned address. Most RISC processors, for example, cannot access memory except at 32-bit address boundaries. To access a short or byte value, some RISC processors require the software to read a 32-bit value and extract the 16-bit or 8-bit value (that is, the CPU forces the software to treat bytes and words as packed data). The extra instructions and memory accesses needed to pack and unpack this data reduce the speed of memory access by a considerable amount (that is, two or more instructions—usually more—may be needed to fetch a byte or word from memory). Writing data to memory is even worse because the CPU must first fetch the data from memory, merge

the new data with the old data, and then write the result back to memory. Therefore, most RISC compilers won't create an activation record similar to the one in Figure 8-11. Instead, they will add padding bytes so that every memory object begins at an address boundary that is a multiple of four bytes (see Figure 8-12).

In Figure 8-12 notice that all of the variables are at addresses that are multiples of 32 bits. Therefore, a RISC processor has no problems accessing any of these variables. The cost, of course, is that the activation record is quite a bit larger (the local variables consume 32 bytes rather than 19 bytes).

Although the example in Figure 8-12 is typical for RISC-based compilers, don't get the impression that compilers for CISC CPUs won't do this as well. Many compilers for the 80x86, for example, will also build this activation record in order to improve performance of the code the compiler generates. Although declaring your variables in a misaligned fashion may not slow down your code on a CISC CPU, it may result in additional memory usage.

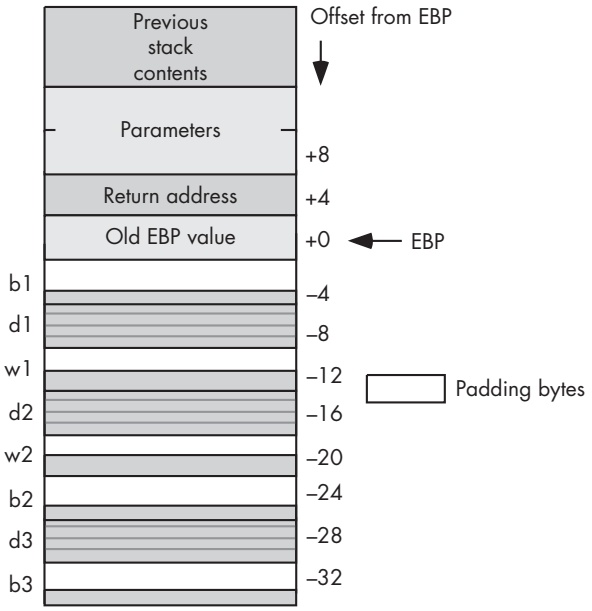

Figure 8-12: RISC compilers force aligned access by adding padding bytes

Of course, if you work in assembly language, it is generally up to you to declare your variables in a manner that is appropriate or efficient for your particular processor. In HLA (on the 80x86), for example, the following two procedure declarations result in the activation records appearing in Figures 8-10, 8-11, and 8-12:

```
procedure someFunction; @nodisplay; @noalignstack;
var
    d1  :dword;
    d2  :dword;
    d3  :dword;
```

```
    w1  :word;
    w2  :word;
    b1  :byte;
    b2  :byte;
    b3  :byte;
begin someFunction;
        .
        .
        .
end someFunction;


procedure someFunction2; @nodisplay; @noalignstack;
var
    b1  :byte;
    d1  :dword;
    w1  :word;
    d2  :dword;
    w2  :word;
    b2  :byte;
    d3  :dword;
    b3  :byte;
begin someFunction2;
        .
        .
        .
end someFunction2;


procedure someFunction3; @nodisplay; @noalignstack;
var
    // HLA align directive forces alignment of the next declaration.

    align(4);
    b1  :byte;
    align(4);
    d1  :dword;
    align(4);
    w1  :word;
    align(4);
    d2  :dword;
    align(4);
    w2  :word;
    align(4);
    b2  :byte;
    align(4);
    d3  :dword;
    align(4);
    b3  :byte;
begin someFunction3;
        .
        .
        .
end someFunction3;
```

HLA procedures `someFunction` and `someFunction3` will produce the fastest-running code on any 80x86 processor because all variables are aligned on an appropriate boundary. HLA procedures `someFunction` and `someFunction2` will produce the most compact activation records on an 80x86 CPU because there is no padding between variables in the activation record. If you're working in assembly language on a RISC CPU, then you'll probably want to choose the equivalent of `someFunction` or `someFunction3` to make it easier to access the variables in memory.

8.6.1 Records and Alignment

Records/structures in HLLs also have alignment issues about which you should worry. Recently, CPU manufacturers have been promoting *Application Binary Interface (ABI)* standards to promote interoperability between different programming languages and implementations of those languages. Although not all languages and compilers adhere to these suggestions, many of the newer compilers do. Among other things, these ABI specifications describe how the compilers should organize fields within a record or structure object in memory. Although the rules vary by CPU, a generic description that is applicable to most ABIs is that a compiler should align a record/structure field at an offset that is a multiple of the object's size. If two adjacent fields in the record or structure have different sizes, and the placement of the first field in the structure would cause the second field to appear at an offset that is not a multiple of that second field's native size, then the compiler will insert some padding bytes to push the second field to a higher offset that is appropriate for that second object's size.

In actual practice, ABIs for different CPUs have minor differences based on the CPUs' ability to access objects at different addresses in memory. Intel, for example, suggests that compiler writers align bytes at any offset, words at even offsets, and everything else at offsets that are a multiple of 4. Some ABIs recommend placing 64-bit objects at 8-byte boundaries within a record. Some CPUs, which have a difficult time accessing objects smaller than 32 bits, may suggest a minimum alignment of 32 bits for all objects in a record/structure. The rules vary depending on the CPU and whether the manufacturer wants to promote faster executing code (the usual case) or smaller data structures.

If you are writing code for a single CPU (e.g., an Intel-based PC) with a single compiler, you should learn that compiler's rules for padding fields and adjust your declarations for maximum performance and minimal waste. However, if you ever need to compile your code using several different compilers, particularly compilers for several different CPUs, following one set of rules will work fine on one machine and produce less efficient code on several others. Fortunately, there are some rules that can help reduce the inefficiencies created by recompiling for a different ABI.

From a performance/memory usage standpoint, the best solution is the same rule we saw earlier for activation records: When declaring fields in a record, group all like-sized objects together and put all the larger (scalar)

objects first and the smaller objects last in the record/structure.[8] This scheme will produce the least amount of waste (padding bytes) and provide the highest performance across most of the ABIs in existence. The only drawback to this approach is that you have to organize the fields by their native size rather than by their logical relationship to one another. However, because all fields of a record/structure are logically related insofar as they are all members of that same record/structure, this problem isn't as bad as employing this organization for all of a particular function's local variables.

Many programmers try to add padding fields themselves to a structure. For example, the following type of code is common in the Linux kernel and other bits and pieces of overly hacked software:

```
typedef struct IveAligned
{
    char byteValue;
    char padding0[3];
    int  dwordValue;
    short wordValue;
    char padding1[2];
    unsigned long dwordValue2;
        .
        .
        .
};
```

The `padding0` and `padding1` fields in this structure were added to manually align the `dwordValue` and `dwordValue2` fields at offsets that are even multiples of 4.

While this padding is not unreasonable, if you're using a compiler that doesn't automatically align the fields, keep in mind that an attempt to compile this code in a different machine can produce unexpected results. For example, if a compiler aligns all fields on a 32-bit boundary, regardless of size, then this structure declaration will consume two extra double words to hold the two `paddingX` arrays. This winds up wasting space for no good reason. So, keep this fact in mind if you decide to manually add the padding fields yourself.

Many compilers that automatically align fields in a structure provide an option to turn off this facility. This is particularly true for compilers generating code for CPUs where the alignment is optional and the compiler only does this to achieve a slight performance boost. If you're going to manually add padding fields to your record/structure, you obviously need to specify this option so that the compiler doesn't realign the fields after you've manually aligned them.

In theory, a compiler is free to rearrange the offsets of local variables within an activation record. However, it would be extremely rare for a compiler to rearrange the fields of a user-defined record or structure. Too many external programs and data structures depend on the fields of a record appearing in the same order as they are declared. This is particularly true

[8] Generally, arrays and records/structures appearing as fields wind up at the end of the list of fields, though you could group arrays with the objects whose size matches the array's element size as well.

when passing record/structure data between code written in two separate languages (for example, when calling a function written in assembly language).

In assembly language, the amount of effort needed to align fields varies from pure manual labor to a rich set of features capable of automatically handling almost any ABI. Some (low-end) assemblers don't even provide record or structure data types. In such systems, the assembly programmer has to manually specify the offsets into a record structure (typically by declaring, as constants, the numeric offsets into the structure). Other assemblers (e.g., NASM) provide macros that automatically generate the equates for you. In such systems as these, the programmer has to manually provide padding fields to align certain fields on a given boundary. Some assemblers, such as MASM and TASM, provide simple alignment facilities. You can specify the value 1, 2, or 4 when declaring a struct in MASM or TASM, and the assembler will align all fields on either the alignment value you specify or at an offset that is a multiple of the object's size, whichever is smaller. It accomplishes this by automatically adding padding bytes to the structure. Also, note that MASM (and TASM) will add a sufficient number of padding bytes to the end of the structure so that the whole structure's length is a multiple of the alignment size. Consider the following struct declaration in MASM:

```
Student  struct  2
score    word    ?   ;offset 0
id       byte    ?   ;offset 2, one byte of padding appears after this field
year     dword   ?   ;offset 4
id2      byte    ?   ;offset 8
Student  ends
```

In this example, MASM will add an extra byte of padding to the end of the structure so that the structure's length is a multiple of 2 bytes.

MASM and TASM also let you control the alignment of individual fields within a structure by using the `align` directive. The following structure declaration is equivalent to the current example (note the absence of the alignment value operand in the `struct` operand field):

```
Student  struct
score    word    ?   ;offset 0
id       byte    ?   ;offset 2
         align   2   ;Injects one byte of padding.
year     dword   ?   ;offset 4
id2      byte    ?   ;offset 8
         align   2   ;Adds one byte of padding to the end of the struct.
Student  ends
```

The default field alignment for MASM/TASM structures is unaligned. That is, a field begins at the next available offset within the structure, regardless of the field's (and the previous field's) size.

The High-Level Assembler (HLA) probably provides the greatest control (both automatic and manual) over record field alignment. Like MASM, the

default record alignment is unaligned. Also, like MASM, you can use HLA's `align` directive to manually align fields in an HLA record. The following is the HLA version of the previous MASM example:

```
type
    Student :record
        score :word;
        id    :byte;
        align(2);
        year  :dword;
        id2   :byte;
        align(2);
    endrecord;
```

HLA also lets you specify an automatic alignment for all fields in a record. For example:

```
type
    Student :record[2]  //This tells HLA to align all
                        // fields on a word boundary
        score :word;
        id    :byte;
        year  :dword;
        id2   :byte;
    endrecord;
```

There is a subtle difference between this HLA record and the earlier MASM structure (with automatic alignment). When you specify a directive of the form `Student struct 2` MASM will align all fields on a boundary that is an multiple of 2 or a multiple of the object's size, *whichever is smaller*. HLA, on the other hand, will always align all fields on a 2-byte boundary using this declaration, even if the field is a byte.

The fact that you can force field alignment to a minimum size is a nice feature if you're working with data structures generated on a different machine (or compiler) that forces this kind of alignment. However, this type of alignment can unnecessarily waste space in a record for certain declarations if you only want the fields to be aligned on their natural boundaries (which is what MASM is doing). Fortunately, HLA provides another syntax for record declarations that let you specify both the maximum and minimum alignment that HLA will apply to a field. That syntax takes the following form:

```
recordID: record[ maxAlign : minAlign ]
    <<fields>>
endrecord;
```

The `maxAlign` item specifies the largest alignment that HLA will use within the record. HLA will align any object whose native size is larger than `maxAlign` on a boundary of `maxAlign` bytes. Similarly, HLA will align any object whose size is smaller than `minAlign` on a boundary of at least `minAlign` bytes. HLA

will align objects whose native size is between `minAlign` and `maxAlign` on a boundary that is a multiple of that object's size. The following HLA and MASM record/struct declarations are equivalent. Here's MASM code:

```
Student  struct  4
score    word    ?   ;offset:0
id       byte    ?   ;offset 2

    ; One byte of padding appears here

year     dword   ?   ;offset 4
id2      byte    ?   ;offset:8

    ; 3 padding bytes appear here

courses  dword   ?   ;offset:12
Student  ends
```

Here's the HLA code:

```
type
    // Align on 4-byte offset, or object's size, whichever
    //  is the smaller of the two. Also, make sure that the
    //  entire record is a multiple of 4 bytes long.

    Student  :record[4:1]
        score   :word;
        id      :byte;
        year    :dword
        id2     :byte;
      courses   :dword;
    endrecord;
```

Although few HLLs provide facilities within the language's design to control the alignment of fields within records (or other data structures), many compilers do provide extensions to those languages, in the form of compiler *pragmas*, that let programmers specifying default variable and field alignment. Because there are no standards for this, you'll have to check your particular compiler's reference manual. Although such extensions are nonstandard, they are often quite useful, especially when linking code compiled by different languages or if you're trying to squeeze the last bit of performance out of a system.

8.7 For More Information

One of the best places to look for more information on how HLLs implement variables is a programming language textbook. Dozens of decent programming design textbooks are available, for example:

- *Programming Languages, Design and Implementation*, Terrence Pratt and Marvin Zelkowitz (Prentice Hall, 2001)

- *Programming Languages, Principles and Practice*, Kenneth Louden (Course Technology, 2002)
- *Concepts of Programming Languages*, Robert Sebesta (Addison-Wesley, 2003)
- *Programming Languages, Structures and Models*, Herbert Dershem and Michael Jipping (Wadsworth, 1990)
- *The Programming Language Landscape*, Henry Ledgard and Michael Marcotty (SRA, 1986)
- *Programming Language Concepts*, Carlo Ghezzi and Jehdi Jazayeri (Wiley, 1997)

Of course, any textbook on compiler design and construction can be a source of information about implementating variables in an HLL. Here are a few examples of compiler-construction textbooks you may want to consider looking at:

- *Compilers, Principles, Techniques, and Tools*, Alfred Aho, Ravi Sethi, and Jeffrey Ullman (Addison-Wesley, 1986)
- *Compiler Construction: Theory and Practice*, William Barret and John Couch (SRA, 1986)
- *A Retargetable C Compiler: Design and Implementation*, Christopher Fraser and David Hansen (Addison-Wesley Professional, 1995)
- *Introduction to Compiler Design*, Thomas Parsons (W. H. Freeman, 1992)
- *Compiler Construction, Principles and Practice*, Kenneth Louden (Course Technology, 1997)

CPU manufacturers' literature, data sheets, and books are also quite useful for determining how compilers will often implement variables. For example, *The PowerPC Compiler Writer's Guide*, edited by Steve Hoxey, Faraydon Karim, Bill Hay, and Hank Warren,[9] is a great reference for programmers writing code to run on a PowerPC processor; most PowerPC compiler writers have used this reference to help them decide how to generate code for the PowerPC processor. Similarly, many compiler writers have used Intel's Pentium manual set (including their *Optimization Guide*) to help them write code generators for their compilers. These manuals may prove handy to someone who wants to understand how 80x86-based compilers generate code.

Of course, the ultimate suggestion is to learn assembly language. If you become an expert assembly language programmer, someone who knows the intricacies of all the machine instructions for a particular processor, then you'll have a much better understanding of how a compiler will generate code for that processor. If you're interested in learning 80x86 assembly language, you might consider *The Art of Assembly Language* (No Starch Press, 2003).

[9] This document is available in PDF format on IBM's website (www.ibm.com).