Chapter 3: The C - Assembly Connection

3.1: Why are We Reading About C?

You probably purchased this book to learn assembly language programming under Windows (after all, that's what the title promises). This chapter is going to spend considerable time talking about the C programming language. Now assembly language programmers fall into two camps with respect to the C programming language: those who already know it and don't really need to learn a whole lot more about C, and those who don't know C and probably don't want to learn it, either. Unfortunately, as the last chapter points out, the vast majority of Windows programming documentation assumes that the reader is fluent in C. This book cannot begin to provide all the information you may need to write effective Win32 applications; therefore, this chapter does the next best thing - it describes how you can translate that C documentation for use in your assembly language programs.

This chapter contains two main sections. The first section provides a basic description of the C programming language for those readers who are not familiar with the C/C++ programming language. It describes various statements in C/C++ and provides their HLA equivalents. Though far from a complete course on the C programming language, this section will provide sufficient information to read some common Win32 programming examples in C and translate them into assembly language. Experienced C/C++ programmers can elect to skip this section (though if you're not comfortable with HLA, you may want to skim over this section because it will help you learn HLA from a C perspective). The second portion of this chapter deals with the Win32 interface and how C passes parameter data to and from Windows. Unless you're well-versed in compiler construction, mixed language calling sequences, and you've examined a lot of compiler code, you'll probably want to take a look at this material.

3.2: Basic C Programming From an Assembly Perspective

The C programming language is a member of the group of programming languages known as the *imperative* or *procedural* programming languages. Languages in this family include FORTRAN, BASIC, Pascal (Delphi/Kylix), Ada, Modula-2, and, of course, C. Generally, if you've learned to write programs in one of these languages, it's relatively easy to learn one of the other languages in the same category. When you attempt to learn a new language from a different class of languages (i.e., you switch *programming paradigms*), it's almost like you're learning to program all over again; learning a new language that is dissimilar to the one(s) you already know is a difficult task. A recent trend in programming language design has been the *hybrid language*. A hybrid language bridges the gap between two different programming paradigms. For example, the C++ language is a hybrid language that shares attributes common to both procedural/imperative languages and object-oriented languages. Although hybrid languages often present some compromises on one side or the other of the gulf they span, the advantage of a hybrid language is that it is easy to learn a new programming paradigm if you're already familiar with one of the programming methodologies that the language presents. For example, programmers who already know find it much easier to learn object-oriented programming via C++ rather than learning the object-oriented programming paradigm from scratch, say by learning Smalltalk (or some other "pure" object-oriented language). So hybrid languages are good in the sense that they help you learn a new way of programming by leveraging your existing knowledge.

The High Level Assembler, HLA, is a good example of a hybrid programming language. While a true assembly language, allowing you to do just about anything that is possible with a traditional (or *low-level*) assembler, HLA also inherits some syntax and many other features from various high-level imperative programming languages. In particular, HLA borrows several control and data structures from the C, Pascal, Ada, and Modula-2 programming languages. The original intent for this design choice was to make it easier to learn assembly lan-

guage if you already knew an high level language like Pascal or C/C++. By borrowing heavily from the syntax of these high-level programming languages, a new assembly language programmer could learn assembly programming much more rapidly by leveraging their C/C++/Pascal knowledge during the early phase of their assembly education.

Note, however, that the reverse is also true. Someone who knows HLA well but doesn't know C can use their HLA knowledge to help them learn the C programming language. HLA's high level control structures are strongly based on languages like C and Modula-2 (or Ada); therefore, if you're familiar with HLA's high level control structures, then learning C's control structures will be a breeze. The sections that immediately follow use this concept to teach some basic C syntax. For those programmers who are not comfortable or familiar with HLA's high level control structures, the following subsections will also describe how to convert between "pure" assembly language and various C control structures. The ultimate goal here is to show you how to convert C code to HLA assembly code; after all, when reading some Win32 programming documentation, you're going to need to convert the examples you're reading in C into assembly language. Although it is always possible (and very easy) to convert any C control structure directly into assembly language, the reverse is not true. That is, it is possible to devise some control flow scheme in assembly language that does not translate directly into a high level language like C. Fortunately, for our purposes, you generally won't need to go in that direction. So even though you're learning about C from an assembly perspective (that is, you're being taught how to read C code by studying the comparable assembly code), this is not a treatise on converting assembly into C (which can be a very difficult task if the assembly code is not well structured).

3.2.1: C Scalar Data Types

The C programming language provides three basic scalar data types[1]: integers, and a couple floating point types. Other data types you'd find in a traditional imperative programming language (e.g., character or boolean values) are generally implemented with integer types in C. Although C only provides three basic scalar types, it does provide several variations of the integer and floating point types. Fortunately, every C data type maps directly to an HLA structured data type, so conversion from C to HLA data types is a trivial process.

3.2.1.1: C and Assembler Integer Data Types

The C programming language specifies (up to) four different integer types: `char` (which, despite its name, is a special case of an integer value), `short, int,` and `long`. A few compilers support a fifth size, "long long". In general, the C programming language does not specify the size of the integer values; that decision is left to whomever implements a specific compiler. However, when working under Windows (Win32), you can make the following assumptions about integer sizes:

- char - one byte
- short - two bytes
- int, long - four bytes

1. For our purposes, a scalar data type is a primitive or atomic data type; one that the language treats as a single unit, that isn't composed of smaller items (like, say, elements of an array or fields of a structure).

The C programming language also specifies two types of integers: signed and unsigned. By default, all integer values are signed. You can explicitly specify unsigned by prefacing one of these types with the keyword `unsigned`. Therefore, C's integral types map to HLA's types as shown in Table 3-1.

Table 3-1: Integer Type Correspondence Between HLA and C

C Type	Corresponding HLA Types
char	char, byte, int8[a]
short	word, int16
int	dword, int32
long	dword, int32
long long	qword, int64
unsigned char	char, byte, uns8
unsigned short	word, uns16
unsigned	dword, uns32
unsigned int	dword, uns32
unsigned long	dword, uns32
unsigned long long	qword, uns64

a.Some compilers have an option that lets you specify the use of unsigned char as the default. In this case, the corresponding HLA type is uns8.

Generic integer literal constants in C take several forms. C uses standard decimal representation for base 10 integer constants, just like most programming languages (including HLA). For example, the sequence of digits:

```
128
```

represents the literal integer constant 128.

If a literal integer constant begins with a zero (followed by one or more octal digits in the range 0..7), then C treats the literal constant as a base-8 (octal) value. HLA doesn't support octal constants, so you will have to manually convert such constants to decimal or hexadecimal prior to using them in an assembly language program. Fortunately, you rarely see octal constants in modern C programs (especially in Win32 programs).

C integer literal constants that begin with "0x" are hexadecimal (base-16) constants. You will replace the "0x" prefix with a "$" prefix when converting the value from C to HLA. For example, the C literal constant "0x1234ABCD" becomes the HLA literal constant "$1234ABCD".

C also allows the use of an "L" suffix on a literal integer constant to tell the compiler that this should be a long integer value. HLA automatically adjusts all literal constants to the appropriate size, so there is no need to tell HLA to extend a smaller constant to a long (32-bit) value. If you encounter an "L" suffix in a C literal constant, just drop the suffix when translating the value to assembly.

3.2.1.2: C and Assembly Character Types

As the previous section notes, C treats character variables and constants as really small (one-byte) integer values. There are some non-intuitive aspects to using C character variables that can trip you up; hence the presence of this section.

The first place to start is with a discussion of C and HLA literal character constants. The two literal forms are quite similar, but there are just enough differences to trip you up if you're not careful. The first thing to note is that both HLA and C treat a character constant differently than a string containing one character. We'll cover character strings a little later in this chapter, but keep in mind that character objects are not a special case of a string object.

A character literal constant in C and HLA usually consists of a single character surrounded by apostrophe characters. E.g., 'a' is a character constant in both of these languages. However, HLA and C differ when dealing with non-printable (i.e., control) and a couple of other characters. C uses an *escape character sequence* to represent the apostrophe character, the backslash character, and the control characters. For example, to represent the apostrophe character itself, you'd use the C literal constant '\''. The backslash tells C to treat the following value specially; in this particular case, the backslash tells C to treat the following apostrophe character as a regular character rather than using it to terminate the character constant. Likewise, you use '\\' to tell C that you want a single backslash character constant. C also uses a backslash followed by a single lowercase alphabetic character to denote common control characters. Table 3-2 lists the escape character sequences that C defines.

Table 3-2: C Escape Character Sequences

C Escape Sequence	Control Character
'\n'	New line (carriage return/line feed under Windows, though C encodes this as a single line feed)
'\r'	Carriage return
'\b'	Backspace
'\a'	Alert (bell character, control-G)
'\f'	Form Feed (control-L)
'\t'	Tab character (control-I)
'\v'	Vertical tab character (control-k)

C also allows the specification of the character's numeric code by following a backslash with an octal or hexadecimal constant in the range 0..0xff, e.g., '\0x1b'.

HLA does not support escape character sequences using the backslash character. Instead, HLA uses a pound sign ('#') followed immediately by a numeric constant to specify the ASCII character code. Table 3-3 shows how to translate various C escape sequences to their corresponding HLA literal character constants.

Table 3-3: Converting C Escape Sequences to HLA Character Constants

C Escape Sequence	HLA Character Constant	Description
'\n'	#$a #$d	Note that the end of line sequence under Windows is not a character, but rather a string consisting of two characters. If you need to represent newline with a single character, using a linefeed (as see does) whose ASCII code is $A; linefeed is also defined in the HLA Standard Library as stdio.lf. Note that the "nl" symbol an HLA user typically uses for newline is a two-character string containing line feed followed by carriage return.
'\r'	#$d	Carriage return character. This is defined in the HLA Standard Library as stdio.cr.
'\b'	#8	Backspace character. This is defined in the HLA Standard Library as stdio.bs.
'\a'	#7	Alert (bell) character. This is defined in the HLA Standard Library as stdio.bell.
'\f'	#$c	Form feed character.
'\t'	#9	Tab character. This is defined in the HLA Standard Library as stdio.tab.
'\v'	#$b	Vertical tab character.

Because C treats character values as single-byte integer values, there is another interesting aspect to character values in C - they can be negative. One might wonder what "minus 'z'" means, but the truth is, there really is no such thing as a negative character; C simply uses signed characters to represent small integer values in the range -128..+127 (versus unsigned characters that represent values in the range 0..255). For the standard seven-bit ASCII characters, the values are always positive, regardless of whether you're using a signed character or an unsigned character object. Note, however, that many C functions return a signed character value to specify certain error conditions or other states that you cannot normally represent within the ASCII character set. For example, many functions return the character value minus one to indicate end of file.

3.2.1.3: C and Assembly Floating Point (Real) Data Types

The C programming language defines three different floating point sizes: *float, double,* and *long double*[2]. Like the integer data type sizes, the C language makes no guarantees about the size or precision of floating point values other than to claim that *double* is at least as large as *float* and *long double* is at least as large as *double*. However, while nearly every C/C++ compiler that generates code for Windows uses the same sizes for integers (8, 16, and 32 bits for *char, short,* and *int/long*), there are differences in the sizes of floating objects among compilers. In particular, some compilers use a 10-byte extended precision format for *long double* (e.g., Borland) while others use an eight-byte double precision format (e.g., Microsoft). Fortunately, all (reasonable) compilers running under Windows use the IEEE 32-bit single-precision format for *float* and the IEEE 64-bit double-precision format for *double*. If you encounter a *long double* object in C code, you will have to check with the com-

2. Only recent versions of the C programming language support the "long double" floating point type.

piler's vendor to determine the size of the object in order to convert it to assembly language. Of course, for many applications it won't really matter if you go ahead and use a 10-byte real value whenever you encounter a *long double* object. After all, the original programmer is probably expecting something bigger than a *double* object anyway. Do keep in mind, however, that some algorithms involving real arithmetic may not be stable when run with a different floating point size other than the sized used when they were developed.

T shows the correspondence between C/C++ floating point data types and HLA's floating point data types.

Table 3-4: Real Type Correspondence Between C and Assembly

C Real Type	Corresponding HLA Type	Comment
float	real32	32-bit IEEE format floating point value.
double	real64	64-bit IEEE format floating point value.
long double	real64	64-bit IEEE format floating point value on certain compilers (e.g., Microsoft).
	real80	80-bit IEEE format floating point value on certain compilers (e.g., Borland)

C and HLA floating point literal constants are quite similar. They may begin with an optional sign, followed by one or more decimal digits. Then you can have an optional decimal point followed by another optional sequence of decimal digits; following that, you can have an optional exponent specified as an 'e' or 'E', an optional sign, and one or more decimal digits. The final result must not look like an integer constant (i.e., the decimal point or an exponent must be present).

C allows an optional "F" suffix on a floating point constant to specify single precision, e.g., 1.2f. Similarly, you can attach an "L" suffix to a float value to indicate *long double*, e.g., 1.2L. By default, C literal floating point constants are always double precision values. HLA always maintains all constants as 80-bit floating point values internally and converts them to 64 or 32 bits as necessary, so there is no need for such a suffix. So by dropping the "F" or "L" suffix, if it is present, you can use any C floating point literal constant as-is in the HLA code.

3.2.2: C and Assembly Composite Data Types

C provides four major composite (or aggregate) data types: arrays, structures, unions, and pointers. C++ adds the class types well. A composite type is a type that is built up from smaller types (e.g., an array is a collection of smaller objects, each element having the same type as all other elements in the array). In the following sub-sections, we'll explore these composite data types in C and provide the corresponding type in HLA.

As is the case throughout this chapter, this section assumes that you already know assembly language and may not know C. This section provides the correspondence between C and HLA, but doesn't attempt to teach you how to do things like access an array in assembly language; the previous chapter already covered that material.

3.2.2.1: C and Assembly Array Types

Since HLA's syntax was based on the C and Pascal programming languages, it should come as no surprise that array declarations in HLA are very similar to those in C. This makes the translation from C to HLA very easy.

The syntax for an array declaration in C is the following:

```
elementType  arrayName[ elements ] <<additional dimension infor-
mation>>;
```

elementType is the type of an individual element of the array. *arrayName* is the name of the array object. *elements* is an integer constant value the specifies the number of array elements. Here are some sample C array declarations:

```
int intArray[4];
char grid[3][3];  // 3x3 two-dimensional array
double flts[16];
userType userData[2][2][2]
```

In HLA, multiple dimension arrays use a comma-delimited list to separate each of the maximum bounds (rather than using separate sets of brackets). Here are the corresponding declarations in HLA:

```
intArray :int32[ 4 ];
grid     :char[3,3];
flts     :real64[16];
userData :userType[2,2,2];
```

Both C and HLA index arrays from zero to *n*-1, where *n* is the value specified as the array bound in the declaration.

C stores arrays in memory using row-major ordering. Therefore, when accessing elements of a multi-dimensional array, always use the row-major ordering algorithm (see *The Art of Assembly Language* for details if you're unfamiliar with accessing elements of a multi-dimensional array in assembly language).

In C, it is possible to provide an initial value for an array when declaring an array. The following C example demonstrates the syntax for this operation:

```
int iArray[4] = {1,2,3,4};
```

HLA also allows initialization of array variables, but only for static objects. Here's the HLA version of this C code:

```
static
   iArray :int32[4] := [1,2,3,4];
```

C allows the same syntax for an array initialization to automatic variables. However, you cannot initialize an automatic variable at compile time (this is true for C and HLA); therefore, the C compiler automatically emits code to copy the data from some static memory somewhere into the automatic array, e.g., a C declaration like the following appearing in a function (i.e., as an automatic local variable):

```
int autoArray[4] = {1,2,3,4};
```

gets translated into machine code that looks something like the following:

```
readonly
   staticInitializeData :dword := [1,2,3,4];
      .
      .
      .
var
   autoArray: int32[4];
      .
      .
      .
   mov( &staticInitializerData, esi );
   lea( edi, autoArray );
   mov( 4, ecx );
   rep.movsd();
```

(yes, compilers really do generate code like this). This code is pretty disgusting. If you see an automatic array variable with an initializer, it's probably a better idea to try and figure out if you really need a new copy of the initial data every time you enter the function.

3.2.2.2: C and Assembly Record/Structure Types

C implements the equivalent of HLA'S records using the `struct` keyword. Structure declarations in C look just like standard variable declarations sandwiched inside a "struct {...}" block. Conversion to HLA is relatively simple: just stick the HLA equivalent of those field declarations in a `record..endrecord` block. The only point of confusion is C's syntax for declaring tags, types, and variables of some `struct` type.

C allows you to declare structure variables and types several different ways. First, consider the following structure variable declaration:

```
   struct
   {
      int fieldA;
      float fieldB;
      char fieldC;
   }
      structVar;
```

Assuming this is a global variable in C (i.e., not within a function) then this creates a static variable `structVar` that has three fields: `structVar.fieldA`, `structVar.fieldB`, and `structVar.fieldC`. The corresponding HLA declaration is the following (again, assuming a static variable):

```
static
   structVar:
      record
         fieldA  :int32;
         fieldB  :real32;
         fieldC  :char;
      endrecord;
```

In an HLA program, you'd access the fields of the structVar variable using the exact same syntax as C, specifically: `structVar.fieldA`, `structVar.fieldB`, and `structVar.fieldC`.

There are two ways to declare structure types in C: using `typedef` and using *tag fields*. Here's the version using C's `typedef` facility (along with a variable declaration of the structure type):

```
typedef struct
   {
      int fieldA;
      float fieldB;
      char fieldC;
   }
      structType;

   structType structVar;
```

Once again, you access the fields of `structVar` as `structVar.fieldA`, `structVar.fieldB`, and `structVar.fieldC`.

The `typedef` keyword was added to the C language well after it's original design. In the original C language, you'd declare a structure type using a structure tag as follows:

```
struct structType
   {
      int fieldA;
      float fieldB;
      char fieldC;
   } /* Note: you could actually declare structType variables here */ ;

   struct structType strucVar;
```

HLA provides a single mechanism for declaring a record type - by declaring the type in HLA's type section. The syntax for an HLA record in the type section takes this form:

```
type
   structType:
      record
         fieldA  :int32;
         fieldB  :real32;
         fieldC  :char;
      endrecord;

static
   structVar  :structType;
```

C also allows the initialization of structure variables by using initializers in a variable declaration. The syntax is similar to that for an array (i.e., a list of comma separated values within braces). C associates the values in the list with each of the fields by position. Here is an example of the `structVar` declaration given earlier with an initializer:

```
   struct structType structVar = { 1234, 5.678, '9' };
```

Like the array initializers you read about in the previous section, the C compiler will initialize these fields at compile time if the variable is a static or global variable. however, if it's an automatic variable, then the C compiler emits code to copy the data into the structure upon each entry into the function defining the initialized structure (i.e., this is an expensive operation).

Like arrays, HLA allows the initialization of static objects you define in a `static`, `storage`, or `readonly` section. Here is the previous C example translated to HLA:

```
static
   structVar  :structType := structType:[1234, 5.678, '9' ];
```

One major difference between HLA and C with respect to structures is the alignment of the fields within the structure. By default, HLA packs all the fields in a record so that there are no extra bytes within a structure. The `structType` structure, for example, would consume exactly nine bytes in HLA (four bytes each for the `int32` and `real32` fields and one byte for the `char` field). C structures, on the other hand, generally adhere to certain alignment and padding rules. Although the rules vary by compiler implementation, most Win32 C/C++ compilers align each field of a struct on an offset boundary that is an even multiple of that field's size (up to four bytes). In the current `structType` example, `fieldA` would fall at offset zero, `fieldB` would fall on offset four within the structure, and `FieldC` would appear at offset eight in the structure. Since each field has an offset that is an even multiple of the object's size, C doesn't manipulate the field offsets for the record. Suppose, however, that we have the following C declaration in our code:

```
typedef struct
   {
      char fieldC;
      int fieldA;
      short fieldD;
      float fieldB;
   }
      structType2;
```

If C, like HLA, didn't align the fields by default, you'd find `fieldC` appearing at offset zero, `fieldA` at offset 1, `fieldD` at offset five, and `fieldB` at offset seven. Unfortunately, these fields would appear at less than optimal addresses in memory. C, however, tends to insert padding between fields so that each field is aligned at an offset within the structure that is an even multiple of the object's size (up to four bytes, which is the maximum field alignment that compilers support under Win32). This padding is generally transparent to the C programmer; however, when converting C structures to assembly language records, the assembly language programmer must take this padding into consideration in order to correctly communicate information between an assembly language program and Windows. The traditional way to handle this (in assembly language) has been to explicitly add padding fields to the assembly language structure, e.g.,

```
type
   structType2:
      record
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
      endrecord;
```

HLA provides a set of features that automatically and semi-automatically align record fields. First of all, you can use the HLA `align` directive to align a field to a given offset within the record; this mechanism is great for programs that need absolute control over the alignment of each field. However, this kind of control is not necessary when implementing C records in assembly language, so a better approach is to use HLA's automatic record field alignment facilities[3]. The rules for C/C++ (under Windows, at least) are pretty standard among compilers; the rule is this: each object (up to four bytes) is aligned at a starting offset that is an even multiple of that object's size. If the object is larger than four bytes, then it gets aligned to an offset that is an even multiple of four bytes. In HLA, you can easily do this using a declaration like the following:

```
type
   structType2:
      record[4:1];
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
      endrecord;
```

The "[4:1];" appendage to the record keyword tells HLA to align objects between one and four bytes on their natural boundary and larger objects on a four-byte boundary, just like C. This is the only record alignment option you should need for C/C++ code (and, in fact, you should always use this alignment option when converting C/C++ structs to HLA records). If you would like more information on this alignment option, please see the HLA reference manual.

Specifying the field alignment via the "[4:1];" option only ensures that each field in the record starts at an appropriate offset. It does not guarantee that the record's length is an even multiple of four bytes (and most C/C++ compilers will add padding to the end of a `struct` to ensure the length is an even multiple of four bytes). However, you can easily tell HLA to ensure that the record's length is an even multiple of four bytes by adding an align directive to the end of the field list in the record, e.g.,

```
type
   structType2:
      record[4:1];
         fieldC  :char;
         pad0    :byte[3]; // Three bytes of padding.
         fieldA  :int32;
         fieldD  :int16;
         pad1    :byte[2]; // Two bytes of padding.
         fieldB  :real32;
         align(4);
      endrecord;
```

Another important thing to remember is that the fields of a record or structure are only properly aligned in memory if the starting address of the record or structure is aligned at an appropriate address in memory. Specifying the alignment of the fields in an HLA record does not guarantee this. Instead, you will have to use HLA's `align` directive when declaring variables in your HLA programs. The best thing to do is ensure that an instance

3. For those who are interested in using the align directive and other advanced record field alignment facilities, please consult the HLA reference manual.

of a record (i.e., a record variable) begins at an address that is an even multiple of four bytes[4]. You can do this by declaring your record variables as follows:

```
static
   align(4);  // Align following record variable on a four-byte address in memory.
   recVar   :structType2;
```

3.2.2.3: C and Assembly Union Types

A C union is a special type of structure where all the fields have the same offset (in C, the offset is zero, in HLA you can actually select the offset though the default is zero). That is, all the fields of an instance of a union overlay one another in memory. Because the fields of a union all have the same starting address, there are no issues regarding field offset alignment between C and HLA. Therefore, all you need really do is directly convert the C syntax to the HLA syntax for a union declaration.

In C, union type declarations can take one of two forms:

```
union unionTag
{
   <<fields that look like variable declarations>>
} <<optional union variable declarations>>;

typedef union
{
   <<fields that look like variable declarations>>
}
   unionTag2; /* The type declaration */
```

You may also declare `union` variables directly using syntax like the following:

```
union
{
   << fields that look like variable declarations >>
}
   unionVariable1, unionVariable2, unionVariable3;

   union unionTag unionVar3, unionVar4;
   unionTag2 unionVar5;
```

In HLA, you declare a union type in HLA's `type` section using the `union/endunion` reserved words as follows:

```
type
   unionType:
      union
         << fields that look like HLA variable declarations >>
      endunion;
```

4. Technically, you should align a record object on a boundary that is an even multiple of the largest field's size, up to four bytes. Aligning record variables on a four-byte boundary, however, will also work.

You can declare actual HLA union variables using declarations like the following:

```
storage
   unionVar1:
      union
         << fields that look like HLA variable declarations >>
      endunion;

   unionvar2: unionType;
```

The size of a union object in HLA is the size of the largest field in the `union` declaration. You can force the size of the union object to some fixed size by placing an align directive at the end of the union declaration. For example, the following HLA type declaration defines a union type that consumes an even multiple of four bytes:

```
type
   union4_t:
      union
         b    :boolean;
         c    :char[3];
         w    :word;
         align(4);
      endunion;
```

Without the "align(4);" field in this union declaration, HLA would only allocate three bytes for a object of type `union4_t` because the single largest field is `c`, which consumes three bytes. The presence of the "align(4);" directive, however, tells HLA to align the end of the union on a four-byte boundary (that is, make the union's size an even multiple of four bytes). You'll need to check with your compiler to see what it will do with unions, but most compilers probably extend the union so that it's size is an even multiple of the largest scalar (e.g., non-array) object in the field list (in the example above, a typical C compiler would probably ensure that and instance of the `union4_t` type is an even multiple of two bytes long).

As for records, the fact that a union type is an even multiple of some size does not guarantee that a variable of that type (i.e., an instance) is aligned on that particular boundary in memory. As with records, if you want to ensure that a union variable begins at a desired address boundary, you need to stick an align directive before the declaration, e.g.,

```
var
   align(4);
   u  :union4_t;
```

3.2.2.4: C and Assembly Character String Types

The C/C++ programming language does not support a true string type. Instead, C/C++ uses an array of characters with a zero terminating byte to represent a character string. C/C++ uses a pointer to the first character of the character sequence as a "string object". HLA defines a true character string type, though it's internal representation is a little different than C/C++'s. Fortunately, HLA's string format is upwards compatible with the zero-terminated string format that C/C++ uses, so it's very easy to convert HLA strings into the C/C++ format (indeed, the conversion is trivial). There are a few problems going in the other direction (at least, at run time). So a special discussion of HLA versus C/C++ strings is in order.

Character string declarations are somewhat confused in C/C++ because C/C++ often treats pointers to an object and arrays of that same object type equivalently. For example, in an arithmetic expression, C/C++ does not differentiate between the use of a `char*` object (a character pointer) and an array of characters. Because of this syntactical confusion in C/C++, you'll often see strings in this language declared one of two different ways: as an array of characters or as a pointer to a character. For example, the following are both typical string variable declarations in C/C++:

```
char stringA[ 256 ]; // Holds up to 255 characters plus a zero terminating byte.
char *stringB;       // Must allocate storage for this string dynamically.
```

The big difference between these two declarations is that the `stringA` declaration actually reserves the storage for the character string while the `stringB` declaration only reserves storage for a pointer. Later, when the program is running, the programmer must allocate storage for the string associated with `stringB` (or assign the address of some previously allocated string to `stringB`). Interestingly enough, once you have two declarations like `stringA` and `stringB` in this example, you can access characters in either string variable using either pointer or array syntax. That is, all of the following are perfectly legal given these declarations for `stringA` and `stringB` (and they all do basically the same thing):

```
char ch;

ch = stringA[i];       // Access the char at position i in stringA.
ch = *(stringB + i);   // Access the char at position i in stringB.
ch = stringB[i];       // Access the char at position i in stringB.
ch = *(stringA + i);   // Access the char at position i in stringA.
```

String literal constants in C/C++ are interesting. Syntactically, a C/C++ string literal looks very similar to an HLA string literal: it consists of a sequence of zero or more characters surrounded by quotes. The big difference between HLA string literals and C/C++ string literals is that C/C++ uses escape sequences to represent control characters and non-graphic characters within a string (as well as the backslash and quote characters). HLA does not support the escape character sequences (see the section on character constants for more details on C/C++ escape character sequences). To convert a C/C++ string literal that contains escape sequences into an HLA character string, there are four rules you need follow:

- Replace the escape sequence \" with "". HLA uses doubled-up quotes to represent a single quote within a string literal constant.
- Replace the escape sequence \\ with a single backslash. Since HLA doesn't use the backslash as a special character within a string literal, you need only one instance of it in the string to represent a single backslash character
- Convert special control-character escape sequences, e.g., \n, \r, \a, \b, \t, \f, and \v, to their corresponding ASCII codes (see Table 3-3) and splice that character into the string using HLA's #nn character literal, e.g., the C/C++ string "hello\nworld\n" becomes the following:

  ```
  "hello" #$d #$a "world" #$d #$a    //HLA automatically splices all
  this together.
  ```

- Whenever a C/C++ numeric escape sequence appears in a string (e.g., \0nn or \0Xnn) then simply convert the octal constant to a hexadecimal constant (or just use the hexadecimal constant as-is) along with the HLA #$nn literal constant specification and splice the object into the string as before. For example, the C/C++ string "hello\0xaworld0xa" becomes:

```
“hello” #$a “world” #$a
```

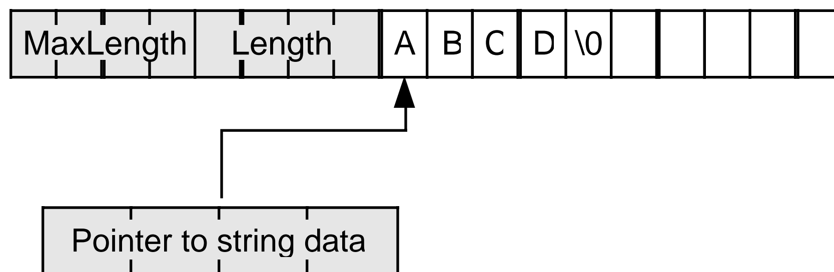
Whenever a C/C++ compiler sees a string literal constant in a source file, the compiler will allocate storage for each character in that constant plus one extra byte to hold the zero terminating byte[5] and, of course, the compiler will initialize each byte of this storage with the successive characters in the string literal constant. C/C++ compilers will typically (though not always) place this string they've created in a read-only section of memory. Once the compiler does this, it replaces the string literal constant in the program with the address of the first character it has created in this read-only memory segment. To see this in action, consider the following C/C++ code fragment:

```
char* str;

str = “Some Literal String Constant”;
```

This does not copy the character sequence “Some Literal String Constant” into `str`. Instead, the compiler creates a sequence of bytes somewhere in memory, initializes them with the characters in this string (plus a zero terminating byte), and then stores the address of the first character of this sequence into the `str` character pointer variable. This is very efficient, as C/C++ needs to only move a four-byte pointer around at run-time rather than moving a 29 byte sequence.

The HLA language provides an explicit `string` data type. Internally, however, HLA represents strings using a four-byte pointer variable, just as C/C++ does. There is an important difference, however, between the type of data that an HLA string variable references and a C/C++ character pointer references: HLA includes some additional information with its string data: specifically, a length field and a maximum length field. Like C/C++ string objects, the address held in an HLA `string` variable points at the first character of the character string (and HLA strings also end with a zero terminating byte). Unlike C/C++ however, the four bytes immediately preceding the first character of the string contain the current length of the string (as an `uns32` value). The four bytes preceding the length field contain the maximum possible length of the string (that is, how much memory is reserved for the string variable, which can be larger than the current number of characters actually found in the string). Figure 3-1 shows the HLA string format in memory.

Figure 3-1: HLA String Format in Memory



The interesting thing to note about HLA strings is that they are downwards compatible with C/C++ strings. That is, if you've got a function, procedure, or some other piece of code that operates on a C/C++ string, you can generally pass it a pointer to an HLA string and the operation will work as expected[6]. This was done by design

5. Some compilers may actually allocate a few extra bytes of padding at the end of the string to ensure that the literal string constant's length is an even multiple of four bytes. However, this is not universal among compilers so don't count on it.
6. The only ‘gotcha’ is that you cannot expect the string to remain in a consistent HLA format if your code that operates on C/C++ strings makes any modifications to the HLA string data.

and is one of the reasons HLA interfaces so well with the Win32 API. The Win32 API generally expects C/C++ zero terminated strings when you pass it string data. Usually, you can pass an HLA string directly to a Win32 API function and everything will work as expected (few Win32 API function calls actually change the string data).

Going the other direction, converting a C/C++ string to an HLA string, is not quite as trivial (though still easy to accomplish using appropriate code in the HLA Standard Library). Before describing how to convert C/C++ strings to the HLA format, however, it *is* important to point out that HLA is assembly language and assembly language is capable of working with any string format you can dream up (including C/C++ strings). If you've got some code that produces a zero-terminated C/C++ string, you *don't* have to convert that string to an HLA string in order to manipulate it within your HLA programs. Although the HLA string format is generally more efficient (faster) because the algorithms that process HLA strings can be written more efficiently, you can write your own zero-terminated string functions and, in fact, there are several common zero-terminated string functions found in the HLA Standard Library. However, as just noted, HLA strings are generally more efficient, so if you're going to be doing a bit of processing on a zero terminated string, it's probably wise to convert it to an HLA string first. On the other hand, if you're only going to do one or two trivial operations on a zero-terminated string that the Win32 API returns, you're better off manipulating it directly as a zero-terminated string and not bothering with the conversion (of course, if convenience is your goal rather than efficiency, it's probably always better to convert the string to the HLA string format upon return from a Win32 API call just so you don't have to work with two different string formats in your assembly code). Of course, the other option open to you is to work exclusively with zero-terminated strings in your assembly code (though the HLA Standard Library doesn't provide anywhere near the number of zero-terminated string functions).

The purpose of this chapter is to explain how to convert C/C++ based Windows documentation to a form suitable for assembly language programmers and describe how assembly language programmers can call the C-based functions that make up the Win32 API set. Therefore, we won't get into the details of converting string function calls in C/C++ to their equivalent (or similar) HLA Standard Library calls. Rest assured, the HLA Standard Library provides a much richer set of string functions than does the C Standard Library; so anything you find someone doing in C/C++ can be done just as easily (and usually more efficiently) using an HLA Standard Library function. Please consult the HLA Standard Library documentation for more details. In this section, we'll simply cover those routines that are necessary for converting between the two different string formats and copying string data from one location to another.

In C/C++ code, it is very common to see the program declare a string object (character pointer) and initialize the pointer with the address of a literal string constant all in one operation, e.g.,

```
char *str = "Literal String Constant";
```

Remember, this doesn't actually copy any character data; it simply loads the character pointer variable `str` with the address of the 'L' character in this string literal constant. The compiler places the literal string data somewhere in memory and either initializes `str` with the address of the start of that character data (if `str` is a global or static variable) or it generates a short (generally one instruction) machine code sequence to initialize a run-time variable with the address of this string constant. Note, *and this is very important*, many compilers will allocate the actual string data in read-only memory. Therefore, as long as `str` continues to point at this literal string constant's data in memory, any attempt to store data into the string will likely produce a memory protection fault. Some very old C programs may have made the assumption that literal string data appears in read/write memory and would overwrite the characters in the string literal constant at run-time. This is generally not allowed by modern C/C++ compilers. HLA, by default, also places string literal data in write-protected memory. So if you encounter some older C code that overwrites the characters in a string literal constant, be aware of the fact that you will not (generally) be able to get away with this in HLA.

If `str` (in the previous example) is a static or global variable, converting this declaration to assembly language is very easy, you can use a declaration like the following:

```
static
   HLAstr  :string := "Literal String Constant";
```

Remember, like C/C++, HLA implements string objects using pointers. So `HLAstr` is actually a four-byte pointer variable and this code initializes that pointer with the address of the first character in the literal string constant.

Because HLA string variables are pointers that (when properly initialized) point at a sequence of characters that end with a zero byte (among other things), you can use an HLA string variable just about anywhere C/C++ expects a string object. So, for example, if you want to make some Win32 API call that requires a string parameter, you can pass in `HLAstr` as that parameter, e.g.,

```
Win32APIcall( HLAstr );
```

Of course, you don't have to assign the address of a string literal constant to an HLA string variable to make this function call, HLA allows you to pass the literal string constant directly:

```
Win32APIcall( "Literal String Constant" );
```

Note that this call does not pass the string data directly to the function; like C/C++, HLA continues to allocate and initialize the literal string constant in write-protected memory somewhere else and simply passes the address of the literal string constant as the function's parameter.

For actual string variable objects (that is, strings whose character values you can change at run-time), you'll typically find two different ways of variable declaration/allocation in a C/C++ program: the programmer will either create a character array with sufficient storage to hold the characters in the string, or the programmer will simply declare a character pointer variable and allocate storage for the string at run-time using a function like `malloc` (C) or new (C++). We'll look at both of these mechanisms in HLA in the following paragraphs.

One way to allocate storage for a C string variable is to simply declare a character array with sufficient room to hold the longest possible string value you will assign plus an extra byte for the zero terminator. To do this, a C/C++ programmer simply declares a character array with n+1 elements, where n is the maximum number of characters they expect to put into the string. The following is a typical "string" declaration in C/C++ that creates a string capable of holding up to 255 characters:

```
char myStr[256];  // 255 chars plus a zero terminating byte
```

Although HLA most certainly allows you to create arrays of characters, such arrays are not directly compatible with HLA strings (because they don't reserve extra storage prior to the string object for the length and maximum length fields that are present in the HLA string format). Therefore, you cannot allocate storage for an HLA string variable by simply doing the following:

```
static
      myStr :char[256];  // Does NOT create an HLA string!
```

You *can* use this technique to pre-allocate storage for a zero-terminated string in HLA, but this does not create the proper storage for an HLA string variable.

In general, HLA expects you to allocate storage for string objects dynamically (that is, at run-time). However, for static strings (i.e., non-automatic string variables) the HLA Standard Library does provide a macro,

str.strvar, that lets you declare string variable and allocate storage for the string variable at compile-time. Here is an example of the use of this macro:

```
static  //Note: str.strvar only makes sense in the STATIC section
   preAllocated :str.strvar( 255 ); // Allows strings up to 255 characters long.
```

There are a couple of important things to note about this declaration. First of all, note that you surround the size of the string with parentheses, not square brackets. This is because str.strvar is actually a macro, not a type and the 255 that appears in this example is a macro argument. The second thing to note is that the use of the str.strvar macro only makes sense in the static declaration section. It is illegal in a var or storage section because this macro initializes the variable you're declaring (something you can't do in a var or storage declaration section). While the use of the str.strvar macro is technically legal in HLA's readonly declaration section, it doesn't make sense to use it there since doing so would allocate the string data storage in write-protected memory, so you'd never be able to store any character data into the string. Also note that the str.strvar macro doesn't provide any mechanism for initializing the character data in the string within the declaration (not that it would be that hard to create a new macro that does this). Finally, note that you specify the maximum number of characters you want to allow in the string as the str.strvar argument. You do not have to account for any zero-terminating bytes, the current string length field, or the maximum length field.

Although the str.strvar macro is very convenient to use, it does have a couple of serious limitations. Specifically, it only makes sense to use it in the static section of an HLA program and you must know the maximum length of the string before you can use the str.strvar macro to declare a string object. Both of these issues are easy to resolve by using dynamically allocated string objects. C/C++ programmers can also create dynamically allocated strings by declaring their string variables as pointers to characters, e.g.,

```
   char *strAsPtr;
      .
      .
      .
   // Allocate storage for a 255 character string at run-time

   strAsPtr = malloc( 256 );  // 255 chars + zero terminating byte

   // Or (in C++):

   strAsPtr = new char[256];
```

In HLA, the declaration and allocation is very similar:

```
var  // Could also be static or storage section
   strAsPtr :string;
      .
      .
      .
   strmalloc( 255 );
   mov( eax, strAsPtr );
```

Do note a couple of things about the HLA usage. First, you should note that the specify the actual number of characters you wish to allow in the string; you don't have to worry about the zero terminating byte. Second, you should call the HLA Standard Library stralloc function (rather than malloc, which HLA also provides) in order to allocate storage for a string at run time; in addition to allocating storage for the character data in the string, stralloc also allocates (additional) storage for the zero terminating byte, the maximum length value, and

the current length value. The `stralloc` function also initializes the string to the empty string (by setting the current length to zero and storing a zero in the first character position of the string data area). Finally, note that stralloc returns the address of the first character of the string in EAX so you must store that address into the `string` variable. HLA does support a feature known as instruction composition, so you could actually write these two instructions as follows:

```
mov( stralloc( 255 ), strAsPtr );
```

The drawback to this form, however, is that it hides the fact that this code sequence wipes out the EAX register. So be careful if you decide to use this form.

Once you've initialized an HLA string variable so that it points at valid string data, you can generally pass that string variable to a C/C++ function (i.e., a Win32 API function) that expects a string value. The only restriction is that if the C/C++ function modifies the string data in such a way that it changes the length of the string, HLA will not recognize the change if you continue to treat the string as an HLA type string upon return from the C/C++ function. Likewise, if a C/C++ function returns a pointer to a string that the function has created itself, your HLA functions cannot treat that string as an HLA string because it's probably just a zero-terminated string. Fortunately, there are only a small number of Win32 API functions that return a string or modify an existing string, so you don't have to worry about this very often. However, that's a problem in and of itself; you may have to deal with this problem so infrequently that it slips your mind whenever you call a function that returns such a string. As a concrete example, consider the following (HLA) prototype for the Win32 `GetFullPathName` function:

```
static
   GetFullPathName: procedure
   (
          lpFileName    : string;
          nBufferLength : dword;
      var lpBuffer      : var;
      var lpFilePart    : var
   );
   @stdcall; @returns( "eax" ); @external( "__imp__GetFullPathNameA@16" );
```

This function, given the address of a zero-terminated string containing a filename (`lpFileName`), the length length of a buffer (`nBufferLength`), the address of a buffer (`lpBuffer`), and the address of a pointer variable (`lpFilePart`) will convert the filename you pass in to a full drive letter/path/filename string. Specifically, this function returns a string in the buffer whose address you specify in the `lpbuffer` parameter. However, this function does not create an HLA-compatible string; it simply creates a zero-terminated string.

Fortunately, most functions in the Win32 API that return string data do two things for you that will help make your life easier. First of all, Win32 API functions that store string data into your variables generally require that you pass a maximum length for the string (e.g., the `nBufferLength` parameter in the prototype above). The function will not write any characters to the buffer beyond this maximum length (doing so could corrupt other data in memory). The other important thing this function does (which is true for most Win32 API functions that return string data) is that it returns the length of the string in the EAX register; the function returns zero if there was some sort of error. Because of the way this function works, converting the return result to an HLA string is nearly trivial. Consider the following call to `GetFullPathName`:

```
static
   fullName :string;
   namePtr  :pointer to char;
      .
      .
```

```
    .
stralloc( 256 );       // Allocate sufficient storage to hold the string data.
mov( eax, fullName );
    .
    .
    .
mov( fullName, edx );
GetFullPathName
(
    "myfile.exe",                           // File to get the full path for.
    (type str.strRec [edx]).MaxStrLen,      // Maximum string size
    [edx],                                  // Pointer to buffer
    namePtr                                 // Address of base name gets stored here
);
mov( fullName, edx );                       // Note: Win32 calls don't preserve EDX
mov( eax, (type str.strRec [edx]).length  // Set the actual string length
```

The nice thing about most Win32 API calls that return string data is that they guarantee that they won't overwrite a buffer you pass in (you also pass in the maximum string length, which is available in the `MaxStrLen` field of the string object, that is, at offset -8 from the string pointer). These string functions also return the string length as the function's result, so you can shove this into the HLA string's `length` field (at offset -4 from the string's base address) immediately upon return. This is a very efficient way to convert C/C++ strings to HLA format strings upon return from a function.

Of course, converting a C/C++ string to an HLA string is only easy if the C/C++ function you're calling returns the length of the string it has processed. It also helps if the function guarantees that it won't overstep the bounds of the string variable you've passed it (i.e., it accepts a `MaxStrLen` parameter and won't write any data beyond the maximum buffer size you've specified). Although most Win32 API functions that return string data operate this way (respect a maximum buffer size and return the length of the actual string), there are many C/C++ functions you may need to call that won't do this. In such a case, you've got to compute the length of the string yourself (and guarantee that your character buffer is large enough to hold the maximum possible string the function will produce). Fortunately, there is a function in the HLA Standard Library, str.zlen, that will compute the length of a zero terminated string so you can easily update the length field of an HLA string object that a C/C++ function has changed (without respect to the HLA string's length field). For example, suppose you have a C/C++ function named `fstr` that expects the address of a character buffer where it can store (or modify) a zero-terminated string. Since HLA strings are zero-terminated, you can pass an HLA string to this function. However, if `fstr` changes the length of the string, the function will not update the HLA string's `length` field and the result will be inconsistent. You can easily correct this by computing the length of the string yourself and storing the length into the HLA string's `length` field, e.g.,

```
static
    someStr  :str.strvar( 255 );  // Allow strings up to 255 characters in length.
        .
        .
        .
    fstr( someStr );     // Assume this changes the length of someStr.

    mov( someStr, edx ); // Get the pointer to the string data structure.
    str.zlen( edx );     // Compute the new length of someStr.
    mov( eax, (type str.strRec [edx]).length );  // Update the length field.
```

One thing to keep in mind is that `str.zlen` has to search past every character in the string to find the zero-terminating byte in order to compute the string's length. This is not a particularly efficient operation, particularly if the

string is long. Although `str.zlen` uses a fairly decent algorithm to search for the terminating zero byte, the amount of time it takes to execute is still proportional to the length of the string. Therefore, you want to avoid calling `str.zlen` if at all possible. Fortunately, many C/C++ string functions (that modify string data) return the length as the function result, so calling `str.zlen` isn't always necessary.

Although not explicitly shown in the example above, do keep in mind that many string functions (especially Win32 API functions) assign double-duty to the function return result; they'll return a positive value if the function successfully produces a string and they'll return a negative result (or sometimes a zero result) if there was an error. For example, the `GetFullPathName` function in the earlier example returns zero if it there was a problem producing the string. You code should check for errors on return from these functions to prevent problems. While shoving a zero into a string length field isn't cause for concern (indeed, that's perfectly reasonable), a negative number will create all kinds of problems (since the HLA string functions treat the length field as an `uns32` value, those functions will interpret a negative number as a really large positive value).

3.2.2.5: C++ and Assembly Class Types

C++ and HLA both support classes. Since HLA is an assembly language, it should be obvious that anything you can do with a C++ class you can do with in HLA (since C++ compilers convert C++ source code into x86 machine code, it should be obvious that you can achieve anything possible in C++ using assembly language). However, directly translating C++ classes into HLA classes is not a trivial matter. Many constructs transfer across directly. Some constructs in C++, however, do not have a direct correspondence in HLA. Quite honestly, the conversion of C++ classes to HLA classes is beyond the scope of this book; fortunately, we're dealing with the Win32 API in this book so we won't see much Windows source code that uses classes in typical documentation we're interested it. There is quite a bit of C++-based object-oriented code out there for Windows but most of it uses the Microsoft Foundation Classes (MFC) class library, and that's not applicable to assembly language programming (at least, not today; someday there might be an "Assembly Language Foundation Class" library but until that day arrives we don't have to worry about MFC). Since this book covers Win32 API programming, there really is no need to worry about converting C++ classes into assembly - the Win32 API documentation doesn't use classes. This book may very well use HLA classes and objects in certain programming examples, but that will be pure assembly language, not the result of translating C++ code into HLA code.

3.2.3: C and Assembly Pointer Types[7]

When running under 32-bit versions of the Windows operating system, C pointers are always 32-bit values and map directly to 32-bit flat/linear machine addresses. This is true regardless of what the pointer references (including both data and functions in C). At one level, the conversion of a pointer from C to HLA/assembly is fairly trivial, just create a `dword` variable in you HLA program and keep the pointer there. However, HLA (like C) supports typed pointers and procedure pointers; generally, it's a good idea to map C pointers to their HLA typed-pointer equivalent.

In C, you can apply the address-of operator, '&' to an object (e.g., a static variable or a function name) to take the address of that object. In HLA, you may use this same operator to take the address of a *static* object (a `static/storage/readonly` variable, statement label, or procedure name). The result is a 32-bit value that you may load into a register or 32-bit memory variable. For example, if in C you have a statement like the following (`pi` is a pointer to an integer and `i` is a static integer variable):

7. Many programmers (and languages) consider pointers to be a scalar type because the value is not composed of other types. This book treats pointers as a separate classification simply because it has to discuss both scalar and composite data types before being able to discuss pointers.

```
pi = &i;
```

You convert this to HLA syntax as follows:

```
mov( &i, pi );
```

If the object you're taking the address of with the '&' operator is not a function name or a static (non-indexed) variable, then you must compute the address of the object at run-time using the `lea` instruction. For example, automatic variables (non-static local variables) fall into this class. Consider the following C function fragment:

```
int f( void )
{
   int *pi;   // Declares a variable that is a pointer to an integer.
   int i;     // Declares a local integer variable using automatic allocation.

   pi = &i;
      .
      .
      .
}
```

Because `i` is an automatic variable (the default storage class for local variables in a function) HLA cannot static compute this address for you at compile time. Instead, you would use the `lea` instruction as follows:

```
procedure f;
var
   pi: pointer to int32;  // Declare HLA pointers this way.
   i:  int32;             // Declares an automatic variable in HLA (in the VAR section)
      .
      .
      .
   lea( eax, i );    // Compute the run-time address of i
   mov( eax, pi );   // Save address in pi.
      .
      .
      .
```

You should also note that you need to use the `lea` instruction when accessing an indexed object, even if the base address is a static variable. For example, consider the following C/C++ code that takes the address of an array element:

```
int *pi;
static int array[ 16 ];
    .
    .
    .
pi = &array[i];
```

In order to access an array element in assembly language, you will need to use a 32-bit register as an index register and compute the actual element address at run-time rather than at compile-time (assuming that `i` in this example is a variable rather than a constant). Therefore, you cannot use the '&' operator to statically compute the address of this array element at compile-time, instead you should use the `lea` instruction as follows:

```
mov( i, ebx );                 // Move index into a 32-bit register.
lea( eax, array[ ebx*4 ] );    // int objects are four bytes under Win32
```

```
mov( eax, pi );
```

As you've seen in these simple examples, C uses the unary '*' operator to declare pointer objects. Anytime you see a declaration like the following in C:

```
typename *x;
```

you can convert it to a comparable HLA declaration as follows:

```
x:pointer to hla_typename;  // hla_typename is the HLA equivalent of C's typename
```

Of course, since all pointers are simply 32-bit objects, you can also convert to assembly language using a statement like the following:

```
x:dword;
```

Both forms are equivalent in assembly language, though the former version is preferable since it's a little more descriptive of what's going on here.

Function pointers in C is one area where the C syntax can get absolutely weird. A simple C function pointer declaration takes the following form:

```
returnType (*functionPtrName) ( parameters );
```

For example,

```
int (*ptrToFuncReturnsInt) ( int i, int j );
```

This example declares a pointer to a function that takes two integer parameters and returns an integer value. Note that the following is *not* equivalent to this example:

```
int *ptrToFuncReturnsInt( int i, int j );  //Not a function pointer declaration!
```

This example is a prototype for a function that returns a pointer to an integer, not a function pointer. C's syntax is a little messed up (i.e., it wasn't thought out properly during the early stages of the design), so you can get some *real* interesting function pointer declarations; some are nearly indecipherable.

Of course, at the assembly language level all pointers are just `dword` variables. So all you really need to implement the C function pointer in HLA is a statement like the following:

```
ptrToFuncReturnsInt: dword;
```

You can all this function in HLA using the `call` instruction as follows:

```
call( ptrToFuncReturnsInt );
```

As for data pointers, however, HLA provides a better solution: procedure variables. A procedure variable is a pointer object that (presumably) contains the address of some HLA procedure. The advantage of a procedure variable over a `dword` variable is that you can use HLA's high-level syntax calling convention with procedure

variables; something you cannot do with a `dword` variable. Here's an example of a procedure variable declaration in HLA:

```
ptrToFuncReturnsInt: procedure( i:int32; j:int32 );
```

HLA allows you to call the code whose address this variable contains using the standard HLA procedure call syntax, e.g.,

```
ptrToFuncReturnsInt( 5, intVar );
```

To make this same call using a `dword` variable[8], you'd have to manually pass the parameters yourself as follows (assuming you pass the parameters on the stack):

```
push( 5 );       // Pass first parameter (i)
push( intVar );  // Pass second parameter.
call( ptrToFuncReturnsInt );
```

In C, any time a function name appears without the call operator attached to it (the "call operator" is a set of parenthesis that may contain optional parameters), C substitutes the address of the function in place of the name. You do not have to supply the address-of operator ('&') to extract the function's address (though it is legal to go ahead and do so). So if you see something like the following in a C program:

```
ptrToFuncReturnsInt = funcReturnsInt;
```

where `funcReturnsInt` is the name of a function that is compatible with `ptrToFuncReturnsInt's` declaration (e.g., it returns an integer result and has two integer parameters in our current examples), this code is simply taking the address of the function and shoving it into `ptrToFuncReturnsInt` exactly as though you'd stuck the '&' operator in front of the whole thing. In HLA, you can use the '&' operator to take the address of a function (they are always static objects as far as the compiler is concerned) and move it into a 32-bit register or variable (e.g., a procedure variable). Here's the code above rewritten in HLA:

```
mov( &funcReturnsInt, ptrToFuncReturnsInt );
```

Both C and HLA allow you to initialize static variables (including pointer variables) at compile time. In C, you could do this as follows:

```
static int (*ptrToFuncReturnsInt) ( int i, int j) = funcRe-
turnsInt;
```

The comparable statement in HLA looks like this:

```
procedure funcReturnsInt( i:int32; j:int32 );
begin funcReturnsInt;
      .
      .
      .
end funcReturnsInt;
      .
      .
      .
static
```

8. Actually, this calling scheme works for HLA procedure variables, too.

```
   ptrToFuncReturnsInt:procedure( i:int32; j:int32 ) := &funcReturnsInt;
```

Especially note that in HLA, unlike C, you still have to use the ‘&’ operator when taking the address of a function name.

Note that you cannot initialize HLA automatic variables (`var` variables) using a statement like the one in this example. Instead, you must move the address of the function into the pointer variable using the `mov` instruction given a little earlier.

Arrays are another object that C treats specially with respect to pointers. Like functions, C will automatically supply the address of an array if you specify the name of an array variable without the corresponding index operator (the square brackets). HLA requires that you explicitly take the address of the array variable. If the array is a static object (static/readonly/storage) then you may use the (static) address-of operator, ‘&’; however, if the variable is an automatic (`var`) object, then you have to take the address of the object at run-time using the `lea` instruction:

```
static
   staticArray  :byte[10];
var
   autoArray    :byte[10];
       .
       .
       .
   mov( &staticArray, eax );   // Can use ‘&’ on static objects
   lea( ebx, autoArray );      // Must use lea on automatic (VAR) objects.
```

C doesn’t automatically substitute the address of a structure or union whenever it encounters a struct or union variable. You have to explicitly take the address of the object using the ‘&’ operator. In HLA, taking the address of a structure or union operator is very easy - if it’s a static object you can use the ‘&’ operator, if it’s an automatic (`var`) object, you have to use the lea instruction to compute the address of the object at run-time, just like other variables in HLA.

There are a couple of different ways that C/C++ allows you to dereference a pointer variable[9]. First, as you’ve already seen, to dereference a function pointer you simply “call” the function pointer the same way you would directly call a function: you append the call operator (parenthesis and possible parameters) to the pointer name. As you’ve already seen, you can do the same thing in HLA as well. Technically, you could also dereference a function pointer in C/C++ as follows:

```
            (*ptrToFuncReturnsInt)( 5, intVar );
```

However, you’ll rarely see this syntax in an actual C/C++ source file. You may convert an indirect function call in C/C++ to either HLA’s high-level or low-level syntax, e.g., the following two calls are equivalent:

```
      push( 5 );
      push( intVar );
      call( ptrToFuncReturnsInt );

// -or-

      ptrToFuncReturnsInt( 5, intVar );
```

9. Dereferencing means to access the data pointed at by a pointer variable.

Dereferencing a pointer to a data object is a bit more exciting in C/C++. There are several ways to dereference pointers depending upon the underlying data type that the pointer references. If you've got a pointer that refers to a scalar data object in memory, C/C++ uses the unary '*' operator. For example, if `pi` is a pointer that contains the address of some integer in memory, you can reference the integer value using the following C/C++ syntax:

```
*pi = i+2;  // Store the sum i+2 into the integer that pi points at.
j = *pi;    // Grab a copy of the integer's value and store it into j.
```

Converting an indirect reference to assembly language is fairly simple. The only gotcha, of course, is that you must first move the pointer value into an 80x86 register before dereferencing the pointer. The following HLA examples demonstrate how to convert the two C/C++ statements in this example to their equivalent HLA code:

```
// *pi = i + 2;

mov( pi, ebx );    // Move pointer into a 32-bit register first!
mov( i, eax );     // Compute i + 2 and leave sum in EAX
add( 2, eax );
mov( eax, [ebx] ); // Store i+2's sum into the location pointed at by EBX.

// j = *pi;

mov( pi, ebx );    // Only necessary if EBX no longer contains pi's value!
mov( [ebx], eax ); // Only necessary if EAX no longer contains *pi's value!
mov( eax, j );     // Store the value of *pi into j.
```

If you've got a pointer that holds the address of a sequence of data values (e.g., an array), then there are two completely different (but equivalent) ways you can indirectly access those values. One way is to use C/C++'s *pointer arithmetic syntax*, the other is to use *array syntax*. Assuming `pa` is a pointer to an array of integers, the following example demonstrates these two different syntactical forms in action:

```
*(pa+i) = j;   // Stores j into the ith object beyond the address held in pa.
pa[i] = j;     // Stores j into the ith element of the array pointed at by pa.
```

The important thing to note here is that both forms are absolutely equivalent and almost every C/C++ compiler on the planet generates exactly the same code for these two statements. Since compilers generally produce exactly the same code for these two statements, it should come as no surprise that you would manually convert these two statements to the same assembly code sequence. Conversion to assembly language is slightly complicated by the fact that you must remember to multiply the index into an array (or sequence of objects) by the size of each array element. You might be tempted to convert the statements above to something like the following:

```
mov( j, XXX );          // XXX represents some register that will hold j's value.
mov( pa, ebx );         // Get base address of array/sequence in memory
mov( i, ecx );          // Grab index
mov( XXX, [ebx][ecx] ); // XXX as above
```

The problem with this sequence is that it only works properly if each element of the array is exactly one byte in size. For larger objects, you must multiply the index by the size of an array element (in bytes). For example, if each element of the array is six bytes long, you'd probably use code like the following to implement these two C/C++ statements:

```
mov( j, eax );              // Get the L.O. four bytes of j.
mov( j[4], dx );            // Get the H.O. two bytes of j.
mov( pa, ebx );             // Get the base address of the array into EBX
mov( i, ecx );              // Grab the index
intmul( 6, ecx );           // Multiply the index by the element size (six bytes).
mov( eax, [ebx][ecx] );     // Store away L.O. four bytes
mov( dx, [ebx][ecx][4] ); // Store away H.O. two bytes.
```

Of course, if the size of your array elements is one of the four magic sizes of one, two, four, or eight bytes, then you don't need to do an explicit multiplication. You can get by using the 80x86 scaled indexed addressing mode as the following HLA example demonstrates:

```
mov( j, eax );
mov( pa, ebx );             // Get base address of array/sequence in memory
mov( i, ecx );              // Grab index
mov( eax, [ebx][ecx*4] ); // Store j's value into pa[i].
```

C uses yet another syntax when accessing fields of a structure indirectly (that is, you have a pointer to some structure in memory and you want to access a field of that structure via the pointer). The problem is that C's unary '*' (dereference) operator has a lower precedence than C's '.' (field access) operator. In order to access a field of some structure to which you have a pointer, you'd have to write an expression like the following when using the '*' operator:

```
(*ptrToStruct).field
```

Avoid the temptation to write this as follows:

```
*ptrToStruct.field
```

The problem with this latter expression is that '.' has a higher precedence than '*', so this expression tells the C compiler to dereference the thing that `ptrToStruct.field` points at. That is, `ptrToStruct` must be an actual struct object and it must have a field, `field`, that is a pointer to some object. This syntax indirectly references the value whose address `field` contains. An expression of the form "*(ptrToStruct).field" tells the compiler to first dereference the pointer `ptrToStruct` and then access `field` at the given offset from that indirect address.

Because accessing fields of a structure object indirectly is a common operation, using the syntax "(*ptrToStruct).field" tends to clutter up a program and make it less readable. In order to reduce the clutter the C/C++ programming language defines a second dereferencing operator that you use specifically to access fields of a structure (or union) via a pointer: the "->" operator. The "->" operator has the same precedence as the field selection operator (".") and they are both left associative. This allows you to write the following expression rather than the ungainly one given earlier:

```
ptrToStruct->field
```

Regardless of which syntax you find in the C/C++ code, in assembly language you wind up using the same code sequence to access a field of a structure indirectly. The first step is to always load the pointer into a 32-bit register and then access the field at some offset from that indirect address. In HLA, the best way to do this is to coerce an indirect expression like "[ebx]" to the structure type (using the "(type *XXX* [ebx])" syntax) and then use the "." field reference operator. For example,

```
type
   Struct:
      record
         field  :int32;
```

```
            .
            .
            .
      endrecord;
         .
         .
         .
static
   ptrToStruct :pointer to Struct;
      .
      .
      .
   // Access field "field" indirectly via "ptrToStruct"

   mov( ptrToStruct, ebx );
   mov( (type Struct [ebx]).field, eax );
```

For more details on the HLA type coercion operator, please consult the HLA language reference manual. Note that you use this same technique to indirectly access fields of a `union` or a `class` in HLA.

You may combine structure, array, and pointer types in C/C++ to form *recursive* and *nested* types. That is, you can have an array of structs, a struct may contain a field that is an array (or a struct), or you could even have a structure that has a field that is an array of structs whose fields are arrays of pointers to structs whose fields... In general, a C/C++ programmer can create an arbitrarily complex data structure by nesting array, struct, union, class, and pointer data types. Translating such objects into assembly language is equally complex, often taking a half dozen or more instructions to access the final object. Although such constructs rarely appear in real-world C/C++ programs, they do appear every now an then, so you'll need to know how to translate them into assembly language.

Although a complete description of every possible data structure access conversion would require too much space here, an example that demonstrates the process you would go through is probably worthwhile. For our purposes, consider the following complex C/C++ expression:

```
per->field[i].p->x[j].y
```

This expression uses three separate operators: "->", "[ ]", and ".". These three operators all have the same precedence and are left associative, so we process this expression strictly on a left-to-right basis[10]. The first object in this expression, `per`, is a pointer to some structure. So the first step in the conversion to HLA is to get this pointer value into a 32-bit register so the code can refer to the indirect object:

```
mov( per, ebx );
```

The next step is to access a field of the structure that `per` references. In this example, `field` is an array of structures and the code accesses element `i` of this array. To access this particular object, we need to compute the index into the array (by multiplying the index by the size of an array element). Assuming `field` is an array of `fieldpiece` objects, you'd use code like the following to reference the `i`th object of `field`:

```
mov( i, ecx );                      // Get the index into the field array.
intmul( @size( fieldpiece ), ecx );  // Multiply index by the size of an element
```

10.A later section in this chapter will discuss C/C++ operator precedence and associativity. Please see that section for more details concerning operator precedence and associativity.

The next step in the conversion of this expression is to access the `p` field of the `i`th array element of `field`. The following code does this:

```
mov( (type ptrsType[ebx]).p[ecx], edx );
```

The interesting thing to note here is that the index into the field array is tacked on to the end of the HLA address expression we've created, i.e., we write "(type ptrsType[ebx]).p[ecx]" rather than "(type ptrsType[ebx])[ecx].p". This is done simply because HLA doesn't allow this latter syntax. Because the "." and "[ ]" operators both involve addition and addition is commutative, it doesn't matter which syntax we use. Note that HLA would allow an address expression of the form "(type ptrsType [ebx][ecx]).p" but this tends to (incorrectly) imply that EBX points at an array of pointers, so we'll not use this form here[11].

The array element that "(type ptrsType [ebx]).p[ecx]" access is a pointer object. Therefore, we have to move this pointer into a 32-bit register in order to dereference the pointer. That's why the previous HLA statement moved this value into the EDX register. Now this pointer points at an array of array objects (the x field) and the code then accesses the `j`th element of this array (of structures). To do this, we can use the following HLA statements:

```
mov( j, esi );                  // Get the index into the array.
intmul( @size( xType ), esi );  // Multiply by the size of an array element.

// Now [edx][esi] references the jth element of the x field
```

The last step in our code sequence is to reference the `y` field of the `j`th element of the `x` array. Assuming that `y` is a double-word object (just to make things easy for this example), here's the code to achieve this:

```
mov( (type xType [edx]).y[esi], eax );
```

Here's the complete code sequence:

```
// ptr->field[i].p->x[j].y

    mov( ptr, ebx );
    mov( i, ecx );                      // Get the index into the field array.
    intmul( @size( fieldType ), ecx );  // Multiply index by the size of an element
    mov( (type ptrsType [ebx]).p[ecx], edx );
    mov( j, esi );                  // Get the index into the array.
    intmul( @size( xType ), esi );  // Multiply by the size of an array element.
    mov( (type xType [edx]).y[esi], eax );
```

3.2.4: C and Assembly Language Constants

Although we've already looked at literal constants in C/C++ and assembly language, we must still consider symbolic constants in C/C++ and their equivalents in assembly language. A *symbolic constant* is one that we refer to by name (an identifier) rather than the constant's value (i.e., a literal constant). Syntactically, a symbolic constant is an identifier and you use it as you would a variable identifier; semantically of course, there are certain limitations on symbolic constants (such as you cannot assign the value of an expression to a symbol constant). There are two types of symbolic constants you'll find in the C and C++ languages: *manifest constants* and *storage constants*.

11.The implication is only visual. This form is completely equivalent to the previous form since addition is *still* commutative.

The first type of constant to consider is a *manifest constant*. A manifest constant is a symbolic identifier to which you've bound (assigned) some value. During compilation, the compiler simply substitutes the actual value of the constant everywhere it encounters the manifest constant's identifier. From the standpoint of the compiler's code generation algorithms, there is no difference between a manifest constant and a literal constant. By the time the code generator sees the constants, it's a literal value. In C/C++, you can use the `#define` preprocessor directive to create manifest constants. Note that you can use a manifest constant anywhere a literal constant is legal.

There are two ways to convert a C/C++ manifest constant into assembly language. The first way, of course, is to manually do the translation from symbolic to literal constant. That is, whenever you encounter a manifest constant in C/C++, you simply translate it to the corresponding literal constant in your assembly language code. E.g.,

```
#define someConst 55
       .
       .
       .
   i = someConst;
```

in assembly language becomes:

```
   mov( 55, i );
```

Obviously, manually expanding manifest constants when converting C/C++ code into assembly language is not a good idea. The reasons for using symbolic constants in C/C++ apply equally well to assembly language programs. Therefore, the best solution is to keep the constants symbolic when translating the C/C++ code to assembly language. In HLA, the way to create manifest constants is by using the `const` declaration section. The exact form of the translation depends how the C/C++ code uses the `#define` preprocessor directive. Technically, the `#define` preprocessor directive in C/C++ doesn't define a constant, it defines a macro. There are two basic forms of the `#define` directive:

```
#define someID  some text...

#define someID(parameter list) some text...
```

We'll not consider the second form here, since that's a true macro declaration. We'll return to macros and how to convert this second example to HLA a little later in this chapter.

The first `#define` directive in this example defines a *textual substitution macro*. The C/C++ preprocessor will substitute the text following `someID` for each occurrence of `someID` appearing after this declaration in the source file. Note that the text following the declaration can be *anything*, it isn't limited to being a literal constant. For the moment, however, let's just consider the case where the text following the `#define` directive and the identifier is a single literal constant. This begin the case, you can create an equivalent HLA manifest constant declaration using code like the following:

```
// #define fiftyFive 55

const
   fiftyFive := 55;
```

Like C/C++, HLA will substitute the literal value 55 for the identifier `fiftyFive` everywhere it appears in the HLA source file.

There is a subtle difference between HLA manifest constants you define in a `const` section and manifest constants you define with C/C++'s `#define` directive: HLA's constants are typed while C/C++ `#define` constants are untyped textual substitutions. Generally, however, you will not notice a difference between the two. However, one special case does deserve additional comment: the case where a program specifies a constant expression rather than a single literal constant (which both languages allow). Consider the following statements in C/C++ and in HLA:

```
// C/C++ constant expression:

#define constExpr i*2+j

// HLA constant expression:

const
   constExpr := i*2+j;
```

The difference between these two is that the C/C++ preprocessor simply saves up the text "i*2+j" and emits that string whenever it encounters `constExpr` in the source file. C/C++ does not require that `i` and `j` be defined prior to the `#define` statement. As long as these identifiers have a valid definition prior to the first appearance of `constExpr` in the source file, the C/C++ code will compile correctly. HLA, on the other hand, evaluates the constant expression at the point of the declaration. So `i` and `j` must be defined at the point of the constant declaration (another difference is that HLA requires `i` and `j` to both be constants whereas C/C++ doesn't require this; though if `i` and `j` are not constant objects then this isn't really a manifest constant declaration as we're defining it here, so we won't worry about that).

Beyond the fact that C/C++ relaxes the requirement that i and j be defined before the manifest constant declaration, there is another subtle difference between C/C++ constant declarations and HLA constant declarations: *late binding*. HLA evaluates the value of the expression at the point you declare the constant in your source file (which is why `i` and `j` have to be defined at that point in the previous example). C/C++, on the other hand, only evaluates the constant expression when it actually expands the symbolic identifier in your source file. Consider the following C/C++ source fragment:

```
#define constExpr i*2 + j
#define i 2
#define j 3

   printf( "1:%d\n", constExpr );

#define i 4  //The compiler may issue a warning about this

   printf( "2:%d\n", constExpr );
```

The first `printf` statement in this example will display the value seven (2*2+3) whereas the second example will display 11 (4*2+3). Were you to do the equivalent in HLA (using `val` constants and the "?" statement in HLA, see the HLA reference manual for more details), you would get a different result, e.g.,

```
program t;
#include( "stdlib.hhf" )

? i := 2; // Defines i as a redefinable constant
? j := 3;
const
```

```
   constExpr := i*2 + j;

begin t;

   stdout.put( "1:", constExpr, nl );
   ? i := 4;
   stdout.put( "2:", constExpr, nl );

end t;
```

The HLA code prints the strings "1:7" and "2:7" since HLA only computes the expression "i*2+j" once, when you define the `constExpr` manifest constant.

HLA does allow the definition of *textual substitution constants* using the `text` data type in the const section. For example, consider the following HLA `const` declaration:

```
const
   constExpr :text := "i*2+j";
```

This declaration is totally equivalent to the C/C++ `#define` declaration. However, as you cannot drop in arithmetic expressions into assembly code at arbitrary points in your source file, this textual substitution isn't always legal in an assembly file as it might be in a C/C++ source file. So best not to attempt to use textual substitution constants like this. For completeness' sake, however, the following HLA example demonstrates how to embed textual substitution constant expressions in an HLA source file (and have the compiler calculate the expression at the point of expansion):

```
program t;
#include( "stdlib.hhf" )

? i := 2; // Defines i as a redefinable constant
? j := 3;

const

   // Note: "@eval" tells HLA to evaluate the constant expression inside
   //       the parentheses at the point of expansion.  This avoids some
   //       syntax problems with the stdout.put statements below.

   constExpr :text := "@eval(i*2 + j)";

begin t;

   stdout.put( "1:", constExpr, nl );
   ? i := 4;
   stdout.put( "2:", constExpr, nl );

end t;
```

This HLA example prints "1:7" and "2:11" just like the C/C++ example. Again, however, if the C/C++ manifest constant expansion depends upon late binding (that is, computing the value of the expression at the point of use rather than the point of declaration in the source file) then you should probably expand the text manually at each point of use to avoid problems.

The other way to define constant objects in C/C++ is to use the `const` keyword. By prefacing what looks like an initialized C/C++ variable declaration with the const keyword, you can create constant (immutable at run-time) values in your program, e.g.,

```
const int cConst = 4;
```

Although C/C++ lets you declare constant objects using the `const` keyword, such constants possess different semantics than manifest and literal constants. For example, in C/C++ you may declare an array as follows:

```
#define maxArray 16

int array[ maxArray ];
int anotherArray[ maxArray ];
```

However, the following is generally not legal in C/C++:

```
const int maxBounds = 8;
int iArray[ maxBounds ];
```

The difference between manifest constants and `const` objects in C/C++ has to do with how the program treats the constant object at run-time. Semantically, C++ treats `const` objects as read-only variables. If the CPU and operating system support write-protected memory, the compiler may very well place the const object's value in write-protected memory to enforce the read-only semantics at run-time. Other than the compiler doesn't allow you to store the result of some expression into a `const` object, there is little difference between `const` and `static` variable declarations in C++. This is why a declaration like the one for `iArray` earlier is illegal. C/C++ does not allow you to specify an array bounds using a variable and (with the exception of the read-only attribute) `const` objects are semantically equivalent to variables. To understand why C/C++ `const` objects are not manifest constants and why such declarations are even necessary in C/C++ (given the presence of the `#define` preprocessor directive), we need to look at how CPUs encode constants at the machine code level.

The 80x86 provides special instructions that can encode certain constants directly in a machine instruction. Consider the following two 80x86 assembly language instructions:

```
mov( maxBound, eax );  // Copy maxBound's value into eax
mov( 8, eax );         // Copy the value eight into eax
```

In both cases the machine code the CPU executes consists of three components: an opcode that tells the processor that it needs to move data from one location to another; an addressing mode specification that specifies the register, whether the register is a destination or source register, and the format the other operand takes (it could be a register, a memory location, or a constant); and the third component is the encoding of the memory address or the actual constant. The instruction that copies `maxBound's` value into EAX encodes the address of the variable as part of the instruction whereas the instruction that copies the value eight into EAX encodes the 32-bit value for eight into the instruction. At the machine level, there is a fundamental difference between the execution of these two instructions – the CPU requires an extra step to fetch `maxBound's` value from memory (and this fact remains true even if you initialize `maxBound` to eight and place `maxBound` in write-protected memory). Therefore, the CPU treats certain types of literal and manifest constants differently than it does other types of constants.

Note that a constant (literal, manifest, or otherwise) object in a high level language does not imply that the language encodes that constant as part of a machine instruction. Most CPUs only allow you to encode integer constants (and in some cases, *small* integer constants) directly in an instruction's opcode. The 80x86, for example, does not allow you to encode a floating point constant within a machine instruction. Even if the CPU were capable of encoding floating point and all supported integer values as immediate constants within an opcode,

high level languages like C/C++ support the declaration of large data objects as constant data. For example, you could create a constant array in C/C++ as follows:

```
const int constArray[4] = {0,1,2,3};
```

Few processors, if any, support the ability to encode an arbitrary array constant as immediate data within a machine instruction. Similarly, you'll rarely find structure/record constants or string constants encoded directly within an instruction. Support for large structured constants is the main reason C/C++ adds a another class of constants to the language.

A high level language compiler may encode a literal constant or a manifest constant as an instruction's immediate operand. There is no guarantee, however, that the compiler might actually do this; the CPU must support immediate constants of the specified type and the compiler write must choose to emit the appropriate immediate addressing mode along with the instruction. On the other hand, constants that are not manifest constants (e.g., `const` objects in C/C++) are almost always encoded as memory references rather than as immediate data to an instruction.

So why would you care whether the compiler emits a machine instruction that encodes a constant as part of the opcode versus accessing that constant value appearing in memory somewhere? After all, since machine instructions appear in memory, an immediate constant encoded as part of an instruction also appears in memory. So what's the difference? Well, the principle difference is that accessing a constant value appearing elsewhere in memory (i.e., not as immediate data attached to the instruction) requires twice as much memory to encode. First, you need the constant itself, consuming memory somewhere; second, you need the address that constant encoded as part of the instruction. Since the address of a constant value typically consumes 32-bits, it typically takes twice as much memory to encode the access to the constant. Of course, if you reference the same constant value throughout your code, the compiler should only store one copy of the constant in memory and every instruction that references that constant would reference the same memory location. However, even if you amortize the size of the constant access over any number of instructions, the bottom line is that encoding constants as memory location still takes more room than encoding immediate constants.

Another difference between manifest/literal constants and read-only objects is that decent compilers will compute the result of constant expressions at compile-time, something that it may not be able to do with read-only objects. Consider the following C++ code:

```
#define one 1
#define two 2
const int three = 3;
const int four = 4;
int i;
int j;

   i = one + two;
   j = three + four;
```

Most decent C/C++ compilers will replace the first assignment statement above with the following:

```
i = 3;  // Compiler compute 1+2 at compile-time
```

On the 80x86 processor this statement takes a single machine instruction to encode (this is generally true for most processors). Some compilers, however, may not precompute the value of the expression "three+four" and will, instead, emit machine instructions to fetch these values from their memory locations and add them at run-time.

HLA provides a mechanism whereby you can create immutable "variables" in your code if you need the storage semantics of a variable that the program must not change at run-time. You can use HLA's `readonly` declaration section to declare such objects, e.g.,

```
readonly
   constValue :int32 := -2;
```

For all intents and purposes, HLA treats the `readonly` declaration like a `static` declaration. The two major differences are than HLA requires an initializer associated with all `readonly` objects and HLA attempts to place such objects in read-only memory at run time. Note that HLA doesn't prevent you from attempting to store a value into a `readonly` object. That is, the following is perfect legal and HLA will compile the program without complaint:

```
mov( 56, constValue );
```

Of course, if you attempt to execute the program containing this statement, the program will probably abort with an *illegal access* violation when the program attempts to execute this statement. This is because HLA will place this object in write-protected memory and the operating system will probably raise an exception when you attempt to execute this statement.

A compiler may not be able to efficiently process a constant simply because it is a literal constant or a manifest constant. For example most CPUs are not capable of encoding a string constant in an instruction. Using a manifest string constant may actually make your program less efficient. Consider the following C code:

```
#define strConst "A string constant"
      .
      .
      .
   printf( "string: %s\n", strConst );
      .
      .
      .
   sptr = strConst;
      .
      .
      .
   result = strcmp( s, strConst );
      .
      .
      .
```

Because the compiler (actually, the C preprocessor) expands the macro `strConst` to the string literal "A string constant" everywhere the identifier `strConst` appears in the source file, the above code is actually equivalent to:

```
      .
      .
      .
   printf( "string: %s\n", "A string constant" );
      .
      .
      .
   sptr = "A string constant";
      .
```

```
    .
    .
result = strcmp( s, "A string constant" );
    .
    .
    .
```

The problem with this code is that the same string constant appears at different places throughout the program. In C/C++, the compiler places the string constant off in memory somewhere and substitutes a pointer to that string for the string literal constant. A naive compiler would wind up making three separate copies of the string in memory, thus wasting space since the data is exactly the same in all three cases. Compiler writers figured this out a couple of decades ago and modified their compilers to keep track of all the strings the compiler had already emitted; when the program used the same literal string constant again, the compiler would not allocate storage for a second copy of the string, it would simply return the address of the earlier string appearing in memory. Such an optimization could reduce the size of the code the compiler produced by a fair amount if the same string appears through some program. Unfortunately, this optimization probably lasted about a week before the compiler vendors figured out that there were problems with this approach. One major problem with this approach is that a lot of C programs would assign a string literal constant to a character pointer variable and then proceed to change the characters in that literal string, e.g.,

```
sptr = "A String Constant";
    .
    .
    .
*(sptr+2) = 's';
    .
    .
    .
printf( "string: '%s'\n", sptr ); /* displays "string: 'A string Constant'" */
    .
    .
    .
printf( "A String Constant" );    /* Prints "A string Constant"! */
```

Compilers that used the same string constant in memory for multiple occurrences of the same string literal appearing in the program quickly discovered that this trick wouldn't work if the user stored data into the string object, as the code above demonstrates. Although this is a bad programming practice, it did occur frequently enough that the compiler vendors could not use the same storage for multiple copies of the same string literal. Even if the compiler vendor were to place the string literal constant into write-protected memory to prevent this problem, there are other semantic issues that this optimization raise. Consider the following C/C++ code:

```
sptr1 = "A String Constant";
sptr2 = "A String Constant";
s1EQs2 = sptr1 == sptr2;
```

Will `s1EQs2` contain true (1) or false (0) after executing this instruction sequence? In programs written before C compilers had strong optimizers available, this sequence of statements would leave false in `s1EQs2` because the compiler created two different copies of the same string data and placed those strings at different addresses in memory (so the addresses the program assigns to `sptr1` and `sptr2` would be different). In a later compiler, that kept only a single copy of the string data in memory, this code sequence would leave true sitting in `s1EQs2` since

both `sptr1` and `sptr2` would be pointing at the same address in memory; this difference exists regardless of whether the string data appears in write-protected memory.

Of course, when converting the C/C++ code to assembly language, it is your responsibility to determine whether you can merge strings and use a common copy of the data or whether you will have to use a separate copy of the string data for each instance of the symbolic constant throughout your assembly code.

C/C++ supports other composite constant types as well (e.g., arrays and structures/records). This discussion of string constants in a program applies equally well to these other data types. Large data structures that the CPU cannot represent as a primitive data type (i.e., hold in a general purpose register) almost always wind up stored in memory and the program access the constant data exactly as it would access a variable of that type. On many modern systems, the compiler may place the constant data in write-protected memory to prevent the program from accidentally overwriting the constant data, but otherwise the "constant" is structurally equivalent to a variable in every sense except the ability to change its value. You can place such "constant" declarations in an HLA `readonly` declaration section to achieve the HLA equivalent to the C/C++ code.

HLA also allows the declaration of composite constants in the `const` section. For example, the following is perfect legal in HLA:

```
const
   constArray :int32[4] := [1,2,3,4];
```

However, you should note that HLA maintains this array strictly at compile-time within the compiler. You cannot, for example, write HLA code like the following once you have the above declaration:

```
   for( mov( 0, ebx ); ebx < 4; inc( ebx )) do

      mov( constArray[ ebx*4 ], eax );
      stdout.puti32( eax );

   endfor;
```

The problem with this code is that constArray is not a memory location so you cannot refer to it using an 80x86 addressing mode. In order for this array constant to be accessible (as an array) at run time, you have to make a copy of it in memory. You can do this with an HLA declaration like the following:

```
readonly
   rtConstArray :int32[4] := constArray; //Assuming the declaration given earlier.
```

Please consult the HLA documentation for more details on structured (composite) constants. Although these are quite useful for HLA programmers, they aren't generally necessary when converting C/C++ code to HLA. As such, they're a bit beyond the scope of this chapter so we won't deal any farther with this issue here.

3.2.5: Arithmetic Expressions in C and Assembly Language

One of the major advances that high level languages provided over low level languages was the use of algebraic-like expressions. High level language arithmetic expressions are an order of magnitude more readable than the sequence of machine instructions the compiler converts them into. However, this conversion process (from arithmetic expressions into machine code) is also one of the more difficult transformation to do efficiently and a fair percentage of a typical compiler's optimization phase is dedicated to handling this transformation.

Computer architects have made extensive studies of typical source files and one thing they've discovered is that a large percentage of assignment statements in such programs take one of the following forms:

```
var = var2;
var = constant;
var = op var2;
var = var op var2;
var = var2 op var3;
```

Although other assignments do exist, the set of statements in a program that takes one of these form is generally larger than any other group of assignment statements. Therefore, computer architects have generally optimized their CPUs to efficiently handle one of these forms.

The 80x86 architecture is what is known as a *two-address machine*. In a two-address machine, one of the source operands is also the destination operand. Consider the following 80x86/HLA `add` instruction:

```
add( ebx, eax );  ; computes eax := eax + ebx;
```

Two-address machines, like the 80x86, can handle the first four forms of the assignment statement given earlier with a single instruction. The last form, however, requires two or more instructions and a temporary register. For example, to compute "var1 = var2 + var3;" you would need to use the following code (assuming `var2` and `var3` are memory variables and the compiler is keeping `var1` in the EAX register):

```
mov( var2, eax );
add( var3, eax );  //Result (var1) is in EAX.
```

Once your expressions get more complex than the five forms given earlier, the compiler will have to generate a sequence of two or more instructions to evaluate the expression. When compiling the code, most compilers will internally translate complex expressions into a sequence of "three address statements" that are semantically equivalent to the more complex expression. The following is an example of a more complex expression and a sequence of three-address instructions that are representative of what a typical compiler might produce:

```
// complex = ( a + b ) * ( c - d ) - e/f;

temp1 = a + b;
temp2 = c - d;
temp1 = temp1 * temp2;
temp2 = e / f;
complex = temp1 - temp2;
```

If you study the five statements above, you should be able to convince yourself that they are semantically equivalent to the complex expression appearing in the comment. The major difference in the computation is the introduction of two temporary values (`temp1` and `temp2`). Most compilers will attempt to use machine registers to maintain these temporary values (assuming there are free registers available for the compiler to use).

Table 3-5 lists most of the arithmetic operators used by C/C++ programs as well as their associativity.

Table 3-5: C/C++ Operators, Precedence, and Associativity

Precedence	Operator Classification	Associativity	C/C++ Operators
1 (highest)	Primary Scope Resolution (C++)	left to right	::
2	Primary	left to right	( ) [ ] . ->
3	Unary (monadic[a])	right to left	++ -- + - ! ~ & * (*type*) sizeof new delete
4	Multiplicative (dyadic[b])	left to right	* / %
5	Additive (dyadic)	left to right	+ -
6	Bitwise Shift (dyadic)	left to right	<< >>
7	Relational (dyadic)	left to right	< > <= >=
8	Equality (dyadic)	left to right	== !=
9	Bitwise AND (dyadic)	left to right	&
10	Bitwise Exclusive OR (dyadic)	left to right	^
11	Bitwise Inclusive OR (dyadic)	left to right	|
12	Logical AND (dyadic)	left to right	&&
13	Logical OR (dyadic)	left to right	||
14	Conditional (triadic[c])	right to left	? :
15	Assignment	right to left	= += -= *= /= <<= >>= %= &= ^= |=
16 (lowest)	Comma	left to right	,

a. Monadic means single operand .

b.Dyadic means two operand .
c.Triadic means three operands .

3.2.5.1: Converting Simple Expressions Into Assembly Language

In this section we will consider the case where we need to convert a simple C/C++ expressions into assembly language. We've already discussed the conversion of the primary operators (in Table 3-5) into assembly language, so we won't bother repeating that discussion here. Likewise, we've already discussed the address-of operator ("&") and the dereferencing operator ("*") so we'll skip their discussions here as well.

Although the conversion of the remaining operators into assembly language is generally obvious, there are a few peculiarities. So it's worthwhile to quickly discuss how to convert a simple expression of the form @X, X@, or X@Y (where '@' represents one of the operators found in Table 3-5) into assembly language. Note that the discussion that follows deals with integer (signed or unsigned) only. The conversion of floating point expressions into assembly language is actually easier than converting integer expressions. This book will not deal with converting floating point expressions into assembly language. There are two reasons for this: (1) once you see how to convert integer expressions to assembly, you'll discover than floating point expression conversion is very similar; (2) the Win32 API uses very few floating point values. The whole reason for this chapter is to describe the C/C++ to assembly conversion process so you can read and understand existing C/C++ documentation when writing Windows assembly code. Since you won't find much Windows programming documentation that involves the use of floating point arithmetic, there is little need to present that information here. If you're interested in the subject, be sure to check out the discussion of this process in *The Art of Assembly Language Programming*.

Each of the examples appearing in this section will assume that you're operating on 32-bit integers producing a 32-bit result (except in the case of boolean results, where this book will assume an 8-bit result is probably sufficient). If you need to operate on eight or sixteen bit values, so sweat, just substitute the 8-bit or 16-bit registers in place of the 32-bit registers you'll find in these examples. If you need to deal with larger values (e.g., long long ints), well, that's beyond the scope of this book; please see the section on extended precision arithmetic in *The Art of Assembly Language* for details on those operations.

Translating the ++ and -- (increment and decrement) operators from C/C++ to assembly language looks, at first, like a trivial operation. You simply substitute an `inc` or `dec` instruction for these operators. However, there are two details that complicate this conversion by a slight amount: pre- and post- increment/decrement operations and pointer increment/decrement operations.

Normally, when you apply the ++ (increment) operator to an integer variable, the ++ operator increments the value of that variable by one. Similarly, when you apply the -- (decrement) operator to an integer variable, the -- operator decrements that variable by one. However, C/C++ also allows you to apply the ++ and -- operators to pointer variables as well as integers (pointers and integer variables are the only legal objects to which you may apply these operators, though). The semantics of a pointer increment are different than the semantics of an integer increment; applying the ++ operator to a pointer increments that pointer variable *by the size of the object at which the pointer refers*. For example, if `pi` is a pointer that points at a 32-bit integer value somewhere in memory, then `++pi` adds four to `pi` (rather than one); this cause `pi` to point at the next sequential memory location that can hold a 32-bit integer (without overlapping the current integer in memory). Similarly, the -- (decrement) operator subtracts the size of the object at which a pointer refers from the pointer's value. So `--pi` would subtract

four from `pi` if `pi` points at a 32-bit integer. So the basic conversion of the ++ and -- operator to assembly language is as Table 3-6 describes.

Table 3-6: Converting C/C++ Increment/Decrement Operators to Assembly

C/C++	HLA[a]
int i; ++i; i++;	inc(i);
int *pi; ++pi; pi++;	add(@size(int32), i);
int i; --i; i--;	dec(i);
int *pi; --pi; pi--;	sub(@size(int32), i);

a.In the pointer examples, substitute the appropriate type identifier when incrementing a pointer to some type other than int32.

The increment and decrement operators may appear before or after a variable. If a C/C++ statement consists of a single variable with one of these operators, then whether you use the *pre-increment/decrement form* (sticking the ++ or -- before the variable) or the *post-increment/decrement* form (placing the ++ or -- operator after the variable) is irrelevant. In either case the end result is that the program will increment or decrement the variable accordingly:

```
    c++;  // is equivalent to
    ++c;

// and

    --c;  // is equivalent to
    c--;
```

If the C/C++ increment and decrement operators are attached to a variable within a larger expression, then the issue of pre-increment/decrement versus post-increment/decrement makes a big difference in the final result. Consider the statements “a = ++c;” and “a = c++;”. In both cases the program will add one to variable `c` (assuming c is an integer rather than a pointer to some object). However, these two statements are quite different with respect to the value they assign to variable `a`. The first example here first increments the value in `c` and then assigns the value in `c` to `a` (hence the term *pre-increment* since the program first increments `c` and then uses its value in the expression). The second statement here first grabs the value in `c`, assigns that to `a`, and then increments `c` (hence the term *post-increment* since this expression increments `c` after using its value). Here’s some sample HLA code that implements these two statements:

```
// a = ++c;

    inc( c );          // pre-increment the value in c.
    mov( c, eax );
    mov( eax, a  );

// a = c++;

    mov( c, eax );
    mov( eax, a );
    inc( c );          // post-increment the value in c.
```

The C/C++ compiler effectively ignores the unary "+" operator. If you attach this operator to an operand, it does not affect that value of that operand in any way. It's presence in the language is mainly for notational purposes. It lets you specify positive numeric constants like +123.456 in the source file. Sometimes explicitly place the "+" in front of such a constant can make the program more readable. However, since this operator rarely appears in real-world C/C++ programs, you're unlikely to see it.

The unary "-" operator negates the expression/variable that immediately follows it. The important thing to note is that this operator negates the value of the operand immediately following the "-" for use in the expression containing the operator. In particular, if a simple variable immediately follows the unary "-" this operator doesn't negate that operator directly. Therefore, you cannot use the 80x86 neg instruction on the variable except for the very special case where you have a statement like the following:

```
i = -i;
```

Instead, you must move the value of the variable into a register, negate the value of that register, and then use that register's value within the expression. For example, consider the following:

```
//  j = -i;

   mov( i, eax );
   neg( eax );
   mov( eax, j );
```

The unary "!" operator is the logical not operator. If the sub-expression (i.e., variable) appearing immediately to the left of this operator is zero, the "!" operator returns one. If the value of that sub-expression is non-zero, this operator returns zero. To convert this to assembly language, what you would do is test the operand for zero and set the result to one if it is zero, zero if the operand is not zero. You can use the `cmp` (or `test`) instruction along with the 80x86 `setne` instruction to achieve this:

```
// Convert !i to assembly, assume i is an int variable, leave result in AL(or EAX)

cmp( i, 0 );
setne( al );
// movsx( al, eax );  // Do this if you need a 32-bit boolean result.
```

A very common construct you'll see in many C/C++ programs is a sub-expression like "!!i" (that is, apply the logical not operator twice to the same value. This *double logical negation* converts the value in `i` to zero if it was previously zero, to one if it was previously non-zero. Rather than execute the previous code fragment twice, you can easily achieve this effect as follows:

```
// Convert !!i to assembly, assume i is an int variable, leave result in AL(or EAX)
```

```
cmp( i, 0 );
sete( al );
// movsx( al, eax );  // Do this if you need a 32-bit boolean result.
```

The C/C++ unary "~" operator does a bitwise logical not on its operand (that is, it inverts all the bits of the operand). This is easily achieved using the 80x86 `not` instruction as follows:

```
// j = ~i

    mov( i, eax );
    not( eax );
    mov( i, j );
```

For the special case of "i = ~i;" you can use the 80x86 `not` instruction to negate i directly, i.e., "not(i);".

A simple C/C++ expression like "x = y * z;" is easily converted to assembly language using a code sequence like the following:

```
// x = y * z;

mov( y, eax );
intmul( z, eax );
mov( eax, x );

// Note: if y is a constant, can do the following:
// (because multiplication is commutative, this also works if z is a constant,
//  just swap z and y in this code if that is the case):

intmul( y, z, eax );
mov( eax, x );
```

Technically, the `intmul` instruction expects signed integer operands so you would normally use it only with signed integer variables. However, if you're not checking for overflow (and C/C++ doesn't so you probably won't need to either), then a two's complement signed integer multiply produces exactly the same result as an unsigned multiply. See *The Art of Assembly Language* if you need to do a true unsigned multiply or an extended precision multiply. Also note that the intmul instruction only allows 16-bit and 32-bit operands. If you need to multiple two 8-bit operands, you can either zero extend them to 16 (or 32) bits or you can use the 80x86 `imul` or `mul` instructions (see *The Art of Assembly Language* for more details).

The C/C++ division and modulo operators ("/" and "%", respectively) almost translate into the same code sequence. This is because the 80x86 `div` and `idiv` instructions calculate both the quotient and remainder of a division at the same time.

Unlike integer multiplication, division of signed versus unsigned operands does not produce the same result. Therefore, when dividing values that could potentially be negative, you must use the `idiv` instruction. Only use the `div` instruction when dividing unsigned operands.

Another complication with the division operation is that the 80x86 does a 64/32-bit division (that is, it divides a 64-bit number by a 32-bit number). Since both C/C++ operands are 32-bits you will need to sign extend (for signed integer operands) or zero extend (for unsigned integer operands) the numerator to 64 bits. Also remember that the `div` and `idiv` instructions expect the numerator in the EDX:EAX register pair (or DX:AX for 32/16 divisions, or AH:AL for 16/8 divisions, see *The Art of Assembly Language* for more details). The last thing to note is

that these instructions return the quotient in EAX (AX/AL) and they return the remainder in EDX (DX/AH). Here's the code to translate an expression of the form "x=y/z;" into 80x86 assembly code:

```
// x = y / z;   -- assume all operands are unsigned.

mov( y, eax );
xor( edx, edx );  // zero extend EAX to 64 bits in EDX:EAX
div( z );
mov( eax, x );    // Quotient winds up in EAX


// x = y % z;   -- assume all operands are unsigned.

mov( y, eax );
xor( edx, edx );  // zero extend EAX to 64 bits in EDX:EAX
div( z );
mov( edx, x );    // Remainder winds up in EDX


// x = y / z;   -- assume all operands are signed.

mov( y, eax );
cdq();            // sign extend EAX to 64 bits in EDX:EAX
idiv( z );
mov( eax, x );    // Quotient winds up in EAX


// x = y % z;   -- assume all operands are signed.

mov( y, eax );
cdq();            // sign extend EAX to 64 bits in EDX:EAX
idiv( z );
mov( edx, x );    // Remainder winds up in EDX
```

Converting C/C++ expressions involving the "+", "-", "&", "|", and "&" operators into assembly language is quite easy. A simple C/C++ expression of the form "a = b @ c;" (where '@' represents one of these operators) translates into the following assembly code:

```
//  a = b @ c;

   mov( b, eax );
   instr( c, eax );  //instr = add, sub, and, or, xor, as appropriate
   mov( eax, a );
```

The C/C++ programming language provides a shift left operator ("<<"). This dyadic operator returns the result of its left operand shifted to the left the number of bits specified by its right operand. An expression of the form "a=b<<c;" is easily converted to one of two different HLA instruction sequences (chosen by whether `c` is a constant or a variable expression) as follows:

```
// a = b << c;   -- c is a constant value.

mov( b, eax );
shl( c, eax );
mov( eax, a );
```

```
// a = b << c;   -- c is a variable value in the range 0..31.

mov( b, eax );
mov( (type byte c), cl );  //assume H.O. bytes of c are all zero.
shl( cl, eax );
mov( eax, a );
```

C/C++ also provides a shift right operator, ">>". This translates to a sequence that is very similar to the conversion of the "<<" operator with one caveat: the 80x86 supports two different shift right instructions: `shr` (shift logical right) and `sar` (shift arithmetic right). The C/C++ language doesn't specify which shift you should use. Some compilers always use a logical shift right operation, some use a logical shift right for unsigned operands and they use an arithmetic shift right for signed operands. If you don't know what you're supposed to use when converting code, using a logical (unsigned) shift right is probably the best choice because this is what most programmers will expect. That being the case, the shift right operator (">>") appearing in an expression like "a=b>>c;" translates into 80x86 code thusly:

```
// a = b >> c;   -- c is a constant value.

mov( b, eax );
shr( c, eax );
mov( eax, a );

// a = b >> c;   -- c is a variable value in the range 0..31.

mov( b, eax );
mov( (type byte c), cl );  //assume H.O. bytes of c are all zero.
shr( cl, eax );
mov( eax, a );
```

If you decide you need to use an arithmetic (signed) shift right operation, simply substitute `sar` for `shr` in this code.

The logical OR and logical AND operators ("||" and "&&") return the values zero or one based on the values of their two operands. The logical OR operator ("||") returns one if either or both operands are non-zero; it returns zero if both operands are zero. The logical AND operator ("&&") returns zero if either operand is zero, it returns one only if both operands are non-zero. There is, however, one additional issue to consider: these operators employ *short-circuit boolean evaluation*. When computing "X && Y" the logical AND operator will not evaluate Y if it turns out that X is false (there is no need because if X is false, the full expression is always false). Likewise, when computing "X || Y" the logical OR operator will not evaluate Y if it turns out that X is true (again, there will be no need to evaluate Y for if X is true the result is true regardless of Y's value). Probably for the majority of expressions it doesn't really matter whether the program evaluates the expression using short-circuit evaluation or *complete boolean evaluation*; the result is always the same. However, because C/C++ promises short-circuit boolean evaluation semantics, many programs are written to depend on these semantics and will fail if you recode the expression using complete boolean evaluation. Consider the following two examples that demonstrate two such situations:

```
if( pi != NULL && *pi == 5 )
{
   // do something if *pi == 5...
}
    .
    .
    .
```

```
if( --x == 0 || ++y < 10 )
{
   // do something if x == 0 or y < 10
}
```

The first `if` statement in this example only works properly in all cases when using short-circuit evaluation. The left-hand operand of the "&&" operator evaluates false if `pi` is NULL. In that case, the code will not evaluate the right operand and this is a good thing for if it did it would dereference a NULL pointer (which will raise an exception under Windows). In the second example above, the result is not as drastic were the system to use short-circuit evaluation rather than complete boolean evaluation, but the program would produce a different result in `y` when using complete boolean evaluation versus short-circuit boolean evaluation. The reason for this difference is that the right-hand side of the expression increments `y`, something that doesn't happen if the left operand evaluates true.

Handling short-circuit boolean evaluation almost always means using conditional jumps to skip around an expression. For example, given the expression "Z = X && Y" the way you would encode this using pure short-circuit evaluation is as follows:

```
   xor( eax, eax );   // Assume the result is false.
   cmp( eax, X );     // See if X is false.
   je isFalse;
   cmp( eax, Y );     // See if Y is false.
   je isFalse;

      inc( eax );     // Set EAX to 1 (true);

isFalse:
   mov( eax, Z );     // Save 0/1 in Z
```

Encoding the logical OR operator using short-circuit boolean evaluation isn't much more difficult. Here's an example of how you could do it:

```
   xor( eax, eax );   // Assume the result is false.
   cmp( eax, X );     // See if X is true.
   jne isTrue;
   cmp( eax, Y );     // See if Y is false.
   je isFalse;
isTrue:
      inc( eax );     // Set EAX to 1 (true);

isFalse:
   mov( eax, Z );     // Save 0/1 in Z
```

Although short-circuit evaluation semantics are crucial for the proper operation of certain algorithms, most of the time the logical AND and OR operands are simple variables or simple sub-expressions whose results are independent of one another and quickly computed. In such cases the cost of the conditional jumps may be more expensive than some simple straight-line code that computes the same result (this, of course, depends entirely on which processor you're using in the 80x86 family). The following code sequence demonstrates one (somewhat tricky) way to convert "Z = X && Y" to assembly code, assuming `X` and `Y` are both 32-bit integer variables:

```
   xor( eax, eax ); // Initialize EAX with zero
   cmp( X, 1 );     // Note: sets carry flag if X == 0, clears carry in all other cases.
   adc( 0, eax );   // EAX = 0 if X != 1, EAX = 1 if X = 0.
```

```
    cmp( Y, 1 );      // Sets carry flag if Y == 0, clears it otherwise.
    adc( 0, eax );    // Adds in one if Y = 0, adds in zero if Y != 0.
    setz( al );       // EAX = X && Y
    mov( eax, Z );
```

Note that you cannot use the 80x86 and instruction to merge `X` and `Y` together to test if they are both non-zero. For if `X` contained $55 (0x55) and `Y` contained $aa (0xaa) their bitwise AND (which the `and` instruction produces) is zero, even though both values are logically true and the result should be true. You may, however, use the 80x86 `or` instruction to compute the logical OR of two operands. The following code sequence demonstrates how to compute "Z = X || Y;" using the 80x86 `or` instruction:

```
    xor( eax, eax );  // Clear EAX's H.O. bytes
    mov( X, ebx );
    or( Y, ebx );
    setnz( al );      // Sets EAX to one if X || Y (in EBX) is non-zero.
```

The conditional expression in C/C++ is unusual insofar as it is the only ternary (three-operand) operator that C/C++ provides. An assignment involving the conditional expression might be

```
a = (x != y) ? trueVal : falseVal;
```

The program evaluates the expression immediately to the left of the "?" operator. If this expression evaluates true (non-zero) then the compiler returns the value of the sub-expression immediately to the right of the "?" operator as the conditional expression's result. If the boolean expression to the left of the "?" operator evaluates false (i.e., zero) then the conditional expression returns the result of the sub-expression to the right of the ":" in the expression. Note that the conditional operator does not evaluate the true expression (`trueVal` in this example) if the condition evaluates false. Likewise, the conditional operator does not evaluate the false expression (`falseVal` in this example) if the expression evaluates true. This is similar to short-circuit boolean evaluation in the "&&" and "||" operators. You encode the conditional expression in assembly as though it were an `if/else` statement, e.g.,

```
    mov( falseVal, edx );   // Assume expression evaluates false
    mov( x, eax );
    cmp( eax, y );
    jne TheyreNotEqual
    mov( trueVal, edx );    // Assumption was incorrect, set EDX to trueVal
TheyreNotEqual:
    mov( edx, a );          // Save trueVal or falseVal (as appropriate) in a.
```

C/C++ provides a set of assignment operators. The are the following:

```
=   +=   -=   &=   ^=   |=   *=   /=   %=   <<=   >>=
```

Generally, C/C++ programmers use these assignment operators as stand-alone C/C++ statements, but they may appear as subexpressions as well. If the C/C++ program uses these expressions as stand-alone statements (e.g., "x += y;") then the statement "x @= y;" is completely equivalent to "x = x @ y;" where '@' represents the operator above. Therefore, the conversion to assembly code is fairly trivial, you simply use the conversions we've

been studying throughout this section. Table 3-7 lists the equivalent operations for each of the assignment operators.

Table 3-7: Converting Assignment Operators To Assembly Language

<table>
<tr><th>C/C++ Operator</th><th>C/C++ Example</th><th>Equivalent To This C/ C++ Code</th><th>HLA Encoding</th></tr>
<tr><td>=</td><td>x = y;</td><td>x = y;</td><td>mov(y, eax);
mov(eax, x);</td></tr>
<tr><td>+=</td><td>a += b;</td><td>a = a + b;</td><td>mov(b, eax);
add(eax, a);</td></tr>
<tr><td>-=</td><td>a -= b;</td><td>a = a - b;</td><td>mov(b, eax);
sub(eax, a);</td></tr>
<tr><td>&=</td><td>a &= b;</td><td>a = a & b;</td><td>mov(b, eax);
and(eax, a);</td></tr>
<tr><td>|=</td><td>a |= b;</td><td>a = a | b;</td><td>mov(b, eax);
or(eax, a);</td></tr>
<tr><td>^=</td><td>a ^= b;</td><td>a = a ^ b;</td><td>mov(b, eax);
xor(eax, a);</td></tr>
<tr><td><<=</td><td>a <<= b;</td><td>a = a << b;</td><td>mov((type byte b), cl);
shl(cl, a);</td></tr>
<tr><td>>>=</td><td>a >>= b;</td><td>a = a >> b;</td><td>mov((type byte b), cl);
shr(cl, a);</td></tr>
<tr><td>*=</td><td>a *= b;</td><td>a = a * b;</td><td>mov(a, eax);
intmul(b, eax);
mov(eax, a);</td></tr>
<tr><td>/=</td><td>a /= b;</td><td>a = a / b;</td><td>mov(a, eax);
cdq; // or xor(edx, edx);
div(b);
mov(eax, a); // Store away quotient</td></tr>
<tr><td>%=</td><td>a %= b;</td><td>a = a % b;</td><td>mov(a, eax);
cdq; // or xor(edx, edx);
div(b);
mov(edx, a); // Store away remainder</td></tr>
</table>

The comma operator in C/C++ evaluates two subexpressions and then throws the result of the first expression away (i.e., it computes the value of the first/left expression strictly for any side effects it produces). In general, just convert both sub-expressions to assembly using the rules in this section and then use the result of the second sub-expression in the greater expression, e.g.,

```
//  x = ( y=z, a+b );

   mov( z, eax );
   mov( eax, y );
   mov( a, eax );
   add( b, eax );
   mov( eax, x );
```

3.2.5.2: Operator Precedence

The precedence of an operator resolves the ambiguity that is present in an expression involving several different operands. For example, given the arithmetic expression “4+2*3” there are two possible values we could logically claim this expression produces: 18 or 10 (18 is achieve by adding four and two then multiplying their sum by three; 10 is achieved by multiplying two times three and adding their product together with four). Now you may be thoroughly convinced that 10 is the *correct* answer, but that’s only because by convention most people agree that multiplication has a higher precedence than addition, so you must do the multiplication first in this expression (that is, you’ve followed an existing convention in order to resolve the ambiguity). C/C++ also has its own precedence rules for eliminating ambiguity. In Table 3-5 the *precedence level* appears in the left-most column. Operators with a lower precedence level have a higher precedence and, therefore, take precedence over other operators at a lower precedence. You’ll notice that the multiplication operator in C/C++ (“*”) has a higher precedence than addition (“+”) so C/C++ will produce 10 for the expression “4+2*3” just as you’ve been taught to expect.

Of course, you can always eliminate the ambiguity by explicitly specifying parentheses in your expressions. Indeed, the whole purpose of precedence is to implicitly specify where the parentheses go. If you have two operators with different precedences (say ‘#’ and ‘@’) and three operands, and you have an expression of the form X#Y@Z then you must place parentheses around the operands connected by the operator with the higher precedence. In this example, if ‘@’ has a higher precedence than ‘#’ you’d wind up with X#(Y@Z). Conversely, if ‘#’ has a higher precedence than ‘@’ you’d wind up with (X#Y)@Z.

An important fact to realize when converting C/C++ code into assembly language is that precedence only controls the implicit placement of parentheses within an expression. That is, precedence controls which operands we associate with a given operator. Precedence does not necessarily control the order of evaluation of the operands. For example, consider the expression “5*4+2+3”. Since multiplication has higher precedence than addition, the “5” and “4” operands attach themselves to the “*” operator (rather than “4” attaching itself to the “+” operator). That is, this expression is equivalent to “(5*4)+2+3”. The operator precedence, contrary to popular opinion, does not control the order of the evaluation of this expression. We could, for example, compute the sub-expression “2+3” prior to computing “5*4”. You still get the correct result when computing this particular addition first.

When converting a complex expression to assembly language, the first step is to explicitly add in the parentheses implied by operator precedence. The presence of these parentheses will help guide the conversion to assembly language (we’ll cover the exact process a little later in this chapter).

3.2.5.3: Associativity

Precedence defines where the parentheses go when you have three or more operands separated by different operators at different precedence levels. Precedence does not deal with the situation where you have three or more operands separated by operators at the same precedence level. For example, consider the following expression:

```
5 - 4 - 3;
```

Does this compute "5 - (4 - 3);" or does it compute " (5 - 4) -3;"? Precedence doesn't answer the question for us because the operators are all the same. These two expressions definitely produce different results (the first expression produces 5-1=4 while the second produces 1-3=-2). Associativity is the mechanism by which we determine the placement of parentheses around adjacent operators that have the same precedence level.

Operators generally have one of three different associativities: left, right, and none. C/C++ doesn't have any non-associative operators, so we'll only consider left associative and right associative operators here. Table 3-5 lists the associativity of each of the C/C++ operators (left or right). If two left associative operators are adjacent to one another, then you place the left pair of operands and their operator inside parentheses. If two right associative operators appear adjacent to one another in an expression, then you place a pair of parentheses around the right-most pair of operands and their operator, e.g.,

```
5 - 4 - 3      -becomes-    (5 - 4) - 3
x = y = z      -becomes-    x = (y = z)
```

Like precedence, associativity only controls the implied placement of the parentheses within an expression. It does not necessarily suggest the order of evaluation. In particular, consider the following arithmetic expression:

```
5 + 4 + 3 + 2 + 1
```

Because addition is left associative, the implied parentheses are as follows:

```
(((5 + 4) + 3) + 2) + 1
```

However, a compiler is not forced to first compute 5+4, then 9 + 3, then 12 + 2, etc. Because addition is commutative, a compiler can rearrange this computation in any way it sees fit as long as it produces the same result as this second expression.

3.2.5.4: Side Effects and Sequence Points

A side effect is any modification to the global state of a program other than the immediate result a piece of code is producing. The primary purpose of an arithmetic expression is to produce the expression's result. Any other changes to the system's state in an expression is a side effect. The C/C++ language is especially guilty of allowing side effects in an arithmetic expression. For example, consider the following C/C++ code fragment:

```
i = i + *pi++ + (j = 2) * --k
```

This expression exhibits four separate side effects: the decrement of `k` at the end of the expression, the assignment to `j` prior to using `j's` value, the increment of the pointer `pi` after dereferencing `pi`, and the assignment to `i` (generally, if this expression is converted to a stand-alone statement by placing a semicolon after the expression, we consider the assignment to `i` to be the purpose of the statement, not a side effect).

Another way to create side effects within an expression is via a function call. Consider the following C++ code fragment:

```
int k;
int m;
int n;

int hasSideEffect( int i, int& j )
{
```

```
    k = k + 1;
    hasSideEffect = i + j;
    j = i;
}
        .
        .
        .
    m = hasSideEffect( 5, n );
```

In this example, the call to the `hasSideEffect` function produces two different side effects: (1) the modification of the global variable `k` and the modification of the pass by reference parameter `j` (actual parameter is `n` in this code fragment). The real purpose of the function is to compute the function's return result; any modification of global values or reference parameters constitutes a side effect of that function, hence the invocation of such a function within an expression causes the expression to produce side effects. Note that although C does not provide "pass by reference" parameters as C++ does, you can still pass a pointer as a parameter and modify the dereferenced object, thus achieving the same effect.

The problem with side effects in an expression is that most C/C++ compilers do not guarantee the order of evaluation of the components that make up an expression. Many naive programmers (incorrectly!) assume that when they write an expression like the following:

```
i = f(x) + g(x);
```

the compiler will emit code that first calls function `f` and then calls function `g`. The C and C++ programming languages, however, do not specify this order of execution. That is, some compilers will indeed call `f` , then call `g`, and then add their return results together; some other compilers, however, may call `g` first, then `f`, and then compute the sum of the function return results. That is, the compiler could translate the expression above into either of the following simplified code sequences before actually generating native machine code:

```
// Conversion #1 for "i = f(x) + g(x);"

    temp1 = f(x);
    temp2 = g(x);
    i:= temp1 + temp2;

// Conversion #2 for "i = f(x) + g(x);"

    temp1 = g(x);
    temp2 = f(x);
    i = temp2 + temp1;
```

Note that issues like precedence, associativity, and commutativity have no bearing on whether the compiler evaluates one sub-component of an expression before another. For example, consider the following arithmetic expression and several possible intermediate forms for the expression:

```
    j = f(x) - g(x) * h(x);

// Conversion #1 for this expression:

    temp1 = f(x);
    temp2 = g(x);
    temp3 = h(x);
    temp4 = temp2 * temp3
    j = temp1 - temp4;
```

```
// Conversion #2 for this expression:

   temp2 = g(x);
   temp3 = h(x);
   temp1 = f(x);
   temp4 = temp2 * temp3
   j = temp1 - temp4;

// Conversion #3 for this expression:

   temp3 = h(x);
   temp1 = f(x);
   temp2 = g(x);
   temp4 = temp2 * temp3
   j = temp1 - temp4;
```

Many other combinations are possible.

The specification for the C/C++ programming languages explicitly leave the order of evaluation undefined. This may seem somewhat bizarre, but there is a good reason for this: sometimes a compiler can produce better machine code by rearranging the order it uses to evaluate certain sub-expressions within an expression. Any attempt on the part of the language designer to force a particular order of evaluation on a compiler's implementor may limit the range of optimizations possible. Therefore, very few languages explicitly state the order of evaluation for an arbitrary expression.

There are, of course, certain rules that most languages do enforce. Though the rules vary by language, there are some fairly obvious rules that most languages (and their implementation) always follow because intuition suggests the behavior. Probably the two most common rules that you can count on are the facts that all side effects within an expression occur prior to the completion of that statement's execution. For example, if the function `f` modifies the global variable `x`, then the following statements will always print the value of `x` after `f` modifies it:

```
   i = f(x);
   printf( "x= %d\n", x );
```

Another rule you can count on is that the assignment to a variable on the left hand side of an assignment statement does not get modified prior to the use of that same variable on the right hand side of the expression. I.e., the following will not write a temporary value into variable `n` until it uses the previous value of `n` within the expression:

```
n = f(x) + g(x) - n;
```

Because the order of the production of side effects within an expression is undefined in C/C++, the result of the following code is generally undefined:

```
int incN( void )
{
   incN = n;
   n := n + 1;
}
      .
      .
      .
```

```
n = 2;
printf( "%d\n", incN() + n*2 );
```

The compiler is free to first call the `incN` function (so `n` will contain three prior to executing the sub-expression "n*2") or the compiler may first compute "n*2" and then call the `incN` function. As a result, one compilation of this statement could produce the output "8" while a different compilation of this statement might produce the output "6". In both cases `n` would contain three after the execution of the `writeln` statement, but the order of computation of the expression in the writeln statement could vary.

Don't make the mistake of thinking you can run some experiments to determine the order of evaluation. At the very best, such experiments will tell you the order a particular compiler uses. A different compiler may very well compute sub-expressions in a different order. Indeed, the same compiler might also compute the components of a subexpression differently based on the context of that subexpression. This means that a compiler might compute the expression using one ordering at one point in the program and using a different ordering somewhere else *in the same program*. Therefore, it is very dangerous to "determine" the ordering your particular compiler uses and rely on that ordering. Even if the compiler is consistent in the ordering of the computation of side effects, what's to prevent the compiler vendor from changing this in a later version of the compiler?

As noted earlier, most languages do guarantee that the computation of side effects completes before certain points in your program's execution. For example, almost every language guarantees the completion of all side effects by the time the statement containing the expression completes execution. The point at which a compiler guarantees that the computation of a side effect is completed is called a *sequence point*. The end of a statement is an example of a sequence point.

In the C programming language, there are several important sequence points in addition to the semicolon at the end of a statement. C provides several important sequence points within expressions, as well. Beyond the end of the statement containing an expression, C provides the following sequence points:

```
expression1, expression2                  (the C comma operator in an expression)
expression1 && expression2                (the C logical AND operator)
expression1 || expression2                (the C logical OR operator)
expression1 ? expression2 : expression3   (the C conditional expression operator)
```

C[12] guarantees that all side effects in `expression1` are completed before the computation of `expression2` or `expression3` in these examples (note that for the conditional expression, C only evaluates one of `expression2` or `expression3`, so only the side effects of one of these sub-expressions is ever done on a given execution of the conditional expression).

To understand how side effects and sequence points can affect the operation of your program in non-obvious ways, consider the following example in C:

```
int array[6] = {0, 0, 0, 0, 0, 0};
int i;
    .
    .
    .
i = 0;
array[i] = i++;
```

12.C++ compilers generally provide the same sequence points at C, although the original C++ standard did not define any sequence points.

Note that C does not define a sequence point across the assignment operator. Therefore, the C language makes no guarantees about whether the expression "i" used as an index into `array` is evaluated before or after the program increments `i` on the right hand side of the assignment operator. Note that the fact that the "++" operator is a post increment operation only implies that "i++" returns the value of `i` prior to the increment; this does not guarantee that the compiler will use the pre-increment value of `i` anywhere else in the expression. The bottom line is that the last statement in this example could be semantically equivalent to either of the following statements:

```
   array[0] = i++;
-or-
   array[1] = i++;
```

The C language definition allows either form and, in particular, does not require the first form simply because the array index appears in the expression before the post-increment operator.

To control the semantics of the assignment to `array` in this example, you will have to ensure that no part of the expression depends upon the side-effects of some other part of the expression. That is, you cannot both use the value of `i` at one point in the expression and apply the post-increment operator to `i` in another part of the expression unless there is a sequence point between the two uses. Since no such sequence point exists between the two uses of `i` in this statement, the result is undefined by the C language standard (note that the standard actually says that the result is *undefined*; therefore, the compiler could legally substitute *any* value for `i` as the array index value, though most compilers will substitute the value of `i` before or after the increment occurs in this particular example).

Though this comment appears earlier in this section, it is worth being redundant to stress an important fact: operator precedence and associativity do not control when a computation takes place in an expression. Even though addition is left associative, the compiler may compute the value of the addition operator's right operand before it computes the value of the addition operator's left operand. Precedence and associativity control how the compiler arranges the computation to produce the final result. They do not control when the program computes the subcomponents of the expression. As long as the final computation produces the results one would expect on the basis of precedence and associativity, the compiler is free to compute the subcomponents in any order and at any time it pleases.

3.2.5.5: Translating C/C++ Expressions to Assembly Language

Armed with the information from the past several sections, it is now possible to intelligently describe how to convert complex arithmetic expressions into assembly language.

The conversion of C/C++ expressions into assembly language must take into consideration the issues of operator precedence, associativity, and sequence points. Fortunately, these rules only describe which operators you must apply to which operands and at which points you must complete the computation of side effects. They do not specify the order that you must use when computing the value of an expression (other than completing the computation of side effects before a given point). Therefore, you have a lot of latitude with respect to how you rearrange the computation during your conversion. Because the C/C++ programming language has some relaxed rules with regard to the order of computation in some expression, the result of a computation that relies on other side effects between a pair of sequence points is undefined. However, just because the language doesn't define the result, some programmers will go ahead and assume that the compiler computes results on a left to right basis within the statement. That is, if two subexpressions modify the value of some variable, the programmer will probably (though errantly) assume that the left-most side effect occurs first. So if you encounter such an undefined operation in a C/C++ code sequence that you're converting to assembly, the best suggestion is to compute the result using a left-to-right evaluation of the expression. Although this is no guarantee that you'll produce

what the original programmer intended, chances are better than 50/50 that this will produce the result that programmer was expecting.

A complex expression that is easy to convert to assembly language is one that involves three terms and two operators, for example:

```
w = w - y - z;
```

Clearly the straight-forward assembly language conversion of this statement will require two `sub` instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```
        mov( w, eax );
        sub( y, eax );
        sub( z, eax );
        mov( eax, w );
and
        mov( y, eax );
        sub( z, eax );
        sub( eax, w );
```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result. Associativity is the problem. The second sequence above computes "W = W - (Y - Z);" which is not the same as "W = (W - Y) - Z;". How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```
        mov( y, eax );
        add( z, eax );
        sub( eax, w );
```

This computes "W=W-(Y+Z);". This is equivalent to "W = (W - Y) - Z;".

Precedence is another issue. Consider the C/C++ expression:

```
X = W * Y + Z;
```

Once again there are two ways we can evaluate this expression:

```
        X = (W * Y) + Z;
or
        X = W * (Y + Z);
```

However, C/C++'s precedence rules dictate the use of the first of these statements.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
// w = x + y * z;

        mov( x, ebx );
        mov( y, eax );        // Must compute y*z first since "*"
        intmul( z, eax );     //  has higher precedence than "+".
        add( ebx, eax );
        mov( eax, w );
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w = x - y - z

      mov( x, eax );   // All the same operator, so we need
      sub( y, eax );   //  to evaluate from left to right
      sub( z, eax );   //  because they all have the same
      mov( eax, w );   //  precedence and are left associative.

// w = x + y * z

      mov( y, eax );      // Must compute Y * Z first since
      intmul( z, eax );   // multiplication has a higher
      add( x, eax );      // precedence than addition.
      mov( eax, w );


// w = x / y - z

      mov( x, eax );      // Here we need to compute division
      cdq();              //  first since it has the highest
      idiv( y, edx:eax ); //  precedence.
      sub( z, eax );
      mov( eax, w );

// w = x * y * z

      mov( y, eax );      // Addition and multiplication are
      intmul( z, eax );   // commutative, therefore the order
      intmul( x, eax );   // of evaluation does not matter
      mov( eax, w );
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is generally better to perform the multiplication first. For example, given an expression of the form:

```
W = X/Y * Z         // Note: this is x/y×z not x/(y×z)!
```

It is usually better to compute `X*Z` and then divide the result by `Y` rather than divide `X` by `Y` and multiply the quotient by `Z`. There are two reasons this approach is better. First, remember that the `imul` instruction always produces a 64 bit result (assuming 32 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the EDX register so you do not have to sign extend EAX prior to the division. This saves the execution of the `cdq` instruction. A second reason for doing the multiplication first is to increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
w = x/y*z;
```

You can usually convert it to the assembly code:

```
mov( x, eax );
imul( z, eax );    // Note the use of IMUL, not INTMUL!
idiv( y, edx:eax );
mov( eax, w );
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following C/C++ statement:

```
w = x - y * x;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute `y * z` and then subtract `x` from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load `x` into a register, multiply `y` and `z` leaving their product in a different register, and then subtract this product from `x`, e.g.,

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of `x` until it computes the product of `y` and `z`. As your expressions increase in complexity, the need for temporaries grows. Consider the following C/C++ statement:

```
w = (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you've computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

$Temp_1 = a + b;$
$Temp_2 = y + z;$
$w = Temp_1 * Temp_2;$

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov( a, eax );
add( b, eax );
mov( eax, Temp1 );
mov( y, eax );
add( z, eax );
mov( eax, Temp2 );
mov( Temp1, eax );
intmul( Temp2, eax );
```

```
mov( eax, w );
```

Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```

Yet another example:

```
x = (y+z) * (a-b) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 = (y+z)
Temp2 = (a-b)
Temp1 = Temp1 * Temp2
X = Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov( y, eax );      // Compute eax = y+z
add( z, eax );
mov( a, ebx );      // Compute ebx = a-b
sub( b, ebx );
imul( ebx, eax );   // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that you should attempt to keep temporary values, in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the result. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

As noted earlier, this text will not consider the conversion of floating point expressions into 80x86 assembly language. Although the conversion is slightly different (because of the stack-oriented nature of the FPU register file), the conversion of floating point expressions into assembly language is so similar to the conversion of integer expressions that it isn't worth the space to discuss it here. For more insight into this type of expression conversion, please see *The Art of Assembly Language*.

3.2.6: Control Structures in C and Assembly Language

The C and C++ languages provide several high-level *structured* control statements. Among these, you will find the `if/else` statement, the `while` statement, the `do/while` statement, the `for` statement, the `break/continue/return` statements, and the `goto` statement. C/C++ also provides the function call, but we'll deal with that control structure later in this chapter. The C++ language provides exception handling facilities. However, as you're unlikely to encounter C++'s `try/catch` statements in Win32 API documentation, we won't bother discussing the conversion of those statements into assembly language in this book. If you have need to incorporate exception handling into your HLA programs, please check out the HLA `try..exception..endtry` statements in the HLA reference manual.

One advantage of a high level assembler like HLA is that it also provides high-level, structured, control statements. Although not as sophisticated as the similar statements you'll find in C/C++ (particularly with respect to the boolean expressions the C/C++ statements allow), it's fairly trivial to convert about 75-90% of the typical C/C++ control statements you'll encounter into assembly language (when using HLA).

This book will not cover the conversion of high level control structures into low-level assembly code (i.e., using conditional jumps and comparisons rather than the high-level control structures found in HLA). If you wish to use that conversion process and you're not comfortable with it, please see *The Art of Assembly Language* for more details.

For the most part, this book assumes that the reader is already an accomplished assembly language programmer. However, because many assembly language programmers might not have bothered to learn HLA's high level control structures, the following sections will describe the semantics of the HLA structured control statements in addition to describing how to convert C/C++ control structures into their equivalent assembly language statements. Since C/C++ does not provide as many control structures as C/C++, this section will not bother describing all of HLA's high level control structures - only those that have a C/C++ counterpart. For more details on HLA's high level control statements, please consult the HLA Reference Manual.

3.2.6.1: Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the `if, while,` and `repeat..until` statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high level control structures. In many cases you cannot convert the corresponding C/C++ statements directly into HLA code.

HLA boolean expressions always take the following forms[13]:

```
flag_specification
!flag_specification
register
!register
Boolean_variable
!Boolean_variable
mem_reg relop mem_reg_const
```

A *flag_specification* may be one of the following symbols:

- @c carry: True if the carry is set (1), false if the carry is clear (0).
- @nc no carry: True if the carry is clear (0), false if the carry is set (1).

13.There are a few additional forms, some of which we'll cover a little later in this section..

- @z zero: True if the zero flag is set, false if it is clear.
- @nz not zero: True if the zero flag is clear, false if it is set.
- @o overflow: True if the overflow flag is set, false if it is clear.
- @no no overflow: True if the overflow flag is clear, false if it is set.
- @s sign: True if the sign flag is set, false if it is clear.
- @ns no sign: True if the sign flag is clear, false if it is set.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a non-zero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or non-zero (true). Since HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires such variables be of type boolean. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. Table 3-8 lists the legal combinations.

Table 3-8: Relational Operators in HLA

<table>
<tr><th>Left Operand</th><th>Relational Operator</th><th>Right Operand</th></tr>
<tr><td rowspan="6">Memory Variable
or
Register</td><td>= or ==</td><td rowspan="6">Memory Variable,
Register,
or
Constant</td></tr>
<tr><td><> or !=</td></tr>
<tr><td><</td></tr>
<tr><td><=</td></tr>
<tr><td>></td></tr>
<tr><td>>=</td></tr>
</table>

Note that both operands cannot be memory operands. In fact, if you think of the *Right Operand* as the source operand and the *Left Operand* as the destination operand, then the two operands must be the same that `cmp` instruction allows. This is the primary limitation to HLA boolean expressions and the biggest source of problems when converting C/C++ high level control statements into HLA code.

Like the `cmp` instruction, the two operands must also be the same size. That is, they must both be byte operands, they must both be word operands, or they must both be double word operands. If the right operand is a constant, it's value must be in the range that is compatible with the left operand.

There is one other issue: if the left operand is a register and the right operand is a positive constant or another register, HLA uses an *unsigned* comparison. You will have to use HLA's type coercion operator (e.g., "(type int32 eax)") if you wish to do a signed comparison.

Here are some examples of legal boolean expressions in HLA:

```
@c
Bool_var
```

```
al
ESI
EAX < EBX
EBX > 5
i32 < -2
i8 > 128
al < i8
```

HLA uses the “&&” operator to denote logical AND in a run-time boolean expression. This is a dyadic (two-operand) operator and the two operands must be legal run-time boolean expressions. This operator evaluates true if both operands evaluate to true. Example using an HLA `if` statement:

```
if( eax > 0 && ch = 'a' ) then

    mov( eax, ebx );
    mov( ' ', ch );

endif;
```

The two `mov` statements appearing here execute only if EAX is greater than zero *and* CH is equal to the character ‘a’. If either of these conditions is false, then program execution skips over these `mov` instructions.

Note that the expressions on either side of the “&&” operator may be any legal boolean expression, these expressions don’t have to be comparisons using the relational operators. For example, the following are all legal expressions:

```
@z && al in 5..10
al in 'a'..'z' && ebx
boolVar && !eax
```

HLA uses *short circuit evaluation* when compiling the “&&” operator. If the left-most operand evaluates false, then the code that HLA generates does not bother evaluating the second operand (since the whole expression must be false at that point). Therefore, in the last expression, the code will not check EAX against zero if `boolVar` contains false.

Note that an expression like “eax < 0 && ebx <> eax” is itself a legal boolean expression and, therefore, may appear as the left or right operand of the “&&” operator. Therefore, expressions like the following are perfectly legal:

eax < 0 && ebx <> eax && !ecx

The “&&” operator is left associative, so the code that HLA generates evaluates the expression above in a left-to-right fashion. If EAX is less than zero, the CPU will not test either of the remaining expressions. Likewise, if EAX is not less than zero but EBX is equal to EAX, this code will not evaluate the third expression since the whole expression is false regardless of ECX’s value.

HLA uses the “||” operator to denote disjunction (logical OR) in a run-time boolean expression. Like the “&&” operator, this operator expects two legal run-time boolean expressions as operands. This operator evaluates true if either (or both) operands evaluate true. Like the “&&” operator, the disjunction operator uses short-circuit evaluation. If the left operand evaluates true, then the code that HLA generates doesn’t bother to test the value of the second operand. Instead, the code will transfer to the location that handles the situation when the boolean expression evaluates true. Examples of legal expressions using the “||” operator:

```
@z || al = 10
```

```
al in 'a'..'z' || ebx
!boolVar || eax
```

As for the "&&" operator, the disjunction operator is left associative so multiple instances of the "||" operator may appear within the same expression. Should this be the case, the code that HLA generates will evaluate the expressions from left to right, e.g.,

```
eax < 0 || ebx <> eax   ||  !ecx
```

The code above executes if either EAX is less than zero, EBX does not equal EAX, or ECX is zero. Note that if the first comparison is true, the code doesn't bother testing the other conditions. Likewise, if the first comparison is false and the second is true, the code doesn't bother checking to see if ECX is zero. The check for ECX equal to zero only occurs if the first two comparisons are false.

If both the conjunction and disjunction operators appear in the same expression then the "&&" operator takes precedence over the "||" operator. Consider the following expression:

```
eax < 0 || ebx <> eax  && !ecx
```

The machine code HLA generates evaluates this as

```
eax < 0 || (ebx <> eax  && !ecx)
```

If EAX is less than zero, then the code HLA generates does not bother to check the remainder of the expression, the entire expression evaluates true. However, if EAX is not less than zero, then both of the following conditions must evaluate true in order for the overall expression to evaluate true.

HLA allows you to use parentheses to surround sub-expressions involving "&&" and "||" if you need to adjust the precedence of the operators. Consider the following expression:

```
(eax < 0 || ebx <> eax)  && !ecx
```

For this expression to evaluate true, ECX must contain zero and either EAX must be less than zero or EBX must not equal EAX. Contrast this to the result the expression produces without the parentheses.

HLA uses the "!" operator to denote logical negation. However, the "!" operator may only prefix a register or boolean variable; you may not use it as part of a larger expression (e.g., "!eax < 0"). To achieve logical negative of an existing boolean expression you must surround that expression with parentheses and prefix the parentheses with the "!" operator, e.g.,

```
!( eax < 0 )
```

This expression evaluates true if EAX is not less than zero.

The logical not operator is primarily useful for surrounding complex expressions involving the conjunction and disjunction operators. While it is occasionally useful for short expressions like the one above, it's usually easier (and more readable) to simply state the logic directly rather than convolute it with the logical not operator.

3.2.6.2: Converting C/C++ Boolean Expressions to HLA Boolean Expressions

Although, superficially, C/C++ boolean expressions that appear within control structures look very similar to those appearing in HLA high-level structured control statements, there are some fundamental differences that will create some conversion problems. Fortunately, most boolean expressions appearing in C/C++ control structures are relatively simple and almost translate directly into an equivalent HLA expression. Nevertheless, a large percentage of expressions will take a bit of work to properly convert to a form usable by HLA.

Although HLA provides boolean expressions involving relation and logical (and/or/not) operators, don't get the impression that HLA supports generic boolean expressions as C/C++ does. For example, an expression like "(x+y) > 10 || a*b < c" is perfectly legal in C/C++, but HLA doesn't allow an expression like this. You might wonder why HLA allows some operators but not others. There is a good reason why HLA supports only a limited number of operators: HLA supports all the operations that don't require the use of any temporary values (i.e., registers). HLA does not allow any code in an expression that would require the use of a register to hold a temporary value; i.e., HLA will not modify any register values behind the assembly programmer's back. This severely limits what HLA can do since subexpressions like "(x+y)" have to be computed in a temporary register (at least, on the 80x86). The previous section presented most of the operators that are legal in an HLA boolean expression. Unfortunately, of course, C/C++ does allow fairly complex arithmetic/boolean expressions within a structured control statement. This section provides some guidelines you can use to convert complex C/C++ arithmetic/boolean expressions to HLA.

The first thing to note is that HLA only allows operands that are legal in a `cmp` instruction around one of the relational operators. Specifically, HLA only allows the operands in Table 3-9 around a relational operator.

Table 3-9: Legal Operands to a Relational Operator in an HLA Expression

Left Operand	Relational Operator	Right Operand
reg	< <= = == <> != > >=	reg
reg		mem
reg		const
mem		reg
mem		const

If you need to convert a boolean expression like "(x+y) > 10" from C/C++ into HLA, the most common approach is to compute the sub-expression "(x+y)" and leave the result in a register, then you can compare that register against the value 10, e.g.,

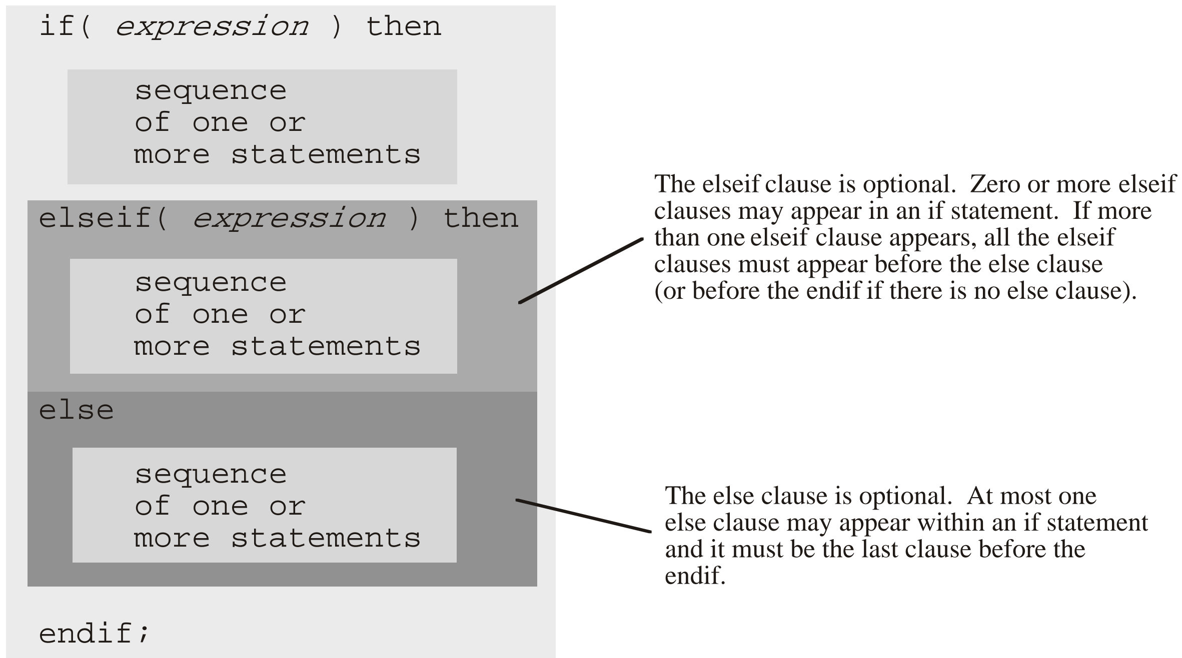
```
mov( x, eax );
add( y, eax );
if( eax > 10 ) then
    .
    .
    .
endif;
```

Unfortunately, the syntax of various high level control structures in HLA don't allow you to place the statements that compute the result before the control structure; we'll take a look at these problems in the sections that follow.

3.2.6.3: The IF Statement

The HLA IF statement uses the syntax shown in Table 3-2.

Figure 3-2: HLA IF Statement Syntax



The expressions appearing in an `if` statement must take one of the forms from the previous sections. If the boolean expression is true, the code after the `then` executes, otherwise control transfers to the next `elseif` or `else` clause in the statement.

Since the `elseif` and `else` clauses are optional, an `if` statement could take the form of a single `if..then` clause, followed by a sequence of statements, and a closing `endif` clause. The following is such a statement:

```
if( eax = 0 ) then

   stdout.put( "error: NULL value", nl );

endif;
```

If, during program execution, the expression evaluates true, then the code between the `then` and the `endif` executes. If the expression evaluates false, then the program skips over the code between the `then` and the `endif`.

Another common form of the `if` statement has a single `else` clause. The following is an example of an `if` statement with an optional `else` clause:

```
if( eax = 0 ) then

   stdout.put( "error: NULL pointer encountered", nl );

else

   stdout.put( "Pointer is valid", nl );

endif;
```

If the expression evaluates true, the code between the `then` and the `else` executes; otherwise the code between the `else` and the `endif` clauses executes.

You can create sophisticated decision-making logic by incorporating the `elseif` clause into an `if` statement. For example, if the CH register contains a character value, you can select from a menu of items using code like the following:

```
if( ch = 'a' ) then

   stdout.put( "You selected the 'a' menu item", nl );

elseif( ch = 'b' ) then

   stdout.put( "You selected the 'b' menu item", nl );

elseif( ch = 'c' ) then

   stdout.put( "You selected the 'c' menu item", nl );

else

   stdout.put( "Error: illegal menu item selection", nl );

endif;
```

Although this simple example doesn't demonstrate it, HLA does not require an `else` clause at the end of a sequence of `elseif` clauses. However, when making multi-way decisions, it's always a good idea to provide an `else` clause just in case an error arises. Even if you think it's impossible for the `else` clause to execute, just keep in mind that future modifications to the code could void this assertion, so it's a good idea to have error reporting statements in your code.

The C/C++ `if` statement is similar, but certainly not identical to, the HLA `if` statement. First of all, the C/C++ if statement is based on an older language design that allows only a single statement after an `if` or `else`. That is, C/C++ supports the following syntaxes for the `if/else` statement:

```
   if( boolean_expression)
      << single statement >>;

   if( boolean_expression )
      << single statement >>;
   else
      << single statement >>;
```

If you need to attach more than a single statement to a C/C++ `if` or `else`, you have to use a compound statement. A compound statement consists of a sequence of zero or more statements surrounded by braces. This means that there are six possible forms of the `if` statement you will find in a typical C/C++ program, as the following syntactical examples demonstrate:

```
1)
   if( boolean_expression)
      << single statement >>;

2)
   if( boolean_expression)
```

```
   {
      << zero or more statements >>
   }

3)
   if( boolean_expression )
      << single statement >>;
   else
      << single statement >>;

4)
   if( boolean_expression )
   {
      << zero or more statements >>
   }
   else
      << single statement >>;

5)
   if( boolean_expression )
      << single statement >>;
   else
   {
      << zero or more statements >>
   }

6)
   if( boolean_expression )
   {
      << zero or more statements >>
   }
   else
   {
      << zero or more statements >>
   }
```

To convert either of the first two forms to HLA is relatively easy. Simply convert the boolean expression to HLA form (including placing any necessary arithmetic computations before the `if` statement), convert the statement or statements attached to the `if` to their HLA equivalents, and then place an `endif` after the last statement attached to the `if`. Here are a couple of examples that demonstrate this conversion for the first two cases:

```
// if( a >= 0 )
//     ++a;

   if( a > 0 ) then

      inc( a );

   endif;

// if( (x*4) >= y && z < -5 )
// {
//    x = x - y;
//    ++z;
//  }
```

```
    mov( x, eax );
    shl( 2, eax );  // x*4
    if( eax >= y && z < -5 ) then

        mov( y, eax );
        sub( eax, x );
        inc( eax );

    endif;
```

Converting one of the other `if/else` forms from C/C++ to HLA is done in a similar fashion except, of course, you also have to include the `else` section in the HLA translation. Here's an example that demonstrates this:

```
// if( a < 256 )
// {
//     ++a;
//     --b;
// }
// else
// {
//     --a;
//     ++b;
// }

    if( a < 256 ) then

        inc( a );
        dec( b );

    else

        dec( a );
        inc( b );

    endif;
```

The C/C++ language does not directly support an `elseif` clause as HLA does, however, C/C++ programs often contain "else if" chains that you may convert to an HLA `elseif` clause. The following example demonstrates this conversion:

```
// if( x >= (y | z))
//     ++x;
//  else if( x >= 10 )
//     --x;
//  else
//  {
//      ++y;
//      --z;
//  }

    mov( y, eax );
    or( z, eax );
    if( x >= eax ) then
```

```
        inc( x );

    elseif( x >= 10 ) then

        dec( x );

    else

        inc( y );
        dec( z );

    endif;
```

Sometimes a C/C++ else-if chain can create some conversion problems. For example, suppose that the boolean expression in the "else if" of this example was "x >= (y & z)" rather than an expression that is trivially convertible to HLA. Unfortunately, you cannot place the computation of the temporary results immediately before the `elseif` in the HLA code (since that section of code executes when `if` clause evaluates true). You could place the computation before the `if` and leave the value in an untouched register, but this scheme has a couple of disadvantages - first, you always compute the result even when it's not necessary (e.g., when the `if` expression evaluates true), second, it consumes a register which is not good considering how few registers there are on the 80x86. A better solution is to use an HLA nested `if` rather than an `elseif`, e.g.,

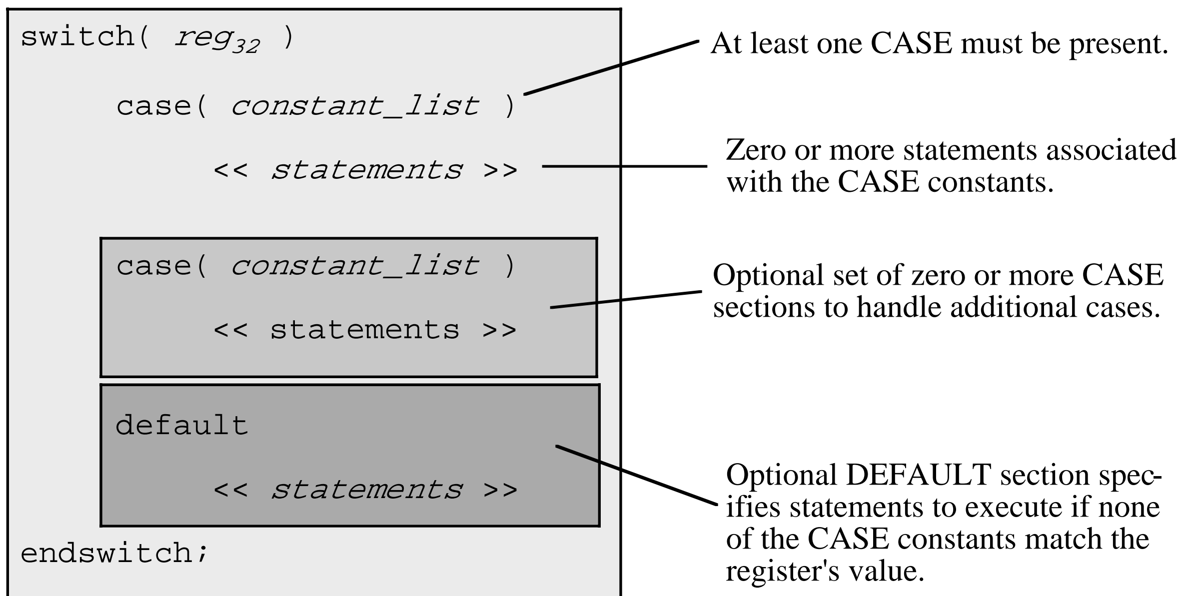
```
// if( x >= (y | z))
//    ++x;
//  else if( x >= (y & z) )
//    --x;
//  else
//  {
//     ++y;
//     --z;
//  }

    mov( y, eax );
    or( z, eax );
    if( x >= eax ) then

        inc( x );

    else

        mov( y, eax );
        and( z, eax );
        if( x >= eax ) then

            dec( x );

        else

            inc( y );
            dec( z );

        endif;

    endif;
```

3.2.6.4: The SWITCH/CASE Statement

The HLA programming language doesn't directly provide a multi-way decision statement (commonly known as a *switch* or *case* statement). However, the HLA Standard Library provides a `switch / case / default / endcase` macro that provides this high level control statement in HLA. If you include the *hll.hhf* header file (which *stdlib.hhf* automatically includes for you), then you can use the `switch` statement exactly as though it were a part of the HLA language.

The HLA Standard Library `switch` statement has the following syntax:

Figure 3-3: Syntax for the Switch..case..default..endswitch Statement



Like most HLA high level language statements, there are several restrictions on the `switch` statement. First of all, the `switch` clause does not allow a general expression as the selection value. The `switch` clause will only allow a value in a 32-bit general purpose register. In general you should only use EAX, EBX, ECX, EDX, ESI, and EDI because EBP and ESP are reserved for special purposes.

The second restriction is that the HLA `switch` statement supports a maximum of 256 different case values. Few `switch` statements use anywhere near this number, so this shouldn't prove to be a problem. Note that each `case` in Figure 3-3 allows a constant list. This could be a single unsigned integer value or a comma separated list of values, e.g.,

```
      case( 10 )
-or-
      case( 5, 6, 8 )
```

Each value in the list of constants counts as one case constant towards the maximum of 256 possible constants. So the second `case` clause above contributes three constants towards the total maximum of 256 constants.

Another restriction on the HLA `switch` statement is that the difference between the largest and smallest values in the case list must be 1,024. Therefore, you cannot have `cases` (in the same `switch` statement) with values like 1, 10, 100, 1,000, and 10,000 since the difference between the smallest and largest values, 9999, exceeds 1,024.

The `default` section, if it appears in a `switch` statement, must be the last section in the `switch` statement. If no `default` section is present and the value in the 32-bit register does not match one of the `case` constants, then control transfers to the first statement following the `endswitch` clause.

Here is a typical example of a `switch..endswitch` statement:

```
switch( eax )

   case( 1 )

      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>

   case( 2, 3 )

      stdout.put( "Selections (2) and (3):" nl );
      << code for cases 2 & 3 >>

   case( 5,6,7,8,9 )

      stdout.put( "Selections (5)..(9)" nl );
      << code for cases 5..9 >

   default

      stdout.put( "Selection outside range 1..9" nl );
      << default case code >>

endswitch;
```

The `switch` statement in a program lets your code choose one of several different code paths depending upon the value of the case selection variable. Among other things, the `switch` statement is ideal for processing user input that selects a menu item and executes different code depending on the user's selection.

The HLA `switch` statement actually supports the semantics of the Pascal `case` statement (as well as multi-way selection statements found in various other languages). The semantics of a C/C++ `switch` statement are slightly different. As it turns out, HLA's `switch` macro provides an option for selecting either Pascal or C/C++ semantics. The *hll.hhf* header file defines a special compile-time boolean variable, `hll.cswitch`, that controls which form of the `switch` statement HLA will use. If this compile-time variable contains false (the default), then HLA uses Pascal semantics for the `switch` statement. If this compile-time variable contains true, then HLA uses C/C++ semantics. You may set this compile-time variable to true or false with either of the following two statements:

```
?hll.cswitch := true;  // Enable C/C++ semantics for the switch statement.
?hll.cswitch := false; // Enable Pascal semantics for the switch statement.
```

The difference between C/C++ and Pascal semantics has to do with what will happen when the statements within some `case` block reach the end of that block (by hitting another `case` or the `default` clause). When using Pascal semantics, HLA automatically transfers control to the first statement following the `endswitch` clause upon hitting a new case. In the previous example, if EAX had contained one, then the switch statement would execute the code sequence:

```
      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>
```

Immediately after the execution of this code, control transfers to the first statement following the `endswitch` (since the next statement following this fragment is the "case(2,3)" clause).

If you select C/C++ semantics by setting the `hll.cswitch` compile-time variable to true, then control does not automatically transfer to the bottom of the `switch` statement; instead, control falls into the first statement of the next `case` clause. In order to transfer control to the first statement following the `endswitch` at the end of a `case` section, you must explicitly place a `break` statement in the code, e.g.,

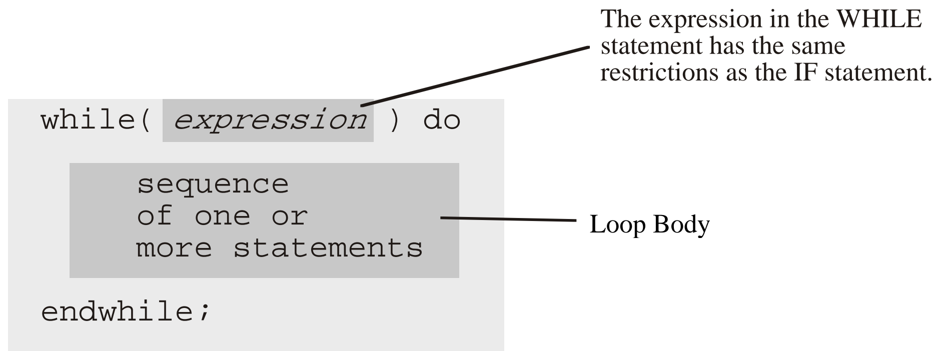
```
?hll.cswitch := true;  // Enable C/C++ semantics for the switch statement.
switch( eax )

   case( 1 )

      stdout.put( "Selection #1:" nl );
      << Code for case #1 >>
      break;

   case( 2, 3 )

      stdout.put( "Selections (2) and (3):" nl );
      << code for cases 2 & 3 >>
      break;

   case( 5,6,7,8,9 )

      stdout.put( "Selections (5)..(9)" nl );
      << code for cases 5..9 >
      break;

   default

      stdout.put( "Selection outside range 1..9" nl );
      << default case code >>

endswitch;
```

Note that you can alternately switch between C/C++ and Pascal semantics throughout your code by setting the `hll.cswitch` compile-time variable to true or false at various points throughout your code. However, as this makes the code harder to read, it's generally not a good idea to do this on a frequent basis. You should pick one form or the other and attempt to stick with it as much as possible. Pascal semantics are actually a little bit nicer (and safer) plus you get to continue using the `break` statement to break out of a loop containing a `switch` statement. On the other hand, some C/C++ `switch` statements need the ability to flow from one case to another, so if you're translating such a statement from C/C++ to HLA, the C/C++ `switch` statement format is easier to deal with. Of course, the purpose of this chapter is not to teach you how to convert a C/C++ Windows program to HLA, but rather to help you read and understand C/C++ documentation. In real life, if you have to convert a C/C++ `switch` statement to assembly language you're probably better off explicitly creating a jump table and using an indirect jump implementation of the `switch` statement (see *The Art of Assembly Language* for details).

3.2.6.5: The WHILE Loop

The HLA `while` statement uses the basic syntax shown in Figure 3-4.

Figure 3-4: HLA WHILE Statement Syntax



The `while` statement evaluates the boolean expression. If it is false, control immediately transfers to the first statement following the `endwhile` clause. If the value of the expression is true, then the CPU executes the body of the loop. After the loop body executes, control transfers back to the top of the loop where the `while` statement retests the loop control expression. This process repeats until the expression evaluates false.

The C/C++ statement uses a similar syntax and identical semantics. There are two principle differences between the HLA while loop and the C/C++ variant: (1) HLA uses "while(expr) do ... endwhile;" whereas C/C++ uses "while(expr) *single_statement*;", as with the C/C++ `if` statement, if you want to attach more than a single statement to the `while` you have to create a compound statement (using braces); (2) HLA's boolean expressions are limited compared to C/C++ boolean expressions (see the discussion in the section on converting boolean expressions from C/C++ to HLA and the section on the `if` statement for details).

One problem with converting C/C++ statements to HLA is the conversion of complex boolean expressions. Unlike an `if` statement, we cannot simply compute portions of a boolean expression prior to the actual test in the `while` statement, i.e., the following conversion doesn't work:

```
// while( (x+y) < z )
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }

   mov( x, eax );   // Note: this won't work!
   add( y, eax );
   while( eax < z ) do
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );
   endwhile;
```

The problem with this conversion, of course, is that the computation of "x+y" needed in the boolean expression only occurs once, when the loop first executes, not on each iteration as is the case with the original C/C++ code. The easiest way to solve this problem is to use the HLA `forever..endfor` loop and a `breakif` statement:

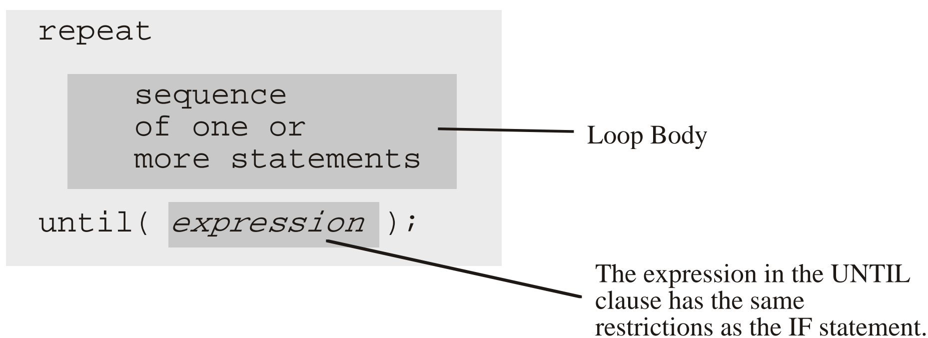
```
// while( (x+y) < z )
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }

   forever
      mov( x, eax );
      add( y, eax );
    breakif( eax < z );
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );
   endfor;
```

3.2.6.6: The DO..WHILE Loop

The C/C++ `do..while` loop is similar to the `while` loop except it tests for loop termination at the bottom of the loop rather than at the top of the loop (i.e., it executes the statements in the loop body at least once, regardless of the value of the boolean control expression the first time the program computes it). Like the `while` loop, the `do..while` loop repeats the execution of the loop body as long as the boolean expression evaluates true. HLA does not provide an exact equivalent of the `do..while` loop, but it does provide a `repeat..until` loop. The difference between these two loops is that a `do..while` loop repeats as long as (while) the expression evaluates true, the `repeat..until` loop repeats until the expression evaluates true (that is, it repeats the loop as long as the expression evaluates false).

The HLA `repeat..until` statement uses the syntax shown in Figure 3-5.

Figure 3-5: HLA repeat..until Statement Syntax



To convert a C/C++ `do..while` statement to an HLA `repeat..until` statement, you must adjust for the semantics of the loop termination condition. Most of the time, the conversion is immediately obvious; in those few cases where you've got a complex boolean expression whose negation is not instantly obvious, you can always use the HLA "!(...)" (not) operator to negate the result of the boolean expression, e.g.,

```
// do
// {
//    <<some code fragment>>
```

```
// }while( (x < 10) && (y > 5 ));

   repeat

      <<some code fragments converted to HLA>>

   until( !((x<10) && (y>5)) );
```

One advantage of the `do..while` loop over C/C++'s while loop is that statements appearing immediately before the `while` clause (and after the `do` clause) will generally execute on each iteration of the loop. Therefore, if you've got a complex boolean expression that tests for loop termination, you may place the computation of portions of that expression immediately before the HLA `until` clause, e.g.,

```
// do
// {
//    printf( "x=%d\n", x );
//    ++x;
//    y = y + x;
// }while( (x+y) < z )

   repeat
      stdout.put( "x=", x, nl );
      inc( x );
      mov( x, eax );
      add( eax, y );

      mov( x, eax );
      add( y, eax );
   until( !(eax < z) );
```

The only time this will not work is if there is a `continue` (or an HLA `continueif`) statement in the loop. The `continue` statement directly transfers control to the loop termination test in the `until` clause. Since `continue` statements in C/C++ appear so infrequently, the best solution is to replace the `continue` with a `jmp` instruction that transfers control the first statement that begins the execution of the termination test expression.

3.2.6.7: The C/C++ FOR Loop

The C/C++ statement `for` statement is a specialized form of the `while` loop. It should come as no surprise, then, that the conversion to HLA is very similar to that for the `while` loop conversion. The syntax for the C/C++ for loop is the following:

```
   for( expression1; expression2; expression3 )
      statement;
```

This C/C++ statement is complete equivalent to the following C/C++ sequence:

```
   expression1;
   while( expression2 )
   {
      statement;
      expression3;
   }
```

Although you can convert a C/C++ `for` statement to an HLA `while` loop, HLA provides a `for` statement that is syntactically similar to the C/C++ `for` statement. Therefore, it's generally easiest to convert such C/C++ statements into HLA `for` statements. The HLA `for` loop takes the following general form:

```
for( Initial_Stmt; Termination_Expression; Post_Body_Statement ) do

   << Loop Body >>

endfor;
```

The following gives a complete example:

```
for( mov( 0, i ); i < 10; add(1, i )) do

   stdout.put( "i=", i, nl );

endfor;


// The above, rewritten as a while loop, becomes:

mov( 0, i );
while( i < 10 ) do

   stdout.put( "i=", i, nl );

   add( 1, i );

endwhile;
```

There are a couple of important differences between the HLA for loop and the C/C++ for loop. First of all, of course, the boolean loop control expression that HLA supports has the usual restrictions. If you've got a complex boolean expression in a C/C++ loop, your best bet is to convert the `for` loop into a C/C++ `while` loop and then convert that `while` loop into an HLA `forever..endfor` loop as the section on the `while` loop describes.

The other difference between the C/C++ and HLA for loops is the fact that C/C++ supports arbitrary arithmetic expressions for the first and third operands whereas HLA supports a single HLA statement. 90% of the C/C++ for loops you'll encounter will simply assign a constant to a variable in the first expression and increment (or decrement) that variable in the third expression. Such for loops are very easy to convert to HLA as the following example demonstrates:

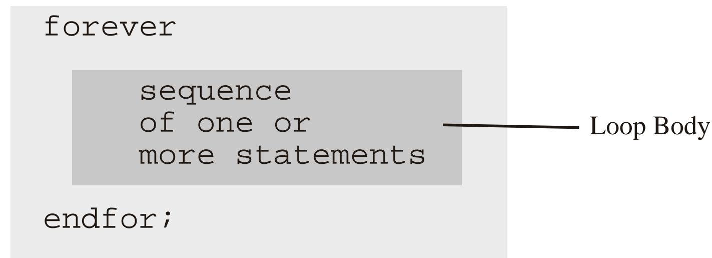
```
// for( i=0; i<10; ++i )
// {
//    printf( "i=%d\n", i );
// }

   for( mov( 0, i ); i<10; inc(i) ) do
      stdout.put( "i=", i , nl );
   endfor;
```

C/C++ allows a bizarre form of the for statement to create an infinite loop. The C/C++ convention for an infinite loop uses the following syntax:

```
for(;;)
   statement;
```

HLA does not allow this same syntax for it's `for` loop. Instead, HLA provides an explicit statement for creating infinite loops: the `forever..endfor` statement. Figure 3-6 shows the syntax for the `forever` statement.

Figure 3-6: HLA forever..endfor Loop Syntax



Although `for(;;)` and `forever..endfor`, by themselves, create infinite loops, the truth is that most of the time a program that employs these statements also uses a `break`, `breakif`, or `return` statement in order to exit the loop somewhere in the middle of the loop. The next section discusses the `break` and `breakif` statements. A little bit later we'll look at C/C++'s `return` statement.

3.2.6.8: Break and Continue

C/C++ supports two specialized forms of the goto statement that immediately exits, or repeats the execution of, the loop containing these statements. The `break` statement exits the loop that contains the statement; the `continue` statement transfers control to the loop control expression (or simply to the top of the loop in the case of the infinite loop). As you've seen earlier, the `break` statement also ends a case sequence in the C/C++ switch statement.

HLA also provides the `break` and `continue` statements that have the same semantics within a loop. Therefore, you can trivially translate these two statements from C/C++ to HLA. HLA also provides `breakif` and `continueif` statements that will test a boolean expression and execute the `break` or `continue` only if the expression evaluates true. Although C/C++ doesn't provide a direct counterpart to these two HLA statements, you'll often see C/C++ statements like the following that you can immediately translate to an HLA `breakif` or `continueif` statement:

```
if( C_expression ) break;
if( C_expression ) continue;
```

3.2.6.9: The GOTO Statement

The C/C++ `goto` statement translates directly into an 80x86 `jmp` instruction. A C/C++ `goto` statement typically takes the following form:

```
   goto someLabel;
      .
      .
      .
```

```
   someLabel:  // The label may appear before the goto statement!
```

This usually translates to the following HLA code:

```
      jmp someLabel;
         .
         .
         .
   someLabel:
```

The only difference, besides substituting `jmp` for `goto`, is the fact that `goto` labels have their own *namespace* in C/C++. In HLA, however, statement labels share the same namespace as other local variables. Therefore, it is possible (though rare) that you'll get a "duplicate name" error if you use the same name in your HLA code that appears in the C/C++ program. If this happens, make a minor change to the statement label when translating the code to HLA.

3.3: Function Calls, Parameters, and the Win32 Interface

This section begins the second major portion of this chapter and, in fact, represents the most important material in this chapter from the perspective of an assembly language programmer: how C/C++ function calls translate into assembly language and how an HLA programmer would call a function written in C/C++. This information represents the major point of this chapter since all Win32 API calls are calls to C code. Furthermore, most Windows documentation that explains the Win32 API explains it in terms of C/C++ function calls, in order to understand how one makes calls to the Win32 API from assembly language, you must understand how C/C++ implements these function calls. Explaining that is the purpose of this section.

3.3.1: C Versus C++ Functions

There are some very important differences, both semantic and syntactical, between functions written in C and functions written in C++. The Win32 API uses the C calling and naming convention. Therefore, all the Win32 API documentation also uses the C calling and naming convention. Therefore, that's what we will concentrate on in this chapter.

C++ functions do offer several advantages over C functions. Function overloading is a good example of such a feature. However, function overloading (using the same function name for different functions and differentiating the actual functions by their parameter lists) requires the use of a facility known as *name mangling* in order to generate unique names that the linker can use. Unfortunately, there is no standard for name mangling among C++ compilers, so every one of them does it differently. Therefore, you rarely see assembly code (or other languages for that matter) interfacing with C++ functions.

In order to allow mixed-language programming with C++ (that is, the use of multiple programming languages on the same project), the C++ language defines a special "C" function syntax that allows you to tell the compiler to generate C linkage rather than C++. This is done with the C++ `extern` attribute:

```
extern "C"
{
   extern char* RetHW( void );
};
```

Please consult a C++ reference manual or your compiler's documentation for more details. Since the Win32 API doesn't use the C++ calling convention, we won't consider it any farther here.

Another useful C++ feature that this chapter will discuss, when appropriate, is pass by reference parameters (since HLA also supports this feature). However, the Win32 API doesn't use any C++ features, so when this chapter gets around to discussing pass by reference parameters, it will mainly be for your own edification.

3.3.2: The Intel 80x86 ABI (Application Binary Interface)

Several years ago, Intel designed what is known as the 80x86 Application Binary Interface, or ABI. The purpose of the ABI was to provide a standard that compiler designers to use to ensure interoperability between modules written in different languages. The ABI specifies what registers a function call should preserve (and which registers a function can modify without preserving), where functions return their results, alignment of data objects in structures, and several other conventions. Since Microsoft's C/C++ compilers (the ones used to compile Windows) adhere to these conventions, you'll want to be familiar with this ABI since the Win32 API uses it.

3.3.2.1: Register Preservation and Scratch Registers in Win32 Calls

The Intel 80x86 ABI specifies that functions must preserve the values of certain registers across a function call. If the function needs to modify the value of any of those registers, it must save the register's value and restore it before returning to the caller. The registers that must be preserved across calls are EBX, ESI, EDI, and EBP. This means two things to an assembly language programmer calling an Win32 function: first of all, Windows preserves the values of these registers across a Win32 API call, so you can place values in these registers, make an OS call, and be confident that they contain the same value upon return. The second implication has to do with *callback functions*. A callback function is a function you write whose address you pass to Windows. At various times Windows may choose to call that function directly. Such callback functions must obey the register preservation rules of the Intel 80x86 ABI. In particular, such callback functions must preserve the value of the EBX, ESI, EDI, and EBP registers.

On the flip side, the Intel 80x86 ABI specifies that a function may freely modify the values of the EAX, ECX and EDX registers without preserving them. This means that you can generally count on Win32 API functions disturbing the values in these registers; as you'll see in a moment, most Win32 API functions return a function result in the EAX register, so it's almost always wiped out. However, most Win32 API functions wipe out the values in ECX and EDX as well. If you need the values of any of these registers preserved across a Win32 API call, you must save their values yourself.

3.3.2.2: The Stack Pointer (ESP)

The ESP register is a special case. Function calls to the Win32 API generally do not preserve ESP because they remove any parameters from the stack that you push onto the stack prior to calling the API function. However, you can generally assume that ESP is pointing at an appropriate top of stack upon return from the function. In particular, any values you push onto the stack before pushing any API parameters (e.g., register values you want to preserve) will still be sitting on the stack when the function returns. Functions that follow the Intel 80x86 ABI do not arbitrarily mess with the value in the ESP register.

All Win32 API functions assume that the stack is aligned on a double-word boundary (that is, ESP contains a value that is an even multiple of four). If you call a Win32 API function and ESP is not aligned at a double-word address, the Win32 API function will fail. By default, HLA automatically emits code at the beginning of each procedure to ensure that ESP contains a value that is an even multiple of four bytes. However, many program-

mers choose to disable this code (to make their programs slightly more efficient). If you do this, always make sure that ESP contains a value whose L.O. two bits contain zeros (that is, an even multiple of four) before calling any Win32 API functions.

3.3.2.3: The Direction Flag

All Win32 functions assume that the direction flag is clear when you call them. The Win32 programming convention is to set the direction flag when you need it set and then immediately clear it when you are done using it in that state. Therefore, in all code where you have not explicitly set the direction flag yourself, you can assume that the direction flag is clear. You code should adhere to this policy as well (and always make sure the direction flag is clear when you make a Win32 API call). You can also assume that the direction flag is clear whenever Windows calls one of your callback routines.

3.3.2.4: Function Return Results

Table 3-10 lists the places that functions should return their result (depending on the size of the function return result). The Win32 API generally adheres to this convention. If a function returns more than eight bytes, Win32 API functions generally require that you pass a pointer (i.e., the address of) some block of memory where the function will store the final result.

Table 3-10: 80x86 ABI Function Return Result Locations

Size of Function Result in Bytes	Returned Here
1	al
2	ax
4	eax
8	edx:eax
other	See Compiler Documentation

3.3.2.5: Data Alignment and Padding

The Intel 80x86 ABI generally expects objects to appear at addresses in memory that are an even multiple of their natural size up to four bytes in length (i.e., byte objects may appear at any address, word objects always appear at even addresses, and larger objects are aligned on double-word addresses). This is true for static objects, automatic variables (local variables within a function), and fields within structures. Although this convention is easily circumvented by setting compiler options, the Win32 API pretty much adheres to this convention throughout.

If an object would normally start at an address that is not an even multiple of it's *natural size*[14] (up to four bytes), then the Microsoft C compiler will align that object at the next highest address that is an even multiple of

14.The natural size of an object is the size of the object if it's a scalar, the size of an element if it's an array, or the size of the largest field (up to four bytes) if it's a structure.

the object's native size. For data, the compiler usually fills (*pads*) the empty bytes with zeros, though you should never count on the values (or even presence) of padding bytes.

Parameters passed on the stack to a function are a special case. Parameters are always an even multiple of four bytes (this is done to ensure that the stack remains double-word aligned in memory). If you pass a parameter that is smaller than four bytes to some function, the Microsoft C compiler will pad it out to exactly four bytes. Likewise, if you pass a larger object that is not an even multiple of four bytes long, the compiler will pad the object with extra bytes so its length is an even multiple of four bytes long.

For information on padding within structures, please see the section on the `struct` data type earlier in this chapter.

3.3.3: The C, Pascal, and Stdcall Calling Conventions

There are many different function calling conventions in use today. Of these different calling conventions, three are of interest to us, the so-called *C, Pascal,* and *Stdcall* calling conventions. The C and Stdcall calling conventions are of interest because they're the ones that Win32 API calls use. The Pascal calling convention is of interest because that's the default calling convention that HLA uses.

The Pascal calling convention is probably the most efficient of the three and the easiest to understand. In the Pascal calling sequence, a compiler (or human programmer writing assembly code) pushes parameters on the stack as they are encountered in the parameter list when processing the parameters in a left-to-right fashion. Another nice feature of the Pascal calling sequence is that the procedure/function is responsible for removing the parameters from the stack upon return from the procedure; so the caller doesn't have to explicitly do this upon return. As an example, consider the following HLA procedure prototype and invocation:

```
// Note: the "@pascal" attribute is optional, since HLA generally uses
// the pascal calling convention by default.

procedure proc( i:int32; j:int32; k:int32 ); @pascal; @external;
    .
    .
    .
proc( 5, a, eax );
```

Whenever HLA encounters the high-level call to `proc` appearing in this example, it emits the following "pure" assembly code:

```
pushd( 5 );
push( a );     // Assumption: a is a 32-bit variable that is type compatible with int32
push( eax );
call proc;
```

Note that you have the choice of using HLA's high-level calling syntax or manually pushing the parameters and calling the procedure directly. HLA allows either form; the high-level calling syntax is generally easier to read and understand and it's less likely you'll make a mistake (that invariably hangs the program) when using the high level syntax. Some assembly programmers, however, prefer the low-level syntax since it doesn't hide what is going on.

The C calling convention does two things differently than the Pascal calling convention. First of all, C functions push their parameters in the opposite order than Pascal (e.g., from right to left). The second difference is that C functions do not automatically pop their parameters from the stack upon return. The advantage of the C

calling convention is that it allows a variable number of parameters (e.g., for C's `printf` function). However, the price for this extra convenience is reduced efficiency (since the caller has to execute extra instructions to remove the parameters from the stack).

Although Windows is mostly written in C, most of the Win32 API functions do not use the C calling convention. In fact, only the API functions that support a variable number of parameters (e.g., `wsprintf`) use the C calling convention. If you need to make a call to one of these API functions (or you want to call some other function that uses the C calling convention), then you've got to ensure that you push the parameters on the stack in the reverse order of their declaration and you've got to remove them from the stack when the function returns. E.g.,

```
// int cProc( int i, int j, int k );
//     .
//     .
//     .
// cProc( a, 5, 2 );

   pushd( 2 );      // push last parameter first!
   pushd( 5 );
   push( a );       //assumes a is a dword variable.
   call cProc;
   add( 12, esp ); // Remove three dword parameters from stack upon return.
```

HLA supports the C calling convention using the `@cdecl` procedure attribute, e.g.,

```
procedure cProc( i:int32; j:int32; k:int32 ); @cdecl; @external;
```

HLA's high-level procedure call syntax will automatically push the actual parameters on the stack in the appropriate order (i.e., in reverse). However, you are still responsible for removing the parameter data from the stack upon returning from the procedure call:

```
   cProc( a, 5, 2 );  // Pushes 2, then 5, then a onto the stack
   add( 12, esp );    // Remove parameter data from the stack.
```

Don't forget that all procedure parameters are an even multiple of four bytes long. Therefore, when removing parameter data from the stack the value you add to ESP must reflect the fact that the Intel ABI rounds parameter sizes up to the next multiple of four bytes.

The last parameter passing mechanism of immediate interest to us is the *Stdcall* (standard call) parameter passing mechanism. The Stdcall scheme is a combination of the C and Pascal calling sequences. Like the C calling sequence, the Stdcall scheme pushes the parameters on the stack in the opposite order of their declaration. Like the Pascal calling sequence, the procedure automatically removes the parameters from the stack before returning. Therefore, the caller does not have to remove the parameter data from the stack (thus improving efficiency by a small amount). Most of the Win32 API functions use the Stdcall calling convention. In HLA, you can use the `@stdcall` procedure attribute to specify the Stdcall calling convention, e.g.,

```
procedure stdProc( i:int32; j:int32; k:int32 ); @stdcall; @exter-
nal;
```

HLA's high level procedure call syntax will automatically push the parameters on the stack in the proper (i.e., reverse) order:

```
stdProc( a, 5, 2 );
```

Of course, you can also manually call a Stdcall procedure yourself. Be sure to push the parameters in the reverse order!

```
pushd( 2 );
pushd( 5 );
push( a );
call stdProc;
```

Notice that this code does not remove any parameters from the stack. That is the function's job.

Some older HLA code (written before the @stdcall facility was added to the language) simulates the Stdcall calling convention by reversing the parameters in the procedure declaration (indeed, some of the HLA standard library code takes this one step farther and uses macros to swap the parameters prior to making calls to these procedures). Such techniques are obsolete and you shouldn't employ them; however, since there is some code lying around that does this, you should be aware of why it does this.

3.3.4: Win32 Parameter Types

Almost all Win32 parameters are exactly four bytes long. This is true even if the formal parameter is one byte (e.g., a `char` object), two bytes (a `short int`), or some other type that is smaller than four bytes. This is done to satisfy the Intel 80x86 ABI and to keep the stack pointer aligned on a four-byte boundary. Since all parameters are exactly four bytes long, a good question to ask is "how do you pass smaller objects, or objects whose size is not an even multiple of four bytes, to a Win32 API function?" This section will briefly discuss this issue.

Whenever you pass a byte parameter to some function, you must pad that byte out to four bytes by pushing an extra three bytes onto the stack. Note that the procedure or function you call cannot assume that those bytes contain valid data (e.g., the procedure/function cannot assume those three bytes all contain zeros). It is perfectly reasonable to push garbage bytes for the upper three bytes of the parameter. HLA will automatically generate code that pushes a byte-sized actual parameter onto the stack as a four-byte object. Most of the time, this code is relatively efficient. Sometimes, however, HLA may generate slightly less efficient code in the interest of safety. For example, if you pass the BH register as a byte-sized parameter (a reasonable thing to do), there is no way that HLA can push BH onto the stack as a double word with a single instruction. Therefore, HLA will emit code like the following:

```
sub( 4, esp );    // make room for the parameter
mov( bh, [esp] ); // Save BH in the L.O. byte of the object on top of stack.
```

Notice that the upper three bytes of this double-word on the stack will contain garbage. This example, in particular, demonstrates why you can't assume the upper three bytes of the double word pushed on the stack contain zeros. In this particular case, they contain whatever happened to be in those three bytes prior to the execution of these two instructions.

Passing the AL, BL, CL, or DL register is fairly efficient on the 80x86. The CPU provides a single byte instruction that will push each of these eight-bite values onto the screen (by passing the entire 32-bit register that contains these registers:

```
push( eax );  // Passes al.
push( ebx );  // Passes bl.
push( ecx );  // Passes cl,
push( edx );  // Passes dl
```

Passing byte-sized memory objects is a bit more problematic. Your average assembly language programmer would probably write code like the following:

```
push( (type dword byteVar) );  // Pushes byteVar plus three following bytes
call funcWithByteParam;
```

HLA, because it observes safety at the expense of efficiency, will not generate this code. The problem is that there is a *tiny* chance that this will cause the system to fail. This situation could occur if `byteVar` is located in memory within the last three bytes of a page of memory (4096 bytes) and the next page in memory is not readable. That would raise a memory access violation. Quite frankly, the likelihood of this ever occurring is so remote that your average programmer would ignore the possibility of it ever happening. However, compilers cannot be so cavalier. Even if the chance that this problem will occur is remote, a compiler must generate safe code (that will never break). Therefore, HLA actually generates code like the following:

```
push( eax );             // Make room for parameter on stack.
push( eax );             // Preserve EAX's value
mov( byteVar, al );
mov( al, [esp+4] );      // Save byteVar's value into parameter location
pop( eax );              // Restore EAX's value.
call funcWithByteParam;  // Call the function.
```

As you can see, the code that HLA generates to pass a byte-sized object as a parameter can be pretty ugly (note that this is only true when passing variables).

Part of the problem with generating code for less-than-dword-sized parameters is that HLA promises to never mess with register values when passing parameters[15]. HLA provides a special procedure attribute, `@use`, that lets you tell HLA that it is okay to modify the value of a 32-bit register if doing so will allow HLA to generate better code. For example, suppose `funcWithByteParam` had the following external declaration:

```
procedure funcWithByteParam( b:byte ); @use EAX; @external;
```

With this declaration, HLA can generate better code when calling the function since it can assume that it's okay to wipe out the value in EAX:

```
// funcWithByteParam( byteVar );

   mov( byteVar, eax );
   push( eax );
   call funcWithByteParam;
```

Because the Intel ABI specifies that EAX (and ECX/EDX) are scratch registers and any function following the Intel ABI is free to modify their values, and because the Win32 functions follow the Intel ABI, and because most Win32 API functions return a function return result in EAX (thereby guaranteeing that they wipe out EAX's value on any Win32 API call), you might wonder why you (or the HLA Standard Library) shouldn't just always specify "@use EAX;" on every Win32 function declaration. Well, there is a slight problem with doing this. Consider the following function declaration and invocation:

15. Indeed, the only time HLA messes with any register value behind your back is when invoking a class method. However, HLA well-documents that fact that class method and procedure calls may wipe out the values in ESI and EDI.

```
procedure func( b:char; c:char; d:boolean ); @use eax; @external;
      .
      .
      .
func( charVar, al, boolVar );
```

Here's code similar to what HLA would generate for this function call:

```
mov( charVar, al );
push( eax );
push( eax );
mov( boolVar, al );
push( eax );
call func;
```

Do you see the problem? Passing the first parameter (when using the `@pascal` calling convention) wipes out the value this code passes as the second parameter in this function invocation. Had we specified the `@cdecl` or `@stdcall` calling convention, then passing the third parameter would have done the dirty deed. For safety reasons, the HLA Standard Library that declares all the Win32 API functions does not attach the `@use` procedure attribute to each procedure declaration. Therefore, certain calls to Win32 API routines (specifically, those that pass memory variables that are less than four bytes long as parameters) will generate exceedingly mediocre code. If having smaller programs[16] is one of your primary goals for writing Windows applications in assembly language, you may want to code calls containing such parameters manually.

If a parameter object is larger than four bytes, HLA will automatically round the size of that object up to the next multiple of four bytes in the parameter declaration. For example, `real80` objects only require ten bytes to represent, but when you pass one as a parameter, HLA sets aside 12 bytes in the stack frame. When HLA generates the code to pass a `real80` object to a procedure, it generates the same code it would use to pass two double word variables and a word variable; in other words, the code needed to pass the last two bytes could get ugly (for the same reasons we've just covered). However, since there aren't any Win32 API functions that expect a `real80` parameter, this shouldn't be an issue.

Table lists the typical C/C++ data types, their HLA equivalents, and how much space they consume when you pass them as parameters to a Win32 API function.

Table 3-11: Space Consumed by Various C Types When Passed as Parameters

C Type	Corresponding HLA Types	Space Consumed on Stack	Padding
char	char, byte, int8[a]	four bytes	three bytes
short	word, int16	four bytes	two bytes
int	dword, int32	four bytes	none
long	dword, int32	four bytes	none
long long	qword, int64	eight bytes	none

16.In the big picture, this extra code is not going to affect the running time of your code by a significant factor. Win32 API functions are sufficiently slow to begin with that the few extra clock cycles consumed by the "safe" code is insignificant.

C Type	Corresponding HLA Types	Space Consumed on Stack	Padding
unsigned char	char, byte, uns8	four bytes	none
unsigned short	word, uns16	four bytes	none
unsigned	dword, uns32	four bytes	none
unsigned int	dword, uns32	four bytes	none
unsigned long	dword, uns32	four bytes	none
unsigned long long	qword, uns64	eight bytes	none
float	real32	four bytes	none
double	real64	eight bytes	none
long double	real64 (on some compilers)	eight bytes	none
	real80 (on other compilers)	twelve bytes	two bytes

a.Some compilers have an option that lets you specify the use of unsigned char as the default. In this case, the corresponding HLA type is uns8.

3.3.5: Pass by Value Versus Pass by Reference

The C programming language only supports *pass by value* parameters. To simulate pass by reference parameters, C requires that you explicitly take the address of the object you wish to pass and then pass this address through a pass by value parameter that is some pointer type. The following code demonstrates how this is done:

```
/* C code that passes some parameter by reference via pointers */

int someFunc( int *ptr )
{
   *ptr = 0;
}
       .
       .
       .
/* Invocation of this function, assume i is an int */

   someFunc( &i );
```

This function passes the address of `i` as the value of the `ptr` parameter. Within `someFunc`, the function dereferences this pointer and stores a zero at the address passed in through the pointer variable (since, in this example, we've passed in the address of `i`, this code stores a zero into the `i` variable).

HLA, like the C++ language, directly supports both pass by value and pass by reference parameters[17]. So when coding a prototype for some Win32 API function that has a pointer parameter, you've got the choice of specifying a pointer type as a value parameter or the pointer's base type as a reference parameter in the HLA dec-

17.Actually, HLA supports several different parameter passing mechanisms. However, pass by value and pass by reference are the only ones that are of interest when calling Win32 API functions, so we'll discuss only those here. See the HLA reference manual for more details on the more advanced parameter passing mechanisms.

laration. The `someFunc` C function in the current example could be encoded with any of the following to HLA declarations:

```
   // A "typeless" variable declaration (since pointers are always 32-bit values)
   // that passes the parameter by value:

   procedure someFunc( ptr:dword );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;



   // A typed version passing a pointer to an int32 object as a value parameter:

type
   pInt32 :pointer to int32;
      .
      .
      .
   procedure someFunc( ptr:pInt32 );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;



   // A version using pass by reference parameters

   procedure someFunc( var ptr:int32 );
   begin someFunc;

      mov( ptr, eax );
      mov( 0, (type dword [eax]) );

   end someFunc;
```

Note that the function's body is exactly the same in all three cases. The function has to grab the address passed on the stack and store a zero at that memory address (just as the C code does). If you manually call `someFunc` (that is, if you use low-level assembly syntax rather than HLA's high-level procedure calling syntax), then the code you write to call any of these three versions is also identical. It is

```
// someFunc( i );

   lea( eax, i );    // Take the address of i, could use "pushd( &i );" if i is static.
   push( eax );      // This code assumes that it is okay to wipe out EAX's value.
   call someFunc;    // We're also assuming @pascal or @stdcall convention here.
```

The difference between these procedure declarations is only evident when you use HLA's high level procedure calling syntax. When the parameter is a double word or pointer value, the caller must explicitly write the code to calculate the address of the actual parameter and pass this computed address as the parameter's value, as the following example demonstrates:

```
// procedure someFunc( ptr:dword );
// -or-
// procedure someFunc( ptr:pInt32 );
//
// call someFunc, passing the address of "i":

   lea( eax, i );
   someFunc( eax );
```

When calling a procedure that has a pass by reference parameter, all you need do is pass the variable itself. HLA will automatically generate the code that takes the address of the variable:

```
// procedure someFunc( var ptr:int32 );

   someFunc( i );
```

If i is a `static, storage,` or `readonly` variable without any indexing applied to it, then HLA generates the following code for this statement:

```
   push( &i );
   call someFunc;
```

However, if the actual parameter (i in this case) is an indexed static object, or is a local variable, then HLA will have to generate code like the following:

```
   push( eax );
   push( eax );
   lea( eax, i );
   mov( eax, [esp+4] );
   pop( eax );
   call someFunc;
```

This happens because HLA promises not to mess with register values when passing parameters. Of course, you can improve the quality of the code that HLA generates by using the "@use" procedure attribute, remembering the caveats given earlier:

```
// procedure someFunc( var i:int32 ); @use EAX;

   someFunc( i );  // Assume i is a local (automatic) variable

// is equivalent to

   lea( eax, i );
   push( eax );
   call someFunc;
```

HLA's pass by reference parameter passing mechanism requires that you specify a memory address as an actual reference parameter. So what happens if you run into a situation when the address you want to pass is in a register and you've specified a pass by reference parameter? If you try to call the function with code like the following HLA will complain that you've not specified a valid memory address:

```
someFunc( esi );
```

The trick is to give HLA what it wants: a memory address. This is easily achieved by specifying the following function call to `someFunc`:

```
someFunc( [esi] );
```

This generates the following assembly code:

```
push( esi );
call someFunc;
```

HLA usually requires the type of the actual parameter (the parameter you pass in a procedure call) to *exactly* match the type of the formal parameter (the parameter appearing in the declaration of the procedure). You cannot pass the address of a `char` variable as a parameter when the original function calls for a `boolean` variable (even though both parameter types are one byte in length). There are a couple of exceptions worth noting. You may pass a `byte` variable as an actual parameter whenever the formal parameter is one byte in length. Similarly, HLA will allow an actual parameter whose type is `word` if the formal parameter's size is two bytes and HLA will allow an actual `dword` parameter whenever the formal parameter is four bytes. Also, if the formal parameter is a `byte`, `word`, or `dword` type, then HLA will allow you to pass an actual parameter that is one byte long, two bytes long, or four bytes long, respectively. HLA will also allow an anonymous memory object (e.g., "[eax]") as an actual parameter for any pass by reference parameter; such a parameter will simply pass the value of the specified register as the address for the reference parameter.

One feature that HLA supports as a convenience (especially for Win32 API function calls) is that if you pass a pointer variable as an actual pass by reference parameter, where the formal type of the reference parameter is the base type of the pointer, HLA will go ahead and pass the value of the pointer rather than returning an error (or passing the address of the pointer variable), e.g., the following demonstrates this:

```
type
   pi  :pointer to int32;
       .
       .
       .
   procedure hasRefParm( var i:int32 );
   begin hasRefParm;
       .
       .
       .
   end hasRefParm;

static
   myInt :int32;
   pInt  :pi;
       .
       .
       .
```

```
   hasRefParm( myInt );  // Computes and passes address of myInt.
      .
      .
      .
   hasRefParm( pInt );   // Passes the value of the pInt pointer variable
```

The choice of whether to pass a parameter as a pointer by value or as a variable by reference is mainly a matter of convenience. If you are usually passing an actual parameter that is a memory variable whose type matches the formal parameter's type, then pass by reference is probably the way to go. However, if you're doing pointer arithmetic or constantly passing the address of objects whose type doesn't exactly match the formal parameter's type (and you're sure you know what you're doing when you do this), then passing a pointer by value is probably going to be more convenient.

Many Win32 API functions accept the address of some buffer as a parameter. Often, the prototype for the function specifies the pointer type as "void *". This means that the caller is supplying the address of a block of memory and the abstract type attached to that block of memory is irrelevant to the compiler. HLA also provides a special form of the pass by reference parameter passing mechanism that suspends type checking on the actual parameters you pass to the procedure. Consider the following HLA procedure prototype:

```
procedure untypedVarParm( var parm:var ); @external;
```

Specifying "var" as the parameter's type tells HLA that this is an untyped pass by reference parameter. The caller can supply any memory address as the parameter and HLA will pass that address on to the function. Like normal pass by reference parameters, the actual parameter you supply to this function must be a memory location, you cannot supply a constant or a register as an operand (though you can specify "[reg_{32}]" as a parameter and HLA will pass the value of the specified 32-bit general purpose register as the memory address). This special pass by reference form is especially useful when passing Win32 API functions the address of some buffer where it can place data that Windows returns to the caller. There is, however, one big "gotcha" associated with untyped pass by reference parameters: HLA always passes the address of the variable you pass the function. This is true even if the variable you pass as a parameter is a pointer variable. The following is syntactically acceptable to HLA, but probably doesn't do what the programmer expects:

```
   procedure hasUntypedParm( var i:var );
   begin hasUntypedParm;
      .
      .
      .
   end hasUntypedParm;

static
   myInt :int32;
   pInt  :pi;
      .
      .
      .
   hasUntypedParm( myInt );  // Computes and passes address of myInt.
      .
      .
      .
   hasUntypedParm( pInt );   // Computes and passes address of pInt
```

In particular, note that this code does not pass the value of the pointer variable in the second call. Instead, it takes the address of the pointer variable and passes that address on to the `hasUntypedParm` procedure. So take care

when choosing to use untyped pass by reference parameters; their behavior is slightly different than regular pass by reference parameters as this example shows.

There is one important issue that HLA programmers often forget: *HLA string variables are pointers!* Most Win32 API functions that return data via a pass by reference parameter return character (string) data. It's tempting to be lazy and just declare all pass by reference parameters as untyped parameters. However, this can create havoc when calling certain Win32 API functions that return string data. Consider the following Win32 API procedure prototype:

```
static
   GetFullPathName: procedure
   (
          lpFileName    : string;
          nBufferLength : dword;
      var lpBuffer      : var;
      var lpFilePart    : var
   );
   @stdcall; @returns( "eax" ); @external( "__imp__GetFullPathNameA@16" );
```

This function stores a zero-terminated string into the block of memory pointed at by `lpBuffer`. It might be tempting to call this procedure as follows:

```
static
   s  :string;
   fp :pointer to char;
      .
      .
      .
   stralloc( 256 );
   mov( eax, s );
      .
      .
      .
   GetFullPathName( "myfile.data", 256, s, fp );
   mov( s, ebx );                                // GetFullPathName returns the actual
   mov( eax, (type str.strRec [ebx]).length );   // string length in EAX.
```

The objective of this code is (obviously) to have the call to `GetFullPathName` place the full path name of the *myfile.data* file into the string variable `s`. Unfortunately, this code does not work as advertised. The problem is that the `lpBuffer` variable is an untyped reference parameter. As a result, the call to `GetFullPathName` takes the address of whatever variable you pass it, even if that variable is a pointer variable. Since strings are four-byte pointers (that contain the address of the actual character data), this example code doesn't do what the author probably intended. Rather than passing the address of the character string data buffer as you might expect, this code passes the address of the four-byte pointer variable `s` to `GetFullPathName` as the buffer address. On return, this function will have overwritten the pointer value (and probably the values of other variables appearing in memory immediately after `s`). Notice how the original example of this function call appearing earlier in the chapter handled this situation:

```
static
   fullName :string;
   namePtr  :pointer to char;
      .
      .
      .
```

```
stralloc( 256 );       // Allocate sufficient storage to hold the string data.
mov( eax, fullName );
    .
    .
    .
mov( fullName, edx );  // Get the address of the data buffer into EDX!
GetFullPathName
(
   "myfile.exe",                          // File to get the full path for.
   (type str.strRec [edx]).MaxStrLen,     // Maximum string size
   [edx],                                 // Pointer to buffer
   namePtr                                // Address of base name gets stored here
);
mov( fullName, edx );                     // Note: Win32 calls don't preserve EDX
mov( eax, (type str.strRec [edx]).length  // Set the actual string length
```

We'll return to this issue a little later in this chapter when we discuss the conversion of Win32 API function prototypes from C to HLA.

The C language always passes arrays by reference. Whenever the C language sees the name of a function without an index operator ("[...]") attached to it, C substitutes the address of the first element of the array for that array. Similarly, if you specify some array as a formal parameter in a C function declaration, C assumes that you will actually be passing in a pointer to an element of that array type[18].

Structures, on the other hand, C always passes by value (unless, of course, you explicitly take the address of a `struct` object using the address-of operator and pass that pointer to the `struct` as your parameter value). Win32 API functions always pass pointers to structures (that is, they expect you to pass structures by reference rather than by value), so when you create a prototype for a Win32 API function call that has a `struct` as a parameter, you'll always specify a pointer to the structure or pass it by reference in the HLA declaration.

3.4: Calling Win32 API Functions

The Windows operating system consists of several dynamic linked library (DLL) modules in memory. Therefore, when you call a Win32 API function, you're not actually calling that function directly. Indeed, unless you declare your function in a special way, there may be two levels of indirection involved before you get to the actual Win32 kernel code within the DLL. This section will give you a quick primer on Win32 DLLs and how to design your Win32 API function prototypes in HLA to make them slightly more efficient.

The phrase "dynamic linked library" means that linkage to a library module is done at run-time. That is, supplying the run-time address of the library function in question could, technically, be done after your program begins execution. The linking process normally involves patching the address fields of all the call instructions that reference a given function in some library code being linked. However, at run-time (that is, after Windows has loaded your program into memory and begun its execution), it's impractical to attempt to locate every call to some function so that you can modify the address field to point at the new location of that function in memory. The solution to this problem is to provide a single object that has to be changed in memory to provide the linkage, put that object in a known location, and then update that single object whenever dynamically linking in the the function. By having a "single point of contact" the OS can easily change the address of that contact object.

There are two ways to add such a "single point of contact" to a machine code program. The first way is to use a pointer that holds the address of the ultimate routine to call. The application code, when it wants to invoke the

18.As noted earlier, C does not differentiate pointer or array access syntax. Both are identical to C, for the most part. This is how C gets away with passing all arrays as a pointer to their first element.

Win32 API function (or any other function in some DLL) would simply issue an indirect call through this pointer. The second way is to place a `jmp` instruction at a known location and modify the `jmp` instruction's address operand field to point at the first instruction of the desired function within the DLL. The indirect call mechanism is a little more efficient, but it requires encoding a special form of the call instruction whenever you call a function in the DLL (and many compilers will not generate this special form of the `call` instruction by default, if they provide the option to do it at all). The use of the `jmp` instruction mechanism is more compatible with existing software development tools, but this scheme requires the execution of an extra instruction in order to transfer control to the actual function in the DLL (specifically, you have to execute the `jmp` instruction after "calling" the function). Windows, as it turns out, combines both of these mechanisms when providing an interface to the Win32 API functions. The API interface consists of an indirect `jmp` instruction that transfers control to some location specified by a double-word pointer. The linking code can use any form of the `call` (or other control transfer) instruction to transfer control to the indirect `jmp` instruction. Then the indirect `jmp` transfers control to the actual function specified by the pointer variable. The operating system can dynamically change the target address of the function within the DLL by simply changing this pointer value in memory.

Of course, there is an efficiency drawback to this scheme. Not only must the code execute that extra `jmp` instruction, but an indirect `jmp` is typically slower than a direct `jmp`. So Windows' solution is the slowest of the three forms: you pay the price for the extra `jmp` instruction and the extra cost associated with the use of indirection. Fortunately, one of the advantages of assembly language is that you can easily circumvent this extra work.

Consider the following HLA procedure prototype to the Win32 `ExitProcess` function:

```
procedure ExitProcess( uExitCode:uns32 ); @stdcall; @external( "_ExitProcess@4" );
```

The `_ExitProcess@4` label is actually the label of an indirect `jmp` instruction that will be linked in with any program that calls `ExitProcess`. In HLA form, the code at the address specified by the `_ExitProcess@4` label looks something like the following (assuming labels like "_ExitProcess@4" were legal in HLA):

```
_ExitProcess@4: jmp( _imp__ExitProcess@4 );
```

The "__imp__ExitProcess@4" symbol is the name of a double word pointer variable that will contain the address of the actual `ExitProcess` function with the Win32 OS kernel, i.e.,

```
static
   _imp__ExitProcess@4 :dword;  // Assuming "@" was actually legal within an HLA ID.
```

Note that the library files (e.g., *kernel32.lib*) that you link your programs with contain definitions for both the symbols `_ExitProcess@4` and `_imp__ExitProcess@4`. The "standard" symbols (e.g., `_ExitProcess@4`) refer to the indirect `jmp` instruction. The symbols with the "_imp_" prefix refer to the double word pointer variable that will ultimately hold the address of the actual kernel code. Therefore, you can circumvent the execution of the extra `jmp` instruction by calling the kernel function indirectly through this pointer yourself, e.g.,

```
   call( _imp__ExitProcess@4 );   // Assuming "@" was actually legal within an HLA ID.
```

The major problem with this approach is that it doesn't allow the use of the HLA high level function call syntax. You would be forced to manually push any parameter(s) on the stack yourself when using this scheme. In a moment, you'll soon see how to circumvent this problem. Another minor issue is that HLA doesn't allow the "@" symbol in an identifier (as all the previous code fragments have noted). This, too, is easily corrected.

HLA allows you to declare both external procedures and variables. We'll use that fact to allow external linkage to both the jmp instruction (that is effectively the Win32 API function's entry point) and the pointer to the variable. The following two declarations demonstrate how you can do this:

```
static
   _imp__ExitProcess :dword; @external( "_imp__ExitProcess@4" );

procedure ExitProcess( uExitCode:uns32 ); @stdcall; @external( "_ExitProcess@4" );
```

HLA also allows the declaration of procedure variables. A procedure variable is a four-byte pointer to a given procedure. HLA procedure variables are perfect for Win32 API declarations because they allow you to use HLA's high level syntax for procedure calls while making an indirect call through a pointer. Consider the following declaration:

```
static
   ExitProcess :procedure( uExitCode:uns32 );
               @stdcall; @external( "_imp__ExitProcess@4" );
```

With this declaration, you can call `ExitProcess` as follows:

```
ExitProcess( 0 );
```

Rather than calling the code beginning with the indirect `jmp` instruction, this HLA high level procedure call does an indirect call through the `_imp__ExitProcess@4` pointer. Since this is both convenient and efficient, this is the scheme this book will generally employ for all Win32 API function calls.

3.5: Win32 API Functions and Unicode Data

Before discussing how to create HLA procedure prototypes for all the Win32 API functions, a short digression is necessary in order to understand certain naming conventions in the Win32 API. For many years, there were actually two different Win32 OS families: the Windows 95/98/2000ME family and the Windows NT/2000/XP family. The branch starting with Windows 95 was based on Windows 3.1 and MS-DOS to a large extent. The OS branch that started with NT was written mostly from scratch without concern about legacy (DOS) compatibility. As such, the internal structure of these two operating system families was quite different. One area where the difference is remarkable is with respect to character data. The Windows 95 family uses standard eight-bit ANSI (ASCII) characters internally while the NT branch uses Unicode internally. Unicode, if you're unfamiliar with it, uses 16-bit character codes allowing the use of up to 65,536 different characters. The beauty of Unicode is that you can represent most character symbols in use by various languages with a single character code (unlike ASCII, which only supports 128 different character values and isn't even really adequate for English, much less English plus dozens of other languages). Since Microsoft was interested in producing an international operating system, Unicode seemed like the right way to go.

Unicode has some great advantages when it comes to write applications that work regardless of the natural language of the application's users. However, Unicode also has some serious disadvantages that prevent it from immediately taking over the world:

- Few software tools directly support Unicode, so it is difficult to develop Unicode-enabled applications (though this is changing as time passes).
- Unicode data requires twice as much storage as ANSI data. This has the effect of doubling the size of many databases and other applications that manipulate a considerable amount of character data.

- Because Unicode characters are twice as long as ANSI characters, processing Unicode data typically takes twice as long as processing ANSI/ASCII characters (a serious defect to most assembly language programmers).
- Many programs that manipulate character data use look-up tables and bit maps (character sets) to operate on that data. An ASCII-based look-up table requires 128 bytes, an ANSI look-up table typically requires 256 bytes, a Unicode-based look-up table would require 65,536 bytes (making it impractical to use a look-up table for all but the most specialized of cases when using Unicode). Even implementing a character set using a power set (i.e., a bit map) would require 8,192 bytes; still too large for most practical purposes.
- There are nowhere near as many Unicode-based string library functions available as there are for ASCII/ANSI based strings. For example, the HLA Standard Library provides almost no Unicode-based string functions at all (actually, it provides none, but a few routines will work with Unicode-based strings).
- Another problem with using Unicode is that HLA v1.x provides only basic support for Unicode characters[19]. At the time this was being written, HLA supported the declaration of `wchar` and `wstring` constants and variables as well as Unicode character and string literal constants (of the form u'A' and u"AAA"). You could also initialize wchar and wstring static objects as the following example demonstrates. However, HLA constant expression parser does not (as of this writing) support Unicode string operations nor does the HLA Standard Library provide much in the way of Unicode support. The following is an example of static initialization of Unicode data (see the HLA reference manual for more details):

```
static
   wCharVar   :wchar := u'w';
   wStringVar :wstring := u"Unicode String";
```

For all these reasons, and many more, Microsoft realized (while designing Windows NT) that they couldn't expect everyone to switch completely over to Unicode when writing applications for Windows NT (or when using applications written for Windows NT). Therefore, Microsoft's engineers provided duomorphic[20] interfaces to Windows NT that involve character data: one routine accepts and returns ANSI data, another routine accepts and returns Unicode data. Internally, of course, Windows NT doesn't really have two sets of routines. Instead, the ANSI routines simply convert incoming data from ANSI to Unicode and the outgoing data from Unicode to ANSI.

In Microsoft Visual C++ (and other high level languages) there is a little bit of macro trickery used to hide the fact that the application has to choose between the Unicode-enabled and the ANSI versions of the Win32 API function calls. By simply changing one macro definition in a properly-written C++ program, it's possible to switch from ANSI to Unicode or from Unicode to ANSI with no other changes to the program. While the same trick is theoretically possible in assembly language (at least, in HLA), the dearth of a good set of Unicode library functions reduces this to the status of an interesting, but not very useful, trick. Therefore, this book will concentrate on producing ANSI-compatible applications with a small discussion of how to do Unicode applications when doing so becomes more practical in assembly language.

Windows duomorphic interface only applies to those functions that accept or return character data. A good example of such a routine is the Win32 API `DeleteFile` function that has the following two interfaces:

```
procedure DeleteFile( lpFileName :string ); @stdcall; @external( "_DeleteFileA@4" );
```

19.Least you use this as an argument against using HLA, note that HLA actually provides *some* Unicode support. Most assemblers provide no Unicode support whatsoever at all.

20.Two-faced.

```
// -or-

   procedure DeleteFile( lpFileName :wstring ); @stdcall; @external( "_DeleteFileW@4" );
```

If you look closely at these two declarations, you'll notice that the only difference between the two is a single character appearing in the external name and the type of the parameter. One of the external names has an "A" (for ANSI) immediately before the "@" while the other has a "W" (for Wide) immediately before the "@" character in the name. Wide, in this context, means a two-byte character format; so the name with the embedded "W" is the Unicode version of the function's name.

The presence of the "A" or the "W" at the end of the function's name in the external declaration (i.e., just before the "@", whose purpose the next section covers) determines whether the function is the ANSI version or the Unicode version ("A"=ANSI, "W"=Unicode). There is only one catch: when reading C/C++ documentation about the Windows API, you'll generally see the function referred to as "DeleteFile" (or whatever), not "DeleteFileA" or "DeleteFileW". So how can you tell whether the function's external name requires the "A" or "W"? Well, if any of the parameters involves character or string data, it's a given that the function will have ANSI and Unicode counterparts. If you're still not sure, you can always run Microsoft's *dumpbin.exe* utility on one of the Win32 API interface libraries (e.g., *kernel32.lib, gdi32.lib, user32.lib*, etc.) to extract all the exported names:

```
dumpbin /exports kernel32.lib
```

This command lists all the Win32 API function names that the *kernel32.lib* library module exports. If you save the output of this command to a text file (by using I/O redirection) you can search for a particular function name with nearly any text editor. Once you find the filename, if there is an "A" or "W" at the end of the name, you know that you've got a duomorphic function that deals with ANSI or Unicode characters. If no such character appears, then the function only works with binary (non-character) data.

Please note that the official name for a Win32 API function does not include the "A" or "W" suffix. That is, the Win32 documentation refers only to names like `DeleteFile`, never to names like `DeleteFileA` or `DeleteFileW`. The assumption is that an application is only going to use one of the two different character types. Either all character data is ANSI, in which case the application will call those functions with the "A" suffix, or all character data is in Unicode form and the application will call those functions with the "W" suffix. Although it's easy enough to switch between the two in an assembly language program, it's probably a good idea to stick to one form or another in a given application (less maintenance issues that way). The examples in this book will all use the ANSI forms of these functions, since assembly language better supports eight-bit character codes.

This book will also adopt the Win32 convention of specifying the API function names without the "A" or "W" suffix. That is, we'll call functions like `DeleteFile` and `GetFullPathName` and not worry about whether it's ANSI or Unicode on each call. The choice will be handled in the declaration of the prototype for the particular Win32 API function. This makes it easy (well, easier) to change from one character format to another should the need arise in the future.

For the most part, this book will stick to the ANSI character set because HLA provides much better support for that character set. If you need to use Unicode in your programs, you'll need to adjust the Win32 API prototypes and HLA `char/string` declarations accordingly.

Note that the names that have the "A" and "W" suffixes are really external names only. C/C++ documentation doesn't mention these suffixes. Again, if you're unsure whether the suffix is necessary, run the *dumpbin* program to get the actual library name.

3.6: Win32 API Functions and the Parameter Byte Count

As you've seen in a few examples appearing in this chapter, the external Win32 API function names typically have an at-sign ("@") and a number appended to the end of the function's external name. This numeric value specifies the number of bytes passed as parameters to the functions. Since most parameters are exactly four bytes long, this number (divided by four) usually tells you how many parameters the API function requires (note that a few API calls have some eight-byte parameters, so this number isn't always an exact indication of the number of parameters, but it does hold true the vast majority of the time).

Note that the names that have the "@nn" suffix are really external names only. C/C++ documentation doesn't mention this suffix. Furthermore, since HLA doesn't allow you to embed at signs ("@") into identifiers, you cannot use these external names as HLA identifiers. Fortunately, HLA's `@external` directive allows you to specify any arbitrary string as the external symbol name.

This book will also adopt the Win32 convention of specifying the API function names without the "@nn" suffix. That is, we'll call functions like `DeleteFile` and `GetFullPathName` and not worry about tacking on the number of bytes of parameters to the name. The full name will be handled in the external prototype declaration for the particular Win32 API function. If you need to determine the exact constant for use in an external declaration, you can run the Microsoft *dumpbin* program on the appropriate .LIB file to determine the actual suffix.

3.7: Creating HLA Procedure Prototypes for Win32 API Functions

Although the HLA distribution includes header files that provide prototypes for most of the Win32 API functions (see the next section for details), there are still some very good reasons why you should be able to create your own HLA external declarations for a Win32 function. Here is a partial list of those reasons:

- HLA provides most, but not all, of the Win32 API Prototypes (e.g., as Microsoft adds new API calls to Windows, HLA's header files may become out of date).
- Not every HLA prototype has been thoroughly tested (there are over 1,500 Win32 API function calls one could make and some of those are quite esoteric). There very well could be a defect in the prototype of some function that you want to call.
- The choice of data type for a give API function may not exactly match what you want to use (e.g., it could specify an `uns32` type when you'd prefer the more general `dword` type).
- You may disagree with the choice of passing a parameter by reference versus passing a pointer by value.
- You may disagree with the choice of an untyped reference parameter versus a typed reference parameter.
- You may disagree with the choice of an HLA string type versus a character buffer.

There are certainly some other reasons for abandoning HLA's external prototypes for various Win32 API functions. Whatever the reason, being able to create an HLA prototype for these functions based on documentation that provides a C prototype is a skill you will need. The following subsections condense the information appearing in the previous sections, toss out a few new ideas, and discuss the "hows and whys" of Win32 API prototyping in HLA.

3.7.1: C/C++ Naming Conventions Versus HLA Naming Conventions

Before jumping in and describing how to translate C/C++ prototypes into HLA format, a slight digression is necessary. Sometimes, you'll run into a minor problem when translating C/C++ code to HLA: identifiers in a C/

C++ program don't always map to legal identifiers in an HLA program. Another area of contention has to do with the fact that Microsoft's programmers have created many user-defined types that the Windows system uses. More often than not, these type names are isomorphisms (that is, a different name for the same thing; for example, Microsoft defines dozens, if not hundreds, of synonyms for `dword`). However, if you understand Microsoft's naming conventions, then figuring out what HLA types to substitute for all these Microsoft names won't prove too difficult.

HLA and C/C++ use roughly the same syntax for identifiers: identifiers may begin with an alphabetic (uppercase or lowercase) character or an underscore, and zero or more alphanumeric or underscore characters may follow that initial character. Given that fact, you'd think that converting C/C++ identifiers to HLA would be fairly trivial (and most of the time, it is). There are, however, two issues that prevent the translation from being completely trivial: HLA reserved words and case sensitivity. We'll discuss these issues shortly.

Even when a C/C++ identifier maps directly to a legal HLA identifiers, questions about that identifier , its readability, applicability, etc., may arise. Unfortunately, C/C++ naming conventions that have been created over the years tend to be rather bad conventions (remember, C was create circa-1970, back in the early days of "software engineering" before people really studied what made one program more readable than another). Unfortunately, there is a lot of inertia behind these bad programming conventions. Someone who is not intimately familiar with those conventions may question why a book such as this one (which covers a different language than C/C++) would continue to propagate such bad programming style. The reason is practical: as this chapter continues to stress, there is a tremendous amount of documentation written about the Win32 API that is C-based. While there is an aesthetic benefit to renaming all the poorly-named identifiers that have become standards in C/C++ Windows source files, doing so almost eliminates the ability to refer to non-HLA based documentation on the Win32 API. That would be a much greater loss than having to deal with some poorly named identifiers. For that reason alone, this book attempts to use standard Windows identifiers (which tend to follow various C/C++ naming conventions) whenever referring to those objects represented by the original Windows identifiers. Changes to the Windows naming scheme are only made where absolutely necessary. However, this book will only use the Windows naming conventions for pre-existing, reknown, Windows (and C/C++) identifiers. This book will adopt the standard "HLA naming convention" (given a little later) for new identifiers.

One problem with C/C++ naming conventions is that they are inconsistent. This is because there isn't a single C/C++ naming convention, but several that have sprung up over the years. Some of them contain mutually exclusive elements, still it isn't unusual to see several of the conventions employed within the same source file. Since the main thrust of this chapter is to prepare you to read Win32 API documentation, the sections that follow will concentrate on those conventions and problems you'll find in typical Windows documentation.

3.7.1.1: Reserved Word Conflicts

The HLA language defines hundreds of reserved words (this is reasonable, since there are hundreds of machine instructions in the 80x86 instruction set, though there is no arguing against that fact that HLA has a large number of reserved words above and beyond the machine instructions). Since not all of HLA's reserved words are reserved words in C/C++, it stands to reason that there are some programs out there than inadvertently use HLA reserved words as identifiers in their source code. This fact is just as true for the Win32 API definitions appearing in Microsoft's C/C++ header files as it is for application programs. There will be some C/C++ identifiers in the Win32 C/C++ documentation that we will not be able to use simply because they are HLA reserved words. Fortunately, such occurrences are rare. This book will deal with such issues on a case-by-case basis, providing a similar name that is not an HLA reserved word when such a conflict arises.

3.7.1.2: Alphabetic Case Conflicts

Another point of conflict between HLA identifiers and C/C++ identifiers is the fact that C/C++ is a *case sensitive* language whereas HLA is a *case neutral* language. HLA treats upper and lower case characters as distinct, but will not allow you to create an identifier that is the same as a different identifier except for alphabetic case. C/C++, on the other hand, will gladly let you create two different identifiers whose only difference is the case of alphabetic characters within the symbols. Worse, some C/C++ programmers have convinced themselves that it's actually a good idea to take advantage of this "feature" in the language (hint: it's a *terrible* idea to do this, it make programs harder to read and understand). Regardless of your beliefs of the utility of this particular programming style, the fact remains that C/C++ allows this (and some programmers take advantage of it) while HLA does not. The question is "which identifier do we modify and how do we modify it?"

Most of the time there is a case neutrality violation in a C/C++ program (that is, two identifiers are the same except for alphabetic case), it's usually the situation where one of the identifiers is either a type definition or a constant definition (the other identifier is usually a function or variable name). This isn't true all the time, but it is true in the majority of the cases where this conflict occurs. When such a conflict occurs, this book will use the following convention (prioritized from first to last):

- If one of the conflicting identifiers is a type name, we'll convert the name to all lowercase characters and append "_t" to the name (a common Unix convention).
- If one of the conflicting identifiers is a constant (and the other is not a type), we'll convert the name to all lowercase and append "_c" to the name (an HLA convention, based on the Unix convention).
- If neither of the above conditions hold, we'll give one of the identifiers a more descriptive name based on what the identifier represents/contains/specifies rather than on its classification (e.g., type of symbol).

A good convention to follow with respect to naming identifiers is the "telephone test." If you can read a line of source code over the telephone and have the listener understand what you're saying without explicitly spelling out an identifier, then that identifier is probably a decent identifier. However, if you have to spell out the identifier (especially when using phrases like "upper case" and "lower case" when spelling out the name), then you should consider using a better name. HLA, of course, prevents abusing and misusing alphabetic case in identifiers (being a case neutral language), so it doesn't even allow one to create identifiers that violate the telephone test (at least, from an alphabetic case perspective).

3.7.1.3: Common C/C++ Naming Conventions

If you search on the Internet for "C Naming Conventions" you'll find hundreds of pages extolling the benefits of that particular web page author's favorite C naming scheme. It seems like nearly every C programmer with an opinion and a web page is willing to tell the world how identifiers should be designed in C. The really funny thing is that almost every one of these pages that specifies some naming convention is mutually exclusive with every other such scheme. That is, if you follow the rules for naming C identifiers found at one web site, you're invariably break one or more rules at nearly every other site that provides a C naming convention. So much for convention; so much for standards.

Interestingly enough, the one convention that nearly everybody agrees upon is also, perhaps, the *worst* naming convention ever invented for programming language identifiers. This is the convention of using all uppercase characters for identifiers that represent constant values. The reason everyone agrees on this one convention is fairly obvious to someone who has been programming in the C programming language for some time: this is one of the few naming conventions proposed by Kernighan and Ritchie in their original descriptive text *The C Pro-*

gramming Language. In *Programming in C: A Tutorial* by Brian W. Kernighan, Mr. Kernighan describes this choice thusly:

> *Good style typically makes the name in the #define upper case; this makes parameters more visible.*

This quote probably offers some insight into why Kernighan and Ritchie proposed the use of all uppercase for constants in the first place. One thing to keep in mind about this naming convention (using all upper case for #define symbols) was that it was developed in the very early 1970s. At the time, many mainframes and programming languages (e.g., FORTRAN) only worked with uppercase alphabetic characters. Therefore, programmers were used to seeing all uppercase alphabetic characters in a source file and lowercase was actually unusual (despite the fact that C was developed on an interactive system that supported lower case). In fact, Kernighan and Ritchie really got it backwards - if they'd wanted the parameters to stand out, they should have made them all uppercase and made the #define name lower case. Another interesting thing to note from this quote was that the all uppercase convention was specifically created for *macros*, not *manifest constants*. The "good style" Brian Kernighan was referring to was an attempt to differentiate the macro name from the macro parameters. Manifest constants (that is, the typical constants you create with a #define definition) don't have parameters, so there is little need to differentiate the name from the macro's parameter list (unless, of course, Mr. Kernighan was treating the remainder of the line as the "parameters" to the #define statement).

Psychologists have long known (long before computer programming languages became popular) that uppercase text is much harder to read than lower case text. Indeed, to a large extent, the whole purpose of uppercase alphabetic text is to slow the reader down and make them take notice of something. All uppercase text makes material harder to read, pure and simple. Don't believe this? Try reading the following:

> PSYCHOLOGISTS HAVE LONG KNOWN (LONG BEFORE COMPUTER PROGRAMING LANGUAGES BECAME POPULAR) THAT UPPERCASE TEXT IS MUCH HARDER TO READ THAN LOWER CASE TEXT. INDEED, TO A LARGE EXTENT, THE WHOLE PURPOSE OF UPPERCASE ALPHBETIC TEXT IS TO SLOW THE READER DOWN AND MAKE THEM TAKE NOTICE OF SOMETHING. ALL UPPERCASE TEXT MAKES MATERIAL HARDER TO READ, PURE AND SIMPLE. DON'T BELEVE THIS? TRY REREADING THIS PARAGRPH.

There are four intentional spelling mistakes in the previous paragraph. Did you spot them all the first time you read this paragraph? They would have been much more obvious had the text been all lowercase rather than all uppercase. Reading all upper case text is so difficult, that most readers (when faced with reading a lot of it) tend to "short-circuit" their reading and automatically fill in words once they've read enough characters to convince them they've recognized the word. That's one of the reasons it's so hard to immediately spot the spelling mistakes in the previous paragraph. Identifiers that cause a lack of attention to the details are obviously problematic in a programming language and they're not something you want to see in source code. A C proponent might argue that this isn't really much of a problem because you don't see as much uppercase text crammed together as you do in the paragraph above. However, some long identifiers can still be quite hard to read in all upper case; consider the following identifier taken from the C/C++ *windows.inc* header file set: CAL_SABBREVMONTHNAME1. Quick, what does it mean?

Sometimes it is useful to make the reader slow down when reading some section of text (be it natural language or a synthetic language like C/C++). However, it's hard to argue that every occurrence of a constant in a source file should cause the reader to slow down and expend extra mental effort just to read the name (thus, hopefully, determining the purpose of the identifier). The fact that it is a constant (or even a macro) is far more easily conveyed using some other convention (e.g., the "_c" convention that this book will adopt).

Now some might argue that making all macro identifiers in a program stand out is a good thing. After all, C's macro preprocessor is not very good and it's macro expander produces some unusual (non-function-like) seman-

tics in certain cases. By forcing the programmer to type and read an all-uppercase identifier, the macro's designer is making them note that this is not just another function call and that it has special semantics. An argument like this is valid for macros (though a suffix like "_m" is probably a better way to do this than by using all uppercase characters in the identifier), but is completely meaningless for simple manifest constants that don't provide any macro parameter expansion. All in all, using all uppercase characters for identifiers in a program is a bad thing and you should avoid it if at all possible.

This text, of course, will continue to use all uppercase names for well-known constants defined in Microsoft's C/C++ header files. The reason is quite simple: they are documented in dozens and dozens of Windows programming books and despite the fact that such identifiers are more difficult to read, changing them in this text would prove to be a disaster because the information appearing herein would not be compatible with most of the other books on Windows programming in C/C++[21]. For arbitrary constant identifiers (i.e., those that are not predefined in the C/C++ Windows header files), this book will generally adopt the "_c" convention for constants.

One C/C++ naming convention that is specified by the original language definition is that identifiers that begin and end with an underscore are reserved for use by the compiler. Therefore, C/C++ programmers should not use such identifiers. This shouldn't prove to be too onerous for HLA programmers because HLA imposes the same restriction (identifiers beginning and ending with an underscore are reserved for use by the compiler and the HLA Standard Library).

Beyond these two conventions that have existed since the very first operational C compilers, there is very little standardization among the various "C Naming Conventions" documents you'll find on the Internet. Many suggestions that one document makes are style violations another document expressedly forbids. So much for standardized conventions! The problem with these myriad of non-standardized "standards" is that unless you include the style guide in the comments of a source file, the guidelines you're following are more likely to confuse someone else reading the source file who is used to operating under a different set of style guidelines.

Perhaps one of the most confusing set of style guidelines people come up with for C/C++ programs is what to do about alphabetic case. Wise programmers using alphabetic case differences for formatting only. They never attach meaning to the case of alphabetic characters within an identifier. All upper case characters for constants is fairly easy to remember (because it is so well ingrained in just about every C/C++ style guide ever written), but how easy is it to remember that "variables must be written in mixed case starting with a lower case character" and "Names representing types must be in mixed case beginning with an uppercase character"? There are some so-called style guidelines that list a dozen different ways to use alphabetic case in an identifier to denote one thing or another. Who can remember all that? What happens when someone comes along and doesn't intimately know the rules? Fortunately, you see little of this nonsense in Windows header files.

As noted earlier in this document, a common (though not ubiquitous) Unix/C programming convention is to append the suffix "_t" to type identifiers. This is actually an excellent convention (since it emphasizes the classification of an identifier rather than its type, scope, or value). The drawback to this scheme is that you rarely see it used consistently even within the same source file. An even bigger drawback is that you almost never see this naming convention in use in Windows code (Windows code has a nasty habit of using all uppercase to denote type names, as well as constant, macro, and enum identifiers, thus eliminating almost any mnemonic value the use of all uppercase might provide; about the only thing you can say about an all-uppercase symbol in a Windows program is that it's probably not a variable or a function name). Once again, this book will use standard Windows identifiers when referencing those IDs, but will typically use the Unix convention of the "_t" suffix when creating new type names.

---

21. Note, however, that there is a precedent for changing the Win32 API identifiers around when programming in a language other than C/C++. Borland's documentation for Delphi, for example, changes the spelling of many Windows identifiers to something more reasonable (note, however, that Pascal is a case insensitive language and some changes were necessary for that reason alone).

Without question, the most common naming convention in use within C/C++ Windows applications is the use of *Hungarian Notation*. Hungarian notation uses special prefix symbols to denote the type of the identifier. Since Hungarian notation is so prevalent in Windows programs, it's worthwhile to spend some time covering it in detail...

3.7.1.4: Hungarian Notation

Hungarian notation is one of those "innovations" that has come out of Microsoft that lots of people love and lots of people hate. Originally developed by Charles Simonyi in a technical paper (searching on the Internet for "Hungarian Notation Microsoft Charles Simonyi" is bound to turn up a large number of copies of his paper (or links to it), Hungarian notation was adopted internally within Microsoft and the popularized outside Microsoft via the Windows include files and Charles Petzold's "Programming Windows" series of books (which push Hungarian notation). As a result of these events, plus the large number of programmers that "cut their teeth" at Microsoft and went on to work at companies elsewhere, Hungarian notation is very a popular naming convention and it's difficult to read a C/C++ Windows program without seeing lots of examples of this notation.

Hungarian notation is one of those conventions that everyone loves to hate. There are lots of good, technical, reasons for not using Hungarian notation. Even many proponents of Hungarian notation will admit that it has its problems. However, people don't use it simply because Microsoft pushes it. In spite of the problems with Hungarian notation, the information it provides is quite useful in large programs. Even if the convention wasn't that useful, we'd still need to explore it here because you have to understand it in order to read C/C++ code; still, because it is somewhat useful, this book will even adopt a subset of the Hungarian notation conventions on an "as useful" basis.

Hungarian notation is a naming convention that allows someone reading a program to quickly determine the type of a symbol (variable, function, constant, type, etc.) without having to look up the declaration for that symbol. Well, in theory that's the idea. To someone accustomed to Hungarian notation, the use of this convention can save some valuable time figuring out what someone else has done. The basic idea behind Hungarian notation is to add a concise prefix to every identifier that specifies the type of that identifier (actually, full Hungarian notation specifies more than that, but few programmers use the full form of Hungarian notation in their programs). In theory, Hungarian notation allows programmers to create their own type prefixes on an application by application basis. In practice, most people stick to the common-predefined type prefixes (tags) let it go at that.

An identifier that employs Hungarian notation usually takes the following generic form:

```
prefix tag qualifier baseIdentifier
```

Each of the components of the identifier are optional, subject of course, to the limitation that the identifier must contain something. Interestingly enough, the `baseIdentifier` component (the name you'd normally think of as the identifier) is itself optional. You'll often find Hungarian notation "identifiers" in Windows programs that consist only of a possible prefix, tag, and/or qualifier. This situation, in fact, is one of the common complaints about Hungarian notation - it encourages the use of meaningless identifiers in a source file. The `baseIdentifier` in Hungarian notation is the symbol you'd normally use if you weren't using Hungarian notation. For the sake of example, we'll use `Variable` in the examples that follow as our base identifier.

The `tag` component in the Hungarian notation is probably the most important item to consider. This item specifies the base type, or use, of the following symbol. Table 3-12 lists many of the basic tags that Windows pro-

grams commonly use; note that Hungarian notation does not limit a program to these particular types, the user is free to create their own tags.

Table 3-12: Basic Tag Values in Hungarian Notation

Tag	Description
f	Flag. This is a true/false boolean variable. Usually one byte in length. Zero represents false, anything else is true.
ch	Character. This is a one-byte character variable.
w	Word. Back in the days of 16-bit Windows systems (e.g., Windows 3.1), this tag meant a 16-bit word. However, as a perfect demonstration of one of the major problems with Hungarian notation, the use of this prefix became ambiguous when Win32 systems started appearing. Sometimes this tag means 16-bit short, sometimes it means a 32-bit value. This prefix doesn't provide much in the way of meaningful information in modern Windows systems.
b	Byte. Always a one-byte value.
l	Long. This is generally a long integer (32 bits).
dw	Double Word. Note that this is not necessarily the same thing as an "l" object. In theory, the usage of this term is as ambiguous as "w", though in 80x86 Windows source files this is almost always a 32-bit double word object.
u	Unsigned. Typically denotes an unsigned integer value (usually 32 bits). Sometimes you will see this symbol used as a prefix to one of the other integer types, e.g., uw is an unsigned word.
r	Real. Four-byte single precision real value.
d	Double. Eight-byte double precision real value.
bit	A single bit. Typically used with field names that are bit fields within some C struct.
bm	Bit map. A collection of bits (e.g., pixel values).
v	Void. An untyped object. Typically used only with the pointer prefix (see the discussion of prefixes). Untyped pointers are always 32 bit objects under Win32.
st	String. Object is a Pascal string with a length prefix.
sz	String, zero terminated. Object is a C/C++ zero terminated string object.

In Table 3-12 you see the basic type values commonly associated with symbols employing Hungarian notation. Table lists some modifier prefixes you may apply to these types (though there is no requirement that a tag

appear after one of the prefixes, a lone prefix followed by the base identifier is perfectly legal, though not as readable as an identifier consisting of a prefix, tag, and base identifier).

Table 3-13: Common Prefix Values in Hungarian Notation

Prefix	Description
p	Pointer to some type.
lp	Long pointer to some type. Today, this is a synonym for "p". Back in the days of 16-bit Windows system, an "lp" object was 32 bits and a "p" object was 16 bits. Today, both pointer types are identical and are 32 bits long. Although you'll see this prefixed used quite a bit in existing code and header files, you shouldn't use this prefix in new code.
hp	Huge pointer to some type. Yet another carry-over from 16-bit Windows days. Today, this is synonymous with lp and p. You shouldn't use this prefix in new code.
rg	Lookup table. Think of an index into an array as a function parameter, the function's result (i.e., the table entry) is the *range* of that function, hence the designation "rg". This one is not common in many Windows programs.
i	An index (e.g., into an array). Also commonly employed for `for` loop control variables.
c	A count. cch, for example, might be the number of characters in some array of characters.
n	A count. Same as c but more commonly used to avoid ambiguity with ch.
d	The difference between two instances of some type. For example, dX might be the difference between to x-coordinate values.
h	A Handle. Handles are used through Windows to maintain resources. Many Win32 API functions require or return a handle value. Handles are 32-bit objects under Win32.
v	A global variable. Many programmers use 'g' rather than 'v' to avoid confusion with the 'v' basic tag specification.
s	A static variable (local or global)
k	A `const` object.

Here are some examples of names you'll commonly see (e.g., from the Windows.hhf header file) that demonstrate the use of these identifiers:

```
char *lpszString;     // Pointer to zero-terminated string of characters.
int  *pchAnswer;      // Pointer to a single character holding an answer.
HANDLE hFile;         // Handle of a file.
```

Note all of these prefixes and tags are equally popular in Windows programs. For example, you'll rarely see the "k" prefix specification in many Windows source files (instead, the programmer will probably use the common C/C++ uppercase convention to denote an identifier). Also, many of the prefix/tag combinations are ambiguous. Fortunately, few people (including the Windows header files) push Hungarian notation to the limit. Usually, you'll just see a small subset of the possibilities in use and there is little ambiguity.

Another component of Hungarian notation, though you'll rarely see this used in real life, is a qualifier. More often that not, qualifiers (if they appear at all) appear in place of the base identifier name. Table lists some of the common qualifiers used in Hungarian notation.

Table 3-14: Common Qualifiers in Hungarian Notation

Qualifier	Description
First	The first item in a set, list, or array that the program is working with (this does not necessarily indicate element zero of an array). E.g., iFirstElement.
Last	The last item in a set, list, or array that the program has worked upon (this does not necessarily indicate the last element of an array or list). E.g., pchLastMember. Note that Last index objects are always valid members of the set, list, or array. E.g., array[iLastIndex] is always a valid array element.
Min	Denotes the minimum index into a set, list, or array. Similar to First, but First specifies the first element you're dealing with rather than the first object actually present.
Max	Denotes an upper limit (plus one, usually) into an array or list.

There are many, many different variants of Hungarian notation. A quick perusal of the Internet will demonstrate that almost no one really agrees on what symbols should be tags, prefixes, or qualifiers, much less what the individual symbols in each of the classes actually mean. Fortunately, the Windows header files are fairly consistent with respect to their use of Hungarian notation (at least, where they use Hungarian notation), so there won't be much difficulty deciphering the names from the header files that we'll use in this book.

As for using Hungarian notation in new identifiers appearing in this book, that will only happen when it's really convenient to do so. In particular, you'll see the "h" (for Handle) and "p" (for pointer) prefixes used quite a bit. Once in a while, you may see some other bits and pieces of Hungarian notation in use (e.g., b, w, and dw for `byte`, `word`, and `dword` objects). Beyond that, this book will attempt to use descriptive names, or at least, commonly used names (e.g., `i, j,` and `k` for array indexes) rather than trying to explain the identifier with a synthetic prefix to the identifier.

3.8: The w.hhf Header File

Provided with the HLA distribution is the *w.hhf* include file that define most of the Win32 API functions, constants, types, variables, and other objects you'll ever want to reference from your assembly language code. All in all, you're talking well over 30,000 lines of source code! It is convenient to simply stick an HLA `#include` statement like the following into your program and automatically include all the Windows definitions:

```
#include( "w.hhf" )
```

The problem with doing this is that it effectively increases the size of your source file by 30,000 lines of code. Fortunately, recent modifications to HLA have boosted the compile speed of this file to about 25,000 lines/second, so it can process this entire include file in just a few seconds. Most people don't really care if an assembly takes two or three seconds, so including everything shouldn't be a problem (note that the inclusion of all this code does not affect the size of the executable nor does it affect the speed of the final program you're writing; it only affects the compile-time of the program). For those who are bothered by even a few seconds of compile time, there is a solution.

A large part of the problem with the HLA/Windows header files is that the vast majority of the time you'll never use more than about 10% of the information appearing in these header files. Windows defines a tremendous number of esoteric types, constants, and API functions that simply don't get used in the vast majority of Windows applications. If we could pick out the 10% of the definitions that you were actually going to use on your next set of projects, we could reduce the HLA compilation overhead to almost nothing, far more acceptable to those programmers that are annoyed by a few seconds of delay. To do this, you've got to extract the declarations you need and put them in a project-specific include file. The examples in this book won't bother with this (because compiling the *w.hhf* file is fast enough), but feel free to do this if HLA compile times bother you.

3.9: And Now, on to Assembly Language!

Okay, we've probably spent enough time discussing C/C++ in a book dedicated to writing Windows programs in assembly language. The aim of this chapter has been to present a firm foundation for those who need to learn additional Windows programming techniques by reading C/C++ based documentation. There is no way a single chapter (even one as long as this one) can completely cover all the details, but there should be enough information in this chapter to get you well on your way to the point of understanding how to interface with Windows in assembly language by reading C/C++ documentation on the subject. Now, however, it's time to turn our attention to writing actual Windows applications in assembly language.