Introduction to Character Strings Chapter Two

2.1 Chapter Overview

This chapter discusses how to declare and use character strings in your programs. While not a complete treatment of this subject (additional material appears later in this text), this chapter will provide sufficient information to allow basic string manipulation within your HLA programs.

2.2 Composite Data Types

Composite data types are those that are built up from other (generally scalar) data types. This chapter will cover one of the more important composite data types – the character string. A string is a good example of a composite data type – it is a data structure built up from a sequence of individual characters and some other data.

2.3 Character Strings

After integer values, character strings are probably the most popular data type that modern programs use. The 80x86 does support a handful of string instructions, but these instructions are really intended for block memory operations, not a specific implementation of a character string. Therefore, this section will concentrate mainly on the HLA definition of character strings and also discuss the string handling routines available in the HLA Standard Library.

In general, a *character string* is a sequence of ASCII characters that possesses two main attributes: a *length* and the *character data*. Different languages use different data structures to represent strings. To better understand the reasoning behind HLA strings, it is probably instructive to look at two different string representations popularized by various high level languages.

Without question, *zero-terminated strings* are probably the most common string representation in use today because this is the native string format for C/C++ and programs written in C/C++. A zero terminated string consists of a sequence of zero or more ASCII characters ending with a byte containing zero. For example, in C/C++, the string “abc” requires four characters: the three characters ‘a’, ‘b’, and ‘c’ followed by a byte containing zero. As you’ll soon see, HLA character strings are upwards compatible with zero terminated strings, but in the meantime you should note that it is very easy to create zero terminated strings in HLA. The easiest place to do this is in the STATIC section using code like the following:

```
static
    zeroTerminatedString: char; @nostorage;
           byte “This is the zero terminated string”, 0;
```

Remember, when using the @NOSTORAGE option, no space is actually reserved for a variable declaration, so the *zeroTerminatedString* variable’s address in memory corresponds to the first character in the following BYTE directive. Whenever a character string appears in the BYTE directive as it does here, HLA emits each character in the string to successive memory locations. The zero value at the end of the string properly terminates this string.

Zero terminated strings have two principle attributes: they are very simple to implement and the strings can be any length. On the other hand, zero terminated string haves a few drawbacks. First, though not usually important, zero terminated strings cannot contain the NUL character (whose ASCII code is zero). Generally, this isn’t a problem, but it does create havoc once in a great while. The second problem with zero terminated strings is that many operations on them are somewhat inefficient. For example, to compute the length of a zero terminated string you must scan the entire string looking for that zero byte (counting each

character as you encounter it). The following program fragment demonstrates how to compute the length of the string above:

```
mov( &zeroTerminatedString, ebx );
mov( 0, eax );
while( (type byte [ebx]) <> 0 ) do

    inc( ebx );
    inc( eax );

endwhile;

// String length is now in EAX.
```

As you can see from this code, the time it takes to compute the length of the string is proportional to the length of the string; as the string gets longer it will take longer to compute its length.

A second string format, *length-prefixed strings*, overcomes some of the problems with zero terminated strings. Length-prefixed strings are common in languages like Pascal; they generally consist of a length byte followed by zero or more character values. The first byte specifies the length of the string, the remaining bytes (up to the specified length) are the character data itself. In a length-prefixed scheme, the string "abc" would consist of the four bytes $03 (the string length) followed by 'a', 'b', and 'c'. You can create length prefixed strings in HLA using code like the following:

```
data
    lengthPrefixedString:char;
           byte 3, "abc";
```

Counting the characters ahead of time and inserting them into the byte statement, as was done here, may seem like a major pain. Fortunately, there are ways to have HLA automatically compute the string length for you.

Length-prefixed strings solve the two major problems associated with zero-terminated strings. It is possible to include the NUL character in length-prefixed strings and those operations on zero terminated strings that are relatively inefficient (e.g., string length) are more efficient when using length prefixed strings. However, length prefixed strings suffer from their own drawbacks. The principal drawback to length-prefixed strings, as described, is that they are limited to a maximum of 255 characters in length (assuming a one-byte length prefix).

HLA uses an expanded scheme for strings that is upwards compatible with both zero-terminated and length-prefixed strings. HLA strings enjoy the advantages of both zero-terminated and length-prefixed strings without the disadvantages. In fact, the only drawback to HLA strings over these other formats is that HLA strings consume a few additional bytes (the overhead for an HLA string is nine bytes compared to one byte for zero-terminated or length-prefixed strings; the overhead being the number of bytes needed above and beyond the actual characters in the string).

An HLA string value consists of four components. The first element is a double word value that specifies the maximum number of characters that the string can hold. The second element is a double word value specifying the current length of the string. The third component is the sequence of characters in the string. The final component is a zero terminating byte. You could create an HLA-compatible string in the STATIC section using the following code[1]:

```
static
       dword 11;
       dword 11;
   TheString: char; @nostorage;
       byte "Hello there";
       byte 0;
```

1. Actually, there are some restrictions on the placement of HLA strings in memory. This text will not cover those issues. See the HLA documentation for more details.

Note that the address associated with the HLA string is the address of the first character, not the maximum or current length values.

"So what is the difference between the current and maximum string lengths?" you're probably wondering. Well, in a fixed string like the above they are usually the same. However, when you allocate storage for a string variable at run-time, you will normally specify the maximum number of characters that can go into the string. When you store actual string data into the string, the number of characters you store must be less than or equal to this maximum value. The HLA Standard Library string routines will raise an exception if you attempt to exceed this maximum length (something the C/C++ and Pascal formats can't do).

The terminating zero byte at the end of the HLA string lets you treat an HLA string as a zero-terminated string if it is more efficient or more convenient to do so. For example, most calls to Windows and Linux require zero-terminated strings for their string parameters. Placing a zero at the end of an HLA string ensures compatibility with Windows, Linux, and other library modules that use zero-terminated strings.

2.4 HLA Strings

As noted in the previous section, HLA strings consist of four components: a maximum length, a current string length, character data, and a zero terminating byte. However, HLA never requires you to create string data by manually emitting these components yourself. HLA is smart enough to automatically construct this data for you whenever it sees a string literal constant. So if you use a string constant like the following, understand that somewhere HLA is creating the four-component string in memory for you:

```
stdout.put( "This gets converted to a four-component string by HLA" );
```

HLA doesn't actually work directly with the string data described in the previous section. Instead, when HLA sees a string object it always works with a *pointer* to that object rather than the object directly. Without question, this is the most important fact to know about HLA strings, and is the biggest source of problems beginning HLA programmers have with strings in HLA: *strings are pointers!* A string variable consumes exactly four bytes, the same as a pointer (because it *is* a pointer!). Having said all that, let's take a look at a simple string variable declaration in HLA:

```
static
      StrVariable:  string;
```

Since a string variable is a pointer, you must initialize it before you can use it. There are three general ways you may initialize a string variable with a legal string address: using static initializers, using the *stralloc* routine, or calling some other HLA Standard Library that initializes a string or returns a pointer to a string.

In one of the static declaration sections that allow initialized variables (STATIC, and READONLY) you can initialize a string variable using the standard initialization syntax, e.g.,

```
static
    InitializedString: string := "This is my string";
```

Note that this does not initialize the string variable with the string data. Instead, HLA creates the string data structure (see the previous section) in a special, hidden, memory segment and initializes the *InitializedString* variable with the address of the first character in this string (the "T" in "This"). *Remember, strings are pointers!* The HLA compiler places the actual string data in a read-only memory segment. Therefore, you cannot modify the characters of this string literal at run-time. However, since the string variable (a pointer, remember) is in the static section, you can change the string variable so that it points at different string data.

Since string variables are pointers, you can load the value of a string variable into a 32-bit register. The pointer itself points at the first character position of the string. You can find the current string length in the double word four bytes prior to this address, you can find the maximum string length in the double word eight bytes prior to this address. The following program demonstrates one way to access this data[2].


```
// Program to demonstrate accessing Length and Maxlength fields of a string.

program StrDemo;
#include( "stdlib.hhf" );

static
    theString:string := "String of length 19";

begin StrDemo;

    mov( theString, ebx );  // Get pointer to the string.

    mov( [ebx-4], eax );    // Get current length
    mov( [ebx-8], ecx );    // Get maximum length

    stdout.put
    (
        "theString = '", theString, "'", nl,
        "length( theString )= ", (type uns32 eax ), nl,
        "maxLength( theString )= ", (type uns32 ecx ), nl
    );

end StrDemo;
```

Program 2.1 Accessing the Length and Maximum Length Fields of a String

When accessing the various fields of a string variable it is not wise to access them using fixed numeric offsets as done in this example. In the future, the definition of an HLA string may change slightly. In particular, the offsets to the maximum length and length fields are subject to change. A safer way to access string data is to coerce your string pointer using the *str.strRec* data type. The *str.strRec* data type is a record data type (see "Records, Unions, and Name Spaces" on page 483) that defines symbolic names for the offsets of the length and maximum length fields in the string data type. Were the offsets to the length and maximum length fields to change in a future version of HLA, then the definitions in *str.strRec* would also change, so if you use *str.strRec* then recompiling your program would automatically make any necessary changes to your program.

To use the *str.strRec* data type properly, you must first load the string pointer into a 32-bit register, e.g., "MOV(SomeString, EBX);" Once the pointer to the string data is in a register, you can coerce that register to the *str.strRec* data type using the HLA construct "(type str.strRec [EBX])". Finally, to access the length or maximum length fields, you would use either "(type str.strRec [EBX]).length" or "(type str.strRec [EBX]).MaxStrLen" (respectively). Although there is a little more typing involved (versus using simple offsets like "-4" or "-8"), these forms are far more descriptive and much safer than straight numeric offsets. The following program corrects the previous example by using the *str.strRec* data type.

```
// Program to demonstrate accessing Length and Maxlength fields of a string.

program LenMaxlenDemo;
#include( "stdlib.hhf" );

static
```

2. Note that this scheme is not recommended. If you need to extract the length information from a string, use the routines provided in the HLA string library for this purpose.


```
    theString:string := "String of length 19";

begin LenMaxlenDemo;

    mov( theString, ebx );  // Get pointer to the string.

    mov( (type str.strRec [ebx]).length, eax );     // Get current length
    mov( (type str.strRec [ebx]).MaxStrLen, ecx );  // Get maximum length

    stdout.put
    (
        "theString = '", theString, "'", nl,
        "length( theString )= ", (type uns32 eax ), nl,
        "maxLength( theString )= ", (type uns32 ecx ), nl
    );

end LenMaxlenDemo;
```

Program 2.2 Correct Way to Access Length and MaxStrLen Fields of a String

A second way to manipulate strings in HLA is to allocate storage on the heap to hold string data. Because strings can't directly use pointers returned by *malloc* (since strings need to access eight bytes prior to the pointer address), you shouldn't use *malloc* to allocate storage for string data. Fortunately, the HLA Standard Library memory module provides a memory allocation routine specifically designed to allocate storage for strings: *stralloc*. Like *malloc*, *stralloc* expects a single dword parameter. This value specifies the (maximum) number of characters needed in the string. The *stralloc* routine will allocate the specified number of bytes of memory, plus between nine and thirteen additional bytes to hold the extra string information[3].

The *stralloc* routine will allocate storage for a string, initialize the maximum length to the value passed as the *stralloc* parameter, initialize the current length to zero, and store a zero (terminating byte) in the first character position of the string. After all this, *stralloc* returns the address of the zero terminating byte (that is, the address of the first character element) in the EAX register.

Once you've allocated storage for a string, you can call various string manipulation routines in the HLA Standard Library to operate on the string. The next section will discuss the HLA string routines in detail; this section will introduce a couple of string related routines for the sake of example. The first such routine is the "stdin.gets(strvar)". This routine reads a string from the user and stores the string data into the string storage pointed at by the string parameter (*strvar* in this case). If the user attempts to enter more characters than you've allocated for the string, then stdin.gets raises the *ex.StringOverflow* exception. The following program demonstrates the use of *stralloc*.

```
// Program to demonstrate stralloc and stdin.gets.

program strallocDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strallocDemo;

    stralloc( 16 );         // Allocate storage for the string and store
```

3. *Stralloc* may allocate more than nine bytes for the overhead data because the memory allocated to an HLA string must always be double word aligned and the total length of the data structure must be an even multiple of four.


```
    mov( eax, theString );  //  the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text (16 chars, max): " );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

end strallocDemo;
```

Program 2.3 Reading a String from the User

If you look closely, you see a slight defect in the program above. It allocates storage for the string by calling *stralloc* but it never frees the storage allocated. Even though the program immediately exits after the last use of the string variable, and the operating system will deallocate the storage anyway, it's always a good idea to explicitly free up any storage you allocate. Doing so keeps you in the habit of freeing allocated storage (so you don't forget to do it when it's important) and, also, programs have a way of growing such that an innocent defect that doesn't affect anything in today's program becomes a show-stopping defect in tomorrow's version.

To free storage allocated via *stralloc*, you must call the corresponding *strfree* routine, passing the string pointer as the single parameter. The following program is a correction of the previous program with this minor defect corrected:

```
// Program to demonstrate stralloc, strfree, and stdin.gets.

program strfreeDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo;

    stralloc( 16 );         // Allocate storage for the string and store
    mov( eax, theString );  //  the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text (16 chars, max): " );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo;
```


Program 2.4 Corrected Program that Reads a String from the User

When looking at this corrected program, please take note that the *stdin.gets* routine expects you to pass it a string parameter that points at an allocated string object. Without question, one of the most common mistakes beginning HLA programmers make is to call *stdin.gets* and pass it a string variable that has not been initialized. This may be getting old now, but keep in mind that *strings are pointers!* Like pointers, if you do not initialize a string with a valid address, your program will probably crash when you attempt to manipulate that string object. The call to *stralloc* plus moving the returned result into *theString* is how the programs above initialize the string pointer. If you are going to use string variables in your programs, you must ensure that you allocate storage for the string data prior to writing data to the string object.

Allocating storage for a string option is such a common operation that many HLA Standard Library routines will automatically do the allocation to save you the effort. Generally, such routines have an "a_" prefix as part of their name. For example, the *stdin.a_gets* combines a call to *stralloc* and *stdin.gets* into the same routine. This routine, which doesn't have any parameters, reads a line of text from the user, allocates a string object to hold the input data, and then returns a pointer to the string in the EAX register. The following program is an adaptation of the previous two programs that uses *stdin.a_gets*:

```
// Program to demonstrate  strfree and stdin.a_gets.

program strfreeDemo2;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo2;


    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo2;
```

Program 2.5 Reading a String from the User with stdin.a_gets

Note that, as before, you must still free up the storage *stdin.a_gets* allocates by calling the *strfree* routine. One big difference between this routine and the previous two is the fact that HLA will automatically allocate exactly enough space for the string read from the user. In the previous programs, the call to *stralloc*

only allocates 16 bytes. If the user types more than this then the program raises an exception and quits. If the user types less than 16 characters, then some space at the end of the string is wasted. The *stdin.a_gets* routine, on the other hand, always allocates the minimum necessary space for the string read from the user. Since it allocates the storage, there is little chance of overflow[4].

2.5 Accessing the Characters Within a String

Extracting individual characters from a string is a very common and easy task. In fact, it is so easy that HLA doesn't provide any specific procedure or language syntax to accomplish this - it's easy enough just to use machine instructions to accomplish this. Once you have a pointer to the string data, a simple indexed addressing mode will do the rest of the work for you.

Of course, the most important thing to keep in mind is that *strings are pointers*. Therefore, you cannot apply an indexed addressing mode directly to a string variable an expect to extract characters from the string. I.e, if *s* is a string variable, then "MOV(s[ebx], al);" does not fetch the character at position EBX in string *s* and place it in the AL register. Remember, *s* is just a pointer variable, an addressing mode like *s[ebx]* will simply fetch the byte at offset EBX in memory starting at the address of *s* (see Figure 2.1).

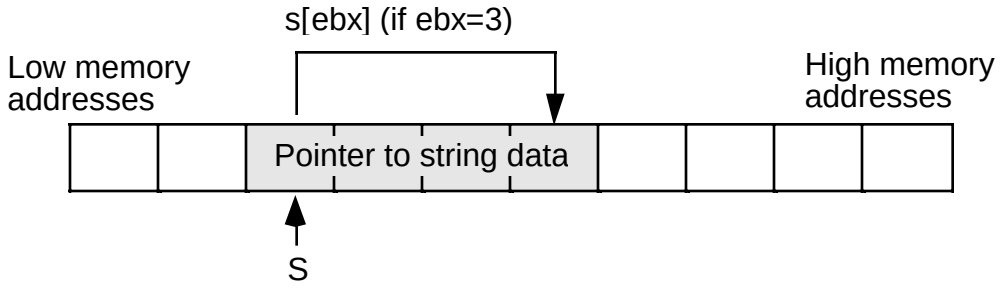

Figure 2.1 Incorrectly Indexing Off a String Variable

In Figure 2.1, assuming EBX contains three, "s[ebx]" does not access the fourth character in the string *s*, instead it fetches the fourth byte of the pointer to the string data. It is very unlikely that this is the desired effect you would want. Figure 2.2 shows the operation that is necessary to fetch a character from the string, assuming EBX contains the value of *s*:

4. Actually, there are limits on the maximum number of characters that stdin.a_gets will allocate. This is typically between 1,024 bytes and 4,096 bytes; See the HLA Standard Library source listings for the exact value.

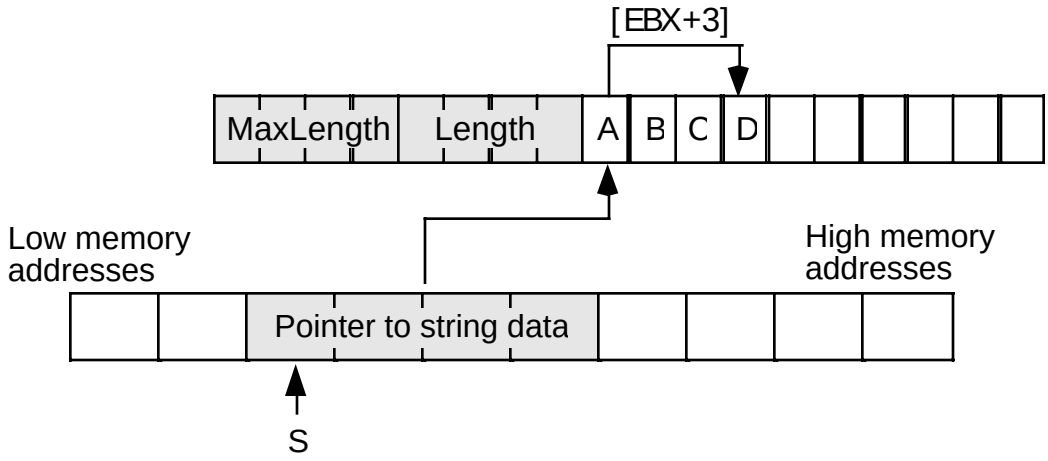

Figure 2.2 Correctly Indexing Off the Value of a String Variable

In Figure 2.2 EBX contains the value of string *s*. The value of *s* is a pointer to the actual string data in memory. Therefore, EBX will point at the first character of the string when you load the value of *s* into EBX. The following code demonstrates how to access the fourth character of string *s* in this fashion:

```
mov( s, ebx );          // Get pointer to string data into EBX.
mov( [ebx+3], al );     // Fetch the fourth character of the string.
```

If you want to load the character at a variable, rather than fixed, offset into the string, then you can use one of the 80x86's scaled indexed addressing modes to fetch the character. For example, if an *uns32* variable index contains the desired offset into the string, you could use the following code to access the character at *s[index]*:

```
mov( s, ebx );          // Get address of string data into EBX.
mov( index, ecx );      // Get desired offset into string.
mov( [ebx+ecx], al );   // Get the desired character into AL.
```

There is only one problem with the code above- it does not check to ensure that the character at offset *index* actually exists. If *index* is greater than the current length of the string, then this code will fetch a garbage byte from memory. Unless you can apriori determine that *index* is always less than the length of the string, code like this is dangerous to use. A better solution is to check the index against the string's current length before attempting to access the character. the following code provides one way to do this.

```
mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).Length ) then

    mov( [ebx+ecx], al );

else

    << error, string index is of bounds >>

endif;
```

In the ELSE portion of this IF statement you could take corrective action, print an error message, or raise an exception. If you want to explicitly raise an exception, you can use the HLA RAISE statement to accomplish this. The syntax for the RAISE statement is

```
raise( integer_constant );
```



```
raise( reg32 );
```

The value of the *integer_constant* or 32-bit register must be an exception number. Usually, this is one of the predefined constants in the excepts.hhf header file. An appropriate exception to raise when a string index is greater than the length of the string is *ex.StringIndexError*. The following code demonstrates raising this exception if the string index is out of bounds:

```
mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).Length ) then

    mov( [ebx+ecx], al );

else

    raise( ex.StringIndexError );

endif;
```

2.6 The HLA String Module and Other String-Related Routines

Although HLA provides a powerful definition for string data, the real power behind HLA's string capabilities lies in the HLA Standard Library, not in the definition of HLA string data. HLA provides several dozen string manipulation routines that far exceed the capabilities found in standard HLLs like C/C++, Java, or Pascal; indeed, HLA's string handling capabilities rival those in string processing languages like Icon or SNOBOL4. While it is premature to introduce all of HLA's character string handling routines, this chapter will discuss many of the string facilities that HLA provides.

Perhaps the most basic string operation you will need is to assign one string to another. There are three different ways to assign strings in HLA: by reference, by copying a string, and by duplicating a string. Of these, assignment by reference is the fastest and easiest. If you have two strings and you wish to assign one string to the other, a simple and fast way to do this is to copy the string pointer. The following code fragment demonstrates this:

```
static
        string1:   string     := "Some String Data";
        string2:   string;
            .
            .
            .
        mov( string1, eax );
        mov( eax, string2 );
            .
            .
            .
```

String assignment by reference is very efficient because it only involves two simple MOV instructions, regardless of the actual length of the string. Assignment by reference works great if you never modify the string data after the assignment operation. Do keep in mind, though, that both string variables (string1 and string2 in the example above) *wind up pointing at the same data*. So if you make a change to the data pointed at by one string variable, you will change the string data pointed at by the second string object since both objects point at the same data. The following program demonstrates this problem:

```
// Program to demonstrate the problem
```



```
// with string assignment by reference.

program strRefAssignDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin strRefAssignDemo;

    // Get a value into string1

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars." nl );

    endfor;

    stdout.put( "You entered: '", string1, "'" nl );

    // Do the string assignment by copying the pointer

    mov( string1, ebx );
    mov( ebx, string2 );

    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we've got at least
    // three characters here).

    mov( 'a', (type char [ebx]) );
    mov( 'b', (type char [ebx+1]) );
    mov( 'c', (type char [ebx+2]) );

    // Okay, demonstrate the problem with assignment via
    // pointer copy.

    stdout.put
    (
        "After assigning 'abc' to the first three characters in string1:"
        nl
        nl
    );
    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    strfree( string1 );     // Don't free string2 as well!


end strRefAssignDemo;
```


Program 2.6 Problem with String Assignment by Copying Pointers

Since both *string1* and *string2* point at the same string data in this example, any change you make to one string is reflected in the other. While this is sometimes acceptable, most programmers expect assignment to produce a different copy of a string; they expect the semantics of string assignment to produce two unique copies of the string data.

An important point to remember when using *copy by reference* (this term means copying a pointer rather than copying the actual data) is that you have created an *alias* to the string data. The term "alias" means that you have two names for the same object in memory (e.g., in the program above, *string1* and *string2* are two different names for the same string data). When you read a program it is reasonable to expect that different variables refer to different memory objects. Aliases violate this rule, thus making your program harder to read and understand because you've got to remember that aliases do not refer to different objects in memory. Failing to keep this in mind can lead to subtle bugs in your program. For instance, in the example above you have to remember that *string1* and *string2* are aliases so as not to free both objects at the end of the program. Worse still, you to remember that *string1* and *string2* are aliases so that you don't continue to use *string2* after freeing *string1* in this code since *string2* would be a dangling reference at that point.

Since using copy by reference makes your programs harder to read and increases the possibility that you might introduce subtle defects in your programs, you might wonder why someone would use copy by reference at all. There are two reasons for this: first, copy by reference is very efficient; it only involves the execution of two MOV instructions. Second, some algorithms actually depend on copy by reference semantics. Nevertheless, you should carefully consider whether copying string pointers is the appropriate way to do a string assignment in your program before using this technique.

The second way to assign one string to another is to actually copy the string data. The HLA Standard Library *str.cpy* routine provides this capability. A call to the *str.cpy* procedure using the following form:

```
str.cpy( source_string, destination_string );
```

The source and destination strings must be string variables (pointers) or 32-bit registers containing the addresses of the string data in memory.

The *str.cpy* routine first checks the maximum length field of the destination string to ensure that it is at least as big as the current length of the source string. If it is not, then *str.cpy* raises the *ex.StringOverflow* exception. If the maximum string length field of the destination string is at least as big as the current string length of the source string, then *str.cpy* copies the string length, the characters, and the zero terminating byte from the source string to the data area at which the destination string points. When this process is complete, the two strings point at identical data, but they do not point at the same data in memory[5]. The following program is a rework of the previous example using *str.cpy* rather than copy by reference.

```
// Program to demonstrate string assignment using str.cpy.

program strcpyDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin strcpyDemo;
```

5. Unless, of course, both string pointers contained the same address to begin with, in which case str.cpy copies the string data over the top of itself.


```
    // Allocate storage for string2:

    stralloc( 64 );
    mov( eax, string2 );

    // Get a value into string1

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars." nl );

    endfor;


    // Do the string assignment via str.cpy

    str.cpy( string1, string2 );

    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we've got at least
    // three characters here).

    mov( string1, ebx );
    mov( 'a', (type char [ebx]) );
    mov( 'b', (type char [ebx+1]) );
    mov( 'c', (type char [ebx+2]) );

    // Okay, demonstrate that we have two different strings
    // since we used str.cpy to copy the data:

    stdout.put
    (
        "After assigning 'abc' to the first three characters in string1:"
        nl
        nl
    );
    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );


    // Note that we have to free the data associated with both
    // strings since they are not aliases of one another.

    strfree( string1 );
    strfree( string2 );


end strcpyDemo;
```


Program 2.7 Copying Strings using str.cpy

There are two really important things to note about this program. First, note that this program begins by allocating storage for *string2*. Remember, the *str.cpy* routine does not allocate storage for the destination string, it assumes that the destination string already has storage allocated to it. Keep in mind that *str.cpy* does not initialize *string2*, it only copies data to the location where *string2* is pointing. It is the program's responsibility to initialize the string by allocating sufficient memory before calling *str.cpy*. The second thing to notice here is that the program calls *strfree* to free up the storage for both *string1* and *string2* before the program quits.

Allocating storage for a string variable prior to calling *str.cpy* is so common that the HLA Standard Library provides a routine that allocates and copies the string: *str.a_cpy*. This routine uses the following call syntax:

```
str.a_cpy( source_string );
```

Note that there is no destination string. This routine looks at the length of the source string, allocates sufficient storage, makes a copy of the string, and then returns a pointer to the new string in the EAX register. The following program demonstrates the current example using the *str.a_cpy* procedure.

```
// Program to demonstrate string assignment using str.a_cpy.

program stra_cpyDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin stra_cpyDemo;


    // Get a value into string1

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars." nl );

    endfor;


    // Do the string assignment via str.a_cpy

    str.a_cpy( string1 );
    mov( eax, string2 );

    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
```



```
    // a string pointer always points at the first character
    // position in the string and we know we've got at least
    // three characters here).

    mov( string1, ebx );
    mov( 'a', (type char [ebx]) );
    mov( 'b', (type char [ebx+1]) );
    mov( 'c', (type char [ebx+2]) );

    // Okay, demonstrate that we have two different strings
    // since we used str.cpy to copy the data:

    stdout.put
    (
        "After assigning 'abc' to the first three characters in string1:"
        nl
        nl
    );
    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );


    // Note that we have to free the data associated with both
    // strings since they are not aliases of one another.

    strfree( string1 );
    strfree( string2 );


end stra_cpyDemo;
```

Program 2.8 Copying Strings using str.a_cpy

Warning: Whenever using copy by reference or *str.a_cpy* to assign a string, don't forget to free the storage associated with the string when you are (completely) done with that string's data. Failure to do so may produce a memory leak if you do not have another pointer to the previous string data laying around.

Obtaining the length of a character string is such a common need that the HLA Standard Library provides a *str.length* routine specifically for this purpose. Of course, you can fetch the length by using the *str.strRec* data type to access the length field directly, but constant use of this mechanism can be tiring since it involves a lot of typing. The *str.length* routine provides a more compact and convenient way to fetch the length information. You call *str.length* using one of the following two formats:

```
str.length( Reg32 );
str.length( string_variable );
```

This routine returns the current string length in the EAX register.

Another pair of useful string routines are the *str.cat* and *str.a_cat* procedures. They use the following calling sequence:

```
str.cat( srcStr, destStr );
str.a_cat( src1Str, src2Str );
```

These two routines concatenate two strings (that is, they create a new string by joining the two strings together). The *str.cat* procedure concatenates the source string to the end of the destination string. Before

the concatenation actually takes place, *str.cat* checks to make sure that the destination string is large enough to hold the concatenated result, it raises the *ex.StringOverflow* exception if the destination string is too small.

The *str.a_cat*, as its name suggests, allocates storage for the resulting string before doing the concatenation. This routine will allocate sufficient storage to hold the concatenated result, then it will copy the *src1Str* to the allocated storage, finally it will append the string data pointed at by *src2Str* to the end of this new string and return a pointer to the new string in the EAX register.

Warning: note a potential source of confusion. The *str.cat* procedure concatenates its first operand to the end of the second operand. Therefore, *str.cat* follows the standard (src, dest) operand format present in many HLA statements. The *str.a_cat* routine, on the other hand, has two source operands rather than a source and destination operand. The *str.a_cat* routine concatenates its two operands in an intuitive left-to-right fashion. This is the opposite of *str.cat*. Keep this in mind when using these two routines.

The following program demonstrates the use of the *str.cat* and *str.a_cat* routines:

```
// Program to demonstrate str.cat and str.a_cat.

program strcatDemo;
#include( "stdlib.hhf" );

static
    UserName:   string;
    Hello:      string;
    a_Hello:    string;

begin strcatDemo;

    // Allocate storage for the concatenated result:

    stralloc( 1024 );
    mov( eax, Hello );

    // Get some user input to use in this example:

    stdout.put( "Enter your name: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, UserName );

    // Use str.cat to combine the two strings:

    str.cpy( "Hello ", Hello );
    str.cat( UserName, Hello );

    // Use str.a_cat to combine the string strings:

    str.a_cat( "Hello ", UserName );
    mov( eax, a_Hello );

    stdout.put( "Concatenated string #1 is '", Hello, "'" nl );
    stdout.put( "Concatenated string #2 is '", a_Hello, "'" nl );

    strfree( UserName );
    strfree( a_Hello );
    strfree( Hello );

end strcatDemo;
```


Program 2.9 Demonstration of str.cat and str.a_cat Routines

The *str.insert* and *str.a_insert* routines are closely related to the string concatenation procedures. However, the *str.insert* and *str.a_insert* routines let you insert one string anywhere into another string, not just at the end of the string. The calling sequences for these two routines are

```
str.insert( src, dest, index );
str.a_insert( StrToInsert, StrToInsertInto, index );
```

These two routines insert the source string (*src* or *StrToInsert*) into the destination string (*dest* or *StrToInsertInto*) starting at character position *index*. The *str.insert* routine inserts the source string directly into the destination string; if the destination string is not large enough to hold both strings, *str.insert* raises an *ex.StringOverflow* exception. The s*tr.a_insert* routine first allocates a new string on the heap, copies the destination string (*StrToInsertInto*) to the new string, and then inserts the source string (*StrToInsert*) into this new string at the specified offset; *str.a_insert* returns a pointer to the new string in the EAX register.

Indexes into a string are zero-based. This means that if you supply the value zero as the index in *str.insert* or *str.a_insert*, then these routines will insert the source string before the first character of the destination string. Likewise, if the index is equal to the length of the string, then these routines will simply concatenate the source string to the end of the destination string. Note: if the index is greater than the length of the string, the *str.insert* and *str.a_insert* procedures will not raise an exception; instead, they will simply append the source string to the end of the destination string.

The *str.delete* and *str.a_delete* routines let you remove characters from a string. They use the following calling sequence:

```
 str.delete( str, StartIndex, Length );
str.a_delete( str, StartIndex, Length );
```

Both routines delete *Length* characters starting at character position *StartIndex* in string *str*. The difference between the two is that *str.delete* deletes the characters directly from *str* whereas *str.a_delete* first allocates storage and copies *str*, then deletes the characters from the new string (leaving *str* untouched). The *str.a_delete* routine returns a pointer to the new string in the EAX register.

The *str.delete* and *str.a_delete* routines are very forgiving with respect to the values you pass in *StartIndex* and *Length*. If *StartIndex* is greater than the current length of the string, these routines do not delete any characters from the string. If *StartIndex* is less than the current length of the string, but *StartIndex+Length* is greater than the length of the string, then these routines will delete all characters from *StartIndex* to the end of the string.

Another very common string operation is the need to copy a portion of a string to a different string without otherwise affecting the source string. The *str.substr* and *str.a_substr* routines provide this capability. These routines use the following calling sequence:

```
str.substr( src, dest, StartIndex, Length );
  str.a_substr( src, StartIndex, Length );
```

The *str.substr* routine copies length characters, starting at position *StartIndex*, from the *src* string to the *dest* string. The *dest* string must have sufficient storage allocated to hold the new string or *str.substr* will raise an *ex.StringOverflow* exception. If the *StartIndex* value is greater than the length of the string, then *str.substr* will raise an *ex.StringIndexError* exception. If *StartIndex+Length* is greater than the length of the source string, but *StartIndex* is less than the length of the string, then *str.substr* will extract only those characters from *StartIndex* to the end of the string.

The *str.a_substr* procedure behaves in a fashion nearly identical to *str.substr* except it allocates storage on the heap for the destination string. Other than overflow never occurs, *str.a_substr* handles exceptions the identically to *str.substr*[6]. As you can probably guess by now, *str.a_substr* returns a pointer to the newly allocated string in the EAX register.

After you begin working with string data for a little while, the need will invariably arise to compare two strings. A first attempt at string comparison, using the standard HLA relational operators, will compile but not necessarily produce the desired results:

```
mov( s1, eax );
if( eax = s2 ) then

    << code to execute if the strings are equal >>

else

    << code to execute if the strings are not equal >>

endif;
```

As stated above, this code will compile and execute just fine. However, it's probably not doing what you expect it to do. Remember *strings are pointers*. This code compares the two pointers to see if they are equal. If they are equal, clearly the two strings are equal (since both *s1* and *s2* point at the exact same string data). However, the fact that the two pointers are different doesn't necessarily mean that the strings are not equivalent. Both *s1* and *s2* could contain different values (that is, they point at different addresses in memory) yet the string data at those two different addresses could be identical. Most programmers expect a string comparison for equality to be true if the data for the two strings is the same. Clearly a pointer comparison does not provide this type of comparison. To overcome this problem, the HLA Standard Library provides a set of string comparison routines that will compare the string data, not just their pointers. These routines use the following calling sequences:

```
str.eq( src1, src2 );
str.ne( src1, src2 );
str.lt( src1, src2 );
str.le( src1, src2 );
str.gt( src1, src2 );
str.ge( src1, src2 );
```

Each of these routines compares the *src1* string to the *src2* string and return true (1) or false (0) in the EAX register depending on the comparison. For example, "str.eq(s1, s2);" returns true in EAX if *s1* is equal to *s2*. HLA provides a small extension that allows you to use the string comparison routines within an IF statement[7]. The following code demonstrates the use of some of these comparison routines within an IF statement:

```
stdout.put( "Enter a single word: " );
stdin.a_gets();
if( str.eq( eax, "Hello" )) then

    stdout.put( "You entered 'Hello'", nl );

endif;
strfree( eax );
```

Note that the string the user enters in this example must exactly match "Hello", including the use of an upper case "H" at the beginning of the string. When processing user input, it is best to ignore alphabetic case in string comparisons because different users have different ideas about when they should be pressing the shift key on the keyboard. An easy solution is to use the HLA case insensitive string comparison functions. These routines compare two strings ignoring any differences in alphabetic case. These routines use the following calling sequences:

```
str.ieq( src1, src2 );
```

6. Technically, *str.a_substr*, like all routines that call *malloc* to allocate storage, can raise an *ex.MemoryAllocationFailure* exception, but this is very unlikely to occur.

7. This extension is actually a little more general than this section describes. A later chapter will explain it fully.


```
str.ine( src1, src2 );
str.ilt( src1, src2 );
str.ile( src1, src2 );
str.igt( src1, src2 );
str.ige( src1, src2 );
```

Other than they treat upper case characters the same as their lower case equivalents, these routines behave exactly like the former routines, returning true or false in EAX depending on the result of the comparison.

Like most high level languages, HLA compares strings using *lexicographical ordering*. This means that two strings are equal if and only if their lengths are the same and the corresponding characters in the two strings are exactly the same. For less than or greater than comparisons, lexicographical ordering corresponds to the way words appear in a dictionary. That is, "a" is less than "b" is less than "c" etc. Actually, HLA compares the strings using the ASCII numeric codes for the characters, so if you are unsure whether "a" is less than a period, simply consult the ASCII character chart (incidentally, "a" is greater than a period in the ASCII character set, just in case you were wondering).

If two strings have different lengths, lexicographical ordering only worries about the length if the two strings exactly match up through the length of the shorter string. If this is the case, then the longer string is greater than the shorter string (and, conversely, the shorter string is less than the longer string). Note, however, that if the characters in the two strings do not match at all, then HLA's string comparison routines ignore the length of the string; e.g., "z" is always greater than "aaaaa" even though it has a shorter length.

The *str.eq* routine checks to see if two strings are equal. Sometimes, however, you might want to know whether one string *contains* another string. For example, you may want to know if some string contains the substring "north" or "south" to determine some action to take in a game. The HLA *str.index* routine lets you check to see if one string is contained as a substring of another. The *str.index* routine uses the following calling sequence:

```
str.index( StrToSearch, SubstrToSearchFor );
```

This function returns, in EAX, the offset into *StrToSearch* where *SubstrToSearchFor* appears. This routine returns -1 in EAX if *SubstrToSearchFor* is not present in *StrToSearch*. Note that str.index will do a case sensitive search. Therefore the strings must exactly match. There is no case insensitive variant of str.index you can use[8].

The HLA strings module contains many additional routines besides those this section presents. Space limitations and prerequisite knowledge prevent the presentation of all the string functions here; however, this does not mean that the remaining string functions are unimportant. You should definitely take a look at the HLA Standard Library documentation to learn everything you can about the powerful HLA string library routines. The chapters on advanced string handling contain more information on HLA string and pattern matching routines.

2.7 In-Memory Conversions

The HLA Standard Library's string module contains dozens of routines for converting between strings and other data formats. Although it's a little premature in this text to present a complete description of those functions, it would be rather criminal not to discuss at least one of the available functions: the *str.put* routine. This one routine (which is actually a macro) encapsulates the capabilities of all the other string conversion functions, so if you learn how to use this one, you'll have most of the capabilities of those other routines at your disposal. For more information on the other string conversions, see the chapters in the volume on Advanced String Handling.

8. However, HLA does provide routines that will convert all the characters in a string to one case or another. So you can make copies of the strings, convert all the characters in both copies to lower case, and then search using these converted strings. This will achieve the same result.

You use the *str.put* routine in a manner very similar to the *stdout.put* routine. The only difference is that the *str.put* routine "writes" its data to a string instead of the standard output device. A call to *str.put* has the following syntax:

```
str.put( destString, values_to_convert );
```

Example of a call to *str.put*:

```
str.put( destString, "I =", i:4, " J= ", j, " s=", s );
```

Note: generally you would not put a newline character seqeuence at the end of the string as you would if you were printing the string to the standard output device.

The *destString* parameter at the beginning of the *str.put* parameter list must be a string variable and it must already have storage associated with it. If *str.put* attempts to store more characters than allowed into the *destString* parameter, then this function raises the *ex.StringOverflow* exception.

Most of the time you won't know the length of the string that *str.put* will produce. In those instances, you should simply allocate sufficient storage for a really large string, one that is way larger than you expect, and use this string data as the first parameter of the *str.put* call. This will prevent an exception from crashing your program. Generally, if you expect to produce about one screen line of text, you should probably allocate at least 256 characters for the destination string. If you're creating longer strings, you should probably use a default of 1024 characters (or more, if you're going to produce *really* large strings).

Example:

```
static
    s: string;
        .
        .
        .
    mov( stralloc( 256 ), s );
        .
        .
        .
    str.put( s, "R: ", r:16:4, " strval: '", strval:-10, "'" );
```

You can use the *str.put* routine to convert any data to a string that you can print using *stdout.put*. You will probably find this routine invaluable for common value-to-string conversions.

At the time this is being written, there is no corresponding *str.get* routine that will read values from an input string (this routine will probably appear in a future version of the HLA Standard Library, so watch out for it). In the meantime, the HLA strings and conversions modules in the Standard Library do provide lots of stand-alone conversion functions you can use to convert string data to some other format. See the volume on "Advanced String Handling" for more details about these routines.

2.8 Putting It All Together

There are many different ways to represent character strings. This chapter began by discussing how the C/C++ and Pascal languages represent strings using zero-terminated and length prefixed strings. HLA uses a hybrid representation for its string. HLA strings consist of a pointer to a zero terminated sequence of character with a pair of prefix length values. HLA's format offers all the advantages of the other two forms with the slight disadvantage of a few extra bytes of overhead.

After discussing string formats, this chapter discussed how to operate on string data. In addition to accessing the characters in a string directly (which is easy, you just index off the pointer to the string data), this chapter described how to manipulate strings using several routines from the HLA Standard Library. This chapter provides a very basic introduction to string handling in HLA. To learn more about string manipulation in assembly language (and the use of the routines in the HLA Standard Library), see the separate volume on "Advanced String Handling" in this text.

