Constants, Variables, and Data Types Chapter One

Volume One discussed the basic format for data in memory. Volume Two covered how a computer system physically organizes that data. This chapter finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with three main topics: constants, variables and data structures. This chapter does not assume that you've had a formal course in data structures, though such experience would be useful.

1.1 Chapter Overview

This chapter discusses how to declare and use constants, scalar variables, integers, reals, data types, pointers, arrays, and structures. You must master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seems to present a multitude of problems to beginning assembly language programmers. However, the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you will pick it up as you need it later. You will need it right away and trying to learn this material along with later material will only confuse you more.

1.2 Some Additional Instructions: INTMUL, BOUND, INTO

This chapter introduces arrays and other concepts that will require the expansion of your 80x86 instruction set knowledge. In particular, you will need to learn how to multiply two values; hence the first instruction we will look at is the intmul (integer multiply) instruction. Another common task when accessing arrays is to check to see if an array index is within bounds. The 80x86 bound instruction provides a convenient way to check a register's value to see if it is within some range. Finally, the into (interrupt on overflow) instruction provides a quick check for signed arithmetic overflow. Although into isn't really necessary for array (or other data type access), its function is very similar to bound, hence the presentation at this point.

The intmul instruction takes one of the following forms:

```
// The following compute destreg = destreg * constant

intmul( constant, destreg16 );
intmul( constant, destreg32 );

// The following compute dest = src * constant

intmul( constant, srcreg16, destreg16 );
intmul( constant, srcmem16, destreg16 );

intmul( constant, srcreg32, destreg32 );
intmul( constant, srcmem32, destreg32 );

// The following compute dest = dest * src

intmul( srcreg16, destreg16 );
intmul( srcmem16, destreg16 );
intmul( srcreg32, destreg32 );
intmul( srcmem32, destreg32 );
```

Note that the syntax of the intmul instruction is different than the add and sub instructions. In particular, note that the destination operand must be a register (add and sub both allow a memory operand as a destination). Also note that intmul allows three operands when the first operand is a constant. Another important difference is that the intmul instruction only allows 16-bit and 32-bit operands; it does not allow eight-bit operands.

intmul computes the product of its specified operands and stores the result into the destination register. If an overflow occurs (which is always a signed overflow, since intmul only multiplies signed integer values), then this instruction sets both the carry and overflow flags. intmul leaves the other condition code flags undefined (so, for example, you cannot check the sign flag or the zero flag after intmul and expect them to tell you anything about the intmul operation).

The bound instruction checks a 16-bit or 32-bit register to see if it is between one of two values. If the value is outside this range, the program raises an exception and aborts. This instruction is particularly useful for checking to see if an array index is within a given range. The bound instruction takes one of the following forms:

bound(*reg_{16}, LBconstant, UBconstant*);
bound(*reg_{32}, LBconstant, UBconstant*);

bound(*reg_{16}, Mem_{16}[2]*);[1]
bound(*reg_{32}, Mem_{32}[2]*);[2]

The bound instruction compares its register operand against an unsigned lower bound value and an unsigned upper bound value to ensure that the register is in the range:

```
lower_bound <= register <= upper_bound
```

The form of the bound instruction with three operands compares the register against the second and third parameters (the lower bound and upper bound, respectively)[3]. The bound instruction with two operands checks the register against one of the following ranges:

Mem_{16}[0] <= $register_{16}$ <= Mem_{16}[2]
Mem_{32}[0] <= $register_{32}$ <= Mem_{32}[4]

If the specified register is not within the given range, then the 80x86 raises an exception. You can trap this exception using the HLA try..endtry exception handling statement. The excepts.hhf header file defines an exception, *ex.BoundInstr*, specifically for this purpose. The following code fragment demonstrates how to use the bound instruction to check some user input:

```
program BoundDemo;
#include( “stdlib.hhf” );

static
    InputValue:int32;
    GoodInput:boolean;

begin BoundDemo;

    // Repeat until the user enters a good value:

    repeat

        // Assume the user enters a bad value.

        mov( false, GoodInput );
```

1. The “[2]” suggests that this variable must be an array of two consecutive word values in memory.
2. Likewise, this memory operand must be two consecutive dwords in memory.
3. This form isn’t a true 80x86 instruction. HLA converts this form of the bound instruction to the two operand form by creating two readonly memory variables initialized with the specified constant.

```
        // Catch bad numeric input via the try..endtry statement.

        try

            stdout.put( “Enter an integer between 1 and 10: “ );
            stdin.flushInput();
            stdin.geti32();

            mov( eax, InputValue );

            // Use the BOUND instruction to verify that the
            // value is in the range 1..10.

            bound( eax, 1, 10 );

            // If we get to this point, the value was in the
            // range 1..10, so set the boolean “GoodInput”
            // flag to true so we can exit the loop.

            mov( true, GoodInput );


            // Handle inputs that are not legal integers.

          exception( ex.ConversionError )

            stdout.put( “Illegal numeric format, reenter”, nl );


            // Handle integer inputs that don’t fit into an int32.

          exception( ex.ValueOutOfRange )

            stdout.put( “Value is *way* too big, reenter”, nl );


            // Handle values outside the range 1..10 (BOUND instruction)

          /*
          exception( ex.BoundInstr )

            stdout.put
            (
                “Value was “,
                InputValue,
                “, it must be between 1 and 10, reenter”,
                nl
            );
          */

        endtry;

    until( GoodInput );
    stdout.put( “The value you entered, “, InputValue, “ is valid.”, nl );

end BoundDemo;
```

Program 1.1   Demonstration of the BOUND Instruction

The into instruction, like bound, also generates an exception under certain conditions. Specifically, into generates an exception if the overflow flag is set. Normally, you would use into immediately after a signed arithmetic operation (e.g., intmul) to see if an overflow occurs. If the overflow flag is not set, the system ignores the into instruction; however, if the overflow flag is set, then the into instruction raises the HLA *ex.IntoInstr* exception. The following code sample demonstrates the use of the into instruction:

```
program INTOdemo;
#include( “stdlib.hhf” );

static
    LOperand:int8;
    ResultOp:int8;

begin INTOdemo;

    // The following try..endtry checks for bad numeric
    // input and handles the integer overflow check:

    try

        // Get the first of two operands:

        stdout.put( “Enter a small integer value (-128..+127):” );
        stdin.geti8();
        mov( al, LOperand );

        // Get the second operand:

        stdout.put( “Enter a second small integer value (-128..+127):” );
        stdin.geti8();

        // Produce their sum and check for overflow:

        add( LOperand, al );
        into();

        // Display the sum:

        stdout.put( “The eight-bit sum is “, (type int8 al), nl );


        // Handle bad input here:

      exception( ex.ConversionError )

        stdout.put( “You entered illegal characters in the number”, nl );


        // Handle values that don’t fit in a byte here:

      exception( ex.ValueOutOfRange )

        stdout.put( “The value must be in the range -128..+127”, nl );


        // Handle integer overflow here:

      /*
      exception( ex.IntoInstr )
```

```
            stdout.put
            (
                “The sum of the two values is outside the range -128..+127”,
                nl
            );
          */

        endtry;

    end INTOdemo;
```

Program 1.2 Demonstration of the INTO Instruction

1.3 The QWORD and TBYTE Data Types

HLA lets you declare eight-byte and ten-byte variables using the *qword*, and *tbyte* data types, respectively. Since HLA does not allow the use of 64-bit or 80-bit non-floating point constants, you may not associate an initializer with these two data types. However, if you wish to reserve storage for a 64-bit or 80-bit variable, you may use these two data types to do so.

The *qword* type lets you declare *quadword* (eight byte) variables. Generally, *qword* variables will hold 64-bit integer or unsigned integer values, although HLA and the 80x86 certainly don’t enforce this. The HLA Standard Library contains several routines to let you input and display 64-bit signed and unsigned integer values. The chapter on advanced arithmetic will discuss how to calculate 64-bit results on the 80x86 if you need integers of this size.

The *tbyte* directive allocates ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. Since you would normally use the *real80* data type for floating point values, about the only purpose of tbyte in HLA is to reserve storage for a 10-byte BCD value (or other data type that needs 80 bits). Once again, the chapter on advanced arithmetic may provide some insight into the use of this data type. However, except for very advanced applications, you could probably ignore this data type and not suffer.

1.4 HLA Constant and Value Declarations

HLA’s CONST and VAL sections let you declare symbolic constants. The CONST section lets you declare identifiers whose value is constant throughout compilation and run-time; the VAL section lets you declare symbolic constants whose value can change at compile time, but whose values are constant at run-time (that is, the same name can have a different value at several points in the source code, but the value of a VAL symbol at a given point in the program cannot change while the program is running).

The CONST section appears in the same declaration section of your program that contains the STATIC, READONLY, STORAGE, and VAR, sections. It begins with the CONST reserved word and has a syntax that is nearly identical to the READONLY section, that is, the CONST section contains a list of identifiers followed by a type and a constant expression. The following example will give you an idea of what the CONST section looks like:

```
const
    pi:           real32 := 3.14159;
    MaxIndex:     uns32  := 15;
    Delimiter:    char   := ‘/’;
    BitMask:      byte   := $F0;
    DebugActive:   boolean:= true;
```

Once you declare these constants in this manner, you may use the symbolic identifiers anywhere the corresponding literal constant is legal. These constants are known as *manifest constants*. A manifest constant is a symbolic representation of a constant that allows you to substitute the literal value for the symbol anywhere in the program. Contrast this with READONLY variables; a READONLY variable is certainly a constant value since you cannot change such a variable at run time. However, there is a memory location associated with READONLY variables and the operating system, not the HLA compiler, enforces the read-only attribute at run-time. Although it will certainly crash your program when it runs, it is perfectly legal to write an instruction like "MOV(EAX, ReadOnlyVar);" On the other hand, it is no more legal to write "MOV(EAX, MaxIndex);" (using the declaration above) than it is to write "MOV(EAX, 15);" In fact, both of these statements are equivalent since the compiler substitutes "15" for *MaxIndex* whenever it encounters this manifest constant.

If there is absolutely no ambiguity about a constant's type, then you may declare a constant by specifying only the name and the constant's value, omitting the type specification. In the example earlier, the *pi*, *Delimiter*, *MaxIndex*, and *DebugActive* constants could use the following declarations:

```
const
    pi              := 3.14159;       // Default type is real80.
    MaxIndex        := 15;            // Default type is uns32.
    Delimiter:      := '/';           // Default type is char.
    DebugActive:    := true;          // Default type is boolean.
```

Symbol constants that have an integer literal constant are always given the type *uns32* if the constant is zero or positive, or *int32* if the value is negative. This is why *MaxIndex* was okay in this CONST declaration but *BitMask* was not. Had we included the statement "BitMask := $F0;" in this latter CONST section, the declaration would have been legal but *BitMask* would be of type *uns32* rather than *byte*.

Constant declarations are great for defining "magic" numbers that might possibly change during program modification. The following provides an example of using constants to parameterize "magic" values in the program.

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

const
    MemToAllocate   := 4_000_000;
    NumDWords       := MemToAllocate div 4;
    MisalignBy      := 62;

    MainRepetitions := 1000;
    DataRepetitions := 999_900;

    CacheLineSize   := 16;

begin ConstDemo;

    //console.cls();
    stdout.put
    (
        "Memory Alignment Exercise",nl,
        nl,
        "Using a watch (preferably a stopwatch), time the execution of", nl
        "the following code to determine how many seconds it takes to", nl
        "execute.", nl
        nl
        "Press Enter to begin timing the code:"
    );
```

```
// Allocate enough dynamic memory to ensure that it does not
// all fit inside the cache.  Note: the machine had better have
// at least four megabytes free or virtual memory will kick in
// and invalidate the timing.

malloc( MemToAllocate );

// Zero out the memory (this loop really exists just to
// ensure that all memory is mapped in by the OS).

mov( NumDWords, ecx );
repeat

    dec( ecx );
    mov( 0, (type dword [eax+ecx*4]));

until( !ecx );  // Repeat until ECX = 0.


// Okay, wait for the user to press the Enter key.

stdin.readLn();

// Note: as processors get faster and faster, you may
// want to increase the size of the following constant.
// Execution time for this loop should be approximately
// 10-30 seconds.

mov( MainRepetitions, edx );
add( MisalignBy, eax );     // Force misalignment of data.

repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );
    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, “Stop timing and record time spent”, nl, nl );


// Okay, time the aligned access.

stdout.put
(
    “Press Enter again to begin timing access to aligned variable:”
);
stdin.readLn();
```

```
    // Note: if you change the constant above, be sure to change
    // this one, too!

    mov( MainRepetitions, edx );
    sub( MisalignBy, eax );     // Realign the data.
    repeat

        mov( DataRepetitions, ecx );
        align( CacheLineSize );
        repeat

            sub( 4, ecx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );
            mov( [eax+ecx*4], ebx );

        until( !ecx );
        dec( edx );

    until( !edx ); // Repeat until EAX is zero.

    stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );
    free( eax );


end ConstDemo;
```

Program 1.3 Data Alignment Program Rewritten Using CONST Definitions

1.4.1 Constant Types

Manifest constants can be any of the HLA primitive types plus a few of the composite types this chapter discusses. Volumes One and Two discussed most of the primitive types; these primitive types include the following:

- Boolean constants (true or false)
- Uns8 constants (0..255)
- Uns16 constants (0..65535)
- Uns32 constants (0..4,294,967,295)
- Int8 constants (-128..+127)
- Int16 constants (-32768..+32767)
- Int32 constants (-2,147,483,648..+2,147,483,647)
- Char constants (any ASCII character with a character code in the range 0..255)
- Byte constants (any eight-bit value including integers, booleans, and characters)
- Word constants (any 16-bit value)
- DWord constants (any 32-bit value)
- Real32 constants (floating point values)
- Real64 constants (floating point values)
- Real80 constants (floating point values)

In addition to the constant types appearing above, the CONST section supports six additional constant types:

- String constants
- Text constants
- Enumerated constant values

- Array constants
- Record/Union constants
- Character set constants

These data types are the subject of this Volume and the discussion of most of them appears in later chapters. However, the string and text constants are sufficiently important to warrant an early discussion of these constant types.

1.4.2 String and Character Literal Constants

HLA, like most programming languages, draws a distinction between a sequence of characters, a *string*, and a single character. This distinction is present both in the type declarations and in the syntax for literal character and string constants. Until now, this text has not drawn a fine distinction between character and string literal constants; now it is time to do so.

String literal constants consist of a sequence of zero or more characters surrounded by the ASCII quote characters. The following are all examples of legal literal string constants:

```
“This is a string”       // String with 16 characters.
““                       // Zero length string.
“a”                      // String with a single character.
“123”                    // String of length three.
```

A string of length one is not the same thing as a character constant. HLA uses two completely different internal representations for character and string values. Hence, “a” is not a character value, it is a string value that just happens to contain a single character.

Character literal constants take a couple forms, but the most common consist of a single character surrounded by ASCII apostrophe characters:

```
‘2’                      // Character constant equivalent to ASCII code $32.
‘a’                      // Character constant for lower case ‘A’.
```

As noted above, “a” and ‘a’ are not equivalent.

Those who are familiar with C/C++/Java probably recognize these literal constant forms, since they are similar to the character and string constants in C/C++/Java. In fact, this text has made a tacit assumption to this point that you are somewhat familiar with C/C++ insofar as examples appearing up to this point use character and string constants without an explicit definition of them[4].

Another similarity between C/C++ strings and HLA’s is the automatic concatenation of adjacent literal string constants within your program. For example, HLA concatenates the two string constants

```
“First part of string, “    “second part of string”
```

to form the single string constant

```
“First part of string, second part of string”
```

Beyond these few similarities, however, HLA strings and C/C++ strings are different. For example, C/C++ strings let you specify special character values using the escape character sequence consisting of a backslash character followed by one or more special characters; HLA does not use this escape character mechanism. HLA does provide, however, several other ways to achieve this same goal.

Since HLA does not allow escape character sequences in literal string and character constants, the first question you might ask is “How does one embed quote characters in string constants and apostrophe characters in character constants?” To solve this problem, HLA uses the same technique as Pascal and many other

4. Apologies are due to those of you who do not know C/C++/Java or a language that shares these string and constant definitions.

languages: you insert two quotes in a string constant to represent a single quote or you place two apostrophes in a character constant to represent a single apostrophe character, e.g.,

```
“He wrote a ““Hello World”” program as an example.”
```

The above is equivalent to:

```
He wrote a “Hello World” program as an example.
```

```
‘‘‘‘
```

The above is equivalent to a single apostrophe character.

HLA provides a couple of other features that eliminate the need for escape characters. In addition to concatenating two adjacent string constants to form a longer string constant, HLA will also concatenate any combination of adjacent character and string constants to form a single string constant:

```
‘1’  ‘2’  ‘3’            // Equivalent to “123”
“He wrote a “  ‘”’ “Hello World”  ‘”’ “ program as an example.”
```

Note that the two “He wrote...” strings in the above examples are identical to HLA.

HLA provides a second way to specify character constants that handles all the other C/C++ escape character sequences: the ASCII code literal character constant. This literal character constant form uses the syntax:

```
#integer_constant
```

This form creates a character constant whose value is the ASCII code specified by *integer_constant*. The numeric constant can be a decimal, hexadecimal, or binary value, e.g.,

```
#13    #$d    #%1101        // All three are the same character, a
                            //   carriage return.
```

Since you may concatenate character literals with strings, and the *#constant* form is a character literal, the following are all legal strings:

```
“Hello World” #13 #10           // #13 #10 is the Windows newline sequence
                                //  (carriage return followed by line feed).

“Error: Bad Value” #7           // #7 is the bell character.
“He wrote a “ #$22 “Hello World” #$22 “ program as an example.”
```

Since $22 is the ASCII code for the quote character, this last example is yet a third form of the “He wrote...” string literal.

1.4.3 String and Text Constants in the CONST Section

String and text constants in the CONST section use the following declaration syntax:

```
const
   AStringConst:     string := “123”;
   ATextConst:       text   := “123”;
```

Other than the data type of these two constants, their declarations are identical. However, their behavior in an HLA program is quite different.

Whenever HLA encounters a symbolic string constant within your program, it substitutes the string literal constant in place of the string name. So a statement like “stdout.put(AStringConst);” prints the string “123” (without quotes, of course) to the display. No real surprise here.

Whenever HLA encounters a symbolic text constant within your program, it substitutes the text of that string (rather than the string literal constant) for the identifier. That is, HLA substitutes the characters between the delimiting quotes in place of the symbolic text constant. Therefore, the following statement is perfectly legal given the declarations above:

```
mov( ATextConst, al );      // equivalent to mov( 123, al );
```

Note that substituting *AStringConst* for *ATextConst* in this example is illegal:

```
mov( AStringConst, al );    // equivalent to mov( “123”, al );
```

This latter example is illegal because you cannot move a string literal constant into the AL register.

Whenever HLA encounters a symbolic text constant in your program, it immediately substitutes the value of the text constant's string for that text constant and continues the compilation as though you had written the text constant's value rather than the symbolic identifier in your program. This can save some typing and help make your programs a little more readable if you often enter some sequence of text in your program. For example, consider the *nl* (newline) text constant declaration found in the HLA stdio.hhf library header file:

```
const
   nl: text := “#$d #$a”;  // Windows version.  Linux is just a line feed.
```

Whenever HLA encounters the symbol *nl*, it immediately substitutes the value of the string "#$d #$a" for the *nl* identifier. When HLA sees the #$d (carriage return) character constant followed by the #$a (line feed) character constants, it concatenates the two to form the string containing the Windows newline sequence (a carriage return followed by a line feed). Consider the following two statements:

```
stdout.put( “Hello World”, nl );
stdout.put( “Hello World”  nl );
```

(Notice that the second statement above does not separate the string literal and the nl symbol with a comma.) In the first example, HLA emits code that prints the string "Hello World" and then emits some additional code that prints a newline sequence. In the second example, HLA expands the *nl* symbol as follows:

```
stdout.put( “Hello World” #$d #$a );
```

Now HLA sees a string literal constant ("Hello World") followed by two character constants. It concatenates the three of them together to form a single string and then prints this string with a single call. Therefore, leaving off the comma between the string literal and the *nl* symbol produces slightly more efficient code. Keep in mind that this only works with string literal constants. You cannot concatenate string variables, or a string variable with a string literal, by using this technique.

Linux users should note that the Linux end of line sequence is just a single linefeed character. Therefore, the declaration for *nl* is slightly different in Linux.

In the constant section, if you specify only a constant identifier and a string constant (i.e., you do not supply a type), HLA defaults to type *string*. If you want to declare a *text* constant you must explicitly supply the type.

```
const
   AStrConst := “String Constant”;
   ATextConst: text := “mov( 0, eax );”;
```

1.4.4 Constant Expressions

Thus far, this chapter has given the impression that a symbolic constant definition consists of an identifier, an optional type, and a literal constant. Actually, HLA constant declarations can be a lot more sophisticated than this because HLA allows the assignment of a constant expression, not just a literal constant, to a symbolic constant. The generic constant declaration takes one of the following two forms:

```
Identifier : typeName := constant_expression ;
Identifier := constant_expression ;
```

Constant expressions take the familiar form you're used to in high level languages like C/C++ and Pascal. They may contain literal constant values, previously declared symbolic constants, and various arithmetic operators. The following lists some of the operations possible in a constant expression:

```
Arithmetic Operators

    -      (unary negation)  Negates the expression immediately following the “-”.
    *      Multiplies the integer or real values around the asterisk.
    div    Divides the left integer operand by the right integer operand
           producing an integer (truncated) result.
    mod    Divides the left integer operand by the right integer operand
           producing an integer remainder.
    /      Divides the left numeric operand by the second numeric operand
           producing a floating point result.
    +      Adds the left and right numeric operands.
    -      Subtracts the right numeric operand from the left numeric operand.

Comparison Operators

    =, ==  Compares left operand with right operand. Returns TRUE if equal.
    <>, != Compares left operand with right operand. Returns TRUE if not equal.
    <      Returns true if left operand is less than right operand.
    <=     Returns true if left operand is <= right operand.
    >      Returns true if left operand is greater than right operand.
    >=     Returns true if left operand is >= right operand.

Logical Operators[5]:

    &      For boolean operands, returns the logical AND of the two operands.
    |      For boolean operands, returns the logical OR of the two operands.
    ^      For boolean operands, returns the logical exclusive-OR.
    !      Returns the logical NOT of the single operand following “!”.

Bitwise Logical Operators:

    &      For integer numeric operands, returns bitwise AND of the operands.
    |      For integer numeric operands, returns bitwise OR of the operands.
    ^      For integer numeric operands, returns bitwise XOR of the operands.
    !      For an integer numeric operand, returns bitwise NOT of the operand.

String Operators:

    ‘+’    Returns the concatenation of the left and right string operands.
```

The constant expression operators follow standard precedence rules; you may use the parentheses to override the precedence if necessary. See the HLA reference in the appendix for the exact precedence relationships between the operators. In general, if the precedence isn't obvious, use parentheses to exactly state the order of evaluation. HLA actually provides a few more operators than these, though the ones above are the ones you will most commonly use. Please see the HLA documentation for a complete list of constant expression operators.

If an identifier appears in a constant expression, that identifier must be a constant identifier that you have previously defined in your program. You may not use variable identifiers in a constant expression; their values are not defined at compile-time when HLA evaluates the constant expression. Also, don't confuse compile-time and run-time operations:

```
// Constant expression, computed while HLA is compiling your program:
```

5. Note to C/C++ and Java users. HLA's constant expressions use complete boolean evaluation rather than short-circuit boolean evaluation. Hence, HLA constant expressions do not behave identically to C/C++/Java expressions.

```
const
        x       := 5;
        y       := 6;
        Sum     := x + y;


// Run-time calculation, computed while your program is running, long after
// HLA has compiled it:

    mov( x, al );
    add( y, al );
```

HLA directly interprets the value of a constant expression during compilation. It does not emit any machine instructions to compute "x+y" in the constant expression above. Instead, it directly computes the sum of these two constant values. From that point forward in the program, HLA associates the value 11 with the constant *Sum* just as if the program had contained the statement "Sum := 11;" rather than "Sum := x+y;" On the other hand, HLA does not precompute the value 11 in AL for the MOV and ADD instructions above[6], it faithfully emits the object code for these two instructions and the 80x86 computes their sum when the program is run (sometime after the compilation is complete).

In general, constant expressions don't get very sophisticated. Usually, you're adding, subtracting, or multiplying two integer values. For example, the following CONST section defines a set of constants that have consecutive values:

```
const
    TapeDAT    :=  1;
    Tape8mm    :=  TapeDAT + 1;
    TapeQIC80  :=  Tape8mm + 1;
    TapeTravan :=  TapeQIC80 + 1;
    TapeDLT    :=  TapeTravan + 1;
```

The constants above have the following values: TapeDAT = 1, Tape8mm = 2, TapeQIC80 = 3, TapeTravan = 4, and TapeDLT = 5.

1.4.5 Multiple CONST Sections and Their Order in an HLA Program

Although CONST sections must appear in the declaration section of an HLA program (e.g., between the "PROGRAM *pgmname*;" header and the corresponding "BEGIN *pgmname*;" statement), they do not have to appear before or after any other items in the declaration section. In fact, like the variable declaration sections, you can place multiple CONST sections in the declaration section. The only restriction on HLA constant declarations is that you must declare any constant symbol before you use it in your program.

Some C/C++ programmers, for example, are more comfortable writing their constant declarations as follows (since this is closer to C/C++'s syntax for declaring constants):

```
const  TapeDAT    :=  1;
const  Tape8mm    :=  TapeDAT + 1;
const  TapeQIC80  :=  Tape8mm + 1;
const  TapeTravan :=  TapeQIC80 + 1;
const  TapeDLT    :=  TapeTravan + 1;
```

The placement of the CONST section in a program seems to be a personal issue among programmers. Other than the requirements of defining all constants before you use them, you may feel free to insert the constant declaration section anywhere in the declaration section. Some programmers prefer to put all their

6. Technically, if HLA had an optimizer it could replace these two instructions with a single "MOV(11, al);" instruction. HLA v1.x, however, does not do this.

CONST declarations at the beginning of their declaration section, some programmers prefer to spread them throughout declaration section, defining the constants just before they need them for some other purpose. Putting all your constants at the beginning of an HLA declaration section is probably the wisest choice right now. Later in this text you'll see reasons why you might want to define your constants later in a declaration section.

1.4.6 The HLA VAL Section

You cannot change the value of a constant you define in the CONST section. While this seems perfectly reasonable (constants after all, are supposed to be, well, constant), there are different ways we can define the term *constant* and CONST objects only follow the rules of one specific definition. HLA's VAL section lets you define constant objects that follow slightly different rules. This section will discuss the VAL section and the difference between VAL constants and CONST constants.

The concept of "*const-ness*" can exist at two different times: while HLA is compiling your program and later when your program executes (and HLA is no longer running). All reasonable definitions of a constant require that a value not change while the program is running. Whether or not the value of a "constant" can change during compilation is a separate issue. The difference between HLA CONST objects and HLA VAL objects is whether the value of the constant can change during compilation.

Once you define a constant in the CONST section, the value of that constant is immutable from that point forward *both at run-time and while HLA is compiling your program*. Therefore, an instruction like "mov(SymbolicCONST, EAX);" always moves the same value into EAX, regardless of where this instruction appears in the HLA main program. Once you define the symbol *SymbolicCONST* in the CONST section, this symbol has the same value from that point forward.

The HLA VAL section lets you declare symbolic constants, just like the CONST section. However, HLA VAL constants can change their value throughout the source code in your program. The following HLA declarations are perfectly legal:

```
val    InitialValue  := 0;
const  SomeVal       := InitialValue + 1;     // = 1
const  AnotherVal    := InitialValue + 2;     // = 2

val    InitialValue  := 100;
const  ALargerVal    := InitialValue;         // = 100
const  LargeValTwo   := InitialValue*2;       // = 200
```

All of the symbols appearing in the CONST sections use the symbolic value *InitialValue* as part of the definition. Note, however, that *InitialValue* has different values at different points in this code sequence; at the beginning of the code sequence *InitialValue* has the value zero, while later it has the value 100.

Remember, at run-time a VAL object is not a variable; it is still a manifest constant and HLA will substitute the current value of a VAL identifier for that identifier[7]. Statements like "MOV(25, InitialValue);" are no more legal than "MOV(25, 0);" or "MOV(25, 100);"

1.4.7 Modifying VAL Objects at Arbitrary Points in Your Programs

If you declare all your VAL objects in the declaration section, it would seem that you would not be able to change the value of a VAL object between the BEGIN and END statements of your program. After all, the VAL section must appear in the declaration section of the program and the declaration section ends before the BEGIN statement. Later, you will learn that most VAL object modifications occur between the BEGIN and END statements; hence, HLA must provide someway to change the value of a VAL object outside the declaration section. The mechanism to do this is the "?" operator.

7. In this context, *current* means the value last assigned to a VAL object looking backward in the source code.

Not only does HLA allow you to change the value of a VAL object outside the declaration section, it allows you to change the value of a VAL object almost *anywhere* in the program. Anywhere a space is allowed inside an HLA program, you can insert a statement of the form:

```
? ValIdentifier := constant_expression ;
```

This means that you could write a short program like the following:

```
program VALdemo;
#include( “stdlib.hhf” );

val
    NotSoConstant := 0;

begin VALdemo;

    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 10;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 20;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

    ?NotSoConstant := 30;
    mov( NotSoConstant, eax );
    stdout.put( “EAX = “, (type uns32 eax ), nl );

end VALdemo;
```

Program 1.4 Demonstration of VAL Redefinition Using “?” Operator

You probably won’t have much use for VAL objects at this time. However, later on you’ll see (in the chapter on the HLA compile-time language) how useful VAL objects can be to you.

1.5 The HLA TYPE Section

Let’s say that you simply do not like the names that HLA uses for declaring byte, word, double word, real, and other variables. Let’s say that you prefer Pascal’s naming convention or, perhaps, C’s naming convention. You want to use terms like *integer*, *float*, *double*, or whatever. If this were Pascal you could redefine the names in the **type** section of the program. With C you could use a **#define** or a **typedef** statement to accomplish the task. Well, HLA, like Pascal, has it’s own TYPE statement that also lets you create aliases of these names. The following example demonstrates how to set up some C/C++/Pascal compatible names in your HLA programs:

```
type
   integer:   int32;
```

```
    float:     real32;
    double:    real64;
    colors:    byte;
```

Now you can declare your variables with more meaningful statements like:

```
static
    i:            integer;
    x:            float;
    HouseColor:   colors;
```

If you are an Ada, C/C++, or FORTRAN programmer (or any other language, for that matter), you can pick type names you're more comfortable with. Of course, this doesn't change how the 80x86 or HLA reacts to these variables one iota, but it does let you create programs that are easier to read and understand since the type names are more indicative of the actual underlying types. One warning for C/C++ programmers: don't get too excited and go off and define an *int* data type. Unfortunately, INT is an 80x86 machine instruction (interrupt) and therefore, this is a reserved word in HLA.

The TYPE section is useful for much more than creating type isomorphism (that is, giving a new name to an existing type). The following sections will demonstrate many of the possible things you can do in the TYPE section.

1.6 ENUM and HLA Enumerated Data Types

In a previous section discussing constants and constant expressions, you saw the following example:

```
const  TapeDAT    :=  1;
const  Tape8mm    :=  TapeDAT + 1;
const  TapeQIC80  :=  Tape8mm + 1;
const  TapeTravan :=  TapeQIC80 + 1;
const  TapeDLT    :=  TapeTravan + 1;
```

This example demonstrates how to use constant expressions to develop a set of constants that contain unique, consecutive, values. There are, however, a couple of problems with this approach. First, it involves a lot of typing (and extra reading when reviewing this program). Second, it's very easy make a mistake when creating long lists of unique constants and reuse or skip some values. The HLA ENUM type provides a better way to create a list of constants with unique values.

ENUM is an HLA type declaration that lets you associate a list of names with a new type. HLA associates a unique value with each name (that is, it *enumerates* the list). The ENUM keyword typically appears in the TYPE section and you use it as follows:

```
type
    enumTypeID:   enum { comma_separated_list_of_names };
```

The symbol *enumTypeID* becomes a new type whose values are specified by the specified list of names. As a concrete example, consider the data type *TapeDrives* and a corresponding variable declaration of type *TypeDrives*:

```
type
    TapeDrives: enum{ TapeDAT, Tape8mm, TapeQIC80, TapeTravan, TapeDLT};

static
    BackupUnit:   TapeDrives := TapeDAT;

    .
    .
    .

    mov( BackupUnit, al );
```

```
if( al = Tape8mm ) then

    ...

endif;

// etc.
```

By default, HLA reserves one byte of storage for enumerated data types. So the *BackupUnit* variable will consume one byte of memory and you would typically use an eight-bit register to access it[8]. As for the constants, HLA associates consecutive *uns8* constant values starting at zero with each of the enumerated identifiers. In the TapeDrives example, the tape drive identifiers would have the values *TapeDAT*=0, *Tape8mm*=1, *TapeQIC80*=2, *TapeTravan*=3, and *TapeDLT*=4. You may use these constants exactly as though you had defined them with these values in a CONST section.

1.7 Pointer Data Types

Some people refer to pointers as scalar data types, others refer to them as composite data types. This text will treat them as scalar data types even though they exhibit some tendencies of both scalar and composite data types.

Of course, the place to start is with the question "What is a pointer?" Now you've probably experienced pointers first hand in the Pascal, C, or Ada programming languages and you're probably getting worried right now. Almost everyone has a real bad experience when they first encounter pointers in a high level language. Well, fear not! Pointers are actually *easier* to deal with in assembly language. Besides, most of the problems you had with pointers probably had nothing to do with pointers, but rather with the linked list and tree data structures you were trying to implement with them. Pointers, on the other hand, have lots of uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and records often involve the use of pointers. So if you've got some deep-rooted fear about pointers, well forget everything you know about them. You're going to learn how *great* pointers really are.

Probably the best place to start is with the definition of a pointer. Just exactly what is a pointer, anyway? Unfortunately, high level languages like Pascal tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity (which exists for good reason, by the way) tends to frighten programmers because *they don't understand what's going on*.

Now if you're afraid of pointers, well, let's just ignore them for the time being and work with an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is pretty easy to understand. M is an array with 1024 integers in it, indexed from *M[0]* to *M[1023]*. Each one of these array elements can hold an integer value that is independent of all the others. In other words, this array gives you 1024 different integer variables each of which you refer to by number (the array index) rather than by name.

If you encountered a program that had the statement "M[0]:=100;" you probably wouldn't have to think at all about what is happening with this statement. It is storing the value 100 into the first element of the array *M*. Now consider the following two statements:

```
i := 0; (* Assume "i" is an integer variable *)
M [i] := 100;
```

You should agree, without too much hesitation, that these two statements perform the same exact operation as "M[0]:=100;". Indeed, you're probably willing to agree that you can use any integer expression in the

8. HLA provides a mechanism by which you can specify that enumerated data types consume two or four bytes of memory. See the HLA documentation for more details.

range 0…1023 as an index into this array. The following statements *still* perform the same operation as our single assignment to index zero:

```
i := 5;    (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

"Okay, so what's the point?" you're probably thinking. "Anything that produces an integer in the range 0…1023 is legal. So what?" Okay, how about the following:

```
M [1] := 0;
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you'll discover that these two instructions perform the exact same operation you've been doing all along. The first statement stores zero into array element *M[1]*. The second statement fetches the value of *M[1]*, which is an integer so you can use it as an array index into *M*, and uses that value (zero) to control where it stores the value 100.

If you're willing to accept the above as reasonable, perhaps bizarre, but usable nonetheless, then you'll have no problems with pointers. *Because m[1] is a pointer!* Well, not really, but if you were to change "M" to "memory" and treat this array as all of memory, this is the exact definition of a pointer.

1.7.1 Using Pointers in Assembly Language

A pointer is simply a memory location whose value is the address (or index, if you prefer) of some other memory location. Pointers are very easy to declare and use in an assembly language program. You don't even have to worry about array indices or anything like that.

An HLA pointer is a 32 bit value that may contain the address of some other variable. If you have a dword variable *p* that contains $1000_0000, then *p* "points" at memory location $1000_0000. To access the dword that *p* points at, you could use code like the following:

```
mov( p, ebx );        // Load EBX with the value of pointer p.
mov( [ebx], eax );    // Fetch the data that p points at.
```

By loading the value of *p* into EBX this code loads the value $1000_0000 into EBX (assuming *p* contains $1000_0000 and, therefore, points at memory location $1000_0000). The second instruction above loads the EAX register with the word starting at the location whose offset appears in EBX. Since EBX now contains $1000_0000, this will load EAX from locations $1000_0000 through $1000_0003.

Why not just load EAX directly from location $1000_0000 using an instruction like "MOV(mem, EAX);" (assuming *mem* is at address $1000_0000)? Well, there are lots of reasons. But the primary reason is that this single instruction always loads EAX from location *mem*. You cannot change the location from which it loads EAX. The former instructions, however, always load EAX from the location where *p* is pointing. This is very easy to change under program control. In fact, the simple instruction "MOV(&mem2, p);" will cause those same two instructions above to load EAX from *mem2* the next time they execute. Consider the following instructions:

```
mov( &i, p );         // Assume all variables are STATIC variables.
    .
    .
    .
if( some_expression ) then


    mov( &j, p );     // Assume the code above skips this instruction and
    .                 // you get to the next instruction by jumping
    .                 // to this point from somewhere else.
    .
```

```
        endif;
        mov( p, ebx );        // Assume both of the above code paths wind up
        mov( [ebx], eax );    // down here.
```

This short example demonstrates two execution paths through the program. The first path loads the variable *p* with the address of the variable *i*. The second path through the code loads *p* with the address of the variable *j*. Both execution paths converge on the last two MOV instructions that load EAX with *i* or *j* depending upon which execution path was taken. In many respects, this is like a *parameter* to a procedure in a high level language like Pascal. Executing the same instructions accesses different variables depending on whose address (*i* or *j*) winds up in *p*.

1.7.2 Declaring Pointers in HLA

Since pointers are 32 bits long, you could simply use the dword directive to allocate storage for your pointers. However, there is a much better way to do this: HLA provides the POINTER TO phrase specifically for declaring pointer variables. Consider the following example:

```
static
    b:              byte;
    d:              dword;
    pByteVar:       pointer to byte := &b;
    pDWordVar:      pointer to dword := &d;
```

This example demonstrates that it is possible to initialize as well as declare pointer variables in HLA. Note that you may only take addresses of static variables (STATIC, READONLY, and STORAGE objects) with the address-of operator, so you can only initialize pointer variables with the addresses of static objects.

You can also define your own pointer types in the TYPE section of an HLA program. For example, if you often use pointers to characters, you'll probably want to use a TYPE declaration like the one in the following example:

```
type
    ptrChar:   pointer to char;

static
    cString: ptrChar;
```

1.7.3 Pointer Constants and Pointer Constant Expressions

HLA allows two literal pointer constant forms: the address-of operator followed by the name of a static variable or the constant zero. In addition to these two literal pointer constants, HLA also supports simple pointer constant expressions.

The constant zero represents the NULL or NIL pointer, that is, an illegal address that does not exist[9]. Programs typically initialize pointers with NULL to indicate that a pointer has explicitly *not* been initialized. The HLA Standard Library predefines both the "NULL" and "nil" constants in the memory.hhf header file[10].

In addition to simple address literals and the value zero, HLA allows very simple constant expressions wherever a pointer constant is legal. Pointer constant expressions take one of the two following forms:

```
        &StaticVarName + PureConstantExpression
        &StaticVarName - PureConstantExpression
```

9. Actually, address zero does exist, but if you try to access it under Windows or Linux you will get a general protection fault.
10. NULL is for C/C++ programmers and nil is familiar to Pascal/Delphi programmers.

The *PureConstantExpression* term is a numeric constant expression that does not involve any pointer constants. This type of expression produces a memory address that is the specified number of bytes before or after ("-" or "+", respectively) the *StaticVarName* variable in memory.

Since you can create pointer constant expressions, it should come as no surprise to discover that HLA lets you define manifest pointer constants in the CONST section. The following program demonstrates how you can do this.

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

static
    b:  byte := 0;
        byte    1, 2, 3, 4, 5, 6, 7;

const
    pb:= &b + 1;

begin PtrConstDemo;

    mov( pb, ebx );
    mov( [ebx], al );
    stdout.put( "Value at address pb = $", al, nl );

end PtrConstDemo;
```

Program 1.5 Pointer Constant Expressions in an HLA Program

Upon execution, this program prints the value of the byte just beyond *b* in memory (which contains the value $01).

1.7.4 Pointer Variables and Dynamic Memory Allocation

Pointer variables are the perfect place to store the return result from the HLA Standard Library *malloc* function. The *malloc* function returns the address of the storage it allocates in the EAX register; therefore, you can store the address directly into a pointer variable with a single MOV instruction immediately after a call to *malloc*:

```
type
    bytePtr:   pointer to byte;

var
    bPtr: bytePtr;
        .
        .
        .
    malloc( 1024 );          // Allocate a block of 1,024 bytes.
    mov( eax, bPtr );        // Store address of block in bPtr.
        .
        .
        .
    free( bPtr );     // Free the allocated block when done using it.
```

```
        .
        .
        .
```

In addition to *malloc* and *free*, the HLA Standard Library provides a *realloc* procedure. The *realloc* routine takes two parameters, a pointer to a block of storage that *malloc* (or *realloc*) previously created, and a new size. If the new size is less than the old size, realloc releases the storage at the end of the allocated block back to the system. If the new size is larger than the current block, then *realloc* will allocate a new block and move the old data to the start of the new block, then free the old block.

Typically, you would use *realloc* to correct a bad guess about a memory size you'd made earlier. For example, suppose you want to read a set of values from the user but you won't know how many memory locations you'll need to hold the values until after the user has entered the last value. You could make a wild guess and then allocate some storage using *malloc* based on your estimate. If, during the input, you discover that your estimate was too low, simply call *realloc* with a larger value. Repeat this as often as required until all the input is read. Once input is complete, you can make a call to *realloc* to release any unused storage at the end of the memory block.

The *realloc* procedure uses the following calling sequence:

```
realloc( ExistingPointer, NewSize );
```

Realloc returns a pointer to the newly allocated block in the EAX register.

One danger exists when using *realloc*. If you've made multiple copies of pointers into a block of storage on the heap and then call *realloc* to resize that block, all the existing pointers are now invalid. Effectively *realloc* frees the existing storage and then allocates a new block. That new block may not be in the same memory location at the old block, so any existing pointers (into the block) that you have will be invalid after the *realloc* call.

1.7.5 Common Pointer Problems

There are five common problems programmers encounter when using pointers. Some of these errors will cause your programs to immediately stop with a diagnostic message; other problems are more subtle, yielding incorrect results without otherwise reporting an error or simply affecting the performance of your program without displaying an error. These five problems are

- Using an uninitialized pointer
- Using a pointer that contains an illegal value (e.g., NULL)
- Continuing to use malloc'd storage after that storage has been free'd
- Failing to free storage once the program is done using it
- Accessing indirect data using the wrong data type.

The first problem above is using a pointer variable before you have assigned a valid memory address to the pointer. Beginning programmers often don't realize that declaring a pointer variable only reserves storage for the pointer itself, it does not reserve storage for the data that the pointer references. The following short program demonstrates this problem:

```
// Program to demonstrate use of
// an uninitialized pointer.  Note
// that this program should terminate
// with a Memory Access Violation exception.

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

static
```

```
    // Note: by default, varibles in the
    // static section are initialized with
    // zero (NULL) hence the following
    // is actually initialized with NULL,
    // but that will still cause our program
    // to fail because we haven’t initialized
    // the pointer with a valid memory address.

    Uninitialized: pointer to byte;

begin UninitPtrDemo;

    mov( Uninitialized, ebx );
    mov( [ebx], al );
    stdout.put( “Value at address Uninitialized: = $”, al, nl );

end UninitPtrDemo;
```

Program 1.6 Uninitialized Pointer Demonstration

Although variables you declare in the STATIC section are, technically, initialized; static initialization still doesn’t initialize the pointer in this program with a valid address.

Of course, there is no such thing as a truly uninitialized variable on the 80x86. What you really have are variables that you’ve explicitly given an initial value and variables that just happen to inherit whatever bit pattern was in memory when storage for the variable was allocated. Much of the time, these garbage bit patterns laying around in memory don’t correspond to a valid memory address. Attempting to *dereference* such a pointer (that is, access the data in memory at which it points) raises a Memory Access Violation exception.

Sometimes, however, those random bits in memory just happen to correspond to a valid memory location you can access. In this situation, the CPU will access the specified memory location without aborting the program. Although to a naive programmer this situation may seem preferable to aborting the program, in reality this is far worse because your defective program continues to run with a defect without alerting you to the problem. If you store data through an uninitialized pointer, you may very well overwrite the values of other important variables in memory. This defect can produce some very difficult to locate problems in your program.

The second problem programmers have with pointers is storing invalid address values into a pointer. The first problem, above, is actually a special case of this second problem (with garbage bits in memory supplying the invalid address rather than you producing via a miscalculation). The effects are the same; if you attempt to dereference a pointer containing an invalid address you will either get a Memory Access Violation exception or you will access an unexpected memory location.

The third problem listed above is also known as the dangling pointer problem. To understand this problem, consider the following code fragment:

```
malloc( 256 );    // Allocate some storage.
mov( eax, ptr );  // Save address away in a pointer variable.
    .
    .             // Code that use the pointer variable “ptr”.
    .
free( ptr );      // Free the storage associated with “ptr”.
    .
    .             // Code that does not change the value in “ptr”.
    .
mov( ptr, ebx );
mov( al, [ebx] );
```

In this example you will note that the program allocates 256 bytes of storage and saves the address of that storage away in the *ptr* variable. Then the code uses this block of 256 bytes for a while and frees the storage, returning it to the system for other uses. Note that calling *free* does not change the value of *ptr* in any way; *ptr* still points at the block of memory allocated by *malloc* earlier. Indeed, *free* does not change any data in this block, so upon return from *free*, *ptr* still points at the data stored into the block by this code. However, note that the call to *free* tells the system that this 256-byte block of memory is no longer needed by the program and the system can use this region of memory for other purposes. The *free* function cannot enforce that fact that you will never access this data again, you are simply promising that you won't. Of course, the code fragment above breaks this promise; as you can see in the last two instructions above the program fetches the value in *ptr* and accesses the data it points at in memory.

The biggest problem with dangling pointers is that you can get away with using them a good part of the time. As long as the system doesn't reuse the storage you've free'd, using a dangling pointer produces no ill effects in your program. However, with each new call to *malloc*, the system may decide to reuse the memory released by that previous call to *free*. When this happens, any attempt to dereference the dangling pointer may produce some unintended consequences. The problems range from reading data that has been overwritten (by the new, legal, use of the data storage), to overwriting the new data, to (the worst case) overwriting system heap management pointers (doing so will probably cause your program to crash). The solution is clear: *never use a pointer value once you free the storage associated with that pointer.*

Of all the problems, the fourth (failing to free allocated storage) will probably have the least impact on the proper operation of your program. The following code fragment demonstrates this problem:

```
malloc( 256 );
mov( eax, ptr );
    .          // Code that uses the data where ptr is pointing.
    .          // This code does not free up the storage
    .          // associated with ptr.
malloc( 512 );
mov( eax, ptr );

// At this point, there is no way to reference the original
// block of 256 bytes pointed at by ptr.
```

In this example the program allocates 256 bytes of storage and references this storage using the *ptr* variable. At some later time, the program allocates another block of bytes and overwrites the value in *ptr* with the address of this new block. Note that the former value in *ptr* is lost. Since this address no longer exists in the program, there is no way to call *free* to return the storage for later use. As a result, this memory is no longer available to your program. While making 256 bytes of memory inaccessible to your program may not seem like a big deal, imagine now that this code is in a loop that repeats over and over again. With each execution of the loop the program loses another 256 bytes of memory. After a sufficient number of loop iterations, the program will exhaust the memory available on the heap. This problem is often called a *memory leak* because the effect is the same as though the memory bits were leaking out of your computer (yielding less and less available storage) during program execution[11].

Memory leaks are far less damaging than using dangling pointers. Indeed, there are only two problems with memory leaks: the danger of running out of heap space (which, ultimately, may cause the program to abort, though this is rare) and performance problems due to virtual memory page swapping. Nevertheless, you should get in the habit of always free all storage once you are done using it. Note that when your program quits, the operating system reclaims all storage including the data lost via memory leaks. Therefore, memory lost via a leak is only lost to your program, not the whole system.

The last problem with pointers is the lack of type-safe access. HLA cannot and does not enforce pointer type checking. For example, consider the following program:

11. Note that the storage isn't lost from you computer; once your program quits it returns all memory (including unfree'd storage) to the O/S. The next time the program runs it will start with a clean slate.

```
// Program to demonstrate use of
// lack of type checking in pointer
// accesses.

program BadTypePtrDemo;
#include( “stdlib.hhf” );

static
    ptr:    pointer to char;
    cnt:    uns32;

begin BadTypePtrDemo;

    // Allocate sufficient characters
    // to hold a line of text input
    // by the user:

    malloc( 256 );
    mov( eax, ptr );


    // Okay, read the text a character
    // at a time by the user:

    stdout.put( “Enter a line of text: “ );
    stdin.flushInput();
    mov( 0, cnt );
    mov( ptr, ebx );
    repeat

        stdin.getc();       // Read a character from the user.
        mov( al, [ebx] );   // Store the character away.
        inc( cnt );         // Bump up count of characters.
        inc( ebx );         // Point at next position in memory.

    until( stdin.eoln());


    // Okay, we’ve read a line of text from the user,
    // now display the data:

    mov( ptr, ebx );
    for( mov( cnt, ecx ); ecx > 0; dec( ecx )) do

        mov( [ebx], eax );
        stdout.put( “Current value is $”, eax, nl );
        inc( ebx );

    endfor;
    free( ptr );


end BadTypePtrDemo;
```

Program 1.7 Type-Unsafe Pointer Access Example

This program reads in data from the user as character values and then displays the data as double word hexadecimal values. While a powerful feature of assembly language is that it lets you ignore data types at will and automatically coerce the data without any effort, this power is a two-edged sword. If you make a mistake and access indirect data using the wrong data type, HLA and the 80x86 may not catch the mistake and your program may produce inaccurate results. Therefore, you need to take care when using pointers and indirection in your programs that you use the data consistently with respect to data type.

1.8 Putting It All Together

This chapter contains an eclectic combination of subjects. It begins with a discussion of the INTMUL, BOUND, and INTO instructions that will prove useful throughout this text. Then this chapter discusses how to declare constants and data types, including enumerated data types. This chapter also introduces constant expressions and pointers. The following chapters in this text will make extensive use of these concepts.