Chapter 6

Compiling to the Assembly Level

The theme of this book is the application of the concept of levels of abstraction to computer science. This chapter continues the theme by showing the relationship between the high-order languages level and the assembly level. It examines features of the C++ language at level HOL6 and shows how a compiler might translate programs that use those features to the equivalent program at level Asmb5.

One major difference between level-HOL6 languages and level-Asmb5 languages is the absence of extensive data types at level Asmb5. In C++, you can define integers, reals, arrays, booleans, and structures in almost any combination. But assembly language has only bits and bytes. If you want to define an array of structures in assembly language, you must partition the bits and bytes accordingly. The compiler does that job automatically when you program at level HOL6.

Another difference between the levels concerns the flow of control. C++ has if, while, do, for, switch, and function statements to alter the normal sequential flow of control. You will see that assembly language is limited by the basic von Neumann design to more primitive control statements. This chapter shows how the compiler must combine several primitive level-Asmb5 control statements to execute a single, more powerful level-HOL6 control statement.

6.1 Stack Addressing and Local Variables

When a program calls a function, the program allocates storage on the run-time stack for the returned value, the parameters, and the return address. Then the function allocates storage for its local variables. Stack-relative addressing allows the function to access the information that was pushed onto the stack.

You can consider main() of a C++ program to be a function that the operating system calls. You might be familiar with the fact that the main program can have parameters named argc and argv as follows:

```c
int main (int argc, char* argv[])
```
With main declared this way, \texttt{argc} and \texttt{argv} are pushed onto the run-time stack, along with the return address and any local variables. To keep things simple, this book always declares \texttt{main()} without the parameters, and it ignores the fact that storage is allocated for the integer returned value and the return address. Hence, the only storage allocated for \texttt{main()} on the run-time stack is for local variables. This section describes how the compiler translates main programs that have local variables.

### Stack-Relative Addressing

In stack-relative addressing, the relationship between the operand and the operand specifier is

\[
\text{Oprnd} = \text{Mem}[\text{SP} + \text{OprndSpec}]
\]

The stack pointer acts as a memory address to which the operand specifier is added. Figure 4.39 shows that the user stack grows upward in main memory starting at address FBCF. When an item is pushed onto the run-time stack, its address is less than the address of the item that was on the top of the stack.

You can think of the operand specifier as the offset from the top of the stack. If the operand specifier is 0, the instruction accesses \text{Mem}[\text{SP}], the value on top of the stack. If the operand specifier is 2, it accesses \text{Mem}[\text{SP} + 2], the value two bytes below the top of the stack.

The Pep/8 instruction set has two instructions for manipulating the stack pointer directly, \texttt{ADDSP} and \texttt{SUBSP}. (\texttt{CALL}, \texttt{RETn}, and \texttt{RETR} manipulate the stack pointer indirectly.) \texttt{ADDSP} simply adds a value to the stack pointer and \texttt{SUBSP} subtracts a value. The RTL specification of \texttt{ADDSP} is

\[
\text{SP} \leftarrow \text{SP} + \text{Oprnd}; \quad \text{N} \leftarrow \text{SP} < 0, \quad \text{Z} \leftarrow \text{SP} = 0, \quad \text{V} \leftarrow \{\text{overflow}\}, \quad \text{C} \leftarrow \{\text{carry}\}
\]

and the RTL specification of \texttt{SUBSP} is

\[
\text{SP} \leftarrow \text{SP} - \text{Oprnd}; \quad \text{N} \leftarrow \text{SP} < 0, \quad \text{Z} \leftarrow \text{SP} = 0, \quad \text{V} \leftarrow \{\text{overflow}\}, \quad \text{C} \leftarrow \{\text{carry}\}
\]

Even though you can add to and subtract from the stack pointer, you cannot set the stack pointer with a load instruction. There is no \texttt{LDSP} instruction. Then how is the stack pointer ever set? When you select the execute option in the Pep/8 simulator the following two actions occur:

\[
\text{SP} \leftarrow \text{Mem}[\text{FFF8}]
\]

\[
\text{PC} \leftarrow 0000
\]

The first action sets the stack pointer to the content of memory location FFF8. That location is part of the operating system ROM, and it contains the address of the
top of the application’s run-time stack. Therefore, when you select the execute option the stack pointer is initialized correctly. The default Pep/8 operating system initializes SP to FBCF. The application never needs to set it to anything else. In general, the application only needs to add to the stack pointer to push items onto the run-time stack, and subtract from the stack pointer to pop items off of the run-time stack.

### Accessing the Run-Time Stack

Figure 6.1 shows how to push data onto the stack, access it with stack-relative addressing, and pop it off the stack. The program pushes the string "BMW" onto the stack followed by the decimal integer 325 followed by the character ‘i’. Then it outputs the items and pops them off the stack.

```
0000 C00042 LDA 'B',i ;push B
0003 F3FFFF STBYTEA -1,s
0006 C0004D LDA 'M',i ;push M
0009 F3FFFE STBYTEA -2,s
000C C00057 LDA 'W',i ;push W
000F F3FFFD STBYTEA -3,s
0012 C00145 LDA 325,i ;push 325
0015 E3FFFB STA -5,s
0018 C00069 LDA 'i',i ;push i
001B F3FFFF STBYTEA -6,s
001E 680006 SUBSP 6,i ;6 bytes on the run-time stack
0021 530005 CHARO 5,s ;output B
0024 530004 CHARO 4,s ;output M
0027 530003 CHARO 3,s ;output W
002A 3B0001 DECO 1,s ;output 325
002D 530000 CHARO 0,s ;output i
0030 600006 ADDSP 6,i ;dealloot stack storage
0033 00 STOP
0034 .END
```

**Output**

```
BMW325i
```

Figure 6.2(a) shows the values in the stack pointer (SP) and main memory before the program executes. The machine initializes the stack pointer to FBCF from the vector at Mem [FFF8].

The first two instructions,

```
LDA 'B',i
STBYTEA -1,s
```
put an ASCII ‘B’ character in the byte just above the top of the stack. \texttt{LDA} puts the ‘B’ byte in the right half of the accumulator, and \texttt{STBYTEA} puts it above the stack. The store instruction uses stack-relative addressing with an operand specifier of \(-1\) (dec) = \(\text{FFFF}\) (hex). Because the stack pointer has the value FBCF, the ‘B’ is stored at Mem \([\text{FBCF} + \text{FFFF}]\) = Mem [FBCE]. The next two instructions put ‘M’ and ‘W’ at Mem [FBCD] and Mem [FBCC], respectively.

The decimal integer 325, however, occupies two bytes. The program must store it at an address that differs from the address of the ‘W’ by two. That is why the instruction to store the 325 is

\texttt{STA -5,s}

and not

\texttt{STA -4,s}

In general, when you push items onto the run-time stack you must take into account how many bytes each item occupies and set the operand specifier accordingly.

The \texttt{SUBSP} instruction subtracts 6 from the stack pointer, as Figure 6.2(b) shows. That completes the push operation.

Tracing a program that uses stack-relative addressing does not require you to know the absolute value in the stack pointer. The push operation would work the same if the stack pointer were initialized to some other value, say FA18. In that case, ‘B’, ‘M’, ‘W’, 325, and ‘i’ would be at Mem [FA17], Mem [FA16], Mem [FA15], Mem [FA13], and Mem [FA12], respectively, and the stack pointer would wind up with a value of FA12. The values would be at the same locations relative to the top of the stack, even though they would be at different absolute memory locations.

Figure 6.3 is a more convenient way of tracing the operation and makes use of the fact that the value in the stack pointer is irrelevant. Rather than show the value in the stack pointer, it shows an arrow pointing to the memory cell whose address is contained in the stack pointer. Rather than show the address of the cells in memory, it shows their offsets from the stack pointer. Figures depicting the state of the run-time stack will use this drawing convention from now on.
The instruction

\textsc{charo} 5, s

outputs the ASCII ‘B’ character from the stack. Note that the stack-relative address of the ‘B’ before \textsc{subsp} executes is –1, but its address after \textsc{subsp} executes is 5. Its stack-relative address is different because the stack pointer has changed. Both

\textsc{stbytea} -1, s

and

\textsc{charo} 5, s

access the same memory cell. The other items are output similarly using their stack offsets shown in Figure 6.3(b).

The instruction

\textsc{addsp} 6, i

deallocates six bytes of storage from the run-time stack by adding 6 to SP. Because the stack grows upward toward smaller addresses, you allocate storage by subtracting from the stack pointer and you deallocate storage by adding to the stack pointer.

\section*{Local Variables}

The previous chapter shows how the compiler translates programs with global variables. It allocates storage for a global variable with a \texttt{.block} dot command and it accesses it with direct addressing. Local variables, however, are allocated on the run-time stack. To translate a program with local variables the compiler

\begin{itemize}
  \item allocates local variables with \texttt{subsp},
  \item accesses local variables with stack-relative addressing, and
  \item deallocates storage with \texttt{addsp}.
\end{itemize}
An important difference between global and local variables is the time at which the allocation takes place. The `.BLOCK` dot command is not an executable statement. Storage for global variables is reserved at a fixed location before the program executes. In contrast, the `SUBSP` statement is executable. Storage for local variables is created on the run-time stack during program execution.

Figure 6.4 is identical to the program of Figure 5.26 except that the variables are declared local to `main()`. Although this difference is not perceptible to the user

---

### High-Order Language

```cpp
#include <iostream>
using namespace std;

int main () {
    const int bonus = 5;
    int exam1;
    int exam2;
    int score;
    cin >> exam1 >> exam2;
    score = (exam1 + exam2) / 2 + bonus;
    cout << "score = " << score << endl;
    return 0;
}
```

### Assembly Language

```
0000 140003  BR         main
    bonus: .EQUATE 5          ;constant
    exam1: .EQUATE 4          ;local variable
    exam2: .EQUATE 2          ;local variable
    score: .EQUATE 0          ;local variable

0003 680006 main:  SUBSP   6,i        ;allocate locals
0006 330004  DECI    exam1,s    ;cin >> exam1
0009 330002  DECI    exam2,s    ;   >> exam2
000C C30004  LDA     exam1,s    ;score = (exam1
000F 730002  ADDA    exam2,s    ;   + exam2)
0012 1E      ASRA               ;   / 2
0013 700005  ADDA    bonus,i    ;   + bonus
0016 500000  STRO    score,s
0019 410006  STRO    msg,d      ;cout << "score = 
001C 3B0000  DECO    score,s    ;   << score
001F 500000A CHARO  'n',i      ;   << endl
0022 600006  ADDSP   6,1       ;deallocate locals
0025 00      STOP
0026 73636F msg:  .ASCII  "score = \x00"
                   726520
                   3D2000
002F .END
```

---
of the program, the translation performed by the compiler is significantly different. Figure 6.5 shows the run-time stack for the program. As in Figure 5.26, bonus is a constant and is defined with the .EQUATE command. However, local variables are also defined with .EQUATE. With a constant, .EQUATE specifies the value of the constant, but with a local variable, .EQUATE specifies the stack offset on the run-time stack. For example, Figure 6.5 shows that the stack offset for local variable exam1 is 6. Therefore, the assembly language program equates the symbol exam1 to 6. Note from the assembly language listing that .EQUATE does not generate any code for the local variables.

Translation of the executable statements in main() differs in two respects from the version with global variables. First, SUBSP and ADDSP allocate and deallocate storage on the run-time stack for the locals. Second, all accesses to the variables use stack-relative addressing instead of direct addressing. Other than these differences, the translation of the assignment and output statements is the same.

### 6.2 Branching Instructions and Flow of Control

The Pep/8 instruction set has eight conditional branches:

- **BRLE**: Branch on less than or equal to
- **BRLT**: Branch on less than
- **BREQ**: Branch on equal to
- **BRNE**: Branch on not equal to
- **BRGE**: Branch on greater than or equal to
- **BRGT**: Branch on greater than
- **BRV**: Branch on V
- **BRC**: Branch on C

![Figure 6.5](image_url)  
The run-time stack for the program of Figure 6.4.
Each of these conditional branches tests one or two of the four status bits, N, Z, V, and C. If the condition is true, the operand is placed in PC, causing the branch. If the condition is not true, the operand is not placed in PC, and the instruction following the conditional branch executes normally. You can think of them as comparing a 16-bit result to 0000 (hex). For example, \textbf{BRLT} checks whether a result is less than zero, which happens if N is 1. \textbf{BRLE} checks whether a result is less than or equal to zero, which happens if N is 1 or Z is 1. Here is the Register Transfer Language (RTL) specification of each conditional branch instruction.

\begin{align*}
\text{BRLT} & : N = 1 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRLE} & : N = 1 \lor Z = 1 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BREQ} & : Z = 1 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRNE} & : Z = 0 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRGE} & : N = 0 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRGT} & : N = 0 \land Z = 0 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRV} & : V = 1 \Rightarrow PC \leftarrow \text{Oprnd} \\
\text{BRC} & : C = 1 \Rightarrow PC \leftarrow \text{Oprnd}
\end{align*}

The conditional branch instructions

Whether a branch occurs depends on the value of the status bits. The status bits are in turn affected by the execution of other instructions. For example,

\begin{verbatim}
LDA num, s
BRLT place
\end{verbatim}

causes the content of \texttt{num} to be loaded into the accumulator. If the word represents a negative number, that is, if its sign bit is 1, then the N bit is set to 1. \textbf{BRLT} tests the N bit and causes a branch to the instruction at \texttt{place}. On the other hand, if the word loaded into the accumulator is not negative, then the N bit is cleared to 0. When \textbf{BRLT} tests the N bit, the branch does not occur and the instruction after \textbf{BRLT} executes next.

\section*{Translating the If Statement}

Figure 6.6 shows how a compiler would translate an \texttt{if} statement from C++ to assembly language. The program computes the absolute value of an integer.

The assembly language comments show the statements that correspond to the high-level program. The \texttt{cin} statement translates to \texttt{DECI} and the \texttt{cout} statement translates to \texttt{DECO}. The assignment statement translates to the sequence \texttt{LDA, NEGA, STA}.

The compiler translates the \texttt{if} statement into the sequence \texttt{LDA, BRGE}. When \texttt{LDA} executes, if the value loaded into the accumulator is positive or zero, the N bit is cleared to 0. That condition calls for skipping the body of the \texttt{if} statement. Figure 6.7(a) shows the structure of the \texttt{if} statement at level HOL6. S1 represents the
statement `cin >> number`, C1 represents the condition `number < 0`, S2 represents the statement `number = -number`, and S3 represents the statement `cout << number`. Figure 6.7(b) shows the structure with the more primitive branching.

\[ S_1 \]
\[
\text{if (C1) (}
\]
\[
\quad S_2
\]
\[
\text{)}
\]
\[
S_3
\]

(a) The structure at Level HOL6.

\[ S_1 \]
\[
\text{if (C1) (}
\]
\[
\quad S_2
\]
\[
\text{)}
\]
\[
S_3
\]

(b) The structure at level Asmb5 for Figure 6.6.

---

**High-Order Language**

```cpp
#include <iostream>
using namespace std;

int main () {
    int number;
    cin >> number;
    if (number < 0) {
        number = -number;
    }
    cout << number;
    return 0;
}
```

---

**Assembly Language**

```
0000 040003            BR      main

number: .EQUATE 0       ;local variable

0003 680002 main:      SUBSP   2,i       ;allocate local
0006 330000           DECI    number,s   ;cin >> number
0009 C30000 if:       LDA     number,s   ;if (number < 0)
000C 0E0016           BRGE    endIf
000F C30000           LDA     number,s   ;   number = -number
0012 1A               NEGA
0013 E30000           STA     number,s
0016 3B0000 endif:    DECO    number,s   ;cout << number
0019 600002          ADDSP   2,i       ;deallocate local
001C 00               STOP
001D                  .END
```

---

Figure 6.6 The `if` statement at level HOL6 and level Asmb5.

---

Figure 6.7 The structure of the `if` statement at level Asmb5.
instructions at level Asmb5. The dot following C1 represents the conditional branch, BRGE.

The braces { and } for delimiting a compound statement have no counterpart in assembly language. The sequence

**Statement 1**

`if (number >= 0) {
    Statement 2
    Statement 3
}
Statement 4`

translates to

```
Statement 1
if:    LDA number,d
BRLT endif
Statement 2
Statement 3
endif: Statement 4
```

---

### Optimizing Compilers

You may have noticed an extra load statement that was not strictly required in Figure 6.6. You can eliminate the LDA at 000F because the value of number will still be in the accumulator from the previous load at 0009.

The question is, what would a compiler do? The answer is that it depends on the compiler. A compiler is a program that must be written and debugged. Imagine that you must design a compiler to translate from C++ to assembly language. When the compiler detects an assignment statement, you program it to generate the following sequence: (a) load accumulator, (b) evaluate expression if necessary, (c) store result to variable. Such a compiler would generate the code of Figure 6.6, with the LDA at 000F.

Imagine how difficult your compiler program would be if you wanted it to eliminate the unnecessary load. When your compiler detected an assignment statement, it would not always generate the initial load. Instead, it would analyze the previous instructions generated and remember the content of the accumulator. If it determined that the value in the accumulator was the same as the value that the initial load put there, it would not generate the initial load. In Figure 6.6, the compiler would need to remember that the value of number was still in the accumulator from the code generated for the if statement.

A compiler that expends extra effort to make the object program shorter and faster is called an optimizing compiler. You can imagine how much more difficult an optimizing compiler is to design than a nonoptimizing one. Not only are opti-
mizing compilers more difficult to write, they also take longer to compile because they must analyze the source program in much greater detail.

Which is better, an optimizing or a nonoptimizing compiler? That depends on the use to which you put the compiler. If you are developing software, a process that requires many compiles for testing and debugging, then you would want a compiler that translates quickly, that is, a nonoptimizing compiler. If you have a large fixed program that will be executed repeatedly by many users, you would want fast execution of the object program, hence, an optimizing compiler. Frequently, software is developed and debugged with a nonoptimizing compiler and then translated one last time with an optimizing compiler for the users.

Real compilers come in all shades of gray between these two extremes. The examples in this chapter occasionally present object code that is partially optimized. Most assignment statements, such as the one in Figure 6.6, are presented in nonoptimized form.

### Translating the If/Else Statement

Figure 6.8 illustrates the translation of the if/else statement. The C++ program is identical to the one in Figure 2.9. The if body requires an extra unconditional branch around the else body. If the compiler omitted the BR at 0015 and the input were 127, the output would be highlow.

Unlike Figure 6.6, the if statement in Figure 6.8 does not compare a variable’s value with zero. It compares it with another nonzero value using CPA, which stands for compare accumulator. CPA subtracts the operand from the accumulator and sets the NZVC status bits accordingly. CPr is identical to SUBr except that SUBr stores the result of the subtraction in register r (accumulator or index register), whereas CPr ignores the result of the subtraction. The RTL specification of CPr is

\[
T \leftarrow r - \text{Oprnd}; N \leftarrow T < 0, \ Z \leftarrow T = 0, \ V \leftarrow \{\text{overflow}\}, \ C \leftarrow \{\text{carry}\}
\]

where T represents a temporary value.

This program computes \(\text{num} - \text{limit}\) and sets the NZVC bits. BRLT tests the N bit, which is set if

\[
\text{num} - \text{limit} < 0
\]

that is, if

\[
\text{num} < \text{limit}
\]

That is the condition under which the else part must execute.
Figure 6.9 shows the structure of the control statements at the two levels. Part a shows the level-HOL6 control statement, and part b shows the level-Asmb5 translation for this program.
6.2 Branching Instructions and Flow of Control

Translating the While Loop

Translating a loop requires branches to previous instructions. Figure 6.10 shows the translation of a while statement. The C++ program is identical to the one in Figure 2.12. It echoes ASCII input characters to the output, using the sentinel technique with * as the sentinel. If the input is happy*, the output is happy.

The test for a while statement is made with a conditional branch at the top of the loop. This program tests a character value, which is a byte quantity. The load instruction at 0007 clears both bytes in the accumulator, so the most significant byte will be 00 (hex) after the load byte instruction at 000A executes. You must guarantee that the most significant byte is 0 because the compare instruction compares a whole word.

Every while loop ends with an unconditional branch to the test at the top of the loop. The branch at 0019 brings control back to the initial test. Figure 6.11 shows the structure of the while statement at the two levels.

```cpp
#include <iostream>
using namespace std;

char letter;

int main () {
    cin >> letter;
    while (letter != '*') {
        cout << letter;
        cin >> letter;
    }
    return 0;
}
```

Figure 6.10
The while statement at level HOL6 and level Asmb5.

Figure 6.9
The structure of the if/else statement at level Asmb5.

(a) The structure at Level HOL6. (b) The structure at level Asmb5 for Figure 6.8.
Translating the Do Loop

A highway patrol officer parks behind a sign. A driver passes by, traveling 20 meters per second, which is faster than the speed limit. When the driver is 40 meters down the road, the officer gets his car up to 25 meters per second to pursue the offender. How far from the sign does the officer catch up to the speeder?

The program in Figure 6.12 solves the problem by simulation. It is identical to the one in Figure 2.13. The values of cop and driver are the positions of the two motorists, initialized to 0 and 40, respectively. Each execution of the do loop represents one second of elapsed time, during which the officer travels 25 meters and the driver 20, until the officer catches the driver.

A do statement has its test at the bottom of the loop. In this program, the compiler translates the while test to the sequence LDA, CPA, BRLT. BRLT executes the branch if N is set to 1. Because CPA computes the difference, cop - driver, N will be 1 if

\[ \text{cop} - \text{driver} < 0 \]
### High-Order Language

```cpp
#include <iostream>
using namespace std;

int cop;
int driver;

int main () {
    cop = 0;
    driver = 40;
    do {
        cop += 25;
        driver += 20;
    } while (cop < driver);
    cout << cop;
    return 0;
}
```

### Assembly Language

```assembly
0000 040007 BR main
0003 0000 cop: .BLOCK 2 ; global variable
0005 0000 driver: .BLOCK 2 ; global variable

0007 C00000 main: LDA 0,i ; cop = 0
000A E10003 STA cop,d
000D C00028 LDA 40,i ; driver = 40
0010 E10005 STA driver,d
0013 C10003 do: LDA cop,d ; cop += 25
0016 700019 ADDA 25,i
0019 E10003 STA cop,d
001C C10005 LDA driver,d ; driver += 20
001F 700014 ADDA 20,i
0022 E10005 STA driver,d
0025 C10003 while: LDA cop,d ; while (cop < driver)
0028 B10005 CPA driver,d
002B 080013 BRLT do
002E 390003 DECO cop,d ; cout << cop
0031 00 STOP
0032 .END
```

---

**Figure 6.12**

The `do` statement at level HOL6 and level Asmb5.
that is, if

cop < driver

That is the condition under which the loop should repeat. Figure 6.13 shows the structure of the \texttt{do} statement at levels 6 and 5.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{structure.png}
\caption{The structure of the \texttt{do} statement at level \texttt{Asmb5}.}
\end{figure}

\section*{Translating the For Loop}

\texttt{for} statements are similar to \texttt{while} statements because the test for both is at the top of the loop. The compiler must generate code to initialize and to increment the control variable. The program in Figure 6.14 shows how a compiler would generate code for the \texttt{for} statement. It translates the \texttt{for} statement into the following sequence at level \texttt{Asmb5}:

\begin{itemize}
  \item Initialize the control variable.
  \item Test the control variable.
  \item Execute the loop body.
  \item Increment the control variable.
  \item Branch to the test.
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{for_structure.png}
\caption{The \texttt{for} statement at level \texttt{HOL6} and level \texttt{Asmb5}.}
\end{figure}

\section*{High-Order Language}

```cpp
#include <iostream>
using namespace std;

int main () {
    int i;
    for (i = 0; i < 3; i++) {
        cout << "i = " << i << endl;
    }
    cout << "i = " << i << endl;
    return 0;
}
```
In this program, \texttt{CPA} computes the difference, \( i - 3 \). \texttt{BRGE} branches out of the loop if \( N \) is 0, that is, if

\[ i - 3 \geq 0 \]

or, equivalently,

\[ i \geq 3 \]

The body executes once each for \( i \) having the values 0, 1, and 2. The last time through the loop, \( i \) increments to 3, which is the value written by the output statement following the loop.

\section*{Spaghetti Code}

At the assembly level, a programmer can write control structures that do not correspond to the control structures in C++. Figure 6.15 shows one possible flow of control that is not directly possible in many level-HOL6 languages. Condition \( C1 \) is
tested, and if it is true, a branch is taken to the middle of a loop whose test is \( C_2 \).
This control flow cannot be written directly in C++.

Assembly language programs generated by a compiler are usually longer than
programs written by humans directly in assembly language. Not only that, but they
often execute more slowly. If human programmers can write shorter, faster assembly
language programs than compilers, why does anyone program in a high-order
language? One reason is the ability of the compiler to perform type checking, as
mentioned in Chapter 5. Another is the additional burden of responsibility that is
placed on the programmer when given the freedom of using primitive branching
instructions. If you are not careful when you write programs at level Asmb5, the
branching instructions can get out of hand, as the next program shows.

The program in Figure 6.16 is an extreme example of the problem that can occur
with unbridled use of primitive branching instructions. It is difficult to understand
because of its lack of comments and indentation and its inconsistent branching style.
Actually, the program performs a very simple task. Can you discover what it does?

```
0000 040009    BR    main
0003 0000    n1:    .BLOCK 2
0005 0000    n2:    .BLOCK 2
0007 0000    n3:    .BLOCK 2
    ;
0009 310005    main:    DECI n2,d
000C 310007    DECI n3,d
000F C10005    LDA n2,d
0012 B10007    CPA n3,d
0015 08002A    BRLT L1
0018 310003    DECI n1,d
001B C10003    LDA n1,d
001E B10007    CPA n1,d
0021 080074    BRLT L7
0024 040065    BR L6
0027 E10007    STA n3,d
002A 310003    L1:    DECI n1,d
002D C10005    LDA n2,d
0030 B10003    CPA n1,d
0033 080053    BRLT L5
0036 390003    DECO n1,d
0039 390005    DECO n2,d
003C 390007    L2:    DECO n3,d
003F 00    STOP
0040 390005    L3:    DECO n2,d
0043 390007    DECO n3,d
0046 040081    BR L9
```

Figure 6.16
A mystery program.
The body of an if statement or a loop in C++ is a block of statements, sometimes contained in a compound statement delimited by braces {}. Additional if statements and loops can be nested entirely within these blocks. Figure 6.17(a) pictures this situation schematically. A flow of control that is limited to nestings of the if/else, switch, while, do, and for statements is called structured flow of control.

The branches in the mystery program do not correspond to the structured control constructs of C++. Although the program’s logic is correct for performing its intended task, it is difficult to decipher because the branching statements branch all over the place. This kind of program is called spaghetti code. If you draw an arrow from each branch statement to the statement to which it branches, the picture looks rather like a bowl of spaghetti, as shown in Figure 6.17(b).

It is often possible to write efficient programs with unstructured branches. Such programs execute faster and require less memory for storage than if they were written in a high-order language with structured flow of control. Some specialized applications require this extra measure of efficiency and are therefore written directly in assembly language.

Balanced against this savings in execution time and memory space is difficulty in comprehension. When programs are hard to understand, they are hard to write, debug, and modify. The problem is economic. Writing, debugging, and modifying
are all human activities, which are labor intensive and, therefore, expensive. The question you must ask is whether the extra efficiency justifies the additional expense.

**Flow of Control in Early Languages**

Computers had been around for many years before structured flow of control was discovered. In the early days there were no high-order languages. Everyone programmed in assembly language. Computer memories were expensive, and CPUs were slow by today’s standards. Efficiency was all-important. Because a large body of software had not yet been generated, the problem of program maintenance was not appreciated.

The first widespread high-order language was FORTRAN, developed in the 1950s. Because people were used to dealing with branch instructions, they included them in the language. An unconditional branch in FORTRAN is

\[
\text{GOTO 260}
\]

where 260 is the statement number of another statement. It is called a goto statement. A conditional branch is

\[
\text{IF (NUMBER .GE. 100) GOTO 500}
\]

where .GE. means “is greater than or equal to.” This statement compares the value of variable NUMBER with 100. If it is greater than or equal to 100, the next statement executed is the one with a statement number of 500. Otherwise the statement after the IF is executed.

FORTRAN’s conditional IF is a big improvement over level-Asmb5 branch instructions. It does not require a separate compare instruction to set the status bits. But notice how the flow of control is similar to level-Asmb5 branching: If the test is true, do the GOTO. Otherwise continue to the next statement.

As people developed more software, they noticed that it would be convenient to group statements into blocks for use in if statements and loops. The most
notable language to make this advance was ALGOL-60, developed in 1960. It was the first widespread block-structured language, although its popularity was limited mainly to Europe.

**The Structured Programming Theorem**

The preceding sections show how high-level structured control statements translate into primitive branch statements at a lower level. They also show how you can write branches at the lower level that do not correspond to the structured constructs. That raises an interesting and practical question: Is it possible to write an algorithm with goto statements that will perform some processing that is impossible to perform with structured constructs? That is, if you limit yourself to structured flow of control, are there some problems you will not be able to solve that you could solve if unstructured goto’s were allowed?

Corrado Bohm and Giuseppe Jacopini answered this important question in a computer science journal article in 1966.¹ They proved mathematically that any algorithm containing goto’s, no matter how complicated or unstructured, can be written with only nested `if` statements and `while` loops. Their result is called the structured programming theorem.

Bohm and Jacopini’s paper was highly theoretical. It did not attract much attention at first because programmers generally had no desire to limit the freedom they had with goto statements. Bohm and Jacopini showed what could be done with nested `if` statements and `while` loops, but left unanswered why programmers would want to limit themselves that way.

People experimented with the concept anyway. They would take an algorithm in spaghetti code and try to rewrite it using structured flow of control without goto statements. Usually the new program was much clearer than the original. Occasionally it was even more efficient.

**The Goto Controversy**

Two years after Bohm and Jacopini’s paper appeared, Edsger W. Dijkstra of the Technological University at Eindhoven, the Netherlands, wrote a letter to the editor of the same journal in which he stated his personal observation that good programmers used fewer goto’s than poor programmers.²

---


Edsger Dijkstra

Born to a Dutch chemist in Rotterdam in 1930, Dijkstra grew up with a formalist predilection toward the world. While studying at the University of Leiden in the Netherlands, Dijkstra planned to take up physics as his career. But his father heard about a summer course on computing in Cambridge, England, and Dijkstra jumped aboard the computing bandwagon just as it was gathering speed around 1950.

One of Dijkstra’s most famous contributions to programming was his strong advocacy of structured programming principles, as exemplified by his famous letter that disparaged the goto statement. He developed a reputation for speaking his mind, often in inflammatory or dramatic ways that most of us couldn’t get away with. For example, Dijkstra once remarked that “the use of COBOL cripples the mind; its teaching should therefore be regarded as a criminal offence.” Not one to single out only one language for his criticism, he also said that “it is practically impossible to teach good programming to students that have had a prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration.”

Besides his work in language design, Dijkstra is also noted for his work in proofs of program correctness. The field of program correctness is an application of mathematics to computer programming. Researchers are trying to construct a language and proof technique that might be used to certify unconditionally that a program will perform according to its specifications—entirely free of bugs. Needless to say, whether your application is customer billing or flight control systems, this would be an extremely valuable claim to make about a program.

Dijkstra worked in practically every area within computer science. He invented the semaphore, described in Chapter 8 of this book, and invented a famous algorithm to solve the shortest path problem. In 1972 the Association for Computing Machinery acknowledged Dijkstra’s rich contributions to the field by awarding him the distinguished Turing Award. Dijkstra died after a long struggle with cancer in 2002 at his home in Nuenen, the Netherlands.

“The question of whether computers can think is like the question of whether submarines can swim.”
—Edsger Dijkstra

In his opinion, a high density of goto’s in a program indicated poor quality. He stated in part:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of goto statements in the programs they produce. More recently I discovered why the use of the goto state-
ment has such disastrous effects, and I became convinced that the goto statement should be abolished from all “higher level” programming languages (i.e., everything except, perhaps, plain machine code). . . . The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program.

To justify these statements, Dijkstra developed the idea of a set of coordinates that are necessary to describe the progress of the program. When a human tries to understand a program, he must maintain this set of coordinates mentally, perhaps unconsciously. Dijkstra showed that the coordinates to be maintained with structured flow of control were vastly simpler than those with unstructured goto’s. Thus he was able to pinpoint the reason that structured flow of control is easier to understand.

Dijkstra acknowledged that the idea of eliminating goto’s was not new. He mentioned several people who influenced him on the subject, one of whom was Niklaus Wirth, who had worked on the ALGOL-60 language.

Dijkstra’s letter set off a storm of protest, now known as the famous goto controversy. To theoretically be able to program without goto was one thing. But to advocate that goto be abolished from high-order languages such as FORTRAN was altogether something else.

Old ideas die hard. However, the controversy has died down and it is now generally recognized that Dijkstra was, in fact, correct. The reason is cost. When software managers began to apply the structured flow of control discipline, along with other structured design concepts, they found that the resulting software was much less expensive to develop, debug, and maintain. It was usually well worth the additional memory requirements and extra execution time.

FORTRAN 77 is a more recent version of FORTRAN standardized in 1977. The goto controversy influenced its design. It contains a block style \texttt{IF} statement with an \texttt{ELSE} part similar to C++. For example,

\begin{verbatim}
IF (NUMBER .GE. 100) THEN
  Statement 1
ELSE
  Statement 2
ENDIF
\end{verbatim}

You can write the \texttt{IF} statement in FORTRAN 77 without goto.

One point to bear in mind is that the absence of goto’s in a program does not guarantee that the program is well structured. It is possible to write a program with three or four nested \texttt{if} statements and \texttt{while} loops when only one or two are necessary. Also, if a language at any level contains only goto statements to alter the flow of control, they can always be used in a structured way to implement \texttt{if} statements and \texttt{while} loops. That is precisely what a C++ compiler does when it translates a program from level HOL6 to level Asmb5.
6.3 Procedure Calls and Parameters

A C++ procedure call changes the flow of control to the first executable statement in the procedure. At the end of the procedure, control returns to the statement following the procedure call. The compiler implements procedure calls with the `CALL` instruction, which has a mechanism for storing the return address on the run-time stack. It implements the return to the calling statement with `RETn`, which uses the saved return address on the run-time stack to determine which instruction to execute next.

Translating a Procedure Call

Figure 6.18 shows how a compiler translates a procedure call without parameters. The program outputs three triangles of asterisks.

The `CALL` instruction pushes the content of the program counter onto the run-time stack, and then loads the operand into the program counter. Here is the RTL specification of the `CALL` instruction:

$$
\text{SP} \leftarrow \text{SP} - 2; \text{Mem[SP]} \leftarrow \text{PC}; \text{PC} \leftarrow \text{Opernd}
$$

In effect, the return address for the procedure call is pushed onto the stack and a branch to the procedure is executed.

As with the branch instructions, `CALL` usually executes in the immediate addressing mode, in which case the operand is the operand specifier. If you do not specify the addressing mode, the Pep/8 assembler will assume immediate addressing.

Figure 5.2 shows that the `RETn` instruction has a three-bit `nnn` field. In general, a procedure can have any number of local variables. There are eight versions of the `RETn` instruction, namely `RET0`, `RET1`, ..., `RET7`, where `n` is the number of bytes occupied by the local variables in the procedure. Procedure `printTri` in Figure 6.18 has no local variables. That is why the compiler generated the `RET0` instruction at 0015. Here is the RTL specification of `RETn`:

$$
\text{SP} \leftarrow \text{SP} + n; \text{PC} \leftarrow \text{Mem[SP]}; \text{SP} \leftarrow \text{SP} + 2
$$

First, the instruction deallocates storage for the local variables by adding `n` to the stack pointer. After the deallocation, the return address should be on top of the run-time stack. Then, the instruction moves the return address from the top of the stack into the program counter. Finally, it adds two to the stack pointer, which completes the pop operation. Of course, it is possible for a procedure to have more than seven bytes of local variables. In that case, the compiler would generate an `ADDSP` instruction to deallocate the storage for the local variables.

In Figure 6.18,
High-Order Language

```cpp
#include <iostream>
using namespace std;

void printTri () {
    cout << "*" << endl;
    cout << "**" << endl;
    cout << "***" << endl;
    cout << "****" << endl;
}

int main () {
    printTri ();
    printTri ();
    printTri ();
    printTri ();
    return 0;
}
```

Assembly Language

```
0000 04001F          BR main

;******** void printTri ()
0003 410016 printTri:STRO msg1,d ;cout << "*"
0006 50000A CHARO \n,i ; << endl
0009 410018 STRO msg2,d ;cout << "**"
000C 50000A CHARO \n,i ; << endl
000F 41001B STRO msg3,d ;cout << "***"
0012 50000A CHARO \n,i ; << endl
0015 58 RET0
0016 2A00 msg1: .ASCII "\x00"
0018 2A2A00 msg2: .ASCII "\x00"
001B 2A2A2A msg3: .ASCII "\x00"

;******** int main ()
001F 160003 main: CALL printTri ;printTri ()
0022 160003 CALL printTri ;printTri ()
0025 160003 CALL printTri ;printTri ()
0028 00 STOP
0029 .END
```

Figure 6.18

A procedure call at level HOL6 and level Asmb5.
puts 001F into the program counter. The next statement to execute is, therefore, the one at 001F, which is the first `CALL` instruction. The discussion of the program in Figure 6.1 explains how the stack pointer is initialized to FBCF. Figure 6.19 shows the runtime stack before and after execution of the first `CALL` statement. As usual, the initial value of the stack pointer is FBCF.

(a) Before execution of the first `CALL`. 
(b) After execution of the first `CALL`.

The operations of `CALL` and `REtn` crucially depend on the von Neumann execution cycle: fetch, decode, increment, execute, repeat. In particular, the increment step happens before the execute step. As a consequence, the statement that is executing is not the statement whose address is in the program counter. It is the statement that was fetched before the program counter was incremented and that is now contained in the instruction register. Why is that so important in the execution of `CALL` and `REtn`?

Figure 6.19(a) shows the content of the program counter as 0022 before execution of the first `CALL` instruction. It is not the address of the first `CALL` instruction, which is 001F. Why not? Because the program counter was incremented to 0022 before execution of the `CALL`. Therefore, during execution of the first `CALL` instruction the program counter contains the address of the instruction in main memory located just after the first `CALL` instruction.

What happens when the first `CALL` executes? First, \( SP \leftarrow SP - 2 \) subtracts two from SP, giving it the value FBCD. Then, \( Mem[SP] \leftarrow PC \) puts the value of the program counter, 0022, into main memory at address FBCD, that is, on top of the run-time stack. Finally, \( PC \leftarrow Oprnd \) puts 0003 into the program counter, because the operand specifier is 0003 and the addressing mode is immediate. The result is Figure 6.19(b).

The von Neumann cycle continues with the next fetch. But now the program counter contains 0003. So, the next instruction to be fetched is the one at address 0003, which is the first instruction of the `printTri` procedure. The output instructions of the procedure execute, producing the pattern of a triangle of asterisks.

Eventually the `REtn` instruction at 0015 executes. Figure 6.20(a) shows the content of the program counter as 0016 just before execution of `REtn`. This might
The reason increment must come before execute in the von Neumann execution cycle seems strange, because 0016 is not even the address of an instruction. It is the address of the string "**\x00". Why? Because RET0 is a unary instruction and the CPU incremented the program counter by one. The first step in the execution of RET0 is SP ← SP + n, which adds zero to SP because n is zero. Then, PC ← Mem[SP] puts 0022 into the program counter. Finally, SP ← SP + 2 changes the stack pointer back to FBCF.

The von Neumann cycle continues with the next fetch. But now the program counter contains the address of the second CALL instruction. The same sequence of events happens as with the first call, producing another triangle of asterisks in the output stream. The third call does the same thing, after which the STOP instruction executes. Note that the value of the program counter after the STOP instruction executes is 0029 and not 0028, which is the address of the STOP instruction.

Now you should see why increment comes before execute in the von Neumann execution cycle. To store the return address on the run-time stack, the CALL instruction needs to store the address of the instruction following the CALL. It can only do that if the program counter has been incremented before the CALL statement executes.

### Translating Call-By-Value Parameters with Global Variables

The allocation process when you call a void function in C++ is

- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

At level HOL6, the instructions that perform these operations on the stack are hidden. The programmer simply writes the function call, and during execution the stack allocation occurs automatically.

At the assembly level, however, the translated program must contain explicit instructions for the allocation. The program in Figure 6.21, which is identical to the program in Figure 2.15, is a level-HOL6 program that prints a bar chart, and the program’s corresponding level-Asmb5 translation. It shows the level-Asmb5 statements, not explicit at level HOL6, that are required to push the parameters.

```cpp
#include <iostream>
using namespace std;

int numPts;
int value;
int i;
```

Figure 6.21
Call-by-value parameters with global variables.
void printBar (int n) {
    int j;
    for (j = 1; j <= n; j++) {
        cout << '*';
    }
    cout << endl;
}

int main () {
    cin >> numPts;
    for (i = 1; i <= numPts; i++) {
        cin >> value;
        printBar (value);
    }
    return 0;
}

Assembly Language

0000 04002B BR main
0003 0000 numPts: .BLOCK 2 ;global variable
0005 0000 value: .BLOCK 2 ;global variable
0007 0000 i: .BLOCK 2 ;global variable

;****** void printBar (int n)
n: .EQUATE 4 ;formal parameter
j: .EQUATE 0 ;local variable
0009 680002 printBar:SUBSP 2,i ;allocate local
000C C00001 LDA 1,i ;for (j = 1
000F E30000 STA j,s
0012 B30004 for1: CPA n,s ;j <= n
0015 100027 BRGT endFor1
0018 50002A CHARO '**',i ; cout << '**'
001B C30000 LDA j,s ;j++
001E 700001 ADDDA 1,i
0021 E30000 STA j,s
0024 040012 BR for1
0027 50000A endFor1: CHARO '\n',i ;cout << endl
002A 5A RET2 ;deallocate local, pop retAddr
The calling procedure is responsible for pushing the actual parameters and executing `CALL`, which pushes the return address onto the stack. The called procedure is responsible for allocating storage on the stack for its local variables. After the called procedure executes, it must deallocate the storage for the local variables, and then pop the return address by executing `RETn`. Before the calling procedure can continue, it must deallocate the storage for the actual parameters.

In summary, the calling and called procedures do the following:

- Calling pushes actual parameters (executes `SUBSP`).
- Calling pushes return address (executes `CALL`).
- Called allocates local variables (executes `SUBSP`).
- Called executes its body.
- Called deallocates local variables and pops return address (executes `RETn`).
- Calling pops actual parameters (executes `ADDSP`).

Note the symmetry of the operations. The last two operations undo the first three operations in reverse order. That order is a consequence of the last-in, first-out property of the stack.

The global variables in the level-HOL6 main program—`numPts`, `value`, and `i`—correspond to the identical level-Asmb5 symbols, whose symbol values are 0003, 0005, and 0007, respectively. These are the addresses of the memory cells that will hold the run-time values of the global variables. Figure 6.22(a) shows the
global variables on the left with their symbols in place of their addresses. The values for the global variables are the ones after
\[ \text{cin} \gg \text{value}; \]
executes for the first time.

What do the formal parameter, \( n \), and the local variable, \( j \), correspond to at level Asmb5? Not absolute addresses, but stack-relative addresses. Procedure printBar defines them with
\[
\begin{align*}
 n & : \text{.EQUATE} 4 \\
 j & : \text{.EQUATE} 0
\end{align*}
\]

Remember that .EQUATE does not generate object code. The assembler does not reserve storage for them at translation time. Instead, storage for \( n \) and \( j \) is allocated on the stack at run time. The decimal numbers 4 and 0 are the stack offsets appropriate for \( n \) and \( j \) during execution of the procedure, as Figure 6.22(b) shows. The procedure refers to them with stack-relative addressing.

The statements that correspond to the procedure call in the calling procedure are
\[
\begin{align*}
 \text{LDA} & \space \text{value,}d \\
 \text{STA} & \space -2,s \\
 \text{SUBSP} & \space 2,i \\
 \text{CALL} & \space \text{printBar} \\
 \text{ADDSP} & \space 2,i
\end{align*}
\]

Because the parameter is a global variable that is called by value, LDA uses direct addressing. That puts the run-time value of variable \( \text{value} \) in the accumulator, which STA then pushes onto the stack. The offset is –2 because \( \text{value} \) is a two-byte integer quantity, as Figure 6.22(a) shows.

The statements that correspond to the procedure call in the called procedure are
\[
\begin{align*}
 \text{SUBSP} & \space 2,i \\
 \cdot & \\
 \cdot & \\
 \cdot & \\
 \text{RET2}
\end{align*}
\]

The SUBSP subtracts 2 because the local variable, \( j \), is a two-byte integer quantity. Figure 6.22(a) shows the run-time stack just after the first input of global
variable value and just before the first procedure call. It corresponds directly to Figure 2.16(d) (page 49). Figure 6.22(b) shows the stack just after the procedure call and corresponds directly to Figure 2.16(g). Note that the return address, which is labeled ra1 in Figure 2.16, is here shown to be 0049, which is the assembly language address of the instruction following the CALL instruction.

The stack address of n is 4 because both j and the return address occupy two bytes on the stack. If there were more local variables, the stack address of n would be correspondingly greater. The compiler must compute the stack addresses from the number and size of the quantities on the stack.

In summary, to translate call-by-value parameters with global variables the compiler generates code as follows:

- To push the actual parameter, it generates a load instruction with direct addressing.
- To access the formal parameter, it generates instructions with stack-relative addressing.

### Translating Call-By-Value Parameters with Local Variables

The program in Figure 6.23 is identical to the one in Figure 6.21 except that the variables in main() are local instead of global. Although the program behaves like the one in Figure 6.21, the memory model and the translation to level Asmb5 are different.

```cpp
#include <iostream>
using namespace std;

void printBar (int n) {
    int j;
    for (j = 1; j <= n; j++) {
        cout << '*';
    }
    cout << endl;
}

int main () {
    int numPts;
    int value;
    int i;
    int main () {
    int numPts;
    int value;
    int i;
```
cin >> numPts;
for (i = 1; i <= numPts; i++) {
    cin >> value;
    printBar (value);
}
return 0;
}

Assembly Language

0000 040025 BR main

;****** void printBar (int n)
  n: .EQUATE 4 ;formal parameter
  j: .EQUATE 0 ;local variable
0003 680002 printBar:SUBSP 2,i ;allocate local
0006 C00001 LDA 1,i ;for (j = 1
0009 E30000 STA j,s
000C B30004 for1: CPA n,s ;j <= n
000F 100021 BRGT endFor1
0012 50002A CHARO ",i ; cout << 
0015 C30000 LDA j,s ;j++)
0018 700001 ADDA 1,i
001B E30000 STA j,s
001E 04000C BR for1
0021 50000A endFor1: CHARO \\
0024 5A RET2 ;deallocation local,

;****** main ()
  numPts: .EQUATE 4 ;local variable
  value: .EQUATE 2 ;local variable
  i: .EQUATE 0 ;local variable
0025 680006 main: SUBSP 6,i ;allocate locals
0028 330004 DECI numPts,s ;cin >> numPts
002B C00001 LDA 1,i ;for (i = 1
002E E30000 STA i,s
0031 B30004 for2: CPA numPts,s ;i <= numPts
0034 100055 BRGT endFor2
0037 330002 DECI value,s ; cin >> value
003A C30000 LDA value,s ; call by value
003D E3FFFE STA -2,s
0040 680002 SUBSP 2,i ; push parameter
0043 160003 CALL printBar ; push retAddr
0046 600002 ADDSP 2,i ; pop parameter
You can see that the versions of void function printTri at level HOL6 are identical in Figure 6.21 and Figure 6.23. Hence, it should not be surprising that the compiler generates identical object code for the two versions of printTri at level Asmb5. The only difference between the two programs is in the definition of main(). Figure 6.24(a) shows the allocation of numPts, value, and i on the runtime stack in the main program. Figure 6.24(b) shows the stack after printTri is called for the first time. Because value is a local variable, the compiler generates LDA value,s with stack-relative addressing to push the actual value of value into the stack cell of formal parameter n.

In summary, to translate call-by-value parameters with local variables the compiler generates code as follows:

- To push the actual parameter, it generates a load instruction with stack-relative addressing.
- To access the formal parameter, it generates instructions with stack-relative addressing.

## Translating Non-Void Function Calls

The allocation process when you call a function is

- Push storage for the returned value.
- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

---

**Figure 6.23**

(Continued)
Allocation for a non-void function call differs from that for a procedure (void function) call by the extra value that you must allocate for the returned function value.

Figure 6.25 shows a program that computes a binomial coefficient recursively and is identical to the one in Figure 2.24. It is based on Pascal’s triangle of coefficients, shown in Figure 2.25. The recursive definition of the binomial coefficient is

\[
\begin{align*}
    b(n,0) &= 1 \\
    b(k,k) &= 1 \\
    b(n,k) &= b(n-1,k) + b(n-1,k-1) \quad \text{for } 0 \leq k \leq n
\end{align*}
\]

The function tests for the base cases with an if statement, using the OR boolean operator. If neither base case is satisfied, it calls itself recursively twice—once to compute \(b(n - 1, k)\) and once to compute \(b(n - 1, k - 1)\). Figure 6.26 shows the run-time stack produced by a call from the main program with actual parameters \((3, 1)\). The function is called twice more with parameters \((2, 1)\) and \((1, 1)\), followed by a return. Then a call with parameters \((1, 0)\) is executed, followed by a second return, and so on. Figure 6.26 shows the run-time stack at the assembly level immediately after the second return. It corresponds directly to the level-HOL6 diagram of Figure 2.28(g) (page 65). The return address labeled ra2 in Figure 2.28 is 0031 in Figure 6.26, the address of the instruction after the first \texttt{CALL} in the function. Similarly, the address labeled ra1 in Figure 2.28 is 007A in Figure 6.26.

\begin{verbatim}
High-Order Language
#include <iostream>
using namespace std;

int binCoeff (int n, int k) {
    int y1, y2;
    if ((k == 0) || (n == k)) {
        return 1;
    }
    else {
        y1 = binCoeff (n - 1, k); // ra2
        y2 = binCoeff (n - 1, k - 1); // ra3
        return y1 + y2;
    }
}

int main () {
    cout << "binCoeff (3, 1) = " << binCoeff (3, 1); // ra1
    cout << endl;
    return 0;
}
\end{verbatim}

Figure 6.25
A recursive nonvoid function at level HOL6 and level Asmb5.
Assembly Language

0000 040065   BR main

; int binomCoeff (int n, int k)
retVal: .EQUATE 10 ; returned value
n: .EQUATE 8 ; formal parameter
k: .EQUATE 6 ; formal parameter
y1: .EQUATE 2 ; local variable
y2: .EQUATE 0 ; local variable

0003 680004 binCoeff: SUBSP 4, i ; allocate locals
0006 C30006 if: LDA k, s ; if ((k == 0)
0009 0A0015 BREQ then
000C C30008 LDA n, s ; || (n == k))
000F B30006 CPA k, s
0012 0C001C BRNE else
0015 C00001 then: LDA 1, i ; return 1
0018 E3000A STA retVal, s
001B 5C RET4 ; deallocate locals, pop retAddr
001C C30008 else: LDA n, s ; push n - 1
001F 800001 SUBA 1, i
0022 E3FFFC STA -4, s
0025 C30006 LDA k, s ; push k
0028 E3FFFA STA -6, s
002B 680006 SUBSP 6, i ; push params and retVal
002E 160003 CALL binCoeff ; binomCoeff (n - 1, k)
0031 600006 ra2: ADDSP 6, i ; pop params and retVal
0034 C3FFFE LDA -2, s ; y1 = binomCoeff (n - 1, k)
0037 E30002 STA y1, s
003A C30008 LDA n, s ; push n - 1
003D 800001 SUBA 1, i
0040 E3FFFC STA -4, s
0043 C30006 LDA k, s ; push k - 1
0046 800001 SUBA 1, i
0049 E3FFFA STA -6, s
004C 680006 SUBSP 6, i ; push params and retVal
004F 160003 CALL binCoeff ; binomCoeff (n - 1, k - 1)
0052 600006 ra3: ADDSP 6, i ; pop params and retVal
0055 C3FFFE LDA -2, s ; y2 = binomCoeff (n - 1, k - 1)
0058 E30000 STA y2, s
005B C30002 LDA y1, s ; return y1 + y2
005E 730000 ADDA y2, s
0061 E3000A STA retVal, s
0064 5C endIf: RET4 ; deallocate locals, pop retAddr
At the start of the main program when the stack pointer has its initial value, the first actual parameter has a stack offset of –4, and the second has a stack offset of –6. In a procedure call (a void function), these offsets would be –2 and –4, respectively. Their magnitudes are greater by 2 because of the two-byte value returned on the stack by the function. The `SUBSP` instruction at 0074 allocates six bytes, two each for the actual parameters and two for the returned value.

When the function returns control to `ADDSP` at 007A, the value it returns will be on the stack below the two actual parameters. `ADDSP` pops the parameters and returned value by adding 6 to the stack pointer, after which it points to the cell directly below the returned value. So `DECO` outputs the value with stack-relative addressing and an offset of –2.

The function calls itself by allocating actual parameters according to the standard technique. For the first recursive call, it computes \( n - 1 \) and \( k \) and pushes those values onto the stack along with storage for the returned value. After the return, the sequence

\[
\text{ADDSP } 6, i \quad ; \text{pop params and retVal} \\
\text{LDA } -2, s \quad ; y_1 = \text{binomCoeff}(n - 1, k) \\
\text{STA } y_1, s
\]

pops the two actual parameters and returned value and assigns the returned value to \( y_1 \). For the second call, it pushes \( n - 1 \) and \( k - 1 \) and assigns the returned value to \( y_2 \) similarly.
C++ provides call-by-reference parameters so that the called procedure can change the value of the actual parameter in the calling procedure. Figure 2.19 shows a program at level HOL6 that uses call by reference to put two global variables $a$ and $b$ in order. Figure 6.27 shows the same program together with the object program that a compiler would produce.

```
High-Order Language

#include <iostream>
using namespace std;

int a, b;

void swap (int& r, int& s) {
    int temp;
    temp = r;
    r = s;
    s = temp;
}

void order (int& x, int& y) {
    if (x > y) {
        swap (x, y);
    }  // ra2
}

int main () {
    cout << "Enter an integer: ";
    cin >> a;
    cout << "Enter an integer: ";
    cin >> b;
    order (a, b);
    cout << "Ordered they are: " << a << ", " << b << endl; // ra1
    return 0;
}
```

Figure 6.27
Call-by-reference parameters with global variables.
Assembly Language

0000 04003C BR main
0003 0000 a: .BLOCK 2 ;global variable
0005 0000 b: .BLOCK 2 ;global variable

;****** void swap (int& r, int& s)
r: .EQUATE 6 ;formal parameter
s: .EQUATE 4 ;formal parameter
temp: .EQUATE 0 ;local variable

0007 680002 swap: SUBSP 2,i ;allocate local
000A C40006 LDA r,sf ;temp = r
000D E30000 STA temp,s
0010 C40004 LDA s,sf ;r = s
0013 E40006 STA r,sf
0016 C30000 LDA temp,s ;s = temp
0019 E40004 STA s,sf
001C 5A RET2 ;deallocate local, pop retAddr

;****** void order (int& x, int& y)
x: .EQUATE 4 ;formal parameter
y: .EQUATE 2 ;formal parameter

001D C40004 order: LDA x,sf ;if (x > y)
0020 B40002 CPA y,sf
0023 06003B BRLE endIf
0026 C30004 LDA x,s ; push x
0029 E3FFFE STA -2,s
002C C30002 LDA y,s ; push y
002F E3FFFC STA -4,s
0032 680004 SUBSP 4,i ; push params
0035 160007 CALL swap ; swap (x, y)
0038 600004 ADDSP 4,i ; pop params
003B 58 endIf: RET0 ; pop retAddr

;****** main ()
003C 41006D main: STRO msg1,d ; cout << "Enter an integer: "
003F 310003 DECI a,d ; cin >> a
0042 41006D STRO msg1,d ; cout << "Enter an integer: "
0045 310005 DECI b,d ; cin >> b
0048 C00003 LDA a,i ; push the address of a
004B E3FFFE STA -2,s
004E C00005 LDA b,i ; push the address of b
0051 E3FFFC STA -4,s
0054 680004 SUBSP 4,i ; push params

Figure 6.27 (Continued)
The main program calls a procedure named `order` with two formal parameters `x` and `y` that are called by reference. `order` in turn calls `swap`, which makes the actual exchange. `swap` has call-by-reference parameters `r` and `s`. Parameter `r` refers to `s`, and `s` refers to `a`. The programmer used call by reference so that when procedure `swap` changes `r` it really changes `a`, because `r` refers to `a` (via `s`).

Parameters called by reference differ from parameters called by value in C++ because the actual parameter provides a reference to a variable in the calling routine instead of a value. At the assembly level, the code that pushes the actual parameter onto the stack pushes the address of the actual parameter. When the actual parameter is a global variable, its address is available as the value of its symbol. So, the code to push the address of a global variable is a load instruction with immediate addressing. In Figure 6.27, the code to push the address of `a` is

```
LDA a,i ;push the address of a
```

The value of the symbol `a` is 0003, the address of where the value of `a` is stored. The machine code for this instruction is

```
C00003
```

`C0` is the instruction specifier for the load accumulator instruction with addressing-aaa field of 000 to indicate immediate addressing. With immediate addressing, the operand specifier is the operand. Consequently, this instruction loads 0003 into the accumulator. The following instruction pushes it onto the run-time stack.

Similarly, the code to push the address of `b` is

```
LDA b,i ;push the address of b
```
The machine code for this instruction is

\texttt{C0005}

where \texttt{0005} is the address of \texttt{b}. This instruction loads \texttt{0005} into the accumulator with immediate addressing, after which the next instruction puts it on the run-time stack.

In Figure 6.27 at \texttt{0026}, procedure \texttt{order} calls \texttt{swap} (\texttt{x}, \texttt{y}). It must push \texttt{x} onto the run-time stack. \texttt{x} is called by reference. Consequently, the address of \texttt{x} is on the run-time stack. The corresponding formal parameter \texttt{r} is also called by reference. Consequently, procedure \texttt{swap} expects the address of \texttt{r} to be on the run-time stack. Procedure \texttt{order} simply transfers the address for \texttt{swap} to use. The statement

\texttt{LDA x,s ;push x}

at \texttt{0026} uses stack-relative addressing to put the address in the accumulator. The next instruction puts it on the run-time stack.

In procedure \texttt{order}, however, the compiler must translate

\texttt{temp = r}

It must load the value of \texttt{r} into the accumulator, and then store it in \texttt{temp}. How does the called procedure access the value of a formal parameter whose address is on the run-time stack? It uses stack-relative deferred addressing.

Remember that the relation between the operand and the operand specifier with stack-relative addressing is

\texttt{Oprnd = Mem [SP + OprndSpec]} \hspace{1cm} \textit{Stack-relative addressing}

The operand is on the run-time stack. But with call-by-reference parameters, the address of the operand is on the run-time stack. The relation between the operand and the operand specifier with stack-relative deferred addressing is

\texttt{Oprnd = Mem [Mem [SP + OprndSpec]]} \hspace{1cm} \textit{Stack-relative deferred addressing}

In other words, \texttt{Mem [SP + OprndSpec]} is the address of the operand, rather than the operand itself.

At lines \texttt{000A} and \texttt{000D}, the compiler generates the following object code to translate the assignment statement:

\texttt{LDA r,sf}
\texttt{STA temp,s}

The letters \texttt{sf} with the load instruction indicate stack-relative deferred addressing.

The object code for the load instruction is

\texttt{C40006}
6.3 Procedure Calls and Parameters

(a) The run-time stack at level HOL6.

(b) The run-time stack at level Asmb5.

Figure 6.28 The run-time stack for Figure 6.27 at level HOL6 and level Asmb5.

0006 is the stack relative address of parameter r, as Figure 6.28(b) shows. It contains 0003, the address of a. The load instruction loads 7, which is the value of a, into the accumulator. The store instruction puts it in temp on the stack.

The next assignment statement in procedure swap

\[ r = s; \]

has parameters on both sides of the assignment operator. The compiler generates LDA to load the value of s and STA to store the value to r, both with stack-relative addressing.

```
LDA s, sf
STA r, sf
```

In summary, to translate call-by-reference parameters with global variables the compiler generates code as follows:

- To push the actual parameter, it generates a load instruction with immediate addressing.
- To access the formal parameter, it generates instructions with stack-relative deferred addressing.

---

Translating Call-By-Reference Parameters with Local Variables

Figure 6.29 shows a program that computes the perimeter of a rectangle given its width and height. The main program prompts the user for the width and the height, which it inputs into two local variables named width and height. A third local variable is named perim. The main program calls a procedure (a void function) named rect passing width and height by value and perim by reference. The figure shows the input and output when the user enters 8 for the width and 5 for the height.
High-Order Language

#include <iostream>
using namespace std;

void rect (int& p, int w, int h) {
    p = (w + h) * 2;
}

int main () {
    int perim, width, height;
    cout << "Enter width: ";
    cin >> width;
    cout << "Enter height: ";
    cin >> height;
    rect (perim, width, height);
    // ra1
    cout << "perim = " << perim << endl;
    return 0;
}

Assembly Language

0000  04000E          BR      main
;
;****** void rect (int & p, int w, int h)
p:    .EQUATE 6          ;formal parameter
w:    .EQUATE 4          ;formal parameter
h:    .EQUATE 2          ;formal parameter
0003  C30004 rect:    LDA     w,s        ;p = (w + h) * 2
0006  730002          ADDA    h,s
0009  1C              ASLA
000A  E40006          STA     p,sf
000D  58     endIf:   RET0               ;pop retAddr
;
;****** main()
perim: .EQUATE 4       ;local variable
width: .EQUATE 2       ;local variable
height: .EQUATE 0      ;local variable
000E  680006 main:    SUBSP   6,i        ;allocate locals
0011  410046 STRO    msg1,d     ;cout <= "Enter width: "
0014  330002 DECI    width,s     ;cin >> width
0017  410054 STRO    msg2,d     ;cout <= "Enter height: "

Figure 6.29
Call-by-reference parameters with local variables.
Figure 6.30 shows the run-time stack at level HOL6 for the program. Compare it to Figure 6.28(a) for a program with global variables that are called by reference. In that program, formal parameters \(x\), \(y\), \(r\), and \(s\) refer to global variables \(a\) and \(b\). At level Asmb5, \(a\) and \(b\) are allocated at translation time with the `.EQUATE` dot command. Their symbols are their addresses. However, Figure 6.30 shows \(\text{perim}\) to be allocated on the run-time stack. The statement

\[
\text{main: SUBSP 6,i}
\]

at 000E allocates storage for \(\text{perim}\), and its symbol is defined by

\[
\text{perim: .EQUATE 4}
\]
Figure 6.31
The run-time stack for Figure 6.29 at level Asmb5.

Its symbol is not its absolute address. Its symbol is its address relative to the top of the run-time stack, as Figure 6.31(a) shows. Its absolute address is FBCD. Why? Because that is the location of the bottom of the application run-time stack, as the memory map in Figure 4.39 shows.

So, the compiler cannot generate code to push parameter perim with

```
LDA perim,i
STA -2,s
```

as it does for global variables. If it generated those instructions, procedure rect would modify the content of Mem [0004], and 0004 is not where perim is located.

The absolute address of perim is FBCD. Figure 6.31(a) shows that you could calculate it by adding the value of perim, 4, to the value of the stack pointer. Fortunately, there is a unary instruction MOVSPA that moves the content of the stack pointer to the accumulator. The RTL specification of MOVSPA is

```
A ← SP
```

The MOVSPA instruction

To push the address of perim the compiler generates the following instructions at 001D in Figure 6.29:

```
MOVSPA
ADDA perim,i
STA -2,s
```

The first instruction moves the content of the stack pointer to the accumulator. The accumulator then contains FBC9. The second instruction adds the value of perim, which is 4, to the accumulator, making it FBCD. The third instruction puts the address of perim in the cell for p, which procedure rect uses to store the perimeter. Figure 6.31(b) shows the result.

Procedure rect uses p as any procedure would use any call-by-reference parameter. Namely, at 000A it stores the value using stack-relative deferred addressing.

```
STA p, sf
```
With stack-relative deferred addressing, the address of the operand is on the stack. The operand is

\[ \text{Oprnd} = \text{Mem} [\text{Mem} [\text{SP} + \text{OprndSpec}]] \]

This instruction adds the stack pointer FBC1 to the operand specifier 6 yielding FBC7. Because \( \text{Mem} [\text{FBC7}] \) is FBCD, it stores the accumulator at \( \text{Mem} [\text{FBCD}] \).

In summary, to translate call-by-reference parameters with local variables the compiler generates code as follows:

- To push the actual parameter, it generates the unary MOVSPA instruction followed by the ADDA instruction with immediate addressing.
- To access the formal parameter, it generates instructions with stack-relative deferred addressing.

### Translating Boolean Types

Several schemes exist for storing boolean values at the assembly level. The one most appropriate for C++ is to treat the values true and false as integer constants. The values are

```c
const int true = 1;
const int false = 0;
```

Figure 6.32 is a program that declares a boolean function named `inRange`. The compiler translates the function as if true and false were declared as above.

```c
#include <iostream>
using namespace std;

const int LOWER = 21;
const int UPPER = 65;

bool inRange (int a) {
    if ((LOWER <= a) && (a <= UPPER)) {
        return true;
    }
    else {
        return false;
    }
}
```
```c
int main () {
    int age;
    cin >> age;
    if (inRange (age)) {
        cout << "Qualified\n";
    }
    else {
        cout << "Unqualified\n";
    }
    return 0;
}
```

**Assembly Language**

```
0000  040023          BR      main
true:    .EQUATE 1
false:   .EQUATE 0
;
LOWER:   .EQUATE 21         ;const int
UPPER:   .EQUATE 65         ;const int
;
;;****** bool inRange (int a)
retVal:  .EQUATE 4          ;returned value
a:       .EQUATE 2          ;formal parameter
0003  C00015 inRange: LDA     LOWER,i    ;if ((LOWER <= a)
0006  B30002 if:      CPA     a,s
0009  10001C          BRGT    else
000C  C30002          LDA     a,s        ;   && (a <= UPPER))
000F  B00041          CPA     UPPER,i
0012  E30004          STA     retVal,s
0015  C00001 then:    LDA     true,i     ;   return true
0018  58              RET0
001C  C00000 else:    LDA     false,i    ;   return false
001F  E30004          STA     retVal,s
0022  58              RET0
;
;;****** main ()
age:     .EQUATE 0          ;local variable
0023  680002 main:    SUBSP   2,i        ;allocate local
0026  330000          DECI    age,s      ;cin >> age
```
Representing false and true at the bit level as 0000 and 0001 (hex) has advantages and disadvantages. Consider the logical operations on boolean quantities and the corresponding assembly instructions \( \text{AND} \), \( \text{OR} \), and \( \text{NOT} \). If \( p \) and \( q \) are global boolean variables, then

\[
p \land q
\]

translates to

\[
\text{LDA } p,d \\
\text{ANDA } q,d
\]

If you AND 0000 and 0001 with this object code, you get 0000 as desired. The OR operation \( \lor \) also works as desired. The NOT operation is a problem, however, because if you apply NOT to 0000, you get FFFF instead of 0001. Also, applying NOT to 0001 gives FFFE instead of 0000. Consequently, the compiler does not generate the \( \text{NOT} \) instruction when it translates the C++ assignment statement

\[
p = !q
\]

Instead, it uses the exclusive-or operation XOR, which has the mathematical symbol \( \oplus \). It has the useful property that if you take the XOR of any bit value \( b \) with 0
you get b. And if you take the XOR of any bit value \( b \) with 1 you get the logical negation of \( b \). Mathematically,

\[
\begin{align*}
   b \oplus 0 &= b \\
   b \oplus 1 &= \neg b 
\end{align*}
\]

Unfortunately, the Pep/8 computer does not have an XOR instruction in its instruction set. If it did have such an instruction, the compiler would generate the following code for the above assignment:

```
LDA q,d 
XORA 0x0001,i 
STA p,d 
```

If \( q \) is false it has the representation 0000 (hex), and 0000 XOR 0001 equals 0001, as desired. Also, if \( q \) is true it has the representation 0001 (hex), and 0001 XOR 0001 equals 0000.

The type `bool` was not included in the C++ language standard until 1996. Older compilers use the convention that the boolean operators operate on integers. They interpret the integer value 0 as false and any nonzero integer value as true. To preserve backward compatibility, current C++ compilers maintain this convention.

### 6.4 Indexed Addressing and Arrays

A variable at level HOL6 is a memory cell at level ISA3. A variable at level HOL6 is referred to by its name, at level ISA3 by its address. A variable at level Asmb5 can be referred to by its symbolic name, but the value of that symbol is the address of the cell in memory.

What about an array of values? An array contains many elements, and so consists of many memory cells. The memory cells of the elements are contiguous; that is, they are adjacent to one another. An array at level HOL6 has a name. At level Asmb5, the corresponding symbol is the address of the first cell of the array. This section shows how the compiler translates source programs that allocate and access elements of one-dimensional arrays. It does so with several forms of indexed addressing.

Figure 6.33 summarizes all the Pep/8 addressing modes. Previous programs illustrate immediate, direct, stack-relative, and stack-relative deferred addressing. Programs with arrays use indexed, stack-indexed, or stack-indexed deferred addressing. The column labeled `aaa` shows the address-aaa field at level ISA3. The
column labeled Letters shows the assembly language designation for the addressing mode at level Asmb5. The column labeled Operand shows how the CPU determines the operand from the operand specifier (OprndSpec).

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>aaa</th>
<th>Letters</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>000</td>
<td>i</td>
<td>OprndSpec</td>
</tr>
<tr>
<td>Direct</td>
<td>001</td>
<td>d</td>
<td>Mem [OprndSpec]</td>
</tr>
<tr>
<td>Indirect</td>
<td>010</td>
<td>n</td>
<td>Mem [Mem [OprndSpec]]</td>
</tr>
<tr>
<td>Stack-relative</td>
<td>011</td>
<td>s</td>
<td>Mem [SP + OprndSpec]</td>
</tr>
<tr>
<td>Stack-relative deferred</td>
<td>100</td>
<td>sf</td>
<td>Mem [Mem [SP + OprndSpec]]</td>
</tr>
<tr>
<td>Indexed</td>
<td>101</td>
<td>x</td>
<td>Mem [OprndSpec + X]</td>
</tr>
<tr>
<td>Stack-indexed</td>
<td>110</td>
<td>sx</td>
<td>Mem [SP + OprndSpec + X]</td>
</tr>
<tr>
<td>Stack-indexed deferred</td>
<td>111</td>
<td>sx</td>
<td>Mem [Mem [SP + OprndSpec] + X]</td>
</tr>
</tbody>
</table>

**Translating Global Arrays**

Figure 6.34 shows a program at level HOL6 that declares a global array of four integers named `vector` and a global integer named `i`. The main program inputs four integers into the array with a `for` loop and outputs them in reverse order together with their indexes.

```cpp
#include <iostream>
using namespace std;

int vector[4];
int i;

int main () {
    for (i = 0; i < 4; i++) {
        cin >> vector[i];
    }
    for (i = 3; i >= 0; i--) {
        cout << i << ' ' << vector[i] << endl;
    }
    return 0;
}
```
Assembly Language

0000 04000D        BR       main
0003 000000 vector: .BLOCK 8 ;global variable
000000
0000
000B 0000  i: .BLOCK 2 ;global variable
;
;******* main ()
00D C80000 main: LDX 0,i ;for (i = 0
0010 E9000B        STX i,d
0013 B80004 for1:  CPX 4,i ; i < 4
0016 0E0029        BRGE endFor1
0019 1D            ASLX
001A 350003        DECI vector,x ; cin >> vector[i]
001D C9000B        LDX i,d ; i++
0020 780001        ADDX 1,i
0023 E9000B        STX i,d
0026 040013        BR for1
0029 C80003 endFor1: LDX 3,i ;for (i = 3
002C E9000B        STX i,d
002F B80000 for2:  CPX 0,i ; i >= 0
0032 08004E        BRLT endFor2
0035 39000B        DECO i,d ; cout << i
0038 500020        CHARO ,i ; " "
003B 1D            ASLX ; an integer is two bytes
003C 3D0003        DECO vector,x ; << vector[i]
003F 50000A        CHARO \n,i ; << endl
0042 C9000B        LDX i,d ; i--)
0045 880001        SUBX 1,i
0048 E9000B        STX i,d
004B 04002F        BR for2
004E 00        endFor2: STOP
004F                  .END

Input
60 70 80 90

Output
3 90
2 80
1 70
0 60
Figure 6.35 shows the memory allocation for integer \( i \) and array \( \text{vector} \). As with all global integers, the compiler translates
\[
\text{int } i;
\]
at level \( \text{HOL}6 \) as the following statement at level \( \text{Asmb}5 \):
\[
\text{i: .BLOCK 2}
\]
The two-byte integer is allocated at address 000B. The compiler translates
\[
\text{int vector[4];}
\]
at level \( \text{HOL}6 \) as the following statement at level \( \text{Asmb}5 \):
\[
\text{vector: .BLOCK 8}
\]
It allocates eight bytes because the array contains four integers, each of which is two bytes. The \( \text{.BLOCK} \) statement is at 0003. Figure 6.35 shows that 0003 is the address of the first element of the array. The second element is at 0005, and each element is at an address two bytes greater than the previous element.

The compiler translates the first \( \text{for} \) statement
\[
\text{for (i = 0; i < 4; i++)}
\]
as usual. It accesses \( i \) with direct addressing because \( i \) is a global variable. But how does it access \( \text{vector}[i] \)? It cannot simply use direct addressing, because the value of symbol \( \text{vector} \) is the address of the first element of the array. If the value of \( i \) is 2, it should access the third element of the array, not the first.

The answer is that it uses indexed addressing. With indexed addressing, the CPU computes the operand as
\[
\text{Oprnd = Mem[OprndSpec + X]}
\]
It adds the operand specifier and the index register and uses the sum as the address in main memory from which it fetches the operand.

In Figure 6.34, the compiler translates
\[
\text{cin >> vector[i];}
\]
at level \( \text{HOL}6 \) as
\[
\text{ASLX DECI vector,x}
\]
at level \( \text{Asmb}5 \). This is an optimized translation. The compiler analyzed the previous code generated and determined that the index register already contained the current value of \( i \). A nonoptimizing compiler would generate the following code:
\[
\text{LDX i,d ASLX DECI vector,x}
\]
Suppose the value of \( i \) is 2. \texttt{LDX} puts the value of \( i \) in the index register. (Or, an optimizing compiler determines that the current value of \( i \) is already in the index register.) \texttt{ASLX} multiplies the 2 times 2, leaving 4 in the index register. \texttt{DECI} uses indexed addressing. So, the operand is computed as

\[
\text{Mem [OprndSpec + X]} \\
\text{Mem [0003 + 4]} \\
\text{Mem [0007]}
\]

which Figure 6.35 shows is \texttt{vector[2]}. Had the array been an array of characters, the \texttt{ASLX} operation would be unnecessary because each character occupies only one byte. In general, if each cell in the array occupies \( n \) bytes, the value of \( i \) is loaded into the index register, multiplied by \( n \), and the array element is accessed with indexed addressing.

Similarly, the compiler translates the output of \texttt{vector[i]} as

\[
\text{ASLX} \\
\text{DECO vector, x}
\]

with indexed addressing.

In summary, to translate global arrays the compiler generates code as follows:

- It allocates storage for the array with \texttt{.BLOCK tot} where \( \text{tot} \) is the total number of bytes occupied by the array.
- It accesses an element of the array by loading the index into the index register, multiplying it by the number of bytes per cell, and using indexed addressing.

### Translating Local Arrays

Like all local variables, local arrays are allocated on the run-time stack during program execution. The \texttt{SUBSP} instruction allocates the array and the \texttt{ADDSP} instruction deallocates it. Figure 6.36 is a program identical to the one of Figure 6.34 except that the index \( i \) and the array \texttt{vector} are local to \texttt{main()}. 

---

**High-Order Language**

```cpp
#include <iostream>
using namespace std;

int main () {
    int vector[4];
    int i;
}
```

---

**Figure 6.36**

A local array.
for (i = 0; i < 4; i++) {
    cin >> vector[i];
}
for (i = 3; i >= 0; i--) {
    cout << i << ' ' << vector[i] << endl;
}
return 0;

Assembly Language

0000    040003  BR    main
;
;******* main ()
vector: .EQUATE 2 ; local variable
i: .EQUATE 0 ; local variable
0003    68000A  main:  SUBSP  10, i ; allocate locals
0006    CB0000  LDX   0, i ; for (i = 0
0009    EB0000  STX   i, s
000C    BB0004  for1:  CPX   4, i ; i < 4
000F    0E0022  BRGE  endFor1
0012    1D    ASLX ; an integer is two bytes
0013    360002  DECI  vector, sx ; cin >> vector[i]
0016    CB0000  LDX   i, s ; i++
0019    780001  ADDX  1, i
001C    EB0000  STX   i, s
001F    04000C  BR    for1
0022    CB0003  endFor1: LDX   3, i ; for (i = 3
0025    EB0000  STX   i, s
0028    BB0000  for2:  CPX   0, i ; i >= 0
002B    080047  BRLT  endFor2
002E    3B0000  DECO  i, s ; cout << i
0031    500020  CHARO  '\n', i ; << '
0034    1D    ASLX ; an integer is two bytes
0035    3E0002  DECO  vector, sx ; << vector[i]
0038    50000A  CHARO  '\n', i ; << endl
003B    CB0000  LDX   i, s ; i--
003E    880001  SUBX  1, i
0041    EB0000  STX   i, s
0044    040028  BR    for2
0047    60000A  endFor2: ADDSP  10, i ; deallocate locals
004A    00    STOP
004B    .END
Figure 6.37 shows the memory allocation on the run-time stack for the program of Figure 6.36. The compiler translates

```c
int vector[4];
int i;
```

at level HOL6 as

```c
main: SUBSP 10,i
```

at level Asmb5. It allocates eight bytes for `vector` and two bytes for `i`, for a total of 10 bytes. It sets the values of the symbols with

```c
vector: .EQUATE 2
i: .EQUATE 0
```

where 2 is the stack-relative address of the first cell of `vector` and 0 is the stack-relative address of `i` as Figure 6.37 shows.

How does the compiler access `vector[i]`? It cannot use indexed addressing, because the value of symbol `vector` is not the address of the first element of the array. It uses stack-indexed addressing. With stack-indexed addressing, the CPU computes the operand as

```
Oprnd = Mem[SP + OprndSpec + X]
```

It adds the stack pointer plus the operand specifier plus the index register and uses the sum as the address in main memory from which it fetches the operand.

In Figure 6.37, the compiler translates

```c
cin >> vector[i];
```

at level HOL6 as

```c
ASLX
DECI vector,sx
```

at level Asmb5. As in the previous program, this is an optimized translation. A nonoptimizing compiler would generate the following code:

```c
LDX i,d
ASLX
DECI vector,sx
```
Suppose the value of \( i \) is 2. \( \text{LDX} \) puts the value of \( i \) in the index register. \( \text{ASLX} \) multiplies the 2 times 2, leaving 4 in the index register. \( \text{DECI} \) uses stack-indexed addressing. So, the operand is computed as:

\[
\text{Mem} [\text{SP} + \text{OprndSpec} + X] \\
\text{Mem} [\text{FBC5} + 2 + 4] \\
\text{Mem} [\text{FBCB}]
\]

which Figure 6.37 shows is \( \text{vector}[2] \). You can see how stack-indexed addressing is made for arrays on the run-time stack. \( \text{SP} \) is the address of the top of the stack. \( \text{OprndSpec} \) is the stack-relative address of the first cell of the array, so \( \text{SP} + \text{OprndSpec} \) is the absolute address of the first cell of the array. With \( i \) in the index register (multiplied by the number of bytes per cell of the array) the sum \( \text{SP} + \text{OprndSpec} + X \) is the address of cell \( i \) of the array.

In summary, to translate local arrays the compiler generates code as follows:

- The array is allocated with \( \text{SUBSP} \) and deallocated with \( \text{ADDSP} \).
- An element of the array is accessed by loading the index into the index register, multiplying it by the number of bytes per cell, and using stack-indexed addressing.

### Translating Arrays Passed as Parameters

In C++, the name of an array is the address of the first element of the array. When you pass an array, even if you do not use the \& designation in the formal parameter list, you are passing the address of the first element of the array. The effect is as if you call the array by reference. The designers of the C language, on which C++ is based, reasoned that programmers almost never want to pass an array by value because such calls are so inefficient. They require large amounts of storage on the run-time stack because the stack must contain the entire array. And they require a large amount of time because the value of every cell must be copied onto the stack. Consequently, the default behavior in C++ is for arrays to be called as if by reference.

Figure 6.38 shows how a compiler translates a program that passes a local array as a parameter. The main program passes an array of integers \( \text{vector} \) and an integer \( \text{numItms} \) to procedures \( \text{getVect} \) and \( \text{putVect} \). \( \text{getVect} \) inputs values into the array and sets \( \text{numItms} \) to the number of items input. \( \text{putVect} \) outputs the values of the array.

```
#include <iostream>
using namespace std;
```

**Figure 6.38**

Passing a local array as a parameter.
void getVect (int v[], int& n) {
    int i;
    cin >> n;
    for (i = 0; i < n; i++) {
        cin >> v[i];
    }
}

void putVect (int v[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        cout << v[i] << ' ';
    }
    cout << endl;
}

int main () {
    int vector[8];
    int numItms;
    getVect (vector, numItms);
    putVect (vector, numItms);
    return 0;
}

Assembly Language

0000 040049         BR   main

;******** getVect (int v[], int& n)
    v: .EQUATE 6          ;formal parameter
    n: .EQUATE 4          ;formal parameter
    i: .EQUATE 0          ;local variable
0003 680002 getVect: SUBSP   2,i        ;allocate local
0006 340004          DECI    n,sf       ;cin >> n
0009 C80000          LDX     0,i        ;for (i = 0
000C EB0000          STX     i,s        ;   i++)
000F BC0004 for1:    CPX     n,sf       ;   i < n
0012 0E0025          BRGE    endFor1
0015 1D              ASLX               ;   an integer is two bytes
0016 370006          DECI    v,sxf      ;   cin >> v[i]
0019 CB0000          LDX     1,i
001C 780001          ADDX    i,s
001F EB0000          STX     i,s
0022 04000F          BR   for1
0025 5A     endFor1: RET2               ;pop local and retAddr
6.4 Indexed Addressing and Arrays

Figure 6.38 (Continued)

;****** putVect (int v[], int n)
v2: .EQUATE 6 ;formal parameter
n2: .EQUATE 4 ;formal parameter
i2: .EQUATE 0 ;local variable

0026 680002 putVect: SUBSP 2, i ;allocate local
0029 C80000 LDX 0, i ;for (i = 0
002C EB0000 STX i2, s
002F BB0004 for2: CPX n2, s ; i < n
0032 0E0048 BRGE endFor2
0035 1D ASLX
0036 3F0006 DECO v2, sxfsf ; cout << v[i]
0039 500020 CHARO ' ', i ; << ' '
003C CB0000 LDX i2, s ; i++)
003F 780001 ADDX 1, i
0042 EB0000 STX i2, s
0045 04002F BR for2
0048 5A endFor2: RET2 ; pop local and retAddr

;****** main ()
vector: .EQUATE 2 ;local variable
numItms: .EQUATE 0 ;local variable

0049 680012 main: SUBSP 18, i ;allocate locals
004C 02 MOVSPA ; push address of vector
004D 700002 ADDA vector, i
0050 E3FFFF STA -2, s
0053 02 MOVSPA ; push address of numItms
0054 700000 ADDA numItms, i
0057 E3FFFF STA -4, s
005A 680004 SUBSP 4, i ; push params
005D 160003 CALL getVect ; getVect (vector, numItms)
0060 600004 ADDSP 4, i ; pop params
0063 02 MOVSPA ; push address of vector
0064 700002 ADDA vector, i
0067 E3FFFF STA -2, s
006A C30000 LDA numItms, s ; push value of numItms
006D E3FFFF STA -4, s
0070 680004 SUBSP 4, i ; push params
0073 160026 CALL putVect ; putVect (vector, numItms)
0076 600004 ADDSP 4, i ; pop params
0079 600012 ADDSP 18, i ; deallocate locals
007C 00 STOP
007D .END
Figure 6.38 shows that the compiler translates the local variables

```c
int vector[8];
int numItms;
```

as

```c
text: .EQUATE 2
text: .EQUATE 0
main: SUBSP 18,i
```

The `SUBSP` instruction allocates 18 bytes on the run-time stack, 16 bytes for the eight integers of the array and 2 bytes for the integer. The `.EQUATE` dot commands set the symbols to their stack offsets, as Figure 6.39(a) shows.

The compiler translates

```c
g vect (vector, numItms);
```

Figure 6.39 (Continued)

![Figure 6.39](image-url)

(a) Before calling `getVect`.

(b) After calling `getVect`.

The run-time stack for the program of Figure 6.38.
6.4 Indexed Addressing and Arrays

by first generating code to push the address of the first cell of \texttt{vector}

\begin{verbatim}
MOVSPA
ADDA    vector, i
STA     -2, s
\end{verbatim}

and then by generating code to push the address of \texttt{numItms}

\begin{verbatim}
MOVSPA
ADDA    numItms, i
STA     -4, s
\end{verbatim}

Even though the signature of the function

\begin{verbatim}
void getVect (int v[], int\& n)
\end{verbatim}

does not have the \& with parameter \texttt{v[]}, the compiler writes code to push the address of \texttt{v} with the \texttt{MOVSPA} and \texttt{ADDA} instructions. Because the signature does have the \& with parameter \texttt{n}, the compiler writes code to push the address of \texttt{n} in the same way. Figure 6.39(b) shows \texttt{v} with FBBF, the address of \texttt{vector[0]} and \texttt{n} with FBBD, the address of \texttt{numItms}.

Figure 6.39(b) also shows the stack offsets for the parameters and local variables in \texttt{getVect}. The compiler defines the symbols

\begin{verbatim}
v: .EQUATE 6
n: .EQUATE 4
i: .EQUATE 0
\end{verbatim}

accordingly. It translates the input statement

\begin{verbatim}
cin >> n;
\end{verbatim}

as

\begin{verbatim}
DECI n, sf
\end{verbatim}

where stack-relative deferred addressing is used because \texttt{n} is called by reference and the address of \texttt{n} is on the stack.

But how does the compiler translate

\begin{verbatim}
cin >> v[i];
\end{verbatim}

It cannot use stack-indexed addressing, because the array of values is not in the stack frame for \texttt{getVect}. The value of \texttt{v} is 6, which means that the address of the first cell of the array is six bytes below the top of the stack. The array of values is in the stack frame for \texttt{main()}. Stack-indexed deferred addressing is designed to access the elements of an array whose address is in the top stack frame but whose actual collection of values is not. With stack-indexed deferred addressing, the CPU computes the operand as

\begin{verbatim}
Oprnd = Mem [Mem [SP + OprndSpec] + X]
\end{verbatim}

\textit{Stack-indexed deferred addressing}
It adds the stack pointer plus the operand specifier and uses the sum as the address of the first element of the array, to which it adds the index register. The compiler translates the input statement as

\[ \text{ASLX DECI } v, sxf \]

where the letters \( sxf \) indicate stack-indexed deferred addressing, and the compiler has determined that the index register will contain the current value of \( i \).

For example, suppose the value of \( i \) is 2. The \texttt{ASLX} instruction doubles it to 4.

The computation of the operand is

\[
\begin{align*}
\text{Mem}[\text{Mem}[\text{SP} + \text{OprndSpec}] + X] \\
\text{Mem}[\text{Mem}[\text{FBB5} + 6] + 4] \\
\text{Mem}[\text{Mem}[\text{FBBB}] + 4] \\
\text{Mem}[\text{FBBF} + 4] \\
\text{Mem}[\text{FBC3}] \\
\end{align*}
\]

which is \texttt{vector[2]} as expected from Figure 6.39(b).

The formal parameters in procedures \texttt{getVect} and \texttt{putVect} in Figure 6.39 have the same names. At level \texttt{HOL6}, the scope of the parameter names is confined to the body of the function. The programmer knows that a statement containing \( n \) in the body of \texttt{getVect} refers to the \( n \) in the parameter list for \texttt{getVect} and not to the \( n \) in the parameter list of \texttt{putVect}. The scope of a symbol name at level \texttt{Asmb5}, however, is the entire assembly language program. The compiler cannot use the same symbol for the \( n \) in \texttt{putVect} that it uses for the \( n \) in \texttt{getVect}, as duplicate symbol definitions would be ambiguous. All compilers must have some mechanism for managing the scope of name declarations in level-\texttt{HOL6} programs when they transform them to symbols at level \texttt{Asmb5}. The compiler in Figure 6.38 makes the identifiers unambiguous by appending the digit 2 to the symbol name. Hence, the compiler translates variable name \( n \) in \texttt{putVect} at level \texttt{HOL6} to symbol \( n2 \) at level \texttt{Asmb5}. It does the same with \( v \) and \( i \).

With procedure \texttt{putVect}, the array is passed as a parameter but \( n \) is called by value. In preparation for the procedure call, the address of \texttt{vector} is pushed onto the stack as before, but this time the value of \texttt{numItms} is pushed. In procedure \texttt{putVect}, \( n2 \) is accessed with stack-relative addressing.

\[ \text{for2: CPX n2,s} \]

because it is called by value. \( v2 \) is accessed with stack-indexed deferred addressing

\[ \begin{align*}
\text{ASLX} \\
\text{DECO } v2, sxf
\end{align*} \]

as it is in \texttt{getVect}.

In Figure 6.38, \texttt{vector} is a local array. If it were a global array, the translations of \texttt{getVect} and \texttt{putVect} would be unchanged. \( v[i] \) would be accessed with stack-indexed deferred addressing, which expects the address of the first element of
the array to be in the top stack frame. The only difference would be in the code to push the address of the first element of the array in preparation of the call. As in the program of Figure 6.34, the value of the symbol of a global array is the address of the first cell of the array. Consequently, to push the address of the first cell of the array the compiler would generate a LDA instruction with immediate addressing followed by a STA instruction with stack-relative addressing to do the push.

In summary, to pass an array as a parameter the compiler generates code as follows:

- The address of the first element of the array is pushed onto the run-time stack, either (a) with MOVSPA followed by ADDA with immediate addressing for a local array, or (b) with LDA with immediate addressing for a global array.
- An element of the array is accessed by loading the index into the index register, multiplying it by the number of bytes per cell, and using stack-indexed deferred addressing.

### Translating the Switch Statement

The program in Figure 6.40, which is also in Figure 2.11, shows how a compiler translates the C++ switch statement. It uses an interesting combination of indexed addressing with the unconditional branch, BR. The switch statement is not the same as a nested if statement. If a user enters 2 for guess, the switch statement branches directly to the third alternative without comparing guess to 0 or 1. An array is a random access data structure because the indexing mechanism allows the programmer to access any element at random without traversing all the previous elements. For example, to access the third element of a vector of integers you can write vector[2] directly without having to traverse vector[0] and vector[1] first. Main memory is in effect an array of bytes whose addresses correspond to the indexes of the array. To translate the switch statement the compiler allocates an array of addresses called a jump table. Each entry in the jump table is the address of the first statement of a section of code that corresponds to one of the cases of the switch statement. With indexed addressing, the program can branch directly to case 2.

```cpp
#include <iostream>
using namespace std;

int main () {
    int guess;
    cout << "Pick a number 0..3: ";
    cin >> guess;
    // Code to handle the switch statement
}
```

Figure 6.40
Translation of a switch statement.
switch (guess) {
    case 0: cout << "Not close"; break;
    case 1: cout << "Close"; break;
    case 2: cout << "Right on"; break;
    case 3: cout << "Too high";
}
cout << endl;
return 0;

Assembly Language

Figure 6.40 (Continued)
Figure 6.40 shows the jump table at 0013 in the assembly language program. The code generated at 0013 is 001B, which is the address of the first statement of case 0. The code generated at 0015 is 0021, which is the address of the first statement of case 1, and so on. The compiler generates the jump table with .ADDRSS pseudo-ops. Every .ADDRSS command must be followed by a symbol. The code generated by .ADDRSS is the value of the symbol. For example, case2 is a symbol whose value is 0027, the address of the code to be executed if guess has a value of 2. Therefore, the object code generated by

```
.ADDRESS case2
```

at 0017 is 0027.

Suppose the user enters 2 for the value of guess. The statement

```
LDX guess,s
ASLX
BR guessJT,x
```

is an unconditional branch with indexed addressing. The value of the operand specifier guessJT is 0013, the address of the first word of the jump table. For indexed addressing, the CPU computes the operand as

\[
Oprnd = Mem[OprndSpec + X]
\]

Therefore, the CPU computes

```
Mem [OprndSpec + X]
Mem [0013 + 4]
Mem [0017]
0027
```

as the operand. The RTL specification for the BR instruction is

```
PC ← Oprnd
```

and so the CPU puts 0027 in the program counter. Because of the von Neumann cycle, the next instruction to be executed is the one at address 0027, which is precisely the first instruction for case 2.
The break statement in C++ is translated as a BR instruction to branch to the end of the switch statement. If you omit the break in your C++ program, the compiler will omit the BR and control will fall through to the next case.

If the user enters a number not in the range 0..3, a run-time error will occur. For example, if the user enters 4 for guess the ASLX instruction will multiply it by 2, leaving 8 in the index register, and the CPU will compute the operand as

\[
\text{Mem} [\text{OprndSpec} + X] \\
\text{Mem} [0013 + 8] \\
\text{Mem} [001B] \\
4100
\]

so the branch will be to memory location 4100 (hex). The problem is that the bits 001B were generated by the assembler for the STRO instruction and were never meant to be interpreted as a branch address. To prevent such indignities from happening to the user, C++ specifies that nothing should happen if the value of guess is not one of the cases. It also provides a default case for the switch statement to handle any case not encountered by the previous cases. The compiler must generate an initial conditional branch on guess to handle the values not covered by the other cases. The problems at the end of the chapter explore this characteristic of the switch statement.

6.5 Dynamic Memory Allocation

The purpose of a compiler is to create a high level of abstraction for the programmer. For example, it lets the programmer think in terms of a single while loop instead of the detailed conditional branches at the assembly level that are necessary to implement the loop on the machine. Hiding the details of a lower level is the essence of abstraction.

But abstraction of program control is only one side of the coin. The other side is abstraction of data. At the assembly and machine levels, the only data types are bits and bytes. Previous programs show how the compiler translates character, integer, and array types. Each of these types can be global, allocated with .BLOCK, or local, allocated with SUBSP on the run-time stack. But C++ programs can also contain structures and pointers, the basic building blocks of many data structures. At level HOL6, pointers access structures allocated from the heap with the new operator. This section shows the operation of a simple heap at level Asmb5 and how the compiler translates programs that contain pointers and structures. It concludes with a description of the translation of boolean values.

Translating Global Pointers

Figure 6.41 shows a C++ program with global pointers and its translation to Pep/8 assembly language. The C++ program is identical to the one in Figure 2.35, and
Figure 2.36 shows the allocation from the heap as the program executes. The heap is a region of memory different from the stack. The compiler, in cooperation with the operating system under which it runs, must generate code to perform the allocation and deallocation from the heap.

---

**High-Order Language**

```cpp
#include <iostream>
using namespace std;

int *a, *b, *c;

int main () {
    a = new int;
    *a = 5;
    b = new int;
    *b = 3;
    c = a;
    a = b;
    *a = 2 + *c;
    cout << "*a = " << *a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c = " << *c << endl;
    return 0;
}
```

**Assembly Language**

```
0000  040009          BR      main
0003  0000   a:       .BLOCK  2          ;global variable
0005  0000   b:       .BLOCK  2          ;global variable
0007  0000   c:       .BLOCK  2          ;global variable

;****** main ()
0009  C00002 main:    LDA     2,i        ;a = new int
000C  16006A          CALL    new
000F  E90003          STX     a,d
0012  C00005          LDA     5,i        ;*a = 5
0015  E20003          STA     a,n
0018  C00005          LDA     2,i        ;b = new int
001B  16006A          CALL    new
001E  E90005          STX     b,d
0021  C00005          LDA     3,i        ;*b = 3
0024  E20005          STA     b,n
0027  C10003          LDA     a,d        ;c = a
```

---

**Figure 6.41**

Translation of global pointers.
When you program with pointers in C++, you allocate storage from the heap with the `new` operator. When your program no longer needs the storage that was allocated, you deallocate it with the `delete` operator. It is possible to allocate sev-
eral cells of memory from the heap and then deallocate one cell from the middle. The memory management algorithms must be able to handle that scenario. To keep things simple at this introductory level, the programs that illustrate the heap do not show the deallocation process. The heap is located in main memory at the end of the application program. Operator `new` works by allocating storage from the heap, so that the heap grows downward. Once memory is allocated it can never be deallocated. This feature of the Pep/8 heap is unrealistic but easier to understand than if it were presented more realistically.

The assembly language program in Figure 6.41 shows the heap starting at address 0076, which is the value of the symbol `heap`. The allocation algorithm maintains a global pointer named `hpPtr`, which stands for heap pointer. The statement

```
hpPtr: .ADDRSS heap
```

at 0074 initializes `hpPtr` to the address of the first byte in the heap. The application supplies the `new` operator with the number of bytes needed. The `new` operator returns the value of `hpPtr` and then increments it by the number of bytes requested. Hence, the invariant maintained by the `new` operator is that `hpPtr` points to the address of the next byte to be allocated from the heap.

The calling protocol for operator `new` is different from the calling protocol for functions. With functions, information is passed via parameters on the run-time stack. With operator `new`, the application puts the number of bytes to be allocated in the accumulator and executes the `CALL` statement to invoke the operator. The operator puts the current value of `hpPtr` in the index register for the application. So, the precondition for the successful operation of `new` is that the accumulator contains the number of bytes to be allocated from the heap. The postcondition is that the index register contains the address in the heap of the first byte allocated by `new`.

The calling protocol for operator `new` is more efficient than the calling protocol for functions. The implementation of `new` requires only four lines of assembly language code including the `RET0` statement. At 006A, the statement

```
new: LDX hpPtr,d
```

puts the current value of the heap pointer in the index register. At 006D, the statement

```
ADDA hpPtr,d
```

adds the number of bytes to be allocated to the heap pointer, and at 0070, the statement

```
STA hpPtr,d
```

updates `hpPtr` to the address of the first unallocated byte in the heap.

This efficient protocol is possible for two reasons. First, there is no long parameter list as is possible with functions. The application only needs to supply one value to operator `new`. The calling protocol for functions must be designed to handle arbitrary numbers of parameters. If a parameter list had, say, four parameters...
there would not be enough registers in the Pep/8 CPU to hold them all. But the run-time stack can store an arbitrary number of parameters. Second, operator new does not call any other function. Specifically, it makes no recursive calls. The calling protocol for functions must be designed in general to allow for functions to call other functions recursively. The run-time stack is essential for such calls but unnecessary for operator new.

Figure 6.42(a) shows the memory allocation for the C++ program at level HOL6 just before the first cout statement. It corresponds to Figure 2.36(h). Figure 6.42(b) shows the same memory allocation at level Asmb5. Global pointers a, b, and c are stored at 0003, 0005, and 0007. As with all global variables, they are allocated with .BLOCK by the statements

```asm
a: .BLOCK 2
b: .BLOCK 2
c: .BLOCK 2
```

A pointer at level HOL6 is an address at level Asmb5. Addresses occupy two bytes. Hence, each global pointer is allocated two bytes. Pointers are addresses.

The compiler translates the statement

```c
a = new int;
```

as

```asm
main: LDA 2, i
CALL new
STX a, d
```

The LDA instruction puts 2 in the accumulator. The CALL instruction calls the new operator, which allocates two bytes of storage from the heap, and puts the pointer to the allocated storage in the index register. The STX instruction stores the returned pointer in the global variable a. Because a is a global variable, STX uses direct addressing. After this sequence of statements executes, a has the value 0076, and hpPtr has the value 0078 because it has been incremented by two.

How does the compiler translate

```c
*a = 5;
```

Figure 6.42

Memory allocation for Figure 6.41 just before the first cout statement.
At this point in the execution of the program, the global variable \( a \) has the address of where the 5 should be stored. (This point does not correspond to Figure 6.42, which is later.) The store instruction cannot use direct addressing, as that would replace the address with 5, which is not the address of the allocated cell in the heap. Pep/8 provides the indirect addressing mode, in which the operand is computed as

\[
\text{Oprnd} = \text{Mem}[\text{Mem}[\text{OprndSpec}]]
\]

With indirect addressing, the operand specifier is the address in memory of the address of the operand. The compiler translates the assignment statement as

\[
\text{LDA } 5,\text{i} \quad \text{STA } a,\text{n}
\]

where \( n \) in the \text{STA} instruction indicates indirect addressing. At this point in the program, the operand is computed as

\[
\text{Mem}[\text{Mem}[\text{OprndSpec}]] \\
\text{Mem}[\text{Mem}[0003]] \\
\text{Mem}[0076]
\]

which is the first cell in the heap. The store instruction stores 5 in main memory at address 0076.

The compiler translates the assignment of global pointers the same as it would translate the assignment of any other type of global variable. It translates

\[
c = a;
\]

as

\[
\text{LDA } a,\text{d} \quad \text{STA } c,\text{d}
\]

using direct addressing. At this point in the program, \( a \) contains 0076, the address of the first cell in the heap. The assignment gives \( c \) the same value, the address of the first cell in the heap, so that \( c \) points to the same cell to which \( a \) points.

Contrast the access of a global pointer to the access of the cell to which it points. The compiler translates

\[
*a = 2 + *c;
\]

as

\[
\text{LDA } 2,\text{i} \quad \text{ADDA } c,\text{n} \quad \text{STA } a,\text{n}
\]
where the `add` and `store` instructions use indirect addressing. Whereas access to a global pointer uses direct addressing, access to the cell to which it points uses indirect addressing. You can see that the same principle applies to the translation of the `cout` statement. Because `cout` outputs `*a`, that is, the cell to which `a` points, the `DECO` instruction at 003F uses indirect addressing.

In summary, to access a global pointer the compiler generates code as follows:

- It allocates storage for the pointer with `.BLOCK 2` because an address occupies two bytes.
- It accesses the pointer with direct addressing.
- It accesses the cell to which the pointer points with indirect addressing.

### Translating Local Pointers

The program in Figure 6.43 is the same as the program in Figure 6.41 except that the pointers `a`, `b`, and `c` are declared to be local instead of global. There is no difference in the output of the program compared to the program where the pointers are declared to be global. But, the memory model is quite different because the pointers are allocated on the run-time stack.

```cpp
#include <iostream>
using namespace std;

int main () {
    int *a, *b, *c;
    a = new int;
    *a = 5;
    b = new int;
    *b = 3;
    c = a;
    a = b;
    *a = 2 + *c;
    cout << "*a = " << *a << endl;
    cout << "*b = " << *b << endl;
    cout << "*c = " << *c << endl;
    return 0;
}
```

Figure 6.43
Translation of local pointers.
Assembly Language

0000 040003     BR     main

;******* main ()
a:    .EQUATE 4 ;local variable
b:    .EQUATE 2 ;local variable
c:    .EQUATE 0 ;local variable

0003 680006     SUBSP   6,i ;allocate locals
0006  C00002     LDA     2,i ;a = new int
0009 16006A     CALL    new
000C  EB0004     STX     a,s
000F  C00005     LDA     5,i ;*a = 5
0012 E40004     STA     a,sf
0015  C00002     LDA     2,i ;b = new int
0018 16006A     CALL    new
001B EB0002     STX     b,s
001E  C00003     LDA     3,i ;*b = 3
0021 E40002     STA     b,sf
0024  C30004     LDA     a,s ;c = a
0027 E30000     STA     c,s
002A  C30002     LDA     b,s ;a = b
002D E30004     STA     a,s
0030  C00002     LDA     2,i ;*a = 2 + *c
0033 740000     ADDA    c,sf
0036 E40004     STA     a,sf
0039 410058     STRO    msg0,d ;cout << "*a = "
003C 3C0004     DECO    a,sf ; << *a
003F 50000A     CHARO   '\n',i ; << endl
0042 41005E     STRO    msg1,d ;cout << "*b = "
0045 3C0002     DECO    b,sf ; << *b
0048 50000A     CHARO   '\n',i ; << endl
004B 410064     STRO    msg2,d ;cout << "*c = "
004E 3C0000     DECO    c,sf ; << *c
0051 50000A     CHARO   '\n',i ; << endl
0054 600006     ADDSP   6,i ;deallocate locals
0057 00     STOP

0058 2A6120     msg0:   .ASCII  "*a = \x00"
3D2000
005E 2A6220     msg1:   .ASCII  "*b = \x00"
3D2000
0064 2A6320     msg2:   .ASCII  "*c = \x00"
3D2000
Figure 6.44 shows the memory allocation for the program in Figure 6.43 just before execution of the first `cout` statement. As with all local variables, `a`, `b`, and `c` are allocated on the run-time stack. Figure 6.44(b) shows their offsets from the top of the stack as 4, 2, and 0. Consequently, the compiler translates

```c
int *a, *b, *c;
```

as

```c
a: .EQUATE 4
b: .EQUATE 2
c: .EQUATE 0
```

Because `a`, `b`, and `c` are local variables, the compiler generates code to allocate storage for them with `SUBSP` and deallocates storage with `ADDSP`.

The compiler translates

```c
a = new int;
```

as

```c
LDA 2,i
CALL new
STX a,s
```

The `LDA` instruction puts 2 in the accumulator in preparation for calling the `new` operator, because an integer occupies two bytes. The `CALL` instruction invokes the `new` operator, which allocates the two bytes from the heap and puts their address in the

```
006A C90074 new:  LDX hpPtr,d ;returned pointer
006D 710074      ADDA hpPtr,d ;allocate from heap
0070 E10074      STA hpPtr,d ;update hpPtr
0073 58          RET0
0074 0076 hpPtr: .ADDRESS heap ;address of next free byte
0076 00 heap:    .BLOCK 1       ;first byte in the heap
0077      .END
```

Figure 6.44 (Continued)
index register. In general, assignments to local variables use stack-relative addressing. Therefore, the \texttt{STX} instruction uses stack-relative addressing to assign the address to \texttt{a}.

How does the compiler translate the assignment

\begin{verbatim}
*a = 5;
\end{verbatim}

\texttt{a} is a pointer, and the assignment gives 5 to the cell to which \texttt{a} points. \texttt{a} is also a local variable. This situation is identical to the one where a parameter is called by reference in the programs of Figures 6.27 and 6.29. Namely, the address of the operand is on the run-time stack. The compiler translates the assignment statement as

\begin{verbatim}
LDA 5, i  
STA a, sf
\end{verbatim}

where the \texttt{store} instruction uses stack-relative deferred addressing.

The compiler translates the assignment of local pointers the same as it would translate the assignment of any other type of local variable. It translates

\begin{verbatim}
c = a;
\end{verbatim}

as

\begin{verbatim}
LDA a, s  
STA c, s
\end{verbatim}

using stack-relative addressing. At this point in the program, \texttt{a} contains 0076, the address of the first cell in the heap. The assignment gives \texttt{c} the same value, the address of the first cell in the heap, so that \texttt{c} points to the same cell to which \texttt{a} points.

The compiler translates

\begin{verbatim}
*a = 2 + *c;
\end{verbatim}

as

\begin{verbatim}
LDA 2, i  
ADDA c, sf  
STA a, sf
\end{verbatim}

where the \texttt{add} instruction uses stack-relative deferred addressing to access the cell to which \texttt{c} points and the \texttt{store} instruction uses stack-relative deferred addressing to access the cell to which \texttt{a} points. The same principle applies to the translation of \texttt{cout} statements where the \texttt{DECO} instructions also use stack-relative deferred addressing.

In summary, to access a local pointer the compiler generates code as follows:

- It allocates storage for the pointer on the run-time stack with \texttt{SUBSP} and deallocates storage with \texttt{ADDSP}.
- It accesses the pointer with stack-relative addressing.
- It accesses the cell to which the pointer points with stack-relative deferred addressing.

\textit{The translation rules for local pointers}
Translating Structures

Structures are the key to data abstraction at level HOL6, the high-order languages level. They let the programmer consolidate variables with primitive types into a single abstract data type. The compiler provides the `struct` construct at level HOL6. At level Asmb5, the assembly level, a structure is a contiguous group of bytes, much like the bytes of an array. However, all cells of an array must have the same type and, therefore, the same size. Each cell is accessed by the numeric integer value of the index.

With a structure, the cells can have different types and, therefore, different sizes. The C++ programmer gives each cell, called a field, a field name. At level Asmb5, the field name corresponds to the offset of the field from the first byte of the structure. The field name of a structure corresponds to the index of an array. It should not be surprising that the fields of a structure are accessed much like the elements of an array. Instead of putting the index of the array in the index register, the compiler generates code to put the field offset from the first byte of the structure in the index register. Apart from this difference, the remaining code for accessing a field of a structure is identical to the code for accessing an element of an array.

Figure 6.45 shows a program that declares a `struct` named `person` that has four fields named `first`, `last`, `age`, and `gender`. It is identical to the program in Figure 2.37. The program declares a global variable name `bill` that has type `person`. Figure 6.46 shows the storage allocation for the structure at levels HOL6 and Asmb5. Fields `first`, `last`, and `gender` have type `char` and occupy one byte each. Field `age` has type `int` and occupies two bytes. Figure 6.46(b) shows the address of each field of the structure. To the left of the address is the offset from the first byte of the structure. The offset of a structure is similar to the offset of an element on the stack except that there is no pointer to the top of the structure that corresponds to SP.

```cpp
#include <iostream>
using namespace std;

struct person {
    char first;
    char last;
    int age;
    char gender;
};
person bill;
```
int main () {
    cin >> bill.first >> bill.last >> bill.age >> bill.gender;
    cout << "Initials: " << bill.first << bill.last << endl;
    cout << "Age: " << bill.age << endl;
    cout << "Gender: ";
    if (bill.gender == 'm') {
        cout << "male\n";
    } else {
        cout << "female\n";
    }
    return 0;
}

Assembly Language

0000 040008 BR main
0003 000000 bill: .BLOCK 5 ;global variable

;******* main ()
0008 C80000 main: LDX first,i ;cin >> bill.first
000B 4D0003 CHARI bill,x
000E C80001 LDX last,i ; >>bill.last
0011 4D0003 CHARI bill,x
0014 C80002 LDX age,i ; >>bill.age
0017 350003 DECI bill,x
001A C80004 LDX gender,i ; >>bill.gender
001D 4D0003 CHARI bill,x
0020 41005A STRO msg0,d ;cout << "Initials: "
0023 C80000 LDX first,i ; << bill.first
0026 550003 CHARO bill,x
0029 C80001 LDX last,i ; << bill.last
002C 550003 CHARO bill,x
002F 50000A CHARO '\n',i ; << endl
0032 410065 STRO msg1,d ;cout << "Age: "
0035 C80002 LDX age,i ; << bill.age
0038 3D0003 DECO bill,x
003B 50000A CHARO '\n',i ; << endl;
003E 41006B STRO msg2,d ;cout << "Gender: "
0041 C80004 LDX gender,i ;if (bill.gender == 'm')
0044 C00000 LDA 0,i
The compiler translates

```c
struct person {
    char first;
    char last;
    int age;
    char gender;
};
```

with equate dot commands as

```c
first:   .EQUATE 0
last:    .EQUATE 1
age:     .EQUATE 2
gender:  .EQUATE 4
```

The name of a field equates to the offset of that field from the first byte of the structure. `first` equates to 0 because it is the first byte of the structure. `last`
equates to 1 because \texttt{first} occupies one byte. \texttt{age} equates to 2 because \texttt{first} and \texttt{last} occupy a total of two bytes. And \texttt{gender} equates to 4 because \texttt{first}, \texttt{last}, and \texttt{age} occupy a total of four bytes. The compiler translates the global variable

\begin{verbatim}
person bill;
\end{verbatim}

\begin{verbatim}
as
\end{verbatim}

\begin{verbatim}
bill: .BLOCK 5
\end{verbatim}

It reserves five bytes because \texttt{first}, \texttt{last}, \texttt{age}, and \texttt{gender} occupy a total of five bytes.

To access a field of a global structure, the compiler generates code to load the index register with the offset of the field from the first byte of the structure. It accesses the field as it would the cell of a global array using indexed addressing. For example, the compiler translates

\begin{verbatim}
cin >> bill.age
\end{verbatim}

\begin{verbatim}
as
LDX age,i
DECI bill,x
\end{verbatim}

The load instruction uses immediate addressing to load the offset of field \texttt{age} into the index register. The decimal input instruction uses indexed addressing to access the field.

The compiler translates

\begin{verbatim}
if (bill.gender == 'm')
\end{verbatim}

\begin{verbatim}
similarly as
LDX gender,i
LDA 0,i
LDBYTEA bill,x
CPA 'm',i
\end{verbatim}

The first load instruction puts the offset of the \texttt{gender} field into the index register. The second load instruction clears the accumulator to ensure that its left-most byte is all zeros for the comparison. The load byte instruction accesses the field of the
structure with indexed addressing and puts it into the right-most byte of the accumulator. Finally, the compare instruction compares `bill.gender` with the letter `m`.

In summary, to access a global structure the compiler generates code as follows:

- It equates each field of the structure to its offset from the first byte of the structure.
- It allocates storage for the structure with `.BLOCK tot` where `tot` is the total number of bytes occupied by the structure.
- It accesses a field of the structure by loading the offset of the field into the index register with immediate addressing followed by an instruction with indexed addressing.

In the same way that accessing the field of a global structure is similar to accessing the element of a global array, accessing the field of a local structure is similar to accessing the element of a local array. Local structures are allocated on the run-time stack. The name of each field equates to its offset from the first byte of the structure. The name of the local structure equates to its offset from the top of the stack. The compiler generates `SUBSP` to allocate storage for the structure and any other local variables, and `ADDSP` to deallocate storage. It accesses a field of the structure by loading the offset of the field into the index register with immediate addressing followed by an instruction with stack-indexed addressing. Translating a program with a local structure is a problem for the student at the end of this chapter.

### Translating Linked Data Structures

Programmers frequently combine pointers and structures to implement linked data structures. The `struct` is usually called a node, a pointer points to a node, and the node has a field that is a pointer. The pointer field of the node serves as a link to another node in the data structure. Figure 6.47 is a program that implements a linked list data structure. It is identical to the program in Figure 2.38.

#### High-Order Language

```cpp
#include <iostream>
using namespace std;

struct node {
  int data;
  node* next;
};
```

Figure 6.47
Translation of a linked list.
int main () {
    node *first, *p;
    int value;
    first = 0;
    cin >> value;
    while (value != -9999) {
        p = first;
        first = new node;
        first->data = value;
        first->next = p;
        cin >> value;
    }
    for (p = first; p != 0; p = p->next) {
        cout << p->data » ' ';
    }
    return 0;
}

Assembly Language

0000 040003    BR     main
  data:  .EQUATE 0       ;struct field
  next:  .EQUATE 2       ;struct field
  ;****** main ()
  first: .EQUATE 4       ;local variable
  p:     .EQUATE 2       ;local variable
  value: .EQUATE 0       ;local variable
0000  680006 main: SUBSP  6,i         ;allocate locals
0000  C00000    LDA    0,i         ;first = 0
0000  E30004    STA    first,s
0001  330000    DECI   value,s     ;cin » value
0001  C30000    while: LDA    value,s     ;while (value != -9999)
0001  B0D8F1    CPA    -9999,i
0002  0A003F    BREQ   endWh
0002  C30000    LDA    first,s     ; p = first
0002  C00004    LDA    first,s     ; first = new node
0003  160067    CALL   new
0003  E80004    STX    first,s
0004  C30000    LDA    value,s     ; first->data = value
0004  C80000    LDX    data,i
0005  E70004    STA    first,sxf
0006  C30002    LDA    p,s         ; first->next = p
0006  C80002    LDX    next,i
The compiler equates the fields of the struct

```c
struct node {
    int data;
    node* next;
};
```
to their offsets from the first byte of the `struct`. `data` is the first field with an offset of 0. `next` is the second field with an offset of 2 because `data` occupies two bytes. The translation is

```plaintext
data: .EQUATE 0
data: .EQUATE 2

The compiler translates the local variables

```plaintext
node *first, *p;
int value;
```

as it does all local variables. It equates the variable names with their offsets from the top of the run-time stack. The translation is

```plaintext
first: .EQUATE 4
p: .EQUATE 2
value: .EQUATE 0
```

Figure 6.48(b) shows the offsets for the local variables. The compiler generates `SUBSP` at 0003 to allocate storage for the locals and `ADDSP` at 0063 to deallocate storage.

When you use the `new` operator in C++, the computer must allocate enough memory from the heap to store the item to which the pointer points. In this program, a node occupies four bytes. Therefore, the compiler translates

```plaintext
first = new node;
```

by allocating four bytes in the code it generates to call the `new` operator. The translation is

```plaintext
LDA 4,i
CALL new
STX first,s
```

**Figure 6.48**
Memory allocation for Figure 6.47 just after the third execution of the `while` loop.

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0073</td>
</tr>
<tr>
<td>0</td>
<td>0075</td>
</tr>
<tr>
<td>20</td>
<td>0077</td>
</tr>
<tr>
<td>30</td>
<td>0079</td>
</tr>
<tr>
<td>0</td>
<td>007D</td>
</tr>
<tr>
<td>-9999</td>
<td>007B</td>
</tr>
<tr>
<td>2</td>
<td>FBCB</td>
</tr>
<tr>
<td>0077</td>
<td>p</td>
</tr>
<tr>
<td>4</td>
<td>FBCD</td>
</tr>
<tr>
<td>007B</td>
<td>first</td>
</tr>
</tbody>
</table>
```

(a) The linked list at level HOL6.

(b) The linked list at level Asmb5.
The load instruction puts 4 in the accumulator in preparation for the call to new. The call instruction calls the new operator, which puts the address of the first byte of the allocated node in the index register. The store index instruction completes the assignment to local variable first using stack-relative addressing.

How does the compiler generate code to access the field of a node to which a local pointer points? Remember that a pointer is an address. A local pointer implies that the address of the node is on the run-time stack. Furthermore, the field of a struct corresponds to the index of an array. If the address of the first cell of an array is on the run-time stack, you access an element of the array with stack-indexed deferred addressing. That is precisely how you access the field of a node. Instead of putting the value of the index in the index register, you put the offset of the field in the index register. The compiler translates

```
first->data = value;
```

as

```
LDA value, s
LDX data, i
STA first, sx
```

Similarly, it translates

```
first->next = p;
```

as

```
LDA p, s
LDX next, i
STA first, sx
```

To see how stack-indexed deferred addressing works for a local pointer to a node, remember that the CPU computes the operand as

```
Oprnd = Mem[Mem[SP + OprndSpec] + X]
```

It adds the stack pointer plus the operand specifier and uses the sum as the address of the first field, to which it adds the index register. Suppose that the third node has been allocated as shown in Figure 6.48(b). The call to new has returned the address of the newly allocated node, 007B, and stored it in first. The LDA instruction above has put the value of p, 0077 at this point in the program, in the accumulator. The LDX instruction has put the value of next, offset 2, in the index register. The STA instruction executes with stack-indexed addressing. The operand specifier is 4, the value of first. The computation of the operand is

```
Mem[Mem[SP + OprndSpec] + X]
Mem[Mem[FBC9 + 4] + 2]
Mem[Mem[FBCD] + 2]
Mem[007B + 2]
Mem[007D]
```
which is the next field of the node to which first points.

In summary, to access a field of a node to which a local pointer points the compiler generates code as follows:

- The field name of the node equates to the offset of the field from the first byte of the node. The offset is loaded into the index register.
- The instruction to access the field of the node uses stack-indexed deferred addressing.

You should be able to determine how the compiler translates programs with global pointers to nodes. Formulation of the translation rules is an exercise for the student at the end of this chapter. Translation of a C++ program that has global pointers to nodes is also a problem for the student.

SUMMARY

A compiler uses conditional branch instructions at the machine level to translate if statements and loops at the high-order languages level. An if/else statement requires a conditional branch instruction to test the if condition and an unconditional branch instruction to branch around the else part. The translation of a while or do loop requires a branch to a previous instruction. The for loop requires, in addition, instructions to initialize and increment the control variable.

The structured programming theorem, proved by Bohm and Jacopini, states that any algorithm containing goto's, no matter how complicated or unstructured, can be written with only nested if statements and while loops. The goto controversy was sparked by Dijkstra's famous letter, which stated that programs without goto's were not only possible but desirable.

The compiler allocates global variables at a fixed location in main memory. Procedures and functions allocate parameters and local variables on the run-time stack. Values are pushed onto the stack by incrementing the stack pointer (SP) and popped off the stack by decrementing SP. The subroutine call instruction pushes the contents of the program counter (PC), which acts as the return address, onto the stack. The subroutine return instruction pops the return address off the stack into the PC. Instructions access global values with direct addressing and values on the run-time stack with stack-relative addressing. A parameter that is called by reference has its address pushed onto the run-time stack. It is accessed with stack-relative deferred addressing. Boolean variables are stored with a value of 0 for false and a value of 1 for true.

Array values are stored in consecutive main memory cells. You access an element of a global array with indexed addressing, and an element of a local array with stack-indexed addressing. In both cases, the index register contains the index value of the array element. An array passed as a parameter always has the address of the first cell of the array pushed onto the run-time stack. You access an element of the array with stack-indexed deferred addressing. The compiler translates the switch statement with an array of addresses, each of which is the address of the first statement of a case.

Pointer and struct types are common building blocks of data structures. A pointer is an address of a memory location in the heap. The new operator allocates memory from the heap. You access a cell to which a global pointer points with indirect addressing. You access a cell to which a local pointer points with stack-relative deferred addressing. A struct has several named fields and is stored as a contiguous group of bytes. You access a field of a global
struct with indexed addressing with the index register containing the offset of the field from the first byte of the struct. Linked data structures commonly have a pointer to a struct called a node, which in turn contains a pointer to yet another node. If a local pointer points to a node, you access a field of the node with stack-indexed deferred addressing.

**EXERCISES**

**Section 6.1**
1. Explain the difference in the memory model between global and local variables. How are each allocated and accessed?

**Section 6.2**
2. What is an optimizing compiler? When would you want to use one? When would you not want to use one? Explain.

*3. The object code for Figure 6.14 has a CPA at 000C to test the value of $i$. Because the program branches to that instruction from the bottom of the loop, why doesn’t the compiler generate a LDA $i,d$ at that point before CPA?

4. Discover the function of the mystery program of Figure 6.16, and state in one short sentence what it does.

5. Read the papers by Bohm and Jacopini and by Dijkstra that are referred to in this chapter and write a summary of them.

**Section 6.3**

*6. Draw the values just before and just after the CALL at 0022 of Figure 6.18 executes as they are drawn in Figure 6.19.

7. Draw the run-time stack, as in Figure 6.26, that corresponds to the time just before the second return.

**Section 6.4**

*8. In the Pep/8 program of Figure 6.40, if you enter 4 for Guess, what statement executes after the branch at 0010? Why?

9. Section 6.4 does not show how to access an element from a two-dimensional array. Describe how a two-dimensional array might be stored and the assembly language object code that would be necessary to access an element from it.

**Section 6.5**

10. What are the translation rules for accessing the field of a node to which a global pointer points?
11. Translate the following C++ program to Pep/8 assembly language.

```
#include <iostream>
using namespace std;

int main () {
    int number;
    cin >> number;
    if (number % 2 == 0) {
        cout << "Even\n";
    } else {
        cout << "Odd\n";
    }
    return 0;
}
```

12. Translate the following C++ program to Pep/8 assembly language.

```
#include <iostream>
using namespace std;

const int limit = 5;

int main () {
    int number;
    cin >> number;
    while (number < limit) {
        number++;
        cout << number << ' ';
    }
    return 0;
}
```

13. Translate the following C++ program to Pep/8 assembly language.

```
#include <iostream>
using namespace std;

int main () {
    char ch;
    cin >> ch;
    if ((ch >= 'A') && (ch <= 'Z')) {
        cout << 'A';
    } else if ((ch >= 'a') && (ch <= 'z')) {
```
14. Translate the C++ program in Figure 6.12 to Pep/8 assembly language but with the `do` loop test changed to

```cpp
while (cop <= driver);
```

15. Translate the following C++ program to Pep/8 assembly language.

```cpp
#include <iostream>
using namespace std;

int main () {
    int numItms, i, data, sum;
    cin >> numItms;
    sum = 0;
    for (i = 1; i <= numItms; i++) {
        cin >> data;
        sum += data;
    }
    cout << "Sum: " << sum << endl;
    return 0;
}
```

**Sample Input**

```
4 8 -3 7 6
```

**Sample Output**

```
Sum: 18
```

### Section 6.3

16. Translate the following C++ program to Pep/8 assembly language.

```cpp
#include <iostream>
using namespace std;

int myAge;

void putNext (int age) {
    int nextYr;
    nextYr = age + 1;
    cout << "Age: " << age << endl;
    cout << "Age next year: " << nextYr << endl;
}
int main () {
    cin >> myAge;
    putNext (myAge);
    putNext (64);
    return 0;
}

17. Translate the C++ program in Problem 16 to Pep/8 assembly language, but declare
   myAge to be a local variable in main().

18. Translate the following C++ program to Pep/8 assembly language. It multiplies two
   integers using a recursive shift-and-add algorithm:

```cpp
#include <iostream>
using namespace std;

int times (int mpr, int mcand) {
    if (mpr == 0) {
        return 0;
    }
    else if (mpr % 2 == 1) {
        return mcand + times (mpr / 2, mcand * 2);
    }
    else {
        return times (mpr / 2, mcand * 2);
    }
}

int main () {
    int n, m;
    cin >> n >> m;
    cout << "Product: " << times (n, m) << endl;
    return 0;
}
```

19. (a) Write a C++ program that converts a lowercase character to an uppercase charac-
    ter. Declare

```
char uppercase (char ch);
```

to do the conversion. If the actual parameter is not a lowercase character, the function
should return that character value unchanged. Test your function in a main program
with interactive I/O. (b) Translate your C++ program to Pep/8 assembly language.

20. (a) Write a C++ program that defines

```
int minimum (int i1, int i2)
```

    which returns the smaller of i1 and i2, and test it with interactive input. (b) Translate
    your C++ program to Pep/8 assembly language.
21. Translate to Pep/8 assembly language your C++ solution from Problem 2.14 that computes a Fibonacci term using a recursive function.

22. Translate to Pep/8 assembly language your C++ solution from Problem 2.15 that outputs the instructions for the Towers of Hanoi puzzle.

23. The recursive binomial coefficient function in Figure 6.25 can be simplified by omitting y1 and y2 as follows:

```c
int binCoeff (int n, int k) {
    if ((k == 0) || (n == k)) {
        return 1;
    }
    else {
        return binCoeff (n - 1, k) + binCoeff (n - 1, k - 1);
    }
}
```

Write a Pep/8 assembly language program that calls this function. Keep the value returned from the `binCoeff (n - 1, k)` call on the stack and allocate the actual parameters for the `binCoeff (n - 1, k - 1)` call on top of it. Figure 6.49 shows a trace of the run-time stack where the stack frame contains four words (for `retVal`, `n`, `k`, and `retAddr`) and the shaded word is the value returned by a function call. The trace is for a call of `binCoeff (3, 1)` from the main program.

24. Translate the following C++ program to Pep/8 assembly language. It multiplies two integers using an iterative shift-and-add algorithm.

```c
#include <iostream>

using namespace std;

int product, n, m;

void times (int& prod, int mpr, int mcand) {
    prod = 0;
    while (mpr != 0) {
```

Figure 6.49
Trace of the run-time stack for Figure 6.25
if (mpr % 2 == 1) {
    prod = prod + mcand;
} 
mpr /= 2;
mcand *= 2;
}

int main () {
    cin >> n >> m;
times (product, n, m);
cout << "Product: " << product << endl;
return 0;
}

25. Translate the C++ program in Problem 24 to Pep/8 assembly language, but declare product, n, and m to be local variables in main().

26. (a) Rewrite the C++ program of Figure 2.21 to compute the factorial recursively, but use procedure times in Problem 24 to do the multiplication. Use one extra local variable in fact to store the product. (b) Translate your C++ program to Pep/8 assembly language.

Section 6.4
27. Translate the following C++ program to Pep/8 assembly language.

#include <iostream>
using namespace std;

int list[16];
int i, numItems;
int temp;

int main () {
    cin >> numItems;
    for (i = 0; i < numItems; i++) {
        cin >> list[i];
    }
temp = list[0];
    for (i = 0; i < numItems - 1; i++) {
        list[i] = list[i + 1];
    }
list[numItems - 1] = temp;
    for (i = 0; i < numItems; i++) {
        cout << list[i] << ' ';
    }
cout << endl;
return 0;
}
Sample Input
5
11 22 33 44 55

Sample Output
22 33 44 55 11

The test in the second for loop is awkward to translate because of the arithmetic expression on the right side of the < operator. You can simplify the translation by transforming the test to the following mathematically equivalent test.

\[ i + 1 < \text{numItems}; \]

28. Translate the C++ program in Problem 27 to Pep/8 assembly language, but declare list, i, numItems, and temp to be local variables in main().

29. Translate the following C++ program to Pep/8 assembly language.

```cpp
#include <iostream>
using namespace std;

void getList (int ls[], int& n) {
    int i;
    cin >> n;
    for (i = 0; i < n; i++) {
        cin >> ls[i];
    }
}

void putList (int ls[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        cout << ls[i] << ' ';    
    }
    cout << endl;
}

void rotate (int ls[], int n) {
    int i;
    int temp;
    temp = ls[0];
    for (i = 0; i < n - 1; i++) {
        ls[i] = ls[i + 1];
    }
    ls[n - 1] = temp;
}

int main () {
    int list[16];
    int numItems;
```
getList (list, numItems);
putList (list, numItems);
rotate (list, numItems);
putList (list, numItems);
return 0;
}

Sample Input
5
11 22 33 44 55

Sample Output
11 22 33 44 55
22 33 44 55 11

30. Translate the C++ program in Problem 29 to Pep/8 assembly language but declare
list and numItems to be global variables.

31. Translate to Pep/8 assembly language the C++ program from Figure 2.23 that adds
four values in an array using a recursive procedure.

32. Translate to Pep/8 assembly language the C++ program from Figure 2.30 that reverses
the elements of an array using a recursive procedure.

33. Translate the following C++ program to Pep/8 assembly language.

```cpp
#include <iostream>
using namespace std;

int main () {
    int guess;
    cout << "Pick a number 0..3: ";
    cin >> guess;
    switch (guess) {
        case 0: case 1: cout << "Too low"; break;
        case 2: cout << "Right on"; break;
        case 3: cout << "Too high";
    }
    cout << endl;
    return 0;
}
```

The program is identical to Figure 6.40 except that two of the cases execute the same
code. Your jump table must have exactly four entries, but your program must have
only three case symbols and three cases.

34. Translate the following C++ program to Pep/8 assembly language.

```cpp
#include <iostream>
using namespace std;

int main () {
    
```
int guess;
cout << "Pick a number 0..3: ";
cin >> guess;
switch (guess) {
    case 0: cout << "Not close"; break;
    case 1: cout << "Too low"; break;
    case 2: cout << "Right on"; break;
    case 3: cout << "Too high"; break;
    default: cout << "Illegal input";
}
cout << endl;
return 0;

Section 6.5
35. Translate to Pep/8 assembly language the C++ program from Figure 6.45 that accesses the fields of a structure, but declare bill as a local variable in main().

36. Translate to Pep/8 assembly language the C++ program from Figure 6.47 that manipulates a linked list, but declare first, p, and value as global variables.

37. Insert the following C++ code fragment in main() of Figure 6.47 just before the return statement

```cpp
sum = 0; p = first;
while (p != 0) {
    sum += p->data;
    p = p->next;
}
cout << "Sum: " << sum << endl;
```

and translate the complete program to Pep/8 assembly language. Declare sum to be a local variable along with the other locals as follows:

```cpp
node *first, *p;
int value, sum;
```

38. Insert the following C++ code fragment between the declaration of node and main() in Figure 6.47

```cpp
void reverse (node* list) {
    if (list != 0) {
        reverse (list->next);
        cout << list->data << ' ';
    }
}
```

and the following code fragment in main() just before the return statement.

```cpp
cout << endl;
reverse (first);
```
Translate the complete C++ program to Pep/8 assembly language. The added code outputs the linked list in reverse order.

39. Insert the following C++ code fragment in main() of Figure 6.47 just before the return statement

```cpp
first2 = 0; p2 = 0;
for (p = first; p != 0; p = p->next) {
    p2 = first2;
    first2 = new node;
    first2->data = p->data;
    first2->next = p2;
}
for (p2 = first2; p2 != 0; p2 = p2->next) {
    cout << p2->data << ' ';
}
```

Declare `first2` and `p2` to be local variables along with the other locals as follows:

```cpp
node *first, *p, *first2, *p2;
int value;
```

Translate the complete program to Pep/8 assembly language. The added code creates a copy of the first list in reverse order and outputs it.

40. (a) Write a C++ program to input an unordered list of integers with –9999 as a sentinel into a binary search tree, then output them with an inorder traversal of the tree. (b) Translate your C++ program to Pep/8 assembly language.