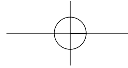
trie_ch01.qxd 8/22/01 1:23 PM Page 1



Introduction

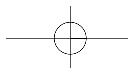
CHAPTER 1

Computers, such as those that use the Intel® Itanium™ processor, cannot solve problems. They are simply machines and must be told what to do. Programmers tell computers what to do by writing sets of instructions known as computer programs or, simply, software.

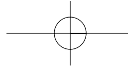
This is not a book about how to program computers or the theory of computer programming. This book is about the software used to program computers configured with the Itanium™ processor. It is about the programming languages used by such machines, and the similarities and the differences in writing software for them. This book presumes some minimal knowledge of computers and a familiarity with at least one programming language, either C or Fortran 90.

One main goal is to show the reader that the fundamental differences between Itanium software and that written for other computers are primarily of scale, not kind—in both cases, the fundamental principles of structured, modular programming are adhered to. In addition, we show the reader instances in which the internal design of the Itanium processor allows the programmer to write more efficient programs than for comparable systems.

To be proficient at writing software, programmers must understand many things; traditionally, however, they have remained insulated from the details of the internal workings of the computer. Throughout this book, we explain low-level architectural details of the Itanium processor that are relevant to creating better high-level language (HLL) programs.



trie_ch01.qxd 8/22/01 1:23 PM Page 2



Explicitly Parallel Instruction Computing

The Itanium processor employs a parallel computer architecture called *explicitly parallel instruction computing* (EPIC). EPIC is a new architecture, a new fundamental basis for microprocessor designs by Intel Corporation. Intel's first hardware implementation of this architecture is the Itanium processor.

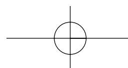
EPIC offers many new features to increase performance, but the most important are explicit instruction-level parallelism (ILP), speculation, predication, and a large register file. Each of these is designed to allow a compiler to produce better code for parallel execution, which allows programs to run faster.

Parallelism

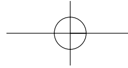
Unlike earlier parallel designs (such as very long instruction word, or VLWI), EPIC does not use a fixed word length encoding. Instead, instructions can be combined to operate in parallel from one to as many as desired. That is, EPIC acknowledges that the parallelism is not consistent from clock to clock, and machine width is not fixed for every microprocessor design. The Itanium processor is six execution units wide, but cannot do every combination of six things at once. Therefore, EPIC allows code to express where the parallelism is, without forcing all code to be six wide forever. Future IA-64 microprocessors may be more flexible about what they can do in parallel or how many things can be done in parallel, or both.

The traditional problem in processing is that there is never enough work ready to keep a machine fully busy. Since the Itanium has all this processing power, the key is to not throw it away by letting it go unused. When the processor is ready to do six more things, you don't want to give it only two things to do and waste 66% of the power.

A breakthrough happens when you decide to stop restricting yourself to doing only the things that you must. The approach is to ask the machine speculatively to do four more things. That is, you pick things that might be needed in the future, but right now you just aren't sure which of the four it will be. Let us say that the four extra things turn out to be needed only half the time. You are still better off than if you



trie_ch01.qxd 8/22/01 1:23 PM Page 3



had not asked the processor to do anything. Speculation is a big part of what EPIC offers, and it offers it in many forms. But, all the speculation features are simply in EPIC to allow us to compute things before they are needed, and before a check is made to ensure that it is not too early to do the work.

In fact, this is the promise of EPIC. As machines get wider and wider, we will speculate more and more. The goal is to compute results before they are needed, so that when a program needs them, the answers are already there.

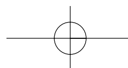
Compiler Technology

EPIC also builds on the abilities of a VLIW machine by allowing the selection of instructions to be executed in parallel to be done at compile time. For this reason, compilers may be the most important component of EPIC. Compilers designed to harness the instruction level parallelism are very complex, but are the critical element in achieving high-performance Itanium code.

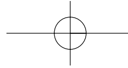
The Itanium processor's EPIC architecture was specifically designed with compilers in mind. It is optimized for compiler writers, to simplify the task of writing the advanced compilers needed for EPIC. The goal of Itanium architectural features such as speculation, predication, and a large register file, is to simplify the problems that a compiler writer faces so that it is possible to write good compilers to use EPIC.

The first thing EPIC offers to compilers is explicit access to the parallelism—hence the term EPIC. Once explicit parallelism is available, the challenge is to make full use of it. As we noted, speculative execution of instructions is the most important concept in EPIC for helping a compiler fill up the available parallelism—it is what makes it possible to believe that programs can be written to use the parallelism. More importantly, speculative execution allows compiler writers to compile programs into parallel code for the processor.

Predication is the second most important concept in EPIC for enabling the compiler to utilize the available parallelism. Practically every instruction that is executed is conditional, based on a predicate register. Each instruction either has an effect, or has no effect, based on the True or False condition in a predicate register. Branches in program control impede full use of parallelism. This predication of



trie_ch01.qxd 8/22/01 1:23 PM Page 4



every instruction, with no extra run-time cost, allows the compiler to eliminate branching, thereby increasing parallelism.

The compiler also benefits from the very large register sets of the Itanium processor. A smaller register set limits performance. The processor must shuffle data in and out of registers instead of doing the work required for the program. The Itanium architecture even offers a feature called rotating registers, which reduces the need for register shuffling and code duplication when doing a type of loop-level parallelism called software pipelining.

Finally, communication between compiler and processor is facilitated by template information contained in bundles of instructions and by completers in the individual instructions. The template information allows the processor to efficiently assign its functional units to the execution of instructions. The compiler has the ability and resources to analyze the behavior of branch operations in a program. The compiler communicates information about the usage of the branch instructions to the Itanium processor hardware through completers. Transfer of this knowledge to the processor helps to reduce penalties associated with the branch operation.

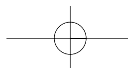
Architecture of the Intel Compiler

The C/C++ and Fortran compilers offered by Intel for the Itanium processor are designed to increase instruction-level parallelism in many ways. For example, the compiler has the ability to reorder instructions to assure efficient use of the Itanium processor's execution units. This technique is known as upward and downward code motion.

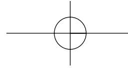
The compiler also exploits the advanced architectural features of Itanium architecture, while being sensitive to practical considerations such as code size and compile time. They utilize the advanced features like speculation, predication, and rotating registers to maximize program performance by

- decreasing overhead in memory accesses,
- reducing the number of branches and the penalty overhead associated with branches, and
- increasing the speed of loops by pipelining their execution.

Instruction-level parallelism is also increased by profiling and scheduling techniques that consider large amounts of a program. Examples are whole program analysis and region formation.



trie_ch01.qxd 8/22/01 1:23 PM Page 5



Profiling a Program

The compiler can take better advantage in an application of the architectural features of the Itanium processor if it understands the execution behavior of the program. Information about key execution properties of a program can be collected by instrumenting, then running, the program. Of special interest is information about which functions are called and how often each branch is taken. This collected information is referred to as the *profile* of the program and is used by the compiler to make better decisions about how to optimize the program.

The first step in the compilation process is the profiling of the program. The Itanium compilers support both *static* and *dynamic* profiling. The software developer makes the choice of static or dynamic profiling at compile time.

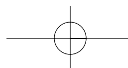
As its name implies, static profiling means that the profile of the execution of the program is estimated without actually running the program. All compilers support static profiling. When the compiler is used with static profiling, it estimates the frequency of execution of the functions and the probability that branches are taken based on known characteristics of a typical program, not the specific application.

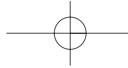
Dynamic profiling, also known as *profile feedback,* is a method for gathering dynamic program control flow information about the operation of a program. A dynamic profile is generally more accurate because it is based on information collected during actual runs of the program, and it therefore more accurately reflects the actual operation of the program.

Many compilers have supported dynamic profiling for other processors, but none have had an architecture designed to take advantage of dynamic profile information in the way that the Itanium architecture can. Over time, as familiarity grows and additional tool support is available, dynamic profiling will become more commonly the preferred method of compilation.

Code Generation

Once the behavior of the program is understood, the compiler must generate code to implement the HLL program. That is, the compiler picks the actual Itanium instructions for the machine to execute. Since the Itanium architecture relies on the compiler to express parallelism in the code, the process of selecting instructions to execute in parallel





is critical. The part of a compiler that selects the instructions to be run in parallel is called the *code scheduler*.

A common theme in the code generator is to understand the most likely paths through a program and to emphasize the optimization of these paths when choices must be made that favor one path or another. This path determination process involves the formation of key regions of the code to optimize.

There are other critical steps in selecting the right code to schedule. Optimization techniques used widely in other compilers have been extended in the Itanium compilers to employ the features and resources of the Itanium architecture. Some examples are loop unrolling, in-lining of functions, and vectorization. Furthermore, new techniques have been developed and used in the compiler to fully extract the parallelism of the architecture. These other steps are concerned with using architectural features like predication, speculation, parallel compares, and rotating registers.

The compilers identify and exploit opportunities for parallel execution in the code, revise and reorder code for parallel execution, and communicate information via hints and instructions to the processor to facilitate high performance. For example, memory access overhead is decreased through speculation and the use of the large number of registers; delays due to branching are reduced by the use of completers and predication; and loops are sped up by applying rotating registers and software pipelining.

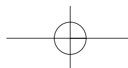
We examine the techniques that the Itanium compilers use to implement high-performance code in Chapters 3 through 5.

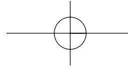
OVERVIEW OF PROGRAMMING LANGUAGES

Although there are many different programming languages, almost all programming is done today in high-level languages. That is because the alternatives—machine language and assembly language—have inherent disadvantages that completely preclude the first, and make the second very difficult at best to use for stand-alone programs. We will briefly describe the evolution from machine language to HLLs, and the advantages and disadvantages of each type of language.

Machine Language

The native language of the computer is machine language. Each machine language instruction is a group of binary digits (bits) or bit





patterns that specify an operation and identify the registers and/or memory cells involved with the operation. For example, you may wish to solve a problem represented by the formula

average = sum/number of items

In a machine-language program, this expression might appear as the following sequence of binary numbers

```
0100  1110  0001  0001  1101  0100  1110  0001
0100  1110  0001  0001  1101  1101  1110  0001
0100  1110  0001  0001  1101  0100  1110  0001
```

In fact, these are the actual Itanium machine-code instructions needed to evaluate the previous expression.

It is of historical interest to note that in the early days of computing—more than fifty years ago—programming was done by setting switches corresponding to 1s and 0s to the desired bit pattern. Fortunately, you need not be concerned with such drudgery.

Assembly Language

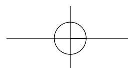
Assembly language provides a great improvement in programming computers over machine language, but still has significant disadvantages. Its instructions consist of commands known as mnemonics that are intended to indicate their purpose to the user. Unfortunately, that is often not true. For instance, an example Itanium assembly-language instruction is

```
(p0)   czx2. r   r10=r5
```

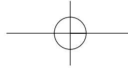
It is not clear what the mnemonic means or what operation is performed. Also, the user must be familiar with number systems other than decimal, including the hexadecimal, or base-16 number system.

Assembly language is time-intensive to program, and therefore more prone to errors than HLLs. Another significant disadvantage is that an assembly language program is unique to the architecture of the processor used in the computer. A computer can only run programs written in its own assembly language. For instance, an Intel® Pentium® 4 processor–based PC cannot run a program written in the instruction set of an Itanium processor.

The main advantage of assembly language is that there is only one assembly language statement for each machine language statement. That means assembly language is more conserving of memory and runs faster than an HLL. Perhaps its biggest advantage is that in order



trie_ch01.qxd 8/22/01 1:23 PM Page 8



to understand the architecture of a computer one must have some knowledge of its assembly language. Such knowledge allows the programmer to write more efficient HLL code and also, when prudent, to use assembly language in conjunction with HLLs—either in functions, or as *in-line*[1] code.

High-Level Languages

As the term implies, a high-level language consists of statements that resemble everyday English language. The HLLs used in this book are C (and to a lesser extent, it's derivative languages, such as C++), and Fortran 90. An HLL has the important advantage of allowing the programmer to write programs using simple-to-understand, English-like instructions.

Frequently, programs written for one type of computer must be ported to run on another type of computer. Another advantage of using HLLs is that they make porting easier. However, HLLs are portable to varying degrees, with C being the most easily ported language.

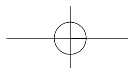
A disadvantage of an HLL is that instructions must be translated into their machine language counterparts and then further processed before they can be executed. There is a many-to-one correspondence between HLL instructions and their equivalent machine language instructions. Therefore, more memory is used by HLLs and their programs will run slightly slower than if written in assembly language—normally an insignificant price to pay for the overall gain in programming efficiency.

Let us look at how the arithmetic statement introduced earlier is coded in C. It is simply written as

```
average = sum/number_of_items;
```

By way of review, note that this statement is read "divide the value contained in the memory variable `sum` by the value contained in the memory variable `number_of_items` and assign that result to the memory variable `average`." Notice the use of variable names that describe the purpose of the variable. For readability and debugging, this is preferable to simply naming variable using unrelated names or letters, such as `a`, `s`, and `n`.

[1]Formally, a *closed subroutine,* but more easily remembered as expanded macro code.



There are many HLLs. In addition to C, C++, and Fortran, other common HLLs are BASIC, COBOL, Prolog, Ada, and Pascal. Although each of these languages was designed for a specific purpose, all are used to write some kind of application software. Additionally, there are languages used to write operating system software—C and C++ are the HLLs used for this purpose. In fact, C was originally created as a system language. It was first used to write the operating system for the Digital Equipment Company's PDP series of minicomputers.

Operating systems currently available for Itanium processor-based computer systems are MS 64-bit Win2000/NT and 64-bit Linux 6.2. WinNT 64-bit-2000 is primarily written in C++, whereas 64-bit Linux 6.2 is coded in C. Both have device-dependent I/O routines written in assembly.

C is a unique language because it has machine language–like statements as well as high-level user-defined data types. It is the most widely used HLL, and is the best vehicle to introduce Itanium software concepts.

Fortran concepts are covered to a lesser extent. Fortran is a more mature language than C, and is widely used for complex floating-point mathematics and engineering applications—known colloquially as number-crunching applications. It has robust mathematical libraries that are used in scientific applications such as radio astronomy and high-energy physics, which require rapid high-precision mathematical computations. Fortran 90 now has all the data types and constructs of C.

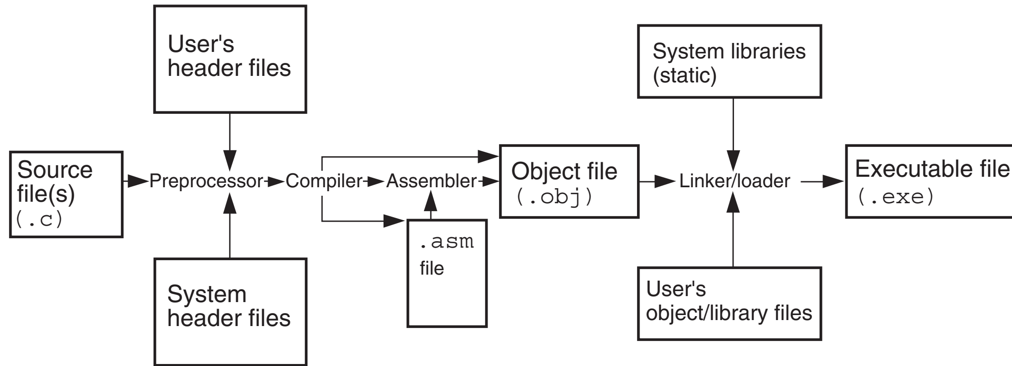
Overview of the Compilation Stages of a C Program

In the last section, we introduced the concept of HLLs and in particular pointed out that our focus is on C and Fortran 90. We will now examine the process of creating and compiling a complete C program, which is illustrated in Figure 1.1. Each step of the process is examined in this section. Fortran compilation is different in its details, but not in principle, and is not covered in this book.

Before a C computer program can be compiled and executed, the user must first create the source code in a text editor (not a word processor), and save it with the file extension `.c` or `.i`. It is considered good programming practice to give the source code a

trie_ch01.qxd 8/22/01 1:23 PM Page 10

FIGURE 1.1 Generic stages in the creation of a C application program



descriptive file name. For example, `average_cost.c` would be a good name for the C source code that determines the average cost of some number of items.

Compilation typically begins with the preprocessor phase.[2] That is, the compiler runs a preprocessor program, such as `cpp`, that modifies the C source code prior to its final compilation. Preprocessor statements are known as directives, and are denoted by lines with a # character in the first column. Directives are not executable instructions. Instead, they tell the compiler to do a variety of tasks; for example, to read in header (`.h`) files, define constants, and evaluate expressions.

Many basic tasks and computations are not defined directly by C. Instead, all C language implementations have collections of related functions and variables called header files. These header files contain definitions and declarations that are associated with their respective object code libraries. ANSI (American National Standards Institute) requires that certain standard library files be provided in every ANSI C/C++ implementation. All Itanium compilers are ANSI C/C++ compliant. In addition to the requisite header file, `stdio.h`, there are numerous other header files, most notably `stdlib.h` and `math.h`. These files supply the definitions and declarations for the library files, `libc.lib` and `libm.lib`, respectively.

After the preprocessor phase is finished, the compiler translates the source code and header files into a machine-readable equivalent

[2]The compilation syntax for the Itanium processor is specified in Chapter 6.

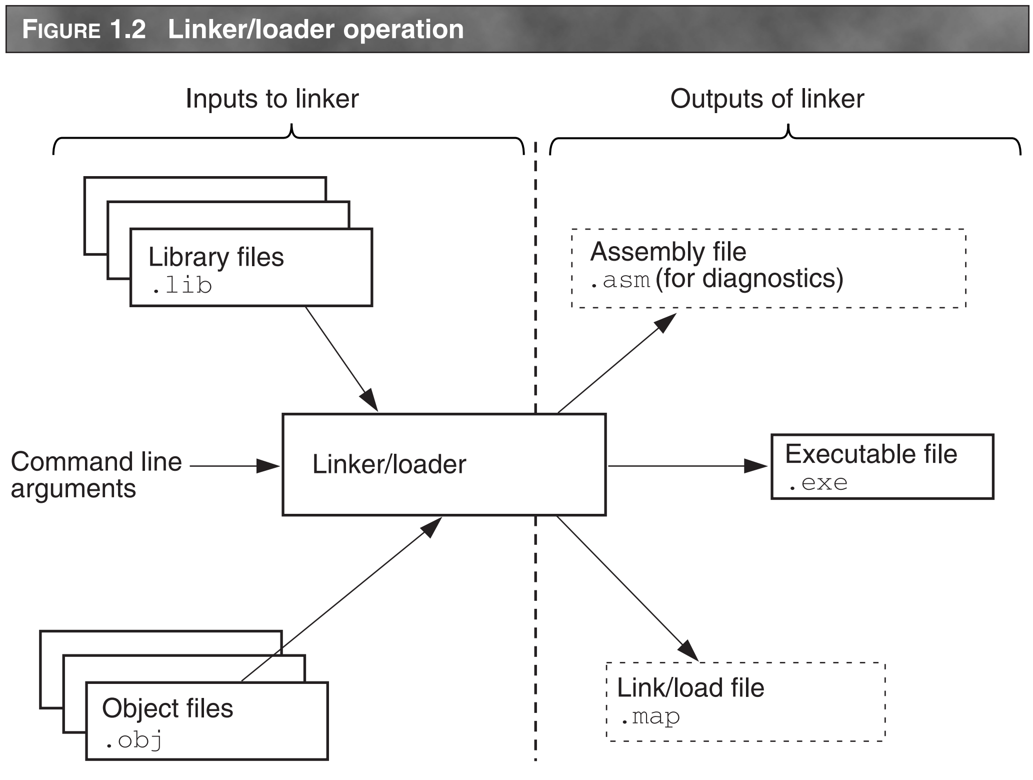
trie_ch01.qxd 8/22/01 1:23 PM Page 11

assembly language file, known as the object file. WinNT object files have the extension `.obj`.

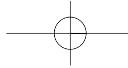
The object file and its associated object library files are not yet executable. The linker must bring them together. As part of the overall compilation process, the compiler calls the linker (`link.exe`) to combine one or more object files into a single program object file. In addition, the linker resolves external references, searches libraries, and performs all other processing required to create object files that are ready for execution. The loader then further manages the resulting object module, and this yields the executable file. This file is loaded into memory by the loader, which may be part of the linker or be a stand-alone program.

Figure 1.2 illustrates the linking process. Notice that the inputs of the linker are the individual object modules of the program. They are not the only inputs of the linker. If the modules reference any routines that are in libraries, the library files must also be supplied as inputs, as

FIGURE 1.2 Linker/loader operation



trie_ch01.qxd 8/22/01 1:23 PM Page 12



well as their associated header files. As shown in Figure 1.2, the executable file output by the compiler is shown with the extension `.exe`. Output files with dashed-line borders are optional.

An optional assembler listing (`.asm`) may be produced if desired by using the `-S` or `/Za` command line switch. Such a file is useful in understanding how the compiler translates the application program and how further optimizations might be accomplished.

THE APPLICATION PROGRAM

Application programs solve problems. For instance, predicting global warming and climatic changes is an example of a problem that must be solved by high-performance Internet servers and workstations. An application that performs this kind of computation typically consists of a massive suite of transaction programs interacting in real time, on clusters of high-end workstations.

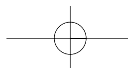
A more tractable problem is one that requires a large number of very accurate computations. For instance, a common type of problem that is performed on a workstation is an $n*n$ matrix multiplication, $n > 10{,}000$, in conjunction with a real-time data-acquisition system. This is an example of a mathematically intensive application with extensive data manipulation. There are specialized ANSI C/C++–compliant libraries for such problems that can be obtained from third-party vendors.

The examples that follow are explicatory—created primarily to demonstrate consistency with the current ANSI C standards, and the reader who is familiar with these principles may choose to skip sections.

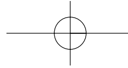
THE SOFTWARE DEVELOPMENT METHOD

Programming is a problem-solving activity that must be carried out in the proper order. This order is formally known as the *software development method*. The steps as stated by the ANSI committee in 1996 are:

1. Specify the problem requirements
2. Analyze the problem
3. Design the algorithm to solve the problem



trie_ch01.qxd 8/22/01 1:23 PM Page 13



4. Implement the algorithm in software
5. Test and verify the completed program
6. Maintain and update the program

Specification of the Problem

Specifying the problem means that you must state the problem clearly and without ambiguity in order to gain a clear understanding of what is required for its solution. The objective is to eliminate unimportant aspects and focus on the underlying or root problem. This may be difficult, because you may have incomplete or inaccurate information from your source.

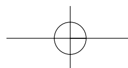
Analysis of the Problem

Analyzing the problem means identifying (a) the problem inputs—that is, data you have to work with; (b) the output(s)—that is, the desired result(s); and (c) additional constraints and requirements. Good practice is to make a list of the problem variables and their relationships to each other, perhaps expressed as formulas. It is often helpful to underline the phrases that identify the inputs and outputs. It is also necessary at this stage to determine the format in which the results must be visualized or displayed. For example, a table with column headings may be needed.

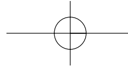
Design of the Algorithm

Designing the algorithm requires you to develop a list of steps that make up the algorithm to solve the problem, and then to verify the correctness of the algorithm. That is, you must verify that the algorithm solves the problem as intended. Writing the algorithm is often the most difficult part of the problem-solving process. Do not attempt to solve every detail at the beginning; instead use what is known as the “divide and conquer” approach in a *top-down manner*. First, list the major steps, or subproblems, that must be solved. Then solve the original problem by solving each of its subproblems. Most computer algorithms consist of at least the following subproblems:

1. Obtain data
2. Perform calculations
3. Display results



trie_ch01.qxd 8/22/01 1:23 PM Page 14



Once the subproblems are known, they can be attacked individually. Usually, the computation step must be broken down into a more detailed list of steps by a process known as *algorithm refinement*. Algorithm refinement is an integral aspect of the top-down approach, and may be thought of as an outline for writing a report or article. You must first create an outline of the major topics, which is refined by filling in subtopics for each major topic. Once the outline is complete, you may begin writing the code.

It is also always wise to *desk check* an algorithm. That is, do sample computations with known inputs that will produce known outputs by hand or with a calculator.

Implementation of the Algorithm

In this step, the algorithm is translated into a program. That is, each step in the algorithm must be converted into one or more statements in the programming language. Often this step is done by writing the program in pseudo-code, or (less frequently) by using flow charts.

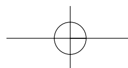
Testing of the Algorithm

This step requires both testing and verifying that the completed program produces the correct results for all possible sets of data. Making sure that it works for every possible situation is impossible for certain classes of problems; we address that problem in the maintenance phase.

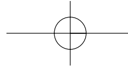
Maintenance of the Program

The maintenance phase involves updating and modifying a program to remove previously undetected errors, or to keep it up to date as dictated by government or the private sector as standards and policies are modified. A program that is more than five years old may well be obsolete and have no one who originally worked on it in-house.

By way of review, a cautionary note concerning errors is appropriate at this point. C is a difficult and unforgiving language, even for experienced programmers. Errors occur in all but the most trivial programs.



trie_ch01.qxd 8/22/01 1:23 PM Page 15



Recall that there are three ways to categorize errors. They are stated here in order of difficulty of correction:

1. Syntax errors—the program fails to compile, and error messages inform you of this. Error messages are especially well documented by the Itanium compiler, because new data types make the possibility of generating syntax errors greater.
2. Run-time errors—the program compiles and runs, but either crashes or continues to run indefinitely. Run-time errors sometimes occur because a library file cannot be found at linkage time. In addition, many new libraries augment the standard C libraries, and the programmer must be aware their existence. It is considered good programming practice to try to catch run-time errors by writing error trap routines. (A classic example is a divide-by-zero function that allows the offending code to be bypassed and corrected later.)
3. Logic errors (the bane of all programmers' existences)—the program executes, but the results are wrong. To help in finding such errors, Intel provides extensive debugging resources that include remote source level symbolic debugging for Itanium applications.

APPLYING THE SOFTWARE DEVELOPMENT METHOD—A CASE STUDY

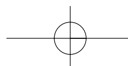
This first example is a simple program, and its purpose is to show some basic ANSI C rules and to review good programming practices.

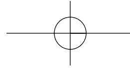
Problem statement: Compute and display the <u>total cost of perishable food items</u>, given the number of <u>items purchased</u> and the <u>cost of each item</u>. The total cost must include a <u>6% sales tax</u>.

This study includes only two items—apples and pumpkins. Once the original algorithm is correctly implemented, support for additional items may be added. Also, a way of inputting the items as they are chosen may be included, so that a device may be provided for the customer to keep a running total cost of items purchased.

First, summarize the important information. You can begin by underlining the phrases that identify the inputs and outputs, as shown.

Problem inputs:
- Number of apples purchased
- Cost per apple (in dollars)
- Number of pumpkins purchased
- Cost per pumpkin (in dollars)
- Sales tax (%)





Problem output:
Total cost of apples and pumpkins (dollars)

Once the problem inputs and outputs are known, you can develop the formulas needed to specify the relationships. The general formulas:

subtotal cost = item cost * number of items
total cost = tax * subtotal cost + subtotal cost

are correct, but do not give us the actual total cost of the items purchased. The algorithm needs specific information; it needs *refinement*. That is, you must re-state the relevant formulas in greater detail and include constant data:

total cost of pumpkins = cost per pumpkin * number of pumpkins
total cost of apples = cost per apple * number of apples
subtotal = total cost of pumpkins + total cost of apples
tax = 0.06
total cost = (tax * subtotal) + subtotal

You may then state the algorithm.

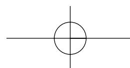
1. Get number of apples purchased
   a. Compute subtotal of apples
2. Get number of pumpkins purchased
   a. Compute subtotal of pumpkins
3. Calculate subtotal of apples and pumpkins purchased
4. Calculate tax for subtotal
5. Calculate total by adding subtotal and tax on subtotal

The implementation is a C program. To accomplish, this each algorithm step must be converted into one or more equivalent C statements. Figure 1.3 shows the C program.

The result produced when this sample program is run on an Itanium-based system is

```
The total is: $7.90
```

The program works, but is not general enough to be implemented as a real-world solution. The first deficiency is that provision is not made for more than two different items. This problem is easily corrected by simply adding more data and the associated assignment and computation statements. The second deficiency is less obvious. How can the user input data dynamically? Such data input is known as interactive input, and the C `stdio.h` library provides a function, `scanf`, for just this purpose and it is used in conjunction with `printf`



trie_ch01.qxd 8/22/01 1:23 PM Page 17

FIGURE 1.3 An example C application program

```
    /* This program performs a computation of a customer's bill.
    Customer bought some pumpkins and apples. Assume tax is 6%. */

    /* preprocessor phase */

#include <stdio.h>  // get I/O modules from C header library
#define TAX 0.06    // declare a constant that is self-documenting
                    // 0.06 is substituted for TAX in main

int main(void)      /* start of program */
{      /* declaration of variables used in the computation */
int num_pumpkins;              // number of pumpkins bought
int num_apples;                // number of apples bought
float price_pumpkin = 1.25;    // current price of 1 pumpkin
float price_apple = .40;       // current price of 1 apple
float subtotal,                // subtotal of bill without tax
      total;                   // total of bill with tax included

      /* quantity of goods bought by customer */
num_pumpkins = 5;        // "num_pumpkins" better than "x" = 5
num_apples = 3;          // ANSI C doesn't require

/* calculate the bill subtotal and total; i.e., implement the algorithm */
subtotal = (num_pumpkins * price_pumpkin) + (num_apples * price_apple);
total = TAX * subtotal + subtotal;      // tax defined in preprocessor

/* The subtotal should be 7.45, total should be 7.90 - testing phase */
/* print out the total; i.e., display results */
printf("The total is: $%5.2f\n",total); // printf is stdio.h function
return 0;                               // tell OS program is finished
}
```

to achieve interactive input as follows

```
      /* example of interactive data input */
printf("Enter number of apples:");        // user prompt
scanf(%d, &num_apples);  // computer waits for user to key in a number

printf("Enter number of pumpkins:");      // user prompt
scanf(%d, &num_pumpkins);                 // user keys in a number
```

These four statements replace the initialization statements in the original program:

```
num_pumpkins = 3;
num_apples = 5;
```

trie_ch01.qxd 8/22/01 1:23 PM Page 18

After modifying to include interactive input, a run of the program produces the following result:

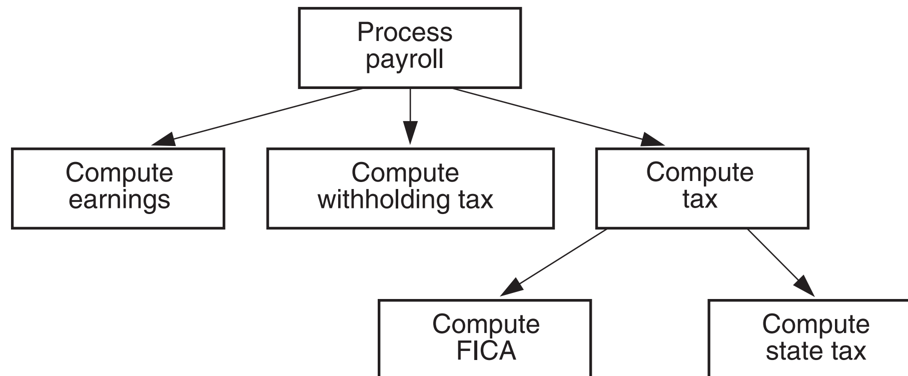
```
Enter number of apples: 1
Enter number of pumpkins: 10
Total is: $5.62
```

MODULAR PROGRAMMING—C VERSUS FORTRAN

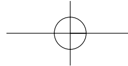
Modular programming techniques are indispensable in the development of modern application programs. They are known by a variety of descriptive terms—such as "divide-and-conquer"—because they feature subprograms that divide each individual task into smaller, more manageable modules known as *subroutines* (or *functions* in C, and *methods* in C++). The use of functions allows an algorithm to be partitioned into subroutines.

Figure 1.4 is a simple illustrative example of the modular programming concept applied to computing an employee's tax. This diagram is called a structure chart. Such charts are a widely used software development tool and are an important initial phase in creating modular applications. Each module (denoted by a rectangular box) performs a task that is needed one or more times during execution of the program.

FIGURE 1.4 Modular program structure chart for a simple payroll system



trie_ch01.qxd 8/22/01 1:23 PM Page 19



The development of modules may be partitioned between a number of programmers or programming groups. Thus, the individual modules may be independently planned, written, and tested. Only after determining that the separate modules are operating correctly will they be integrated to form the complete application program. In some cases, especially in very large applications, a number of completed modules may be combined to form a larger submodule of the program, and its operation tested.

The benefits of modular programming are taught in all computer science and engineering curricula in the first course on programming. For this reason, we shall only summarize some of the benefits:

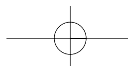
- Faster program development cycle is achieved by dividing the writing of the program between programmers or programming teams.
- A program implemented as separate modules is easier to develop, test, and debug.
- The logical organization of a modular-structured program is easier to read and maintain, thereby resulting in code that is more reliable.

General Structure of a Modular ANSI C/C++ Program

In the preceding case study, you were introduced to the structure of a simple C program consisting of one monolithic block of code, `main`. The building block of a C application program is a module called a function. (Note that C++ modules are known as methods.) In fact, `main` is a function that belongs to the operating system. This section continues by examining the general structure of a simple modular C program consisting of `main` and several other functions.

As its name implies, a function performs a specific operation or function. For instance, a function can be written that performs a specific mathematical computation or sequence of computations, restructures a body of data, or makes a decision or decisions based on computations or the structure of a body of data. It may or may not return a value, and may only return a single value.

Figure 1.5 illustrates the general flow of control in a modular C program. The operation of the program is initiated and controlled by `main`. First, `main` transfers control to function `a`; function `a` performs its operation; and then control is returned to `main`. In the same way, `main` initiates the operation performed by function `b`. Functions `a` and `b` may be called multiple times by the same function.



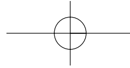
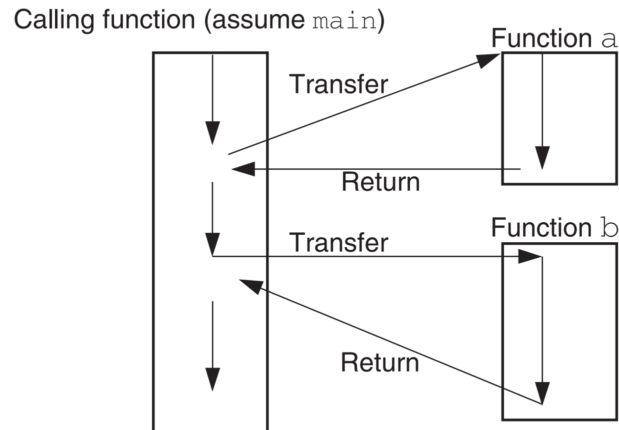


FIGURE 1.5 Flow of control of a modular C/C++ program



General Structure of a C Function

We have reviewed the organization of a C program and the role of functions. Our next step is to look into the general organization of the `main` function. Figure 1.6 presents the general structure of an ANSI C `main` function.

Use of comments to improve readability was discussed earlier. Looking at `main`, you can see that the first statement is the nonexecutable comment

```
/* general structure of main */
```

It is known as a block comment, and is intended primarily for multiline use. The beginning is delineated by `/*` and the end by `*/`. Line-based comments are marked by `//`. An example is

```
// comment for statement 1
```

This latter form originated in C++ and is now part of the ANSI C standard.

All C programs start with the declaration of `main`. This statement

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

identifies `main` to the operating system as the entry point of the C application. `main` is a keyword of the C language and is only used

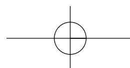


FIGURE 1.6 General structure of `main`

```
/* General structure of main*/

int main(void)
    {
                statement_1    // comment for statement 1
                statement_2    // comment for statement 2
                . . .
                statement_n    // comment for statement n
                return 0;      // must return an int
    }
```

in the definition of the main function. Recall that a keyword is a reserved word in C, and may only be used for its specified purpose. In this statement, `int` is another keyword. This keyword, which precedes the function name, is the return-data type. Typically, arguments to `main` are files of input or output data. The `(void)` keyword means that there are no inputs to `main`. Finally, `main` returns an integer with the statement

```
return 0;
```

The return value is 0 if the program completes successfully, and usually 1 or −1 if it does not.

The format of other functions in a C program is similar to that of `main`. A function, however, cannot run unless it is called by `main` (or another function that has been called by `main`). Figure 1.7 shows the general structure of a function that involves both arguments and a return value.

A function other than `main` should have a name that describes the operation of the function. Here we have simply used a generic name `funct_1`. Finally, a list of arguments, `int a, int b, char c,` is contained within parentheses. ANSI C allows up to 64 arguments of type `int`.

Since functions may return a value, a data type should be specified, even if none is returned. If there were no return value, the return type would be `void; int` is the C keyword for the integer data type, a 32-bit quantity on the Itanium-based computer. Other common data types are `float` for floating-point value and `char` for character value or

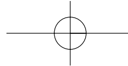


FIGURE 1.7 General structure of a typical function with arguments and a return value

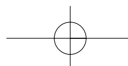
```
/* General structure of a function with 3 arguments that returns an
integer. It could be used as a template for a similar function */

int funct_1(int, int,char)        //function prototype
.
.
.
int funct_1(int a, int b,char c)
  {
              statement_1        // comment for statement 1
              statement_2        // comment for statement 2
              ...
              statement_n        // comment for statement n
              return fun_val;    // must return an int
  }
```

TABLE 1.1 Scalar Data Types Supported by Itanium Processors

Type	C name	Size (in bytes)
Integer	Signed char	1
	Unsigned char (default)	1
	Signed short	2
	Unsigned short (default)	2
	Signed int	4
	Unsigned int (default)	4
	Signed __int 64	8
	Unsigned _int 64 (default)	8
	Signed __int 128	16
	Unsigned _int 128 (default)	16
Pointer	*Any type* *	8
Floating-point	Float	4
	Double	8
	__float80	16
	__float128	16

byte. A summary of the common C scalar data types is presented in Table 1.1, including new Itanium 64-bit data types. Note that an `int` is now 4 bytes, and that 16-byte types are not directly supported by all hardware implementations as of this writing.



Example of a Modular C Program

An example of a C program that calls other functions is listed in Figure 1.8. This simple program executes sequentially to completion. In this program, output values are displayed on the screen and inputs are accepted from the keyboard.

FIGURE 1.8 Example of a modular ANSI C program

```
/* An example modular ANSI C program that calls 3 integer and 1
floating point arithmetic functions - addition, subtraction,
multiplication and division of two variables. It could be thought of
as an embryonic template for a calculator program */

#include <stdio.h>

/* function prototypes go here */
int aug(int, int);              //prototype for addition function
int subt(int, int);             //prototype for subtraction function
int prod(int, int);             //prototype for multiplication function
float quot(int, int);           //prototype for division function

int main(void)
{
/* declare variables: 2 arguments and 4 return values */
int a_in, b_in;                 // arguments to functions
int aug_m, sub_m, mul_m;        // return values
float div_m;

/* interactive input of variables */
printf("Enter first integer:");
scanf(%d, &a_in);                     // user's first number, a_in
printf("Enter second integer:");
scanf(%d, &b_in);                     // user's second number, b_in

      /* function calls */

aug_m = aug(a_in, b_in);      //assign add result to aug_m
sub_m = subt(a_in, b_in);     //assign sub result to sub_m
mul_m = prod(a_in, b_in);     //assign mult result to mul_m
div_m = quot(i a_in, b_in);   //assign div result to div_m
```

(*continued*)

trie_ch01.qxd 8/22/01 1:24 PM Page 24

FIGURE 1.8 (*Continued*)

```
/* display results */
printf("The results are: %d, %d, %d, %f\", aug_m, sub_m, mul_m, div_m);
return 0;                    // tell OS we're done
}

int aug(int x, int y)        // definition of function aug
{
return x + y;                // body of function aug
};
int subt(int x,int y)        // definition of function subt
{
return x - y;                // body of function subt
};
int prod(int x,int y)        // definition of function prod
{
return x * y;                // body of function prod
}
float quot(int x,int y)      // definition of function quot
{
return x / y;                // body of function prod
}
```

The `main` function calls four other functions—`aug, subt, mult`, and `quot`. These functions perform the arithmetic operations described in the comments. Note how the arithmetic statement is incorporated in the return statement of each function. For instance, the computation for the `aug` function is implemented as

```
return x + y;      // body of function aug
```

All ANSI functions must be declared before they are used. This requirement is known as prototyping, and was adopted from the C++ standard. The declarations must be placed before `main`. For example, the `aug` function is declared with the statement

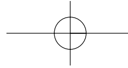
```
int aug(int, int); //prototype for addition function
```

Only the data types should be in the argument list of the prototypes—the compiler ignores any argument names in the prototype.

Functions are typically called by assigning a variable to their name. Notice that `aug` is assigned the name `aug_m` and called as

```
aug_m = aug(a_in, b_in); //assign add result to aug_m
```

trie_ch01.qxd 8/22/01 1:24 PM Page 25



The arguments listed in `main` and in the functions must be in the same order and be of the same type. The values of `a_in` and `b_in` are added, subtracted, multiplied, and then divided in each respective function.

In this example, the function definitions for add, subtract, multiply, and divide are listed in order. This has been done for practicality; the function definitions can be placed in any order.

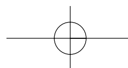
Functions in most programs are typically more complex. They may utilize decision-making statements, require repetitive (iterative) computations, call other functions, call themselves (recursion), and communicate with I/O devices. For this reason, many different C constructs exist. The C language also provides the means to create new or user-defined data types. In the chapters that follow, you will see a number of program constructs that are widely used in C as well as in Fortran.

THE STRUCTURE OF A FORTRAN PROGRAM

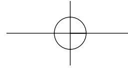
Fortran 90, like C, is a structured language. The syntax of Fortran 90 is quite different from that of C; but, as mentioned earlier, it possesses all the constructs of C. It has eliminated those constructs that tended to dissuade the implementation of modular code. The main difference in the modularity aspect is that Fortran has two kinds of modules. One is also known as the function. It can accept many arguments and returns only one value. The other is the subroutine, and also can accept many arguments, but either does not return any value or returns multiple values.

Fortran is still a card-image-oriented language in many implementations. That means the syntax still adheres to the archaic positional rules imposed by punch cards. More specifically:

1. Comments must begin in column one, and are denoted by an "`*`" or a "`C`."
2. Only columns 7 through 72 may be used for executable statements or data definitions.
3. Column 6 may hold any character, and if one is present it indicates a continuation of the preceding statement.
4. The program must begin with a title—for example, PROGRAM MY_ADD. Although no longer mandatory, upper case letters are typically used in certain constructs, such as the start and end statements.
5. Programs must end with an END statement.



trie_ch01.qxd 8/22/01 1:24 PM Page 26



6. Identifiers may only be six characters long (a rather stringent limitation).
7. Subprograms may not be nested, and in-line code is not supported.

As previously mentioned, Fortran 90 has all the data types and constructs (iterative, decision making, etc.) that are found in C. It also now has the capability of creating pointers and user-defined aggregate data types such as arrays and structures. This means that linked lists, recursion, and stack frames are now part of the Fortran language. One of Fortran's best features is a robust and extensive library of mathematical functions.

Figure 1.9 shows how our example C program is written in Fortran 90, implementing the same functions and degree of modularity. Although not tied to the card-image syntax, it is used in this example for complete generality.

FIGURE 1.9 Example of a complete Fortran program

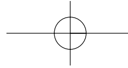
```
      program calc
****************************************************************
Program to calculate 4 simple arithmetic values
*    Input Variables:
*     a_in         : first input
*     b_in         : second input
*    Output Variables :
*     aug_m:       : first return value: a + b
*     sub_m:       : second return value: a - b
*     mul_m:       : third return value: a * b
*     div_m:       : fourth return value: a / b
****************************************************************
     INTEGER a_in, b_in, aug_m, sub_m, mul_m, add, mul, sub
     FLOAT div_m, div
*
* get first user input

      PRINT *, 'Enter first input:'
      READ *, a_in
*
* get second user input
*
      PRINT *, 'Enter second input):'
      READ *, b_in
*
```

FIGURE 1.9 (*Continued*)

```
* call functions:
*
      aug_m = add(a_in, b_in)
      sub_m = sub(a_in, b_in)
      mul_m = mul(a_in, b_in)
      div_m = div(a_in, b_in)
*
*    Print results to screen
     print *, aug_m, sub_m, mul_m, div_m
*
* End of Program CALC
      end
******************************
* start of functions
******************************
        integer function add(a_in, b_in)
        integer a_in, b_in
        add = a_in + b_in
        return
        end
*****************************
        integer function sub(a_in, b_in)
        integer a_in, b_in
        sub = a_in - b_in
        return
        end
*****************************
        integer function mul(a_in, b_in)
        integer a_in, b_in
        sub = a_in * b_in
        return
        end
*****************************
        float function div(a_in, b_in)
        integer a_in, b_in
        div = a_in/b_in
        return
        end
*
*****************************
```

trie_ch01.qxd 8/22/01 1:24 PM Page 28



MS PLATFORM SDK

The key objectives of the Platform SDK are to simplify installation of a development environment by integrating formerly discrete SDKs, introduce the latest technologies, and provide information about existing technologies. Visit the Microsoft Developer Network (MSDN*) Web site at http://msdn.microsoft.com for the latest information on the Platform SDK and MSDN.

The Platform SDK comprises the following main components:

- Headers, libraries, and type libraries needed to build applications
- Documentation providing technology overviews, detailed interface references, and tools usage
- Samples to demonstrate the various available technologies
- Tools to aid in the development and debugging of applications
- Redistributables to support application deployment

To get additional information about the MS Platform SDK, visit the Microsoft website at http://www.microsoft.com/msdownload/platformsdk/setuplauncher.htm.

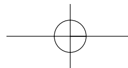
Building Applications with the Platform SDK

At the writing of this book, the current edition of the Platform SDK targets development for Whistler Beta, Microsoft Windows® 2000 64-Bit Edition (prerelease), Microsoft® Windows® Millennium Edition (Windows Me), Microsoft Windows 2000, Microsoft Windows NT® version 4.0, and Windows 98 operating systems.

The Platform SDK must be installed on Microsoft Windows® 2000 to use the tools to target Windows 2000 64-Bit Edition (prerelease). The 64-bit dlls, such as ATL, CRT, and MFC, located in the Platform SDK redist directory, may be loaded onto a 64-bit target machine for development purposes only and are not otherwise redistributable. None of the tools to support 64-bit development are redistributable. Products built with these tools may not be commercially distributed.

The following tools that are included in the SDK and may be of interest are not documented in the Platform SDK Documentation:

- symedit
- tlist
- kill



trie_ch01.qxd 8/22/01 1:24 PM Page 29

- porttool
- Cacls.exe
- NetWatch.exe
- Remote.exe
- Switcher.exe
- Walker.exe
- WinAt.exe

Information may be found by searching http://msdn.microsoft.com.

ITANIUM PROCESSOR SOFTWARE DEVELOPMENT TOOLS

As noted previously, any application program written in either the C or Fortran language must be converted to an executable file. The tools provided for Itanium processor-based computers include C/C++ and Fortran compilers, an assembler, a symbolic debugger, libraries, and a linker/loader. The process of creating an executable file for an application is known as the build process, and was implicitly discussed in the earlier discussion of a generic C language model.

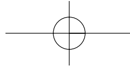
Once an executable file is created, the application program is ready to be tested and debugged. The Intel-supplied debugger is a full-featured source level symbolic debugger. Use of this debugger is presented in detail in Chapter 7.

Intel Corporation conveniently provides all the tools for Itanium processor software development in a standard software package—the Itanium Processor Software Development Environment. You should be sure to keep your SDK current. Information on the Intel Itanium processor software development environment can be found at www.developer.intel.com.

The Intel Itanium compilers use ANSI conventions and options to produce different types of output. For instance, if the version of the program is a prototype, a debug version of the code is required and easily created.

The compiler can also be instructed to produce other output files as part of the compilation process. For example, a source listing file (`.lst` extension) can be created for use in correcting errors or as reference when running the program. The programmer can also request output of an assembly language source file (`.asm` or `.s` extension). This output is useful in understanding how the compiler

trie_ch01.qxd 8/22/01 1:24 PM Page 30



has translated the code, and may also be useful if the program is not operating correctly. The use of the Intel C/C++ compiler to produce such output files is covered in Chapter 6.

Many third-party vendors offer Itanium processor software tools to compile and link with different programming languages. This book explains the Intel tool set. Designs for compilers from other companies will undoubtedly vary, but the concepts will be similar.