INTRODUCTION

TO

PROGRAMMING PRACTICES

© David Scuse
Department of Computer Science
University of Manitoba

INTRODUCTION TO PROGRAMMING PRACTICES

INTRODUCTION

When programming in Java, the Java virtual machine attempts to ensure that you do not perform any invalid operations. For example, Java generates an error condition if you attempt to access an element that does not exist in an array. While many of the C programming language features are very similar to the corresponding features in Java, unfortunately, error-checking is not one of them. In C, you can do almost anything that you want to, whether it makes sense or not. As a result, you must be much more careful when writing C programs.

These notes provide a very basic introduction to the C language and also to some of the tools that are useful when developing C programs. The notes are not a complete introduction to C – you can find that in an introductory C textbook. Instead, these notes cover the basic constructs in C and compare these constructs with the equivalent Java constructs.

Although the first four chapters of these notes focus on the C language, these chapters are just a warm-up to the programming practices discussed in the later chapters. Following good programming practices is necessary in a language such as C but it is also necessary to follow good practices in higher-level languages such as Java even though the compiler provides more support than the C compiler provides.

The programming conventions used in these notes are similar to the standard Java conventions. The particular set of conventions used does not really matter as long as you use the conventions consistently. When you are developing systems as a member of a team or in a programming course, you follow the conventions prescribed by the team or by the course instructor.

David Scuse
Department of Computer Science
University of Manitoba

September, 2008

1 BASIC C CONSTRUCTS

1.1 Introduction

In this chapter, we introduce the basic constructs used to build a C program. Many of these constructs are identical to the corresponding Java contructs.

1.2 Hello World

The "Hello World" program is the program that is most frequently used to illustrate a new programming language. Hello World is an important program not only because it provides a starting point when learning a new language but also because it ensures that the necessary infrastructure (compiler, paths to libraries, etc.) is set up correctly. The following **Java** program prints `Hello World!`.

```
import java.io.*;  // Java

public static void main(String[] parms)
{
     System.out.println("Hello World!");
}
```

The astute programmer will notice that the import statement is not required in this program since the only I/O is directed to System.out. The statement is included only because a similar statement is required in the corresponding C program shown below.

```
#include <stdio.h>

void main(int numParms, char *parms[])
{
     printf("Hello C World!\n");
}
```

As can be seen, the C program is very similar to the Java program. The system library `stdio` contains the I/O functions that are used in C. The parameters that are passed to the main function (`int numParms, char *parms[]`) are equivalent to the parameter passed to Java's main method (`String[] parms`); we will examine the meaning of these parameters in Chapter 4. The function `printf` (which is similar to Java's println method and is identical to Java's printf method) sends its output to the system console, `stdout`. `printf` does not begin a new line after printing the characters and so a newline character (`\n`) is included in the output string. Note the slight difference in terminology between the two languages – in Java, the term "method" is used while in C, the term "function" is used.

If you are using the `gcc` compiler, the program can be compiled using the following statement. The parameter `-o ch1` specifies the name of the output file (the executable file).

(See Chapter 10 for more information about C compilers.)

```
C:\Ch1>gcc –ggdb main.c -o ch1

main.c: In function 'main':
main.c:4: warning: return type of `main' is not 'int'
main.c:6:2: warning: no newline at end of file

C:\Ch1>
```

Even though there were two warnings (which will soon be fixed), the program can still be executed by typing the name of the exe file that was created by the compiler.

```
C:\Ch1>ch1
Hello C World!

C:\Ch1>
```

As was mentioned, the program generates two warning messages. The second warning message is "`no newline at end of file`". The C compiler expects an empty line at the end of each C source file. This is simple to fix, just add an empty line after the final "}".

```
#include <stdio.h>

void main(int numParms, char *parms[])
{
     printf("Hello C World!\n");
}
```

In these notes, the empty line is not always shown at the end of each source file but you should include it at the end of your programs.

The first warning message is "`return type of 'main' is not 'int'`". By convention, the main function should return a value that indicates whether or not the program executed successfully. The following program performs the same processing as the program above but it also indicates that the main function returns an int value (instead of being a void function, that is, not returning a value).

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     printf("Hello C World!\n");
}
```

The program no longer generates a warning but the program also does not explicitly return a return value/code to the operating system. We can explicitly return a return code as shown below.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     printf("Hello C World!\n");
     return 0;
}
```

So now the program compiles correctly, without any warning messages.

With most of the sample programs in these notes, the output generated by the program is shown immediately following the program. For example, the program shown above generates `Hello World!` as its output.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     printf("Hello C World!\n");
     return 0;
}

Hello World!
```

So now that we have a working C program that we can compile and execute, we can move on to bigger and better programs.

1.3 Comments

The C language supports the standard `/* … */` style of comments. These comments may extend over multiple lines. However, many C compilers also support the single-line comment that begins with `//` (this is part of the ANSI/ISO C standard C99). The `//` style of comment is used in these notes for convenience.

1.4 The `int` Data Type

The C language contains the same basic data types as Java. The following C program declares a variable sum to be an `int` (integer) and then computes the sum of the first 5 positive integer values. As in Java, every variable that is used in a program must first be declared to be of the appropriate type.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int sum;

     sum = 1 + 2 + 3 + 4 + 5;
     return 0;
```

```
}
```

It would be nice to know the value that is stored in the variable sum. In Java, we can display the value of sum quite easily using the following statement.

```
     System.out.println("The sum is " +sum);  // Java
```

Unfortunately, C output is slightly more complex. The following C statement generates the same result as the Java statement shown above.

```
     printf("The sum is %d\n", sum);
```

The `printf` statement takes its first parameter, a string, and uses the contents of the string to format the subsequent parameter(s). Each parameter that is to be printed is associated with a format code that begins with a "`%`". The format code for printing the value of an int is `%d`. Format codes are described in additional detail later in this chapter.

1.5 Loops

The program in the previous section that summed the integers from 1 to 5 hard-coded the values to be summed. With a small number of values, this might be acceptable but for a large number of values, hard-coding them is not acceptable. A simple for loop can be used to generalize the processing. Note that the loop is identical to the corresponding loop in Java.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int sum;
     int count;

     sum = 0;
     for (count=1; count<=5; count++)
     {
        sum = sum + count;
     }
     printf("The sum is %d\n", sum);
     return 0;
}

The sum is 15
```

C also includes the += operator which makes the addition of a value to a variable even easier.

```
         sum += count;
```

C also supports a while loop that is also identical to Java's while loop.

```
#include <stdio.h>
```

```
int main(int numParms, char *parms[])
{
     int sum;
     int count;

     sum = 0;
     count = 1;
     while (count<=5)
     {
         sum += count;
         count++;
     }
     printf("The sum is %d\n", sum);
     return 0;
}

The sum is 15
```

1.6 Conditional Execution

C includes a condition statement (if statement) that is identical to Java's condition statement. The following program determines the sum of the even numbers from 1 to 5, inclusive. Note that the remainder operator in C (%) is the same as Java's remainder operator. The condition statement is not actually required (the loop parameters could be modified instead) but the point of the example is to illustrate the condition statement. The logical comparison/equals operator (==) is the same as Java's.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int sum;
     int count;

     sum = 0;
     for (count=1; count<=5; count++)
     {
         if ((count%2)==0)
         {
             sum += count;
         }
     }
     printf("The sum of the even values is %d\n", sum);
     return 0;
}

The sum of the even values is 6
```

C's `if` statement may also include an else clause, as shown below.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int sumEven;
     int sumOdd;
```

```
     int count;

     sumEven = 0;
     sumOdd = 0;
     for (count=1; count<=5; count++)
     {
        if ((count%2)==0)
        {
           sumEven += count;
        }
        else
        {
           sumOdd += count;
        }
     }
     printf("The sum of the even values is %d \n", sumEven);
     printf("The sum of the odd values is %d \n", sumOdd);

     return 0;
}

The sum of the even values is 6
The sum of the odd values is 9
```

Although you may not have encountered Java's conditional element (operator), both Java and C support an identical (and simplified) if statement, the conditional operator.

```
int result;
int value1;
int value2;

result = ( (value1 < value2) ?  value1 : value2);
```

The effect of the statement above is that the logical expression `(value1 < value2)` is evaluated: if the result of the expression is true, the expression that follows the "?" is returned as the result of the conditional operator; if the result of the expression is false, the expression that follows the ":" is returned as the result. The conditional operator above is identical to the if statement shown below (which determines the smaller of two integer values).

```
if (value1 < value2)
{
     result = value1;
}
else
{
     result = value2;
}
```

1.7 Arrays

C also supports arrays that are equivalent to Java arrays, as shown below.

```
#include <stdio.h>

int main(int numParms, char *parms[])
```

```
{
     int sum;
     int count;
     int values[5];

     for (count=0; count<5; count++)
     {
        values[count] = count + 1;
     }

     sum = 0;
     for (count=0; count<5; count++)
     {
        sum += values[count];
     }
     printf("The sum of the values is %d \n", sum);

     return 0;
}
```

```
The sum of the values is 15
```

Note that the size of the array (5 elements) is hard-coded throughout the program. We will examine techniques later in this chapter that remove the need for hard-coding constant values repeatedly in a program.

Unlike Java, one array can **not** be assigned to another array in C. For example, the following assignment statement is **not valid**:

```
     int values[5];
     int moreValues[5];

     moreValues = values;        // Wrong!!
```

The contents of one array can be copied to another array by copying the individual elements one at a time using a loop. Unfortunately, C does not provide a function equivalent to Java's `System.arraycopy` method; although, it is simple enough for the programmer to write such a function (and we will do that in the next section).

Like Java, the contents of an array can not be printed without using a loop. (There is one exception to this statement – it is examined later in this chapter when we examine character strings.)

1.8 Functions

The C language supports user-defined functions in essentially the same way that Java supports user-defined methods. For example, the following program computes `1+2+3+…+N` where `N` is a parameter that is passed to the function.

```
#include <stdio.h>
```

```
int main(int numParms, char *parms[])
{
     int result;

     result = sum(5);
     printf("The sum is %d\n", result);
     return 0;
}

int sum(int n)
{
     int count;
     int result;

     result = 0;
     for (count=1; count<=n; count++)
     {
        result += count;
     }
     return result;
}

The sum is 15
```

Although the program compiles and executes correctly, the compiler generates the following warning message:

```
C:/Ch1/main.c: In function `main':
C:/Ch1/main.c:7: warning: implicit declaration of function `sum'
```

Unlike Java, the C compiler gets upset when a function (in this case `sum`) is used (or referred to) before it is defined. We could move the definition of sum before the definition of main but that would make the program more difficult to read. Instead, in C we define the prototype of the function at the beginning of the source file (the prototype is essentially the same as the signature of a Java method). Now the function `sum` can be used in `main` without having warning messages generated because the C compiler already knows what the `sum` function looks like. We must do this for each function that is referenced before it is defined.

```
#include <stdio.h>

int sum(int);  // function prototype

int main(int numParms, char *parms[])
{
     int result;

     result = sum(5);
     printf("The sum is %d\n", result);
     return 0;
}

int sum(int n)
{
     int count;
     int result;
```

```
     result = 0;
     for (count=1; count<=n; count++)
     {
        result += count;
     }
     return result;
}

The sum is 15
```

As with Java, if a function does not return a value, its type is `void`. A void function is sometimes referred to as a procedure.

In the previous section, we noted that one array can not be assigned directly to another array; nor can an array be printed in one statement. The following program defines two functions that are useful when manipulating arrays, `arrayCopy` copies elements from one array to another array and `arrayPrint` prints the elements in an array. Note that C supports array initialization in the declaration statement in the same manner as does Java.

```
#include <stdio.h>

void arrayCopy(int[], int[], int, int, int);
void printArray(int, int[]);
void zeroArray(int, int[]);

int main(int numParms, char *parms[])
{
     int array1[] = {10, 20, 30, 40, 50};  // array initialization
     int array2[5];

     printArray(5, array1);
     arrayCopy(array1, array2, 0, 0, 5);
     printArray(5, array2);

     zeroArray(5, array2);
     arrayCopy(array1, array2, 0, 1, 4);
     printArray(5, array2);

     zeroArray(5, array2);
     arrayCopy(array1, array2, 1, 0, 4);
     printArray(5, array2);

     zeroArray(5, array2);
     arrayCopy(array1, array2, 1, 2, 2);
     printArray(5, array2);

     return 0;
}

void arrayCopy(int fromArray[],int toArray[],int fromStart,int toStart, int length)
{
     int count;
     for (count=0; count<length; count++)
     {
        toArray[toStart+count] = fromArray[fromStart+count];
     }
}
```

```
void printArray(int length, int array[])
{
     int count;
     for (count=0; count<length; count++)
     {
        printf("%2d ", array[count]);
     }
     printf("\n");
}

void zeroArray(int length, int array[])
{
     int count;
     for (count=0; count<length; count++)
     {
        array[count] = 0;
     }
}

10 20 30 40 50
10 20 30 40 50
 0 10 20 30 40
20 30 40 50  0
 0  0 20 30  0
```

In C, when a basic data type is passed to a function, it is passed by value (also referred to as "call by value"). This is the same as in Java. For example, in C if an `int` is passed to a function, a copy is made of the current contents of the `int` variable and this copy is passed to the function. If the function modifies the value of a parameter, the new value remains in effect for the life of the function but the original value in the calling function is not modified.

In C, when an array is passed to a function, a pointer to the array is passed (this is identical to the processing performed in Java). In the function, the value of the pointer to the array can be modified but this modification is visible only within the function, not in the calling function. However, as with Java, the contents of the array that is pointed to can be modified and these modifications do affect the contents of the array in the calling function. This type of parameter passing is sometimes referred to as "call by reference" even though it is really call by value. This topic will be examined in more detail in Chapter 4.

1.9 More Array Manipulation

In this section, we examine some additional features of array manipulation. In the following program, the variable `MAX_VALUES` is a literal constant that is the same as a Java `final` variable. The use of a literal constant removes the need to hard-code constant values throughout a program.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_VALUES = 5;     // Literal Constant
     int sum;
```

```
     int count;
     int values[MAX_VALUES];

     for (count=0; count<MAX_VALUES; count++)
     {
        values[count] = count + 1;
     }

     sum = 0;
     for (count=0; count<MAX_VALUES; count++)
     {
        sum += values[count];
     }
     printf("The sum of the values is %d \n", sum);

     return 0;
}

Sum of the values is 15
```

If an array is passed to a function, the number of elements in the array must also be passed as a parameter since C does not provide a mechanism for determining the number of elements in an array at run-time.

```
#include <stdio.h>

int sum(int, int[]);

int main(int numParms, char *parms[])
{
     const int MAX_VALUES = 5;
     int result;
     int count;
     int values[MAX_VALUES];

     for (count=0; count<MAX_VALUES; count++)
     {
        values[count] = count + 1;
     }

     result = sum(MAX_VALUES, values);
     printf("The sum of the values is %d \n", result);
     return 0;
}

int sum(int numEntries, int entries[])
{
     int result;
     int count;

     result = 0;
     for (count=0; count<numEntries; count++)
     {
        result += entries[count];
     }
     return result;
}

The sum of the values is 15
```

Actually, the preceding statement is not entirely true. It is possible to determine the number of elements in an array *in the function in which the array is declared* by using the `sizeof` function.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_VALUES = 5;
     int result;
     int count;
     int values[MAX_VALUES];

     for (count=0; count<MAX_VALUES; count++)
     {
        values[count] = count + 1;
     }

     printf("Size of array is %d \n", sizeof(values));
     return 0;
}

Size of array is 20
```

The value 20 is obviously not the number of elements in the array. What C is telling us is the number of bytes of memory that the array occupies. Since each element of the array occupies more than one byte, we need to divide the total amount of memory allocated for the array by the amount of storage that is allocated for one `int` value. The following simple modification causes the correct result to be generated.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_VALUES = 5;
     int result;
     int count;
     int values[MAX_VALUES];

     for (count=0; count<MAX_VALUES; count++)
     {
        values[count] = count + 1;
     }

     printf("Size of array is %d \n", sizeof(values)/sizeof(int));
     return 0;
}

Size of array is 5
```

Unfortunately, as was mentioned, `sizeof` works correctly only in the function in which the array is declared. If an array is passed as a parameter, no information is passed along with the array that permits the function to determine correctly the number of elements that are in the array. While we can use the `sizeof` function in a called function, C generates a value of 1 for

the number of elements, regardless of the actual size of the array. It is for this reason that the actual number of elements in an array is passed to any function that processes the array. (An alternative to passing the number of elements in an array is to place a sentinel value after the last used element in the array.)

```
int sum(int numEntries, int entries[])
{
     int result;
     int count;

     printf("Size of array is %d \n", sizeof(entries)/sizeof(int));

     result = 0;
     for (count=0; count<numEntries; count++)
     {
        result += entries[count];
     }
     return result;
}

Size of array is 1
```

1.10 Variable-Length Arrays

In the updated C standard (C99), the size of an array can be declared at run time (instead of at compile time) as shown in the following program. This type of array is referred to as a "variable-length array".

```
#include <stdio.h>

int sum(int);

int main(int numParms, char *parms[])
{
     int numValues = 5;
     int result;

     result = sum(numValues);
     printf("%d\n", result);

     return 0;
}

int sum(int nentries)
{
     int numEntries = nentries * 2;
     int entries[numEntries];
     int result;
     int count;


     result = 0;
     for (count=0; count<numEntries; count++)
     {
        entries[count] = count+1;
        result += entries[count];
```

```
     }
     return result;
}

55
```

We will examine an alternative mechanism for creating arrays dynamically in Chapter 4.

1.11 Recursive Functions

The C language supports recursive functions in the same way that Java does. For example, the following program uses recursion instead of iteration to compute `1+2+3+…+N` for `N≥1`.

```
#include <stdio.h>

int sum(int);

int main(int numParms, char *parms[])
{
     int result;

     result = sum(5);
     printf("The sum is %d\n", result);
     return 0;
}

int sum(int n)
{
     int result;

     if (n>1)
     {
        result = n + sum(n-1);
     }
     else
     {
        result = 1;
     }
     return result;
}

The sum is 15
```

Note that if the condition `N≥1` is not true, the function returns the value `1`.

1.12 Other C Data Types

The C language contains the same basic data types as does Java (with one exception). These data types are:

Data Type	Format Code	Meaning
int	%d	an integer

float	%f	a floating-point value
double	%f	double-precision value
char	%c	a single character

Some of the data types may be modified by adding an appropriate modifier (`short`, `long`, `signed`, `unsigned`). For example, an `int` may be modified to be an `unsigned int`. An `unsigned int` can store a larger value than a `signed int` but an `unsigned int` can only store non-negative values.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     char charValue = 'a';
     int intValue = 33;
     float floatValue = 4.1;
     double doubleValue = 5.1;

     printf("%c, %d, %f, %f", charValue, intValue, floatValue, doubleValue);
     return 0;
}

a, 33, 4.100000, 5.100000
```

The format codes shown above are just the basic codes. The `printf` statement supports many elaborations on these format codes. For example, the following program illustrates the use of a width specification in the format codes.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     char charValue = 'a';
     int intValue = 33;
     float floatValue = 4.1;
     double doubleValue = -5.1;

     printf("%2c, %3d, %4.1f, %4.1f", charValue, intValue, floatValue,doubleValue);
     return 0;
}

 a,  33,  4.1, -5.1
```

The Java boolean data type (which can be assigned either the value true or the value false) is not included in the list of C basic data types. Instead, C uses integer values to represent true and false. The following program illustrates how the results of logical expressions are manipulated.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
```

```
     int value1;
     int value2;

     value1 = 3 < 5;
     value2 = 4 != 4;
     if (value1 && value2)
     {
        printf("The expression %d and %d is true\n", value1, value2);
     }
     else
     {
        printf("The expression %d and %d is false\n", value1, value2);
     }

     if (value1 || value2)
     {
        printf("The expression %d or %d is true\n", value1, value2);
     }
     else
     {
        printf("The expression %d or %d is false\n", value1, value2);
     }
     return 0;
}

The expression 1 and 0 is false
The expression 1 or 0 is true
```

When C saves the result of evaluating a logical expression, it represents true and false by the values 1 and 0, respectively. However, when C evaluates the result of a logical expression that contains integer values, 0 is interpreted as false and any non-zero value is interpreted as true. The following program illustrates this principle.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int value1;
     int value2;

     value1 = 3;
     value2 = 0;
     if (value1 && value2)
     {
        printf("The expression %d and %d is true\n", value1, value2);
     }
     else
     {
        printf("The expression %d and %d is false\n", value1, value2);
     }

     if (value1 || value2)
     {
        printf("The expression %d or %d is true\n", value1, value2);
     }
     else
     {
        printf("The expression %d or %d is false\n", value1, value2);
     }
     return 0;
```

```
}

The expression 3 and 0 is false
The expression 3 or 0 is true
```

Even though any non-zero value is interpreted as true, it is a good programming practice to use only the value 1 to represent true.

1.13 Casting Between Data Types

In Java, it is occasionally necessary to cast one data type (or object) to another data type (object). The same is true in C.

When converting from one basic data type to another, C does not insist on a cast but a cast may be included. For example,

```
     char char1;
     int int1;
     long int longInt1;
     float float1;
     double double1;

     char1 = 'a';
     longInt1 = 23;
     float1 = 1.0;

     int1 = char1;
     int1 = (int) char1;
     char1 = int1;

     int1 = longInt1;
     longInt1 = int1;

     float1 = (float) double1;
     double1 = float1;
```

We shall see in later chapters why explicit casts are sometimes necessary.

1.14 Structures

In Java, there is no simple mechanism for defining a group of related data items together so that they can be manipulated as a whole. Instead, you can create an object that contains an instance variable for each data item. The following Java program illustrates the use of an object to group related data items together.

```
public class TestPerson          // Java program
{
     public static void main(String[] parms)
     {
        Person person1;

        person1 = new Person();
```

```
        person1.number = 25;
        person1.age = 30;
        System.out.println("Person1: " +person1.number +" " +person1.age);
    }
}

public class Person          // Java program
{
    public int number;
    public int age;
}

Person1: 25 30
```

In C, there is a convenient language feature called a "`struct`" that provides a similar facility. The following is the C language equivalent of the Java program shown above.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
    struct
    {
        int number;
        int age;
    } person1;

    person1.number = 25;
    person1.age = 30;

    printf("Person1: %d %d \n", person1.number, person1.age);
    return 0;
}

Person1: 25 30
```

The structure is defined using the `struct` keyword. Note that there must be a semicolon at the end of the structure definition. The variable, `person1`, at the end of the structure definition is an instance of the specified structure. The variable `person1` can be thought of as similar to an object that contains the public instance variables `number` and `age`. The variables in `person1` are referenced in exactly the same way that a public variable in a Java object is referenced – by specifying the identifier `person1`, a period ("."), and the name of one of the variables defined in the structure.

An alternative method of declaring a structure is shown below. In this program, the structure is given a name which can be used in subsequent declaration statements. The advantage of this method is that once the structure has been defined, it may be used in the declaration of any number of variables.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
```

```
     struct person
     {
        int number;
        int age;
     };

     struct person person1;

     person1.number = 25;
     person1.age = 30;

     printf("Person1: %d %d \n", person1.number, person1.age);
     return 0;
}

Person1: 25 30
```

Just as the object referred to by a variable in Java may be assigned to another variable, the contents of a struct variable may be assigned to another struct variable. Unlike Java (which simply copies the reference of – or pointer to – the object), C copies the contents of the structure. The following example illustrates this process.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     struct person
     {
        int number;
        int age;
     };

     struct person person1;
     struct person person2;

     person1.number = 25;
     person1.age = 30;

     person2 = person1;

     person2.number += 10;

     printf("Person1: %d %d \n", person1.number, person1.age);
     printf("Person2: %d %d \n", person2.number, person2.age);
     return 0;
}

Person1: 25 30
Person2: 35 30
```

Note that modifying `person2` does not have any impact on the contents of `person1`.

Structures can not be compared for equality directly – instead, you must compare corresponding elements in order to determine if the contents of two structures are equal. We will examine how this can be accomplished in the next section.

A structure may include one or more arrays inside the structure in addition to any necessary basic data types. For example, the following is a valid structure:

```
struct student
{
     int number;
     int age;
     int grades[5];
};
```

The array grades is accessed in the same manner as a basic data item.

```
int count;
int sum;
struct student student1;

sum = 0;
for (count=0; count<5; count++)
{
     sum += student1.grades[count];
}
```

1.15 Passing Structures to Functions

A structure is passed as a parameter to a function in exactly the same way that a basic data type is passed (call by value). The following program illustrates passing a structure to a function. Note that the definition of the structure has been moved to the beginning of the program before the main function so that the structure definition is available (i.e. is global) to all functions in this file (more on this in Chapter 3).

```
#include <stdio.h>

struct person
{
     int number;
     int age;
};

void printPerson(struct person);

int main(int numParms, char *parms[])
{
     struct person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}

void printPerson(struct person person1)
{
     printf("Person: %d %d \n", person1.number, person1.age);
}
```

```
Person: 25 30
```

When a structure is passed to a function, it is passed by value. This means that if the contents of the structure are modified by the function, the changes are **not** reflected in the calling function.

The following program illustrates the use of a function that compares 2 instances of the person struct. Again, the definition of the structure has been moved outside of the main function to make the structure global to the entire program. We will examine global variables in more detail in Chapter 3.

```
#include <stdio.h>

struct person
{
     int number;
     int age;
};

void printPerson(struct person);
int equalPersons(struct person, struct person);

int main(int numParms, char *parms[])
{
     struct person person1;
     struct person person2;

     person1.number = 25;
     person1.age = 30;

     person2.number = 50;
     person2.age = 50;

     printPerson(person1);
     printPerson(person2);

     if (equalPersons(person1, person2))
     {
        printf("The two structures are equal.\n");
     }
     else
     {
        printf("The two structures are not equal.\n");
     }

     return 0;
}

void printPerson(struct person person1)
{
     printf("Person: %d %d \n", person1.number, person1.age);
}

int equalPersons(struct person person1, struct person person2)
{
     int result;
```

```
     result = ((person1.number==person2.number) && (person1.age==person2.age));
     return result;
}

Person: 25 30
Person: 50 50
The two structures are not equal.
```

Note that the `equalPersons` function is essentially the same as the Java `equals` method that would be written to compare two Java objects of type `Person`.

Passing a structure by value to a function includes any enclosed arrays. For example, the array `grades[5]` defined in the previous section is passed by value. This is contrary to the normal rule for passing arrays to functions.

1.16 Arrays of Structures

Parallel arrays are used to maintain a collection of groups of related data items. In Java, parallel arrays were used primarily to make life difficult for students – as you should be aware by now, instead of using parallel arrays in Java, you would instead use an array of objects where each object contains the necessary instance variables.

In C, we do exactly the same thing. Although parallel arrays are certainly valid, it is normally more convenient to define an array of structures.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     struct person
     {
        int number;
        int age;
     };

     const int MAX_PERSONS = 5;
     struct person persons[MAX_PERSONS];
     int count;

     for (count=0; count<MAX_PERSONS; count++)
     {
        persons[count].number = 1000 + count;
        persons[count].age = 30 + count;
     }

     for (count=0; count<MAX_PERSONS; count++)
     {
        printf("Person %d: %d %d \n",
                                count, persons[count].number, persons[count].age);
     }

     return 0;
}
```

```
Person 0: 1000 30
Person 1: 1001 31
Person 2: 1002 32
Person 3: 1003 33
Person 4: 1004 34
```

We can also pass an array of structures to a function. The structure definition must be global so that all functions in the source file can access the structure.

```
#include <stdio.h>

struct person
{
     int number;
     int age;
};

void printPersons(int, struct person[]);

int main(int numParms, char *parms[])
{
     const int NUM_PERSONS = 5;

     struct person persons[NUM_PERSONS];
     int count;

     for (count=0; count<NUM_PERSONS; count++)
     {
        persons[count].number = 1000 + count;
        persons[count].age = 30 + count;
     }

     printPersons(NUM_PERSONS, persons);

     return 0;
}

void printPersons(int numPersons, struct person persons[])
{
     int count;

     for (count=0; count<numPersons; count++)
     {
        printf("Person %d: %d %d \n",
                              count, persons[count].number, persons[count].age);
     }
}

Person 0: 1000 30
Person 1: 1001 31
Person 2: 1002 32
Person 3: 1003 33
Person 4: 1004 34
```

1.17 Strings

The `char` data type exists in C as well as in Java. The following program fills a `char` array with the 26 lower-case letters in the alphabet.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     char lowerCase[26];
     int count;

     lowerCase[0] = 'a';
     for (count=1; count<26; count++)
     {
         lowerCase[count] = lowerCase[count-1] + 1;
     }

     for (count=0; count<26; count++)
     {
        printf("%c", lowerCase[count]);
     }
     printf("\n");

     return 0;
}

abcdefghijklmnopqrstuvwxyz
```

Unfortunately, since the size of each array must be declared at compile time, we can not resize an array (as we could in Java) once we know how many characters are to be processed as a collection. As a result, in C there is a convention that a special sentinel character (`'\0'`) is placed after the last valid character in a collection of characters. A char array that is terminated with the sentinel character is referred to as a “string” in the C language. A string is not a special data type, a string is just an array of characters that is properly terminated.

The following program creates a properly terminated string. The entire string can then be printed using the `%s` format code. This is the exception to the rule mentioned earlier that in order to print the contents of an array, a loop must be used to print the array elements one at a time.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     char lowerCase[27];
     int count;

     lowerCase[0] = 'a';
     for (count=1; count<26; count++)
     {
         lowerCase[count] = lowerCase[count-1] + 1;
     }

     lowerCase[26] = '\0';

     printf("%s\n", lowerCase);

     return 0;
```

```
}

abcdefghijklmnopqrstuvwxyz
```

As long as a string is terminated correctly, the size of the array in which the string is stored may be any value, as long as there is sufficient room to hold all of the characters in the string plus the sentinel value. When printed using the `%s` format code, only the characters up to the sentinel value are displayed.

Recall that Java contains a class named String and this class contains a variety of methods that manipulate the contents of a String. Since C does not support objects, an array of char's is the best that we can do, but C does provide the `string` library of functions that can be used to manipulate C's equivalent (a char array that is correctly terminated with the `'\0'` character) of a Java String.

The following are some additional functions in the C string library. (Any good C reference manual should contain a list of the details of each function in the library.) Each function assumes that the source string is properly terminated; if the string is not properly terminated, the results will be unpredictable (and possibly disasterous to the health of your program).

```
strcpy(char destination[], char source[]);   // copy the contents of source
                                             // to destination

strncpy(char destination[], char source[], int n); // copy the first n characters
                                                   // from source to destination

strncat(char destination[], char source[], int n); // append n characters from
                                                   // source to destination

strlen(char source[]);   // return the number of characters in source
                         // (not including the string termination character)

strcmp(char string1[], char string2[]); // compare the contents of the two strings
                                        // return a negative value if s1 < s2
                                        // return zero if s1 == s2
                                        // return a positive value if s1 > s2
```

The following program illustrates the use of the `strcpy(to, from)` function. Note that C permits the use of a string constant in some situations (such as initializing an array).

```
#include <stdio.h>
#include <string.h>

int main(int numParms, char *parms[])
{
     char string1[] = "This is a string.";
     char string2[10];
     char string3[20];

     strcpy(string2, string1);
     strcpy(string3, string1);

     printf("%s\n", string1);
```

```
     printf("%s\n", string2);
     printf("%s\n", string3);

     return 0;
}

.
This is a string.
.
```

However, the output is clearly incorrect. If you examine the program closely, you should notice that `string2` is not long enough to contain all of the characters in `string1` plus the sentinel character. As a result, when `strcpy` copies the characters from `string1`, it overwrites whatever comes after `string2` in memory.

The problem can be fixed by increasing the size of `string2`.

```
#include <stdio.h>
#include <string.h>

int main(int numParms, char *parms[])
{
     char string1[] = "This is a string.";
     char string2[20];
     char string3[20];

     strcpy(string2, string1);
     strcpy(string3, string1);

     printf("%s\n", string1);
     printf("%s\n", string2);
     printf("%s\n", string3);

     return 0;
}

This is a string.
This is a string.
This is a string.
```

This was a simple program but the complications caused by having a destination string too small to receive all of the characters in the source string are significant. One of the difficulties with C programming is that such an error may not be immediately obvious and debugging the program could take a significant amount of time.

Any time that you use the string functions to copy characters from one array to another array, the potential exists for over-writing the memory after an array if the array is not sufficiently large to contain all of the characters or if the source array is not properly terminated.

1.18 The typedef Statement

An alternative mechanism for defining a structure (and other data types as well) is the

`typedef` statement. The `typedef` statement defines a new data type (that is really just an alias for a basic data type or structure).

The following program illustrates the use of the `typedef` statement to define the `person` structure used in the previous sections. Note that the keyword `struct` is only used in the `typedef` statement – subsequently, just the name of the structure (`person` in this case) is used. Also note that the name of the new data type is specified at the end of the declaration. In this program, the `typedef` statement is defined globally (prior to the main function) because the structure is used in more than one function. The `typedef` statement may also be defined within a function if it will be used only within that one function.

```
#include <stdio.h>

typedef struct
{
     int number;
     int age;
} person;


void printPerson(person);

int main(int numParms, char *parms[])
{
     person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

1.19 Enumerated Types

An enumerated type is a collection of symbolic constants that can be used instead of explicit values.

```
enum {FALSE, TRUE} switch;

switch = TRUE;

if (switch == FALSE)
{
     …
}
```

The values used in an enumerated type are integers that begin with 0 and increase in steps of

1. The only exception to the assignment of values to the enumerated constants occurs when the constants are given explicit values in the declaration.

```
enum {VALUE1=10, VALUE2=20} values;
```

Any constant that is not given a value is assigned the value of the previous constant plus 1.

As with the definition of `structs`, there are multiple ways of declaring the same enumerated type. The following definition is the same as the one used in the earlier example, with the exception that the definition below can be used in multiple declaration statements.

```
enum BOOLEAN {FALSE, TRUE};

enum BOOLEAN switch;

switch = TRUE;

if (switch == FALSE)
{
     …
}
```

An enumerated type may be used in a `typedef` statement. This makes the definition of variables somewhat easier because the `enum` does not have to be included in the definition of each variable. The following enumerated type implements the equivalent of the `boolean` data type in Java.

```
typedef enum {FALSE, TRUE} BOOLEAN;

BOOLEAN switch;
```

1.20 Unions

We saw earlier how several variables can be organized as a collection using the `struct` data type. With a `struct`, we can refer to individual values and we can also pass the entire collection of values as one parameter, the name of the `struct`. A related data type is the `union` data type. A `union` is the opposite of a `struct` – any number of variables may be defined in a `union` but only one of the variables can be used at a time and the compiler reserves only enough room for the largest data type in the `union`.

For example, in the following `union`, 2 variables are defined, an `int` and a `double`.

```
union
{
     int intValue;
     double doubleValue;
} union1;
```

On many machines, the C compiler reserves 8 bytes for this `union` (the `sizeof(double)` is 8 bytes and the `sizeof(int)` is 4 bytes). Memory organization will be examined in more detail in Chapter 4.

The variable `union1` is declared to be of type `union` using the same notation as declaring a variable to be a `struct`.

A `union` may also be defined separately from its use in a declaration statement (the same as with a `struct`).

```
union MY_UNION
{
     int intValue;
     double doubleValue;
};

union MY_UNION union1;
```

Similarly, a `union` may be defined using a typedef statement.

```
typedef union
{
     int intValue;
     double doubleValue;
} MY_UNION;

MY_UNION union1;
```

Regardless of which technique is used to declare the variable `union1`, we can then initialize and reference its variables using the same notation as for a `struct`.

```
union1.intValue = 15;
printf("%d\n", union1.intValue);

15
```

Alternatively, we can initialize and refer to `doubleValue`.

```
union1.doubleValue = 123456789.0;
printf("%f\n", union1.doubleValue);

123456789.000000
```

However, **only one value** can be stored and referenced at a time – if after assigning a value to the `double`, as shown above, we then reference the `intValue` in the `union`, the value that is printed is garbage (more or less).

```
printf("%d\n", union1.intValue);

1409286144
```

When programmers use a union, they often include a flag or tag field that identifies the value that is currently stored in the union. What we would like to do is to add a tag field to the beginning of the union. Unfortunately, the instructions shown below do not do what we want. In this union, we can store a value in tag, or in intValue, or in doubleValue but we can not store values in both tag and intValue at the same time.

```
union MY_UNION
{
     enum {INT_TYPE, DOUBLE_TYPE} tag;  // Wrong!!
     int intValue;
     double doubleValue;
};
```

However, if we move the enumerated type out of the union and into a struct that also contains the union, the program will work as desired.

```
struct MY_STRUCT
{
     enum {INT_TYPE, DOUBLE_TYPE} tag;
     union
     {
        int intValue;
        double doubleValue;
     } union1;
};
```

We can now declare a struct variable that contains the tag field plus the union of either an int or a double. Before manipulating either the int or the double, we check the tag field to determine the type of value that is currently stored in the union.

```
struct MY_STRUCT struct1;

struct1.tag = INT_TYPE;
struct1.union1.intValue = 15;

if (struct1.tag == INT_TYPE)
{
     printf("%d\n", struct1.union1.intValue);
}
else if (struct1.tag == DOUBLE_TYPE)
{
     printf("%f\n", struct1.union1.doubleValue);
}
```

Note that in order to reference either of the union variables in the structure, it is necessary to specify both the name of the structure and the name of the union.

As with structs and enumerated types, a union may be defined in a typedef statement. As shown in the example below, once the typedef statement has been defined, variables are declared to be of type defined in the typedef statement.

```
typedef struct
```

```
{
     enum {intType, doubleType} tag;
     union
     {
        int intValue;
        double doubleValue;
     } union1;
} MY_STRUCT;

MY_STRUCT struct1;
```

Unions provide a mechanism for saving space in memory and also disk space if the contents of memory are written to disk. These days most machines have more than enough memory and disk space so the advantage of using unions often doesn't make up for the effort required to use unions properly. However, if you are developing embedded systems where memory and disk memory may be at a premium, saving some memory by using unions is often justified.

1.21 Define Statement

The `#define` statement is used to define a symbolic constant; the value of a symbolic constant is substituted for each occurrence of its name. Note that there is no equals sign in the definition of a symbolic constant.

```
#define MAX_ELEMENTS 100      // Symbolic Constant

int data[MAX_ELEMENTS];
int count;

for (count=0; count<MAX_ELEMENTS; count++)
{
     …
}
```

The use of a symbolic constant instead of literal constant makes a program easier to read and also easier to modify (since you don't have to search through your program looking for all occurrences of the constant and hoping that you don't miss any occurrences).

1.22 Bit Manipulation

Computer memory consists of collections of "bytes" and each byte consists of a fixed number of "bits". It depends on the architecture of a specific computer but in most computers these days, there are 8 bits in each byte. Similarly, each data type in C consists of a machine-dependent number of bytes. For example, on most Intel computers, 4 bytes are allocated for each `int` and 1 byte is allocated for each `char`. However, rather than making an assumption about the number of bytes used to store each data type, use the `sizeof()` function to determine the actual number of bytes used to represent a particular data type. Using good programming practices such as this make a program more machine independent.

When performing low-level processing, it is often necessary to manipulate the individual bits in a data item instead of manipulating all of the bits as a unit. C provides the following bit-manipulation operators:

```
 &    bitwise and
 |    bitwise or (inclusive or)
 ^    bitwise xor (exclusive or)
<<    bitwise left shift
>>    bitwise right shift
 ~    bitwise one's complement
```

These operators work as you would expect, performing the operation on corresponding bits in the operand or operands provided.

When performing bit-wise operations, it is often easier to easier to verify the results if they are displayed in hexadecimal (or octal if that is your number system's base). The following program illustrates a few simple operations and displays the results in hexadecimal (except for the output on the last line which is displayed in octal).

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int value1;
     int value2;
     int value3;

     value1 = 1;
     value2 = -1;

     value3 = value1 & value2;
     printf("Line1: %08X %08X %08X \n", value1, value2, value3);

     value3 = value2 & 0XFFFFFF00;
     printf("Line2: %08X %08X %08X \n", value1, value2, value3);

     value3 = 8 >> 1;
     printf("Line3: %08X %08X \n", 8, value3);

     value3 = value2 >> 2;
     printf("Line4: %08X \n", value3);

     value3 = ((unsigned) value2) >> 4;
     printf("Line5: %08X \n", value3);

     value3 = value2 & 077777777776;
     printf("Line6: %011o %011o %011o \n", value1, value2, value3);

     return 0;
}

Line1: 00000001 FFFFFFFF 00000001
Line2: 00000001 FFFFFFFF FFFFFF00
Line3: 00000008 00000004
Line4: FFFFFFFF
Line5: 0FFFFFFF
```

```
Line6: 00000000001 37777777777 37777777776
```

1.23 Multi-Dimensional Arrays

In earlier sections, we examined the use of one-dimensional arrays. C also supports multi-dimensional arrays that are similar to Java's multi-dimensional arrays. In this section, we will take a quick look at 2-dimensional arrays.

Two-dimensional arrays are declared in the same manner as are two-dimensional arrays in Java. The use of constants to specify the number of rows and the number of columns is a good programming practice. Some C texts do not use the term "multi-dimensional array"; instead, they prefer the term "array of an array". In these notes, we will use the term multi-dimensional array.

The following program illustrates the creation, initialization, and manipulation of a two-dimensional array. The only unusual point to note is that while the size of the first dimension of the array does not have to be defined in the called function (`printArray`), the size of the second dimension **must be explicitly declared**. (We will see how this can be avoided in Chapter 4.)

```
#include <stdio.h>

void printArray(int, int, int[][]);

const int NUM_ROWS = 3;
const int NUM_COLS = 2;

int main(int numParms, char *parms[])
{
     int myArray[NUM_ROWS][NUM_COLS];
     int row, col;

     for (row=0; row<NUM_ROWS; row++)
     {
        for (col=0; col<NUM_COLS; col++)
        {
           myArray[row][col] = row*10 + col;
        }
     }

     printArray(NUM_ROWS, NUM_COLS, myArray);

     return 0;
}


void printArray(int rows, int cols, int myArray[][NUM_COLS])
{
     int row;
     int col;

     for (row=0; row<rows; row++)
     {
```

```
        for (col=0; col<cols; col++)
        {
           printf("%2d ", myArray[row][col]);
        }
        printf("\n");
     }
}
```

As you would expect, multi-dimensional arrays can be declared to be of any valid C data type.

1.24 ArrayList Functions

In this section, we use the techniques that have been developed in this chapter to define a collection of functions that perform much of the same processing as Java's ArrayList class. This example will be continued some of the following chapters.

```
#include <stdio.h>

typedef struct
{
     int size;
     int values[100];
} list;

list newList(list);
list addList(list, int);
int getList(list, int);
int sizeList(list);
list removeList(list, int);
void printList(list);

int main(int numParms, char *parms[])
{
     list myList;

     myList = newList(myList);

     printList(myList);

     myList = addList(myList, 100);
     myList = addList(myList, 200);
     myList = addList(myList, 300);
     myList = addList(myList, 400);

     printList(myList);

     myList = removeList(myList, 3);
     printList(myList);

     myList = removeList(myList, 0);
     printList(myList);

     myList = addList(myList, 500);
     printList(myList);

     myList = removeList(myList, 0);
     myList = removeList(myList, 0);
     myList = removeList(myList, 0);
```

```
     printList(myList);

     printf("\nAll done\n");
     return 0;
}

list newList(list myList)
{
     myList.size = 0;
     return myList;
}

list addList(list myList, int value)
{
     myList.values[myList.size] = value;
     myList.size++;
     return myList;
}

int getList(list myList, int position)
{
     int entry;

     entry = myList.values[position];
     return entry;
}

int sizeList(list myList)
{
     return myList.size;
}

list removeList(list myList, int position)
{
     int count;

     for (count=position; count<(myList.size-1); count++)
     {
        myList.values[count] = myList.values[count+1];
     }
     myList.size--;
     return myList;
}

void printList(list myList)
{
     int count;

     printf("Current list contents:\n");
     if (myList.size > 0)
     {
        for (count=0; count<myList.size; count++)
        {
           printf("Element %d is %d\n", count, getList(myList, count));
        }
        printf("\n");
     }
     else
     {
        printf("The list is empty.\n\n");
     }
}
```

```
Current list contents:
The list is empty.

Current list contents:
Element 0 is 100
Element 1 is 200
Element 2 is 300
Element 3 is 400

Current list contents:
Element 0 is 100
Element 1 is 200
Element 2 is 300

Current list contents:
Element 0 is 200
Element 1 is 300

Current list contents:
Element 0 is 200
Element 1 is 300
Element 2 is 500

Current list contents:
The list is empty.


All done
```

It should be obvious that the arraylist functions defined above are just a beginning. For example, the array used to contain the arraylist elements is an array of int's; also, the array is not automatically resized when the array becomes full.

1.25 Summary

In this chapter, we examined the fundamental C constructs. These constructs are almost identical to the corresponding constructs in Java (since the core of Java was based in large part on the C language).

Unlike Java, C programs are compiled directly into exe files (which consist of machine instructions specific to the machine architecture on which the program was compiled). With the recent increases in speed of the Java Virtual Machine, the difference in performance between executing a C program and a Java program is becoming smaller and smaller.

The C language is a very elegant and compact language. While C does not support objects, we will see in Chapters 3 and 4 that we can create programs that are similar to simple Java objects (not including inheritance).

The C language is not as "safe" a language as Java and the programmer must be careful when writing C programs.

For example, C does not initialize variables – it is the responsibility of the programmer to

initialize all variables (well, almost all variables but it is a good programming practice to initialize all variables).

C does not perform subscript checking when traversing an array.

C does not have a string data type; a “string” in C is an array of characters that is correctly terminated.

The order of evaluation of expressions in C is not always obvious so the programmer should specify the order of evaluation explicitly by including appropriate parentheses.

As you learn more features of the C language, it will be tempting to write very convoluted programs that take advantage of the intricacies of the language – try to avoid this habit. Keep your code as readable as possible and above all, ensure that you follow good programming practices so that your programs do not contain subtle bugs.

1.26 Questions

1. What are the differences between C and Java (as discussed in this chapter)?

2. What is the difference between call by value and call by reference? Which data types are passed by value and which are passed by reference?

3. What happens when a `char[]` that does not contain the termination character is processed using the string manipulation functions?

4. How can you create the equivalent of a very simple Java object in the C language?

2 FILE PROCESSING

2.1 Introduction

In Chapter 1, each program wrote its output to the system console, `stdout`. In this chapter, we examine basic file processing in C. Using C's file processing functions, we can read information from a file and write information to a file. File processing in C is very similar to file processing in Java.

2.2 File Input

The following program reads the contents of the file `in.txt` one character at a time and sends each character to the system console, `stdout`. The definition of the file variable: `FILE *infile;` contains an unusual character, an asterisk ("`*`"). The asterisk has a special meaning in C which we will examine in Chapter 4. For now though, just ignore the asterisk.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     FILE *infile;
     char c;

     infile = fopen("in.txt", "r");

     c = fgetc(infile);
     while (c != EOF)
     {
        printf("%c", c);
        c = fgetc(infile);
     }
     fclose(infile);

     printf("All done\n");
     return 0;
}
```

You should notice that this program is almost identical to the corresponding Java program that reads characters from a file, one character at a time. The value `EOF` that is used in the `while` statement indicates that the end of the file has been reached. This value is defined in `stdio`.

As with Java, the function that reads from a file may be included in the `while` statement as is shown below.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     FILE *infile;
```

```
     char c;

     infile = fopen("in.txt", "r");

     while ((c=fgetc(infile)) != EOF)
     {
        printf("%c", c);
     }
     fclose(infile);

     printf("All done\n");
     return 0;
}
```

However, programmers often forget the inner set of parentheses and instead write the statement as:

```
     while (c=fgetc(infile) != EOF)   // Wrong!!
```

Due to the priority of operators, the statement above is not processed correctly at run-time (although it does compile without any errors or warnings). As a result, in the following examples, we will use the first style for file processing: read from the file before the loop and then read the next piece of information from the file at the end of the loop. It is typically a matter of personal preference which style is used but if you use the second style (with the file input function inside the `while` statement), ensure that you include the appropriate parentheses.

The program below illustrates the point made in Section 10.2 that the C run-time system removes any carriage-return characters that are immediately followed by a newline character (the carriage return and newline combination is used on Windows systems). The program was run with the source program itself being used as the input file. Even though the source file contains both carriage return and newline characters, only the newline characters are presented to the program. This is the expected behaviour for text input files (text is the default when reading a file).

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     FILE *infile;
     char c;

     infile = fopen("main.c", "r");

     c = fgetc(infile);
     while (c != EOF)
     {
        if (c == '\n')
        {
           printf("N\n");
        }
        else if (c == '\r')
```

```
        {
           printf("R");
        }
        else
        {
           printf("%c", c);
        }
        c = fgetc(infile);
     }
     fclose(infile);

     printf("All done\n");

     return 0;
}
```

-- partial program output --

```
#include <stdio.h>N
N
int main(int numParms, char *parms[])N
{N
     FILE *infile;N
     char c;N
     int count1;N
     int count2;N
N
     infile = fopen("main.c", "r");N
N
...
```

If it is necessary to process all characters in a file, the file should be processed as a binary input file by opening the file with the string `"rb"`. As can be seen from the output below, the carriage-return character is now presented to the processing program.

```
#include <stdio.h>RN
RN
int main(int numParms, char *parms[])RN
{RN
     FILE *infile;RN
     char c;RN
     int count1;RN
     int count2;RN
RN
     infile = fopen("checkchars.c", "rb");RN
RN
...
```

The following program is a slight elaboration of the programs above. This program reads characters one at a time, saves the characters in a `char` array, and then when the end-of-line character is encountered, the entire line is printed using the string format code `%s`.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     FILE *infile;
     char line[100];
```

```
     char c;
     int count;

     infile = fopen("in.txt", "r");

     count = 0;
     c = fgetc(infile);
     while (c != EOF)
     {
        line[count] = c;
        if (line[count] == '\n')
        {
           line[count+1] = '\0';
           printf("%s", line);
           count = 0;
        }
        else
        {
           count++;
        }
        c = fgetc(infile);
     }
     if (count > 0)
     {
        line[count] = '\n';
        line[count+1] = '\0';
        printf("%s", line);
     }
     fclose(infile);
     return 0;
}
```

The if statement after the loop handles the case where the last line in the input file contains 1 or more characters but does not end with a newline character. The program above (and the program below) contains a subtle bug. Can you determine what it is?

To avoid having to process a line one character at a time, we could move the processing into its own method, as shown below. The `getLine` function reads the contents of one line or reads until the array is full. The `getLine` function always terminates the array with the sentinel character so you must ensure that the array is large enough to contain the number of characters specified plus one additional character for the sentinel. For example, in the program below, we specify that up to and including 100 characters can be read from a line but we declare the array to be of size 101 to ensure that there is always room for the sentinel character.

```
#include <stdio.h>

char getLine(int, char[], FILE*);

int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;
     FILE *infile;
     char line[MAX_CHARS+1];
     char result;
```

```
     infile = fopen("in.txt", "r");
     result = getLine(MAX_CHARS, line, infile);
     while (result != EOF)
     {
        printf("%s", line);
        result = getLine(MAX_CHARS, line, infile);
     }
     if (line[0]!='\n')  // Check for missing newline at end of file
     {
        printf("%s\n", line);
     }
     fclose(infile);

     return 0;
}

char getLine(int numChars, char line[], FILE *infile)
{
     int count;
     char c;

     c = fgetc(infile);
     for (count = 0; (c != EOF) && (c != '\n'); count++)
     {
        line[count] = c;
        c = fgetc(infile);
     }
     line[count] = '\n';
     count++;
     line[count] = '\0';
     return c;
}
```

In the program above, if the input file is not terminated by a newline character, we would fail to print the contents of the last line in the file if the processing is confined to the loop. It is for this reason that we must check the contents of the line returned when the EOF character is returned.

As we have seen, when the `getc` function detects end of file, it returns a special `EOF` character. One of the features of `getc` is that it can be called again after the end of file is detected – `getc` continues to return the `EOF` character if it is called more than once after end of file. We can take advantage of this feature to improve the program above so that it is no longer the responsibility of the calling function to check for a non-empty line when the end of the file is encountered.

```
#include <stdio.h>

char getLine(int, char[], FILE*);

int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;
     FILE *infile;
     char line[MAX_CHARS+1];
     char result;
```

```
     infile = fopen("in.txt", "r");
     result = getLine(MAX_CHARS, line, infile);
     while (result != EOF)
     {
        printf("%s", line);
        result = getLine(MAX_CHARS, line, infile);
     }
     fclose(infile);

     printf("\n\nAll done\n");
     return 0;
}

char getLine(int numChars, char line[], FILE *infile)
{
     int count;
     char result;
     static char c = ' ';

     // Take advantage of the fact that we can call fgetc after reaching EOF;
     // fgetc keeps returning EOF once the end of the file has been reached.
     c = fgetc(infile);
     for (count = 0; (c != EOF) && (c != '\n'); count++)
     {
        line[count] = c;
        c = fgetc(infile);
     }
     line[count] = '\n';
     count++;
     line[count] = '\0';

     // if we have encountered EOF but there are also characters to return,
     // (i.e. the last line in the file was not terminated by '\n')
     // don't return EOF until the next call to getLine.
     if ((c == EOF) && (count > 1))
     {
        result = ' ';
     }
     else
     {
        result = c;
     }
     return result;
}
```

Finally, C supports line-oriented input using the `fgets` function. (The previous examples were just a warm-up for this function.) The only part that is different is that the variable `result` is defined using the statement `char *result;` Once again, we will examine the purpose of the asterisk in Chapter 4 and the asterisk can be ignored for now.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;
     FILE *infile;
     char line[MAX_CHARS];
     char *result;

     infile = fopen("in.txt", "r");
```

```
     result = fgets(line, MAX_CHARS, infile);
     while (result != NULL)
     {
        printf("%s", line);
        result = fgets(line, MAX_CHARS, infile);
     }
     fclose(infile);

     printf("\n\nAll done\n");
     return 0;
}
```

The `fgets` function reads characters and stores them in the variable `line` (as shown above). Characters are read until a newline character is encountered (the newline character is included with the characters stored in `line`) or until `MAX_CHARS-1` characters have been read. `fgets` terminates the string correctly by adding the `'\0'` character after the last character added to the string. When end of file is encountered or if a file error is detected, the value `NULL` is returned.

2.3 File Output

File output in C is very similar to file output in Java. The example below illustrates how the contents of one file may be copied to another file, one character at a time.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     FILE *infile;
     FILE *outfile;
     char c;

     infile = fopen("in.txt", "r");
     outfile = fopen("out.txt", "w");

     c = fgetc(infile);
     while (c != EOF)
     {
        fputc(c, outfile);
        c = fgetc(infile);
     }
     fclose(infile);
     fclose(outfile);

     printf("All done\n");

     return 0;
}
```

The following program uses string input and string output to copy the contents of one file to another file.

```
#include <stdio.h>
```

```
int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;
     FILE *infile;
     FILE *outfile;
     char line[MAX_CHARS+1];
     char *result;

     infile = fopen("in.txt", "r");
     outfile = fopen("out.txt", "w");
     result = fgets(line, MAX_CHARS, infile);
     while (result != NULL)
     {
        fputs(line, outfile);
        result = fgets(line, MAX_CHARS, infile);
     }
     fclose(infile);
     fclose(outfile);

     printf("All done\n");
     return 0;
}
```

When the program above opens the output file, any information that was in the file is lost and is replaced by the new information written by the program. If it is necessary to append to the end of an existing file instead of replacing the file, the file is opened with the parameter `"a"` instead of `"w"`.

2.4 Formatted File Output

Formatted file output in C works in the same manner as formatted output to the system console. The only difference is that the `fprintf` function is used and this function requires the name of the file as the first parameter. The variables in the parameter list are formatted according to the format codes in the format string. The following program illustrates a simple use of the `fprintf` function.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;
     FILE *infile;
     FILE *outfile;
     char line[MAX_CHARS+1];
     char *result;

     infile = fopen("in.txt", "r");
     outfile = fopen("out.txt", "w");
     result = fgets(line, MAX_CHARS, infile);
     while (result != NULL)
     {
        fprintf(outfile, "%s", line);
        result = fgets(line, MAX_CHARS, infile);
     }
     fclose(infile);
     fclose(outfile);
```

```
     printf("\n\nAll done\n");
     return 0;
}
```

The processing involved in formatted file output can also be used to write formatted output to a string instead of to a file. The `sprintf` function sends its output to a string instead of to a file. The `sprintf` function terminates the output correctly with the `'\0'` character. The function returns the number of characters written to the string, not including the termination character.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     char line[100];
     int value1;
     int value2;

     value1 = 25;
     value2 = -9999;
     sprintf(line, "%d %d", value1, value2);

     printf("\nvalue1 %d value2 %d line %s\n", value1, value2, line);

     printf("All done\n");

     return 0;
}

value1 25 value2 -9999 line 25 -9999
All done
```

2.5 Formatted File Input

Until now, the information read from a file has consisted of character strings. In the same way that we can format output that is sent to stdout or to a file by including a format code, we can specify the format in which input is specified using a format code and one of the scan functions.

The following program reads one integer at a time from an input file and determines the sum of all of the integers in the file. Each line in the file may consist of any number of integer values. The values are separated by one or more white space characters (blank, carriage return, newline, formfeed, tab, vertical tab). There is one aspect of C that has not yet been encountered, the ampersand (“`&`”). The ampersand is placed before the variable in the `fscanf` function. The ampersand operator is related to the asterisk operator and both are discussed in Chapter 4.

```
#include <stdio.h>

int main(int numParms, char *parms[])
```

```
{
     FILE *infile;
     int myInt;
     int sum;
     int result;

     infile = fopen("ints.txt", "r");

     sum = 0;
     result = fscanf(infile, "%d ", &myInt);
     while (result != EOF)
     {
        printf("%d\n", myInt);
        sum += myInt;
        result = fscanf(infile, "%d ", &myInt);
     }

     fclose(infile);
     printf("\nThe sum is %d\n", sum);

     printf("All done\n");
     return 0;
}
```

The `fscanf` function returns an `int`, the number of values **successfully** read. If the number of values read is less than the number of variables specified in the format string, then the first `int` values were read correctly but the next value was not read correctly. When end of file is encountered, the value `EOF` is returned. `fscanf` ignores the newline character (newline is considered to be white space); if a scan ends in the middle of a line, the next scan will resume where the previous scan left off. Similarly, if there are insufficient characters on a line to fill all of the variables in the scan statement, the scan is continued on the next line.

The following program is a slight variation on the program above. The program reads one line at a time into a `char` array (`line`). The program then scans `line` for the first integer value. In this program, reading a line is separated from scanning the line for values. Once an integer value is located, any remaining integer values on the same line are ignored.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     const int MAX_CHARS = 100;

     char line[MAX_CHARS+1];

     FILE *infile;
     int myInt;
     int sum;
     char *result;

     infile = fopen("ints.txt", "r");
     sum = 0;
     result = fgets(line, MAX_CHARS, infile);
     while (result != NULL)
     {
        printf("%s\n", line);
```

```
        sscanf(line, "%d", &myInt);
        sum += myInt;
        result = fgets(line, MAX_CHARS, infile);
    }
    fclose(infile);

    printf("\nThe sum is %d\n", sum);
    printf("All done\n");

    return 0;
}
```

The `sscanf` function is used to scan a string instead of the contents of a file. The function works in the same manner as the `scanf` function. If the end of the string is encountered without having extracted a value for any of the parameters, `sscanf` returns `EOF`. If the first `n` parameters were given values but the remainder of the parameters were not given values, `sscanf` returns the value `n`.

Unfortunately, `sscanf` can not be used to resume a scan where the previous scan left off (`sscanf` always begins its scan at the beginning of the string). One (somewhat ugly) way around this is to write the contents of the string to a temporary file and then use `fscanf` to scan and then resume the scan of the file. There is a more elegant technique that can be used with `sscanf` to perform the equivalent processing (scanning and then resuming the scan where the previous scan terminated). The proof is left to the reader.

The format code in a scan function may also include the width of each value. For example, the format string `"%3d"` specifies that a 3-digit integer value is to be read.

2.6 Console Input

The system console can also be used as an input device. The file input functions described in this chapter can be applied to the system console by using the file name `stdin` or by omitting the character "`f`" at the beginning of the input function name. For example, the following are equivalent:

`scanf(…)`	`fscanf(stdin, …)`
`gets(…)`	`fgets(… stdin)`
`getchar(…)`	`getc(stdin)`

The following program writes a prompt to the system console and then reads an integer from the console using the `scanf` function.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
    int myInt;
```

```
     printf("Please enter an integer: ");
     scanf("%d", &myInt);
     printf("\nThe integer entered was %d\n", myInt);
     return 0;
}
```

2.7 Standard Error

Until now, you have probably written error messages to stdout using the printf statement. For production programs, error information should be written to `stderr` instead of `stdout`. Any output sent to `stderr` is directed to the system console. The following program illustrates writing to `stderr` and also the use of the `exit` command to terminate a program prematurely when an unrecoverable error occurs.

```
#include <stdio.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     int count;
     int sum;

     sum = 0;
     for (count=0; count<100000000; count++)
     {
        sum += count;
        if (sum < 0)
        {
           fprintf(stderr, "Result overflowed into sign bit\n");
           exit(EXIT_FAILURE);
        }
     }

     printf("All done\n");

     return 0;
}
```

```
Result overflowed into sign bit
```

2.8 Summary

In this chapter, we examined the simple file processing. Files may be read and written using character processing or using formatted input and/or output. The descriptions of the various file manipulation functions is very brief to avoid becoming overwhelmed with details. For more detailed information, take a look at a more detailed C language manual or the internet. It is important to note that error checking was not included in the examples in this chapter.

2.9 Questions

1. What does the following portion of code do? (This example is taken from K&R, Second Edition, p. 96; the `getint` function returns one integer value.) This code should convince you to write more readable code (although this code is perfectly readable to an experienced C programmer).

```
int n, array[SIZE], getint(int *);

for (n=0; n<SIZE && getint(&array[n]) != EOF; n++)
```

2. Pick one of the example programs in this chapter and add error checking that ensures that the program will run correctly in all circumstances or will generate an appropriate error message.

3 WORKING WITH MODULES

3.1 Introduction

In Chapters 1 and 2, each program was defined in one source file. For small programs, this is acceptable, but as programs become larger, good programming practices dictate that a program should be subdivided into separate modules, where each module is stored in a separate source file. In this chapter, we examine how a C program can be split into separate modules/files and what additional work has to be done to permit this.

3.2 Modules

In Chapter 1, we created a simple program (shown below) that created a struct and then passed the struct to a print function.

```
#include <stdio.h>

typedef struct
{
     int number;
     int age;
} person;

void printPerson(person);

int main(int numParms, char *parms[])
{
     person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

We will begin our work with modules by breaking this program up into two modules, one module will contain the main function and the other module will contain the printPerson function.

main.c

```
#include <stdio.h>

typedef struct
{
     int number;
     int age;
} person;

int main(int numParms, char *parms[])
{
     person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}
```

module1.c

```
#include <stdio.h>

typedef struct
{
     int number;
     int age;
} person;

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

The two modules can be compiled using the following statement:

```
gcc –ggdb main.c module1.c -o ch3
```

Unfortunately, when the two modules are compiled, multiple error messages are generated because the main module does not know about the module1 module. We need the equivalent of a function prototype that is used to declare a function before it is used in the same module. Fortunately, C supports the definition of "header files" that contain the prototypes of functions that are defined in another module. Such a header file is incorporated into the following `main.c` module using the `include` statement, as shown below.

main.c

```
#include <stdio.h>
#include "module1.h"

typedef struct
{
     int number;
```

```
     int age;
} person;

int main(int numParms, char *parms[])
{
     person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}
```

module1.h

```
void printPerson(person);      // Header file
```

module1.c

```
#include <stdio.h>

typedef struct
{
     int number;
     int age;
} person;

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

Note that the program now consists of 3 separate files: the `main.c` file, the `module1.c` file, and the `module1.h` file. However, the compile statement is the same as before – the header file is not specified in the compile statement, C finds it automatically.

```
gcc –ggdb main.c module1.c -o ch3
```

The program now compiles and executes correctly because the main module knows that the printPerson function is defined in the module1.c module. (Actually, the compiler complains that in main.c, the header file references a structure before the structure has been defined but this problem only results in a warning and could easily be fixed by moving the include "module1.h" statement after the definition of the structure.)

However, even though the program executes correctly, there is one thing that should bother you – the definition of `struct person` is repeated in both the main module and in the module1 module. It is not a good programming practice to duplicate code in this manner because subsequent modifications to the program may not be made to all copies of the structure. So, we would like to move the structure definition to one common location. The header file module1.h is the perfect place for the structure definition. The following revised

program illustrates the use of the header file to store a structure definition that is required by several modules.

main.c

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     person person1;

     person1.number = 25;
     person1.age = 30;

     printPerson(person1);
     return 0;
}
```

module1.h

```
typedef struct
{
     int number;
     int age;
} person;

void printPerson(person);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

Note that now the module1.c module must also include the module1.h header file because the header file contains the definition of the structure.

So a program may be subdivided into any number of modules. Each module must have its own header file that declares the prototypes of each function in the module (plus any structures and/or typedefs that are used in more than one module). When subdividing a program into modules, it is a good idea to think of each module as being similar to a Java class so that related functions are kept in the same module.

3.3 Defining a "Constructor"

In the previous section, we took a program that manipulated a person structure and moved the function that printed a person into its own module. If you examine the main function, you will notice that this function instantiates the values of the two "instance" variables in the structure. This is processing that would be better performed in the module1 module.

The program below illustrates how this processing can be performed. Notice that the structure is declared in the main function and is passed as a parameter to each function that performs processing on the structure.

main.c

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     person person1;

     person1 = newPerson(person1, 25, 30);

     printPerson(person1);
     return 0;
}
```

module1.h

```
typedef struct
{
     int number;
     int age;
} person;

person newPerson(person, int, int);

void printPerson(person);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

person newPerson(person person1, int newNumber, int newAge)
{
     person1.number = newNumber;
     person1.age = newAge;
     return person1;
}

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

As can be seen from this program, we have created a module that is very similar to a Java class: the only significant difference is that the instance variables (the `person struct`) are stored in the main module instead of being encapsulated in the class. We will examine how to encapsulate variables in a C module in Chapter 4.

3.4 Scope of Variables

Until now, we have assumed that the scope of variables that are defined in C programs is the same as the scope of variables defined in a Java program (and, for the most part, this is accurate). However, there are some differences between C and Java that should be examined.

3.4.1 Global Variables

As was mentioned in Chapter 1, variables that are declared before the main function are global to the entire module. In fact, global variables can be accessed in other modules as well if the variable declarations in the other modules are prefixed with the keyword `extern`. The following program illustrates this feature.

main.c

```
#include <stdio.h>
#include "module1.h"

int array1[10];
int numElements = 10;

int main(int numParms, char *parms[])
{
     int count;

     for (count=0; count<numElements; count++)
     {
        array1[count] = count+1;
     }

     myFunction();

     return 0;
}
```

module1.h

```
void myFunction();
```

module1.c

```
#include <stdio.h>
#include "module1.h"

void myFunction()
```

```
{
     extern int array1[];
     extern int numElements;

     int count;

     for (count=0; count<numElements; count++)
     {
        printf("%d ", array1[count]);
     }
     printf("\n");
}
1 2 3 4 5 6 7 8 9 10
```

As can be seen, myFunction which is in a different module is able to access the array and the number of elements in the array by declaring the two variables to be `extern` variables.

3.4.2 Private Global Variables

There are times when defining variables to be global to a module is necessary but you do not want those variables to be accessible in other modules. This is the equivalent of `private` variables in a Java class. In C, you can obtain the same effect if you declare global variables to be `static`. Note that this definition of `static` is not the same as Java's use of `static`.

main.c

```
#include <stdio.h>
#include "module1.h"

static int array1[10];
static int numElements = 10;

int main(int numParms, char *parms[])
{
     int count;

     for (count=0; count<numElements; count++)
     {
        array1[count] = count+1;
     }

     myFunction();

     return 0;
}
```

module1.h

```
void myFunction();
```

module1.c

```
#include <stdio.h>
#include "module1.h"
```

```
void myFunction()
{
     extern int array1[];
     extern int numElements;

     int count;

     for (count=0; count<numElements; count++)
     {
        printf("%d ", array1[count]);
     }
     printf("\n");
}
```

In this program, even though there is an extern variable definition, the compiler chokes on the program and is not able to compile it correctly because the two variables `array1` and `numElements` have been hidden from other modules by the keyword `static`. (The following error message is generated.)

```
module1.o(.text+0x18): In function `myFunction':
C:/Ch3/module1.c:11: undefined reference to `numElements'
module1.o(.text+0x29):C:/Ch3/module1.c:13: undefined reference to `array1'

Tool completed with exit code 2
```

3.4.3 Local Variables

As with Java, local variables are variables that are defined within a C function. Local variables can be accessed anywhere in the function in which they are declared but can not be accessed from outside of the function.

Local variables are re-allocated each time that the function is called so the values of local variables are erased at the end of each function invocation and are not available the next time that the function is called as shown in the program below.

```
#include <stdio.h>

void printElement(void);

int array1[10];
int numElements = 10;

int main(int numParms, char *parms[])
{
     int count;

     for (count=0; count<numElements; count++)
     {
        array1[count] = count+1;
     }

     for (count=0; count<numElements; count++)
     {
```

```
         printElement();
     }

     return 0;
}

void printElement()
{
     int count = 0;

     if (count < numElements)
     {
         printf("%d ", array1[count]);
         count++;
     }
}

1 1 1 1 1 1 1 1 1 1
```

3.4.4 Static Local Variables

Occasionally, it is useful if the value of a local variable is retained from one function invocation to the next invocation of the same function. This can be accomplished by defining the local variables to be `static` variables. Note that this use of `static` is not the same as the use of the keyword `static` that is applied to global variables. (Bad choice of keywords.)

```
#include <stdio.h>

void printElement(void);

int array1[10];
int numElements = 10;

int main(int numParms, char *parms[])
{
     int count;

     for (count=0; count<numElements; count++)
     {
         array1[count] = count+1;
     }

     for (count=0; count<numElements; count++)
     {
         printElement();
     }

     return 0;
}

void printElement()
{
     static int count = 0;  // static local variable

     if (count < numElements)
     {
         printf("%d ", array1[count]);
         count++;
     }
```

```
}

1 2 3 4 5 6 7 8 9 10
```

You should note that the initialization of the static variable count to zero happens only once, when the function is first executed.

3.5 An ArrayList Module

In this section, we use the techniques developed in this chapter to define a module that performs much of the same processing as Java's ArrayList class.

main.c

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     list myList;

     myList = newList(myList);

     printList(myList);

     myList = addList(myList, 100);
     myList = addList(myList, 200);
     myList = addList(myList, 300);
     myList = addList(myList, 400);

     printList(myList);

     myList = removeList(myList, 3);

     printList(myList);

     myList = removeList(myList, 0);

     printList(myList);

     myList = addList(myList, 500);

     printList(myList);

     myList = removeList(myList, 0);
     myList = removeList(myList, 0);
     myList = removeList(myList, 0);

     printList(myList);

     printf("\nAll done\n");
     return 0;
}
```

module1.h

```
typedef struct
{
     int size;
     int values[100];
} list;

list newList(list);

list addList(list, int);

int getList(list, int);

int sizeList(list);

list removeList(list, int);

void printList(list);
```

module1.c

```
#include <stdio.h>

#include "module1.h"

list newList(list myList)
{
     myList.size = 0;
     return myList;
}

list addList(list myList, int value)
{
     myList.values[myList.size] = value;
     myList.size++;
     return myList;
}

int getList(list myList, int position)
{
     int entry;

     entry = myList.values[position];
     return entry;
}

int sizeList(list myList)
{
     return myList.size;
}

list removeList(list myList, int position)
{
     int count;

     for (count=position; count<(myList.size-1); count++)
     {
        myList.values[count] = myList.values[count+1];
     }
     myList.size--;
```

```
     return myList;
}

void printList(list myList)
{
     int count;

     printf("Current list contents:\n");
     if (myList.size > 0)
     {
        for (count=0; count<myList.size; count++)
        {
           printf("Element %d is %d\n", count, getList(myList, count));
        }
        printf("\n");
     }
     else
     {
        printf("The list is empty.\n\n");
     }
}

Current list contents:
The list is empty.

Current list contents:
Element 0 is 100
Element 1 is 200
Element 2 is 300
Element 3 is 400

Current list contents:
Element 0 is 100
Element 1 is 200
Element 2 is 300

Current list contents:
Element 0 is 200
Element 1 is 300

Current list contents:
Element 0 is 200
Element 1 is 300
Element 2 is 500

Current list contents:
The list is empty.
```

Note that there is still a significant limitation in this module – the size of the array can never be increased. We will see in the next chapter how this problem can be resolved.

3.6 Summary

In this chapter we have examined how a large program can be subdivided into a collection of smaller modules. (Although the examples showed only two modules, any number of modules may be defined in a system.) The scope and lifetime of variables in a C program have been expanded by the addition of the keywords `extern` and `static`.

While C does not support object orientation, when programs are modularized, it is possible to observe many of the practices that make object-oriented programming more powerful than procedural programming.

It should also be obvious that the programs developed so far have not included any error-checking. Error-checking is one of the fundamental good programming practices and we will take a closer look at it in subsequent chapters.

3.7 Questions

1. What are the differences in the scope of variables in Java and C?

2. What are the differences in the visibility of variables in Java and C?

3. What is the purpose of a header file?

4. What are the two meanings of the keyword “static”?

5. What does the “lifetime” of a variable mean?

6. There is another type of storage class, the `auto` (automatic) class. What does this storage class do?

4 POINTERS AND MEMORY MANAGEMENT

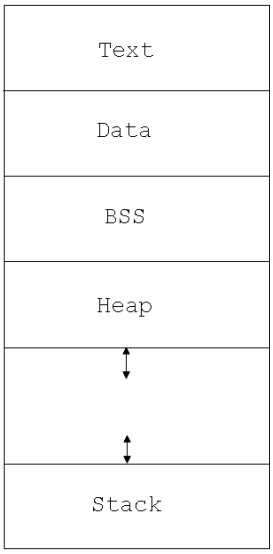
4.1 Introduction

In Chapter 1, we examined the basic C constructs. In Chapter 3, we saw how a program could be decomposed into multiple modules which are similar to Java classes. In this chapter, we examine how data structures can be created dynamically and how pointers (references in Java) can be used to point to dynamically created data structures plus what the `*` and `&` operators mean.

4.2 Memory Organization

We have already seen that a computer's memory consists of a collection of bytes, each of which consists of a collection of bits. The size of a byte (that is, the number of bits in the byte) may vary from machine to machine. The number of bytes used to represent each data type may also vary from machine to machine so it is important that programs be written in as machine-independent a manner as possible. For example, use the `sizeof()` function to determine the number of bytes used to represent a particular data type.

When a program is loaded into memory, different portions of the program are stored in different areas of memory. A generic view of the memory segments used by a C program is shown below. (Understanding the organization of memory in a C program is not really necessary to become a proficient C programmer but it doesn't hurt either.)



The text segment contains the program's executable instructions; the data segment contains initialized global variables; the BSS ("block started by symbol" – an assembly-langugage instruction from the IBM 7094) segment contains unitialized global variables (these variables

are normally initialized to zero but you should never rely on this); the stack contains local variables used within a function; and the heap contains storage explicitly allocated by the programmer. Depending on the operating system, different combinations of access permissions (read, write, execute) may be assigned to each segment.

The `size` command can be used to determine the size of each of the first three segments.

```
C:\Ch4>size ch4.exe
   text    data     bss     dec     hex filename
   2560    1536     112    4208    1070 ch4.exe
C:\Ch4>
```

The amount of memory used by the stack and by the heap grows and shrinks as a program performs its processing so the `size` command can not display the amount of memory that will be used by the stack or heap.

When a function is called, the variables declared in the function are allocated in an area of memory referred to as "local memory" which is allocated on the "stack". When the function terminates, the memory used by the function is released (made available for reuse).

When memory is allocated dynamically (using the `malloc` function, described later in this chapter), that memory is taken from a different area of memory referred to as "dynamic memory" which is allocated on the "heap". Memory allocated in the heap remains dedicated to the program until the programmer explicitly releases the memory (using the `free` function). It is a common programming error to forget to release dynamic memory – this error causes what is referred to as a "memory leak". Memory leaks can be a major problem in programs that execute for a long period of time since the amount of memory that is allocated but no longer required can continue to increase to the point at which there is no more memory available for dynamic allocation.

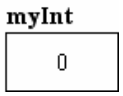
In Java, memory leaks may occur but are quite rare since the Java virtual machine includes a method called the "garbage collector" that is able to identify and free memory allocations that are no longer being used. Thus, with Java, it is not the programmer's responsibility to free memory (such as objects) when the memory is no longer needed. (Although there are some programming practices in Java that make things easier for the garbage collector.)

In C, there are also memory management practices that reduce the chance of memory leaks. We will examine some of them in this chapter and also in a later chapter.

4.3 Memory Allocation for Basic Data Types

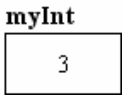
In Java, when a basic data type is declared, memory is reserved for the data item and an initial value of zero is stored in the data item.

```
int myInt;   // Java
```



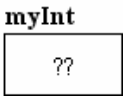
When a value is assigned to the variable, the value is placed in the corresponding memory location.

```
myInt = 3;   // Java
```



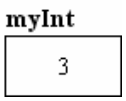
In C, when a basic data type is declared, memory is reserved for the data item and but the data item is **not** initialized. (This statement is not entirely true but it is better to assume that memory is not initialized.)

```
int myInt;
```



Then, when a value is assigned to the variable, the value is placed in the corresponding memory location.

```
myInt = 3;
```



This process is quite simple in both languages: each variable is given an appropriate amount of memory and when a value is assigned to the variable, the value is placed in the memory allocated to that variable. The only difference is that C does not assign an initial value to the memory location.

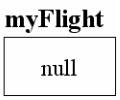
In C, you can determine the amount of memory allocated for a specific data type using the `sizeof()` function. For example, on most computers, `sizeof(int)` returns the value 4. However, `sizeof(int)` may be different on a computer with a different architecture so do not assume that the size of an integer value is always a specific value.

4.4 Java References (Pointers)

Recall that in Java, objects are not stored in a variable – instead, the variable contains a reference (or pointer) to the actual object.

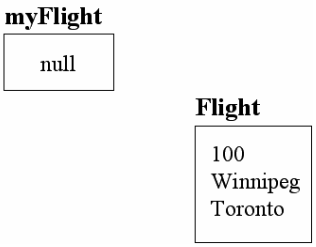
```
public class TestFlight    // Java
{
     public static void main( String[] args )
     {
        Flight myFlight;
        myFlight = new Flight(100, "Winnipeg", "Toronto");
     }
}

public class Flight …   // Java
```

In the example above, myFlight is a variable. When the main method is executed, myFlight is allocated memory and the memory is initialized to null.



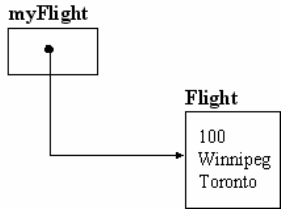
When a new Flight object is created, Java allocates sufficient memory for the object and then creates the object in that memory location.

```
new Flight(100, "Winnipeg", "Toronto")     // Java
```



If the object is assigned to the variable `myFlight`, Java stores a reference/pointer to the object in the variable `myFlight`.

```
myFlight = new Flight(100, "Winnipeg", "Toronto");   // Java
```



Whenever, `myFlight` is accessed, Java automatically follows (dereferences) the pointer to the

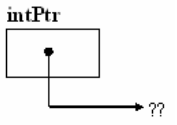
object in such a way that the programmer is never aware of the fact that the variable does not actually contain the object itself.

4.5 C Pointers

In the C language, pointers are an explicit part of the language (instead of being implicit as they are in Java). In C, any data item may be pointed to via a pointer variable.

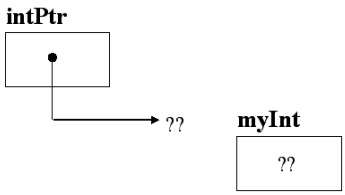
The following statement declares the variable `intPtr` to be a pointer that points to an int memory location. The asterisk in the declaration indicates that the variable is a pointer variable. The declaration of the variable does not cause a value to be assigned to the pointer variable.

```
int *intPtr;
```



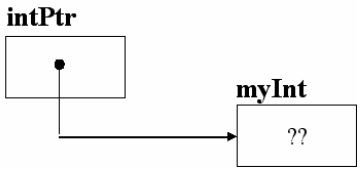
If the int variable `myInt` is subsequently declared, memory is allocated for `myInt`.

```
int myInt;
```



Then, if the pointer variable `intPtr` is assigned the address of the variable `myInt`, `intPtr` points to the same memory location as `myInt`. The **`&`** operator is referred to as the "address of" operator.

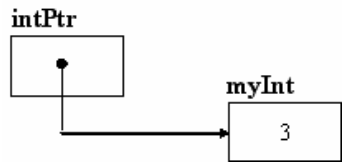
```
intPtr = &myInt;
```



Remember that C does not assign an initial value to variables when they are declared so at this point, `myInt` does not contain a useful value.

Finally, if `myInt` is assigned the value 3, the memory located allocated for `myInt` is given the value 3.

```
myInt = 3;
```



The pointer `intPtr` continues to point to the same memory location and so now points to the value 3. Note that `intPtr` does not contain the value 3, `intPtr` contains a pointer to the memory location allocated for `myInt`, and `myInt` contains the value 3. So `intPtr` points to a memory location that contains the value 3.

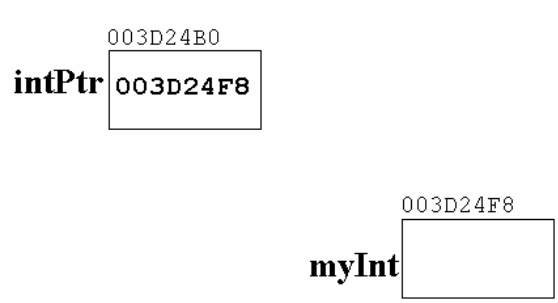
To access the value that `intPtr` points to, we use the asterisk `*` operator to follow (or dereference) the pointer variable to the memory location that it points to. The following statement would print the value 3.

```
printf("%d\n", *intPtr);
```

```
3
```

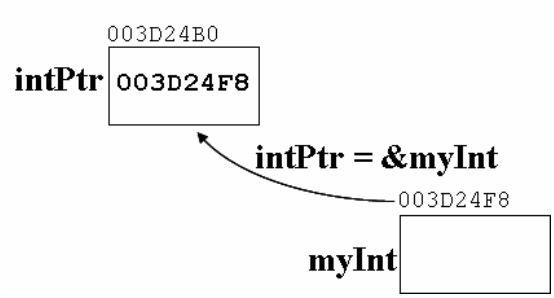
To repeat, each variable is assigned a location in memory. In the following instructions, the variable `intPtr` declared and assigned a memory location (at address `003D24B0`) and the variable `myInt` declared and assigned a memory location (at address `003D24F8`).

```
int myInt;
int *intPtr;
```



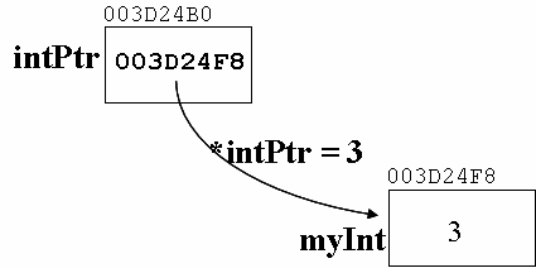
`intPtr` is then assigned the address of the variable `myInt`.

```
intPtr = &myInt;
```



The memory location that `intPtr` points at is then assigned the value 3.

```
*intPtr = 3;
```



Once the pointer variable points to a valid memory location, we can assign a value to that memory location either directly or indirectly. The following two statements each have exactly the same effect – each statement assigns the value 5 to the memory location that used to contain the value 3.

```
myInt = 5;     // assign a value directly to a memory location
*intPtr = 5;   // assign a value indirectly to a memory location
               //   using a pointer variable
```

So a variable that is declared with an `*` is a **pointer variable**. If you recall, the `FILE` variables used in Chapter 2 included an `*` in the declaration – so the `FILE` variable is actually a pointer variable to the `FILE` description.

We now have two new operators that permit us to deal with pointers: the **`&`** operator returns the **address of** the memory location assigned to a variable and the **`*`** operator **dereferences** a pointer variable to the memory location that the variable points to.

The following statements illustrate the use of pointer variables:

```
int intValue1, intValue2;  // declare two int variables
int *intPtr1, *intPtr2;    // declare two (int) pointer variables

intValue1 = intValue2;     // normal assignment; both variables contain the same
                           // value but each variable has its own memory location

intPtr1 = &intValue1;      // assign the address of intValue1 to intPtr1
intPtr2 = &intValue2;      // assign the address of intValue2 to intPtr2

*intPtr1 = *intPtr2;       // assign the value pointed to by intPtr2 to the memory
                           // location pointed to by intPtr1

*intPtr1 = intValue2;      // assign the value in intValue2 to the memory location
                           // pointed to by intPtr1

intPtr1 = intPtr2;         // assign the memory address in intPtr2 to intPtr1 so
                           // that both pointers point to the same memory location
```

The following segment of code will probably execute but since we have not pointed the pointer variable at anything yet, we don't know where the pointer is pointing and it is likely that executing the assignment statement will cause some innocent area of memory to be stomped on by storing the value 5 in it. (If the area of memory that was modified contains some system information, it is likely that the system will start doing weird things at some, possibly later, point.) This is one of the difficulties when programming in C – you are allowed to do stupid things; the Java language and run-time system would not permit such an assignment.

```
int *intPtr;

*intPtr = 5;      // Wrong!
```

Declaring a pointer variable allocates space for the variable itself but does not give the pointer variable a value so the pointer variable does not point at a meaningful memory location.

The following program illustrates some pointer manipulations.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int myInt;
     int *intPtr;

     intPtr = &myInt;
     myInt = 3;

     printf("myInt contains %d \n\n", myInt);

     printf("intPtr points to a memory location that contains the value %d \n\n", *intPtr);

     printf("The memory location assigned to myInt is %p \n\n", &myInt);
```

```
     printf("intPtr points to memory location %p \n\n", intPtr);

     return 0;
}

myInt contains 3

intPtr points to a memory location that contains the value 3

The memory location assigned to myInt is 0022FF8C

intPtr points to memory location 0022FF8C
```

There are two points to mention about this program. First, note that the value of a pointer is printed using the **%p** format code. Secondly, the address of a memory location is printed using the number system used internally by the current computer – base 16 (or hexadecimal) in this case. So the memory location assigned to the variable `myInt` is `0022FF8C` (in base 16) on this particular computer. If the program were run again on the same computer, `myInt` might be assigned a different memory location so you must never assume that a specific memory location will be used for a variable.

4.6 Pointer Types

Each pointer variable has a type, the type that it was declared with. For example, the declaration

```
int *intPtr;
```

declares `intPtr` to be a pointer to an int. A pointer variable may be declared to be of any valid C data type. For example, the following statement declares `charPtr` to be a pointer variable that points to a memory location that contains one char value.

```
char *charPtr;
```

The importance of declaring the type of a pointer will become more obvious when we examine pointer arithmetic later in this chapter.

4.7 Casting a Pointer Variable

As was mentioned in the previous section, each pointer variable has a type. It is valid to cast the value of a pointer variable to a pointer variable of a different type. The following program segment illustrates this technique. We will cast pointer variables later in this chapter.

```
int myInt;
int *intPtr;
char *charPtr;

intPtr = &myInt;
```

```
charPtr = (char*) intPtr;
```

4.8 Passing Values to Functions by Reference

In this section we examine how pointer variables can be passed to functions and what the impact of this technique is.

First, recall that the basic data types (and structures) are passed to a function by value (call by value), so any modifications that are made to a variable in a function are not reflected back in the calling function. For example, in the following program we attempt to modify the value of the parameter in the function. The modification does take effect during the lifetime of the function but once control is returned to the calling program, `myInt` still has its original value.

```
#include <stdio.h>

void modifyValue(int);

int main(int numParms, char *parms[])
{
     int myInt;

     myInt = 3;

     printf("myInt contains %d\n\n", myInt);

     modifyValue(myInt);

     printf("myInt contains %d\n\n", myInt);

     return 0;
}

void modifyValue(int intValue)
{
     intValue++;
     printf("intValue contains %d\n\n", intValue);
}

myInt contains 3

intValue contains 4

myInt contains 3
```

However, if we modify the program so that a pointer to the variable is passed to the function, the modification to the variable is reflected in the calling function (this is essentially call by reference).

```
#include <stdio.h>

void modifyValue(int*);

int main(int numParms, char *parms[])
{
```

```
     int myInt;

     myInt = 3;

     printf("myInt contains %d\n\n", myInt);

     modifyValue(&myInt);

     printf("myInt contains %d\n\n", myInt);

     return 0;
}

void modifyValue(int *intValue)
{
     *intValue = *intValue + 1;
     printf("intValue contains %d\n\n", *intValue);
}

myInt contains 3

intValue contains 4

myInt contains 4
```

If you examine the program, you will notice that instead of passing the parameter `myInt` itself, we now pass a pointer to the parameter.

So we now have a mechanism whereby parameters may be passed by reference instead of by value.

Although this program works correctly, it is generally a good programming practice to return modified values via the return statement instead of using call by reference.

4.9 Arrays and Pointers

Recall from Chapter 1 that arrays are passed by reference, not by value. (Again, saying that arrays are passed by reference is somewhat debatable; but the fact that a pointer to the array is passed to a function is not debatable.)

If a function's prototype is

```
void myFunction(int, int[])
```

and if `myArray` is declared as:

```
int myArray[100];
```

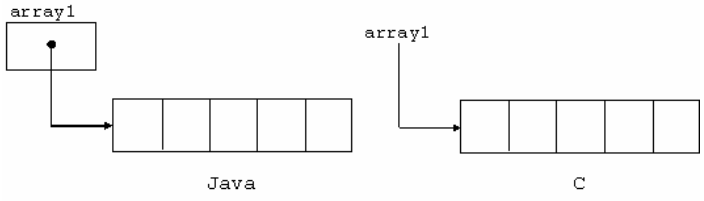
then the following two calls to `myFunction` are equivalent.

```
myFunction(100, myArray)
```

```
myFunction(100, (int*) &myArray[0])
```

Note that the second technique casts the address to the correct type (an `int` pointer variable).

So an array is just the address of the first element in the array. Recall that in Java, the pointer to an array is stored in an array (pointer) variable. Unlike Java, C does not store the address of the first element in a separate variable. C keeps track of the address of the first element of an array and so passing the name of the array is the same as passing a pointer to the first element in the array.



The following program illustrates how an array may be passed by name to a function or it may be passed by address to a function. Both techniques are equivalent.

```
#include <stdio.h>

void sumArray(int, int[]);

int main(int numParms, char *parms[])
{
     int array1[] = {1, 2, 3, 4, 5};

     sumArray(5, array1);
     sumArray(5, (int*) &array1[0]);

     return 0;
}

void sumArray(int numEntries, int entries[])
{
     int count;
     int result;

     result = 0;
     for (count=0; count<numEntries; count++)
     {
        result += entries[count];
     }

     printf("The sum is %d\n", result);
}

The sum is 15
The sum is 15
```

4.10 Pointer Arithmetic

The program in the previous section illustrated how either the name of an array or a pointer to

the first element in an array may be passed to a function that processes the array. The array can then manipulated using conventional array manipulation techniques (`array[subscript]`).

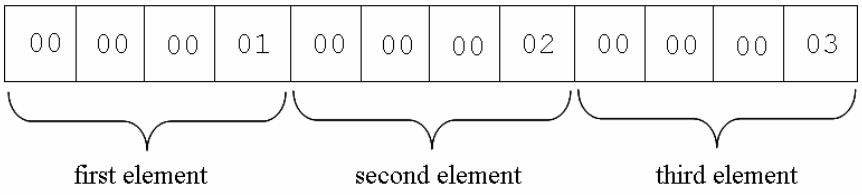
The array can also be manipulated using what is referred to as "pointer arithmetic". When pointer arithmetic is used, we replace the standard `array[subscript]` notation with a pointer variable and an offset. For example,

```
&myArray[0] is equivalent to myArray+0
&myArray[1] is equivalent to myArray+1
&myArray[2] is equivalent to myArray+2
…
```

```
myArray[0] is equivalent to *(myArray+0)
myArray[1] is equivalent to *(myArray+1)
myArray[2] is equivalent to *(myArray+2)
…
```

Adding 1 to a pointer variable looks somewhat strange when each `int` (in this example) element of the array requires more than one byte of memory. (`int` values are typically stored in 4 bytes of memory but this can vary from one computer to another.) The diagram below shows how the following array would be represented internally.

```
int array1[] = {1, 2, 3};
```



Each element of the array is stored in 4 bytes; each byte is represented by 2 hexadecimal digits.

So, if each array element requires 4 bytes of memory, how does adding 1 to a pointer variable cause the pointer variable to point to the next element and not to the next byte in the current element? The answer is that when the value of a pointer variable is modified by adding 1 to it or subtracting 1 from it, C modifies the pointer variable by the `sizeof` the type of the pointer variable. So, for an `int*` pointer variable, C adds/subtracts the value of `sizeof(int)` to/from the pointer variable.

As a result of the way that pointer arithmetic works, the following statement has the effect of modifying the pointer variable to point to the next element in an array.

```
ptr++
```

You must be careful when using the increment (or decrement) operator with pointer variables and dereferencing. For example, the following statement is fairly obvious – add 1 to the memory location pointed to by the pointer variable

```
(*ptr)++
```

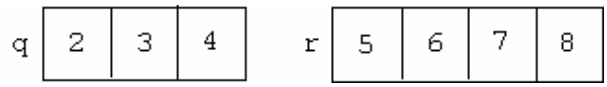
Similarly, the following statement increments the pointer variable by 1 element and then dereferences the new contents of the pointer variable.

```
*(++ptr)
```

However, other variations such as the ones below are not as obvious and should be used with caution. (Try figuring out what they do.)

```
*(ptr++)
*(++ptr)
*ptr++
```

Two pointer variables may be compared (pointer comparison) using the arithmetic comparison operators (<,>,==, etc.) if both pointer variables point at the same data item (or array) in memory. For example, if the following 2 arrays have been declared, two pointer variables that both point to q or that both point to r may be compared. However, comparing one pointer variable that points to q with another pointer variable that points to r does not make any sense since we do not know what the relationship between the memory allocated for q and the memory allocated for r is.



In the following program segments, the pointer variable comparisons all make sense.

```
int q[3], r[4];
int *qPtr, *rPtr;

qPtr = &q[0];
if (qPtr < q+1) …

rPtr = r+1;
if (rPtr == &r[1]) …
```

However, the following comparison statement does not make sense because the programmer has no control over where the arrays q and r are placed in memory.

```
qPtr = &q[0];
rPtr = &r[0];
if (qPtr == rPtr) …   // Wrong
```

The following program rewrites the program from the previous section so that pointer

arithmetic is used instead of the array manipulations used in the `sumArray` function. Note that the `for` statement uses pointer variable comparisons to determine whether or not the end of the array has been reached.

```
#include <stdio.h>

void sumArray(int, int*);

int main(int numParms, char *parms[])
{
     int array1[] = {1, 2, 3, 4, 5};

     sumArray2(5, array1);
     sumArray2(5, (int*) &array1[0]);

     return 0;
}

void sumArray(int numEntries, int *entries)
{
     int *ptr;
     int result;

     result = 0;
     for (ptr=entries; ptr<entries+numEntries; ptr++)
     {
        result += *ptr;
     }

     printf("The sum is %d\n", result);
}


The sum is 15
The sum is 15
```

As a general rule, if you have a choice between using the array notation to access an element of an array or using the pointer notation, it is generally better to use the array notation.

4.11 Casting and Pointer Arithmetic

We saw earlier how a pointer variable could be cast to a different type. The following example illustrates how pointer casting may produce incorrect or unexpected results. The first two loops are defined correctly but the third loop uses an `int` pointer to traverse a `char` array. As a result, only every fourth character in the array is printed since the `int` pointer is incremented by the sizeof an `int` (4) instead of the sizeof a `char` (1).

```
#include <stdio.h>
#include <string.h>

int main(int numParms, char *parms[])
{
     int intArray[] = {1, 2, 3};
     char charArray[] = "This is a string";
```

```
     int *intPtr;
     char *charPtr;

     for (intPtr=&intArray[0]; intPtr<&intArray[0]+3; intPtr++)  // Correct
     {
        printf("%d ", *intPtr);
     }
     printf("\n");

     for (charPtr=&charArray[0]; *charPtr!='\0'; charPtr++)     // Correct
     {
        printf("%c", *charPtr);
     }
     printf("\n");

     intPtr = (int*) &charArray[0];
     for (; intPtr<(int*)(&charArray[0]+strlen(charArray)); intPtr++) // Wrong
     {
        printf("%c", *intPtr);
     }
     printf("\n");

     return 0;
}

1 2 3
This is a string
T ar
```

4.12 Processing Multi-Dimensional Arrays

It was mentioned in Chapter 1 that when multi-dimensional arrays are passed as arguments, the function must explicitly declare the value of the second dimension (and all subsequent dimensions as well). The program below shows how the two-dimensional array was passed to the function `printArray` in Chapter 1 and then illustrates how the array could be passed as a pointer variable and manipulated appropriately in the function `printArray2`. Both methods generate identical output.

```
#include <stdio.h>

void printArray(int, int, int[][]);
void printArray2(int, int, int*);

const int NUM_ROWS = 3;
const int NUM_COLS = 2;

int main(int numParms, char *parms[])
{
     int myArray[NUM_ROWS][NUM_COLS];
     int row, col;
     int* ptr;

     for (row=0; row<NUM_ROWS; row++)
     {
        for (col=0; col<NUM_COLS; col++)
        {
           myArray[row][col] = row*10 + col;
        }
```

```
     }

     printArray(NUM_ROWS, NUM_COLS, myArray);

     printf("\n");
     ptr = (int*) &myArray[0];
     printArray2(NUM_ROWS, NUM_COLS, ptr);

     return 0;
}

void printArray(int rows, int cols, int myArray[][NUM_COLS])
{
     int row;
     int col;

     for (row=0; row<rows; row++)
     {
        for (col=0; col<cols; col++)
        {
           printf("%2d ", myArray[row][col]);
        }
        printf("\n");
     }
}

void printArray2(int rows, int cols, int* ptr)
{
     int row;
     int col;
     int* current;

     current = ptr;
     for (row=0; row<rows; row++)
     {
        for (col=0; col<cols; col++)
        {
           printf("%2d ", *current);
           current++;
        }
        printf("\n");
     }
}
```

```
 0  1
10 11
20 21

 0  1
10 11
20 21
```

4.13 Dynamic Memory Allocation

In the examples used so far in this chapter, the pointers have always pointed at variables that were declared in the corresponding functions. For example, in the program segment below, when array1 is declared in the main function, the memory required to store the 5 values is allocated when the main function begins execution.

```
int main(int numParms, char *parms[])
{
     int array1[] = {1, 2, 3, 4, 5};
     . . .
}
```

Frequently, it is not known how big an array is or how many elements will be used in a data structure (such as a linked list). C permits the dynamic allocation of memory using the `malloc` function (which is defined in the `stdlib` library). The function `malloc` obtains a specific amount of memory from the operating system and then returns a pointer to the memory that is allocated. If there is not sufficient memory available, `malloc` returns a value of `NULL`.

```
int *ptr;
ptr = malloc(bytes);
```

The number of bytes requested must be an integer value. Since the size of certain data types may vary from one machine to another (for example, an `int` may be stored in either 2 bytes or 4 bytes), you should never specify the number of bytes required as an absolute value. Instead, specify the number of elements of a specific type. So, to allocate the space for an int array of size 5, you would use the following statement:

```
int *ptr;
ptr = malloc(5*sizeof(int));
```

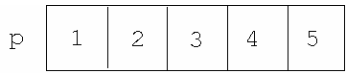
By using the sizeof function instead of an absolute value, your program is portable to other machines instead of being limited to the type of machine on which it was developed.

Once you have finished using some dynamically allocated memory, you should return that memory to the operating system so that the memory can be used by another part of the program. The `free` function is used to return memory that was dynamically allocated.

```
int *ptr;
ptr = malloc(5*sizeof(int));
. . .
free(ptr);
```

Memory that is allocated using `malloc` remains allocated until it is explicitly freed. (This is unlike memory that is allocated for local variables on the stack – this memory is automatically freed when the function that declared the local variables terminates.) Thus, a pointer variable that points to dynamically allocated memory may be passed to other functions and may also be returned to the calling function.

The following program illustrates the use of the `malloc` statement to allocate an `int` array of 5 elements, pointer arithmetic to assign values to the array elements, and finally the `free` statement to return the memory to the operating system.




```
#include <stdio.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     int count;
     int *p;
     int *s;

     p = (int*) malloc(5*sizeof(int));

     for (count=0, s=p; s<p+5; s++, count++)
     {
        *s = count+1;
     }

     ...

     free(p);
     ...

     return 0;
}
```

4.14 NULL Pointers

When the memory pointed to by a pointer variable has been freed, it is a good programming practice to assign the NULL value to that pointer variable. NULL is used in C in a manner that is similar to the way that Java assigns a NULL value to an object reference before the object has been created.

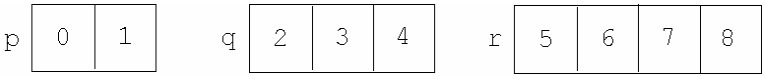
```
#include <stdio.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     int count;
     int *p;
     int *s;

     p = (int*) malloc(5*sizeof(int));

     for (count=0, s=p; s<p+5; s++, count++)
     {
        *s = count+1;
     }

     . . .

     free(p);
     p = NULL;
     ...

     return 0;
}
```

The reason for assigning the value `NULL` to a pointer variable is that if the programmer mistakenly tries to refer to the memory previously pointed to by the pointer variable after the memory has been freed, a `NULL` value will normally cause an error immediately whereas leaving the pointer variable pointing at the memory that had been dynamically allocated will not necessarily cause an error or it may cause an error at a later time.

4.15 Memory Management

When memory is allocated using the `malloc` command, the allocated memory is contiguous. For example, if memory for 5 `int` values is allocated, the values are stored one after the other in one chunk of memory.

However, if multiple calls are made to `malloc`, you must not assume that there is any relationship between the chunks of memory that are allocated. For example, the following program allocates a block of two ints, a block of 3 ints, and a block of 4 ints.

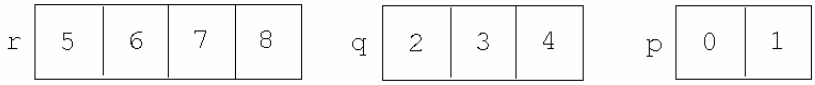


The program then uses pointer arithmetic to assign the values shown above to the 9 memory locations. Note that 3 separate loops are required to perform this initialization.

```
#include <stdio.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     int count;
     int *p;
     int *q;
     int *r;
     int *s;

     p = (int*) malloc(2*sizeof(int));
     q = (int*) malloc(3*sizeof(int));
     r = (int*) malloc(4*sizeof(int));

     for (count=0, s=p; s<p+2; s++, count++)
     {
        *s = count;
     }

     for (s=q; s<q+3; s++, count++)
     {
        *s = count;
     }

     for (s=r; s<r+4; s++, count++)
```

```
    {
       *s = count;
    }

    free(p);
    free(q);
    free(r);
    p = NULL;
    q = NULL;
    r = NULL;
    ...

    return 0;
}
```

The memory locations within each chunk are contiguous but there is no relationship among the 3 chunks. In the diagram shown above, the 3 chunks appear to have been allocated one after the other (with some space in between each pair of chunks), but that organization is not necessarily correct. It is just as likely that the memory will be allocated as shown below.



The only thing that you can be certain of is that the 3 memory allocations will not be contiguous since some system information is reserved before each chunk of allocated memory.

If you require that some memory locations be contiguous, you must allocate the memory in one `malloc` statement. Otherwise if separate `malloc` statements are used, the requested chunks of memory will be allocated but you can not predict in advance where those memory locations will be.

4.16 The Stack and the Heap

As was mentioned earlier, when a function is called, the memory required for the local variables used in the function is automatically allocated on the stack by the C run-time system. The programmer can manipulate those variables within the function in any valid manner. When the function terminates, the memory allocated for those variables is automatically freed by the C run-time system and the programmer must not refer to those memory locations in any of the higher-level (calling) functions. Alternatively, any memory allocated explicitly by the programmer using the `malloc` statement is allocated on the heap. This memory remains available until the programmer explicity frees the memory.

The following program illustrates memory allocation on the stack and on the heap.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int sum1(int, int[]);
int* sum2(int, int[]);
int* sum3(int, int[]);

int main(int numParms, char *parms[])
{
     int myInts[] = {1, 2, 3, 4, 5};
     int result1;
     int* result2;
     int* result3;

     result1 = sum1(5, myInts);
     result2 = sum2(5, myInts);
     result3 = sum3(5, myInts);
     printf("Sum1: %d\n", result1);
     printf("Sum2: %d\n", *result2);
     printf("Sum3: %d\n", *result3);

     free(result3);

     return 0;
}

int sum1(int size, int ints[])
{
     int count;
     int result;

     result = 0;
     for (count=0; count<size; count++)
     {
        result += ints[count];
     }
     return result;
}


int* sum2(int size, int ints[])
{
     int count;
     int result;
     int* ptr;

     result = 0;
     ptr = &result;
     for (count=0; count<size; count++)
     {
        *ptr += ints[count];
     }
     return ptr; // Wrong!!
}


int* sum3(int size, int ints[])
{
     int count;
     int* result;

     result = (int*) malloc(sizeof(int));
     *result = 0;
     for (count=0; count<size; count++)
     {
        *result += ints[count];
```

```
     }
     return result;
}

Sum1: 15
Sum2: 2009291924
Sum3: 15
```

As can be seen from the output, the first function correctly returns its result as an `int` variable. The second function returns a pointer to the variable `result` that is local to the `sum2` function. The memory allocated for this local variable is released at the end of the processing of `sum2` but the pointer still points to the memory location that had been used by the local variable. This is a typical mistake when manipulating pointer variables. In the third function, memory is allocated on the heap for the `result` variable. A pointer to this memory can be returned to the calling program but the calling program must `free` the memory when it has completed its processing (or a memory leak will have been created).

4.17 Void Pointers

The pointer returned by the `malloc` function is of type `void`. A `void` pointer is a pointer that does not have a specific type (such as `int`, `char`, etc.) associated with it. A `void` pointer may be stored in a `void` pointer variable, as is shown below.

```
void *ptr;

ptr = malloc(10*sizeof(int));
```

Although you may store the pointer in a `void` pointer variable, you can not use the variable directly since `void` pointers can not be dereferenced without a corresponding cast. The following statements illustrate how the pointer must be cast before it can be dereferenced.

```
void *ptr;

ptr = malloc(10*sizeof(int));

*((int *) ptr) = 5;
```

C also does not maintain information about the type (if there is a type specified) of each `malloc` statement so it is not possible to determine the type of information that the `void` pointer points to.

As a result, instead of storing the result of a `malloc` statement in a `void` pointer, it is normally preferable to store the result in a pointer of the type for which the memory will be used. The following statements illustrate the intent to use the memory that is allocated to store int values. Although an explicit cast to `int*` is performed, this cast is not required.

```
int *ptr;
```

```
ptr = (int*) malloc(10*sizeof(int));

*ptr = 5;
```

A `void` pointer can be thought of as being similar to Java's Object class – any pointer may be stored in a `void` pointer; however, unlike Java, the `void` pointer can not be used without a corresponding downcast to a specific data type.

4.18 Resizing a Memory Allocation

If it is necessary to resize an array (or any other dynamically allocated chunk of memory) in C, there are several functions that assist with this process.

The first technique involves allocating a larger chunk of memory using `malloc`, then using the C function `memcopy` to copy the contents of the original chunk of memory into the newly allocated chunk of memory, and finally using `free` to release the originally allocated chunk of memory. Be aware that the new chunk of memory will not normally have the same memory address as the original chunk so you must ensure that any pointers are updated appropriately.

However, since resizing a chunk of memory is a common practice in C, another function, `realloc`, is available to perform all tasks (`malloc`, `memcopy`, and `free`) together.

4.19 Pointers to Pointers

A pointer contains the address of a data item. It is perfectly valid in C for a pointer to point to another pointer. A variable that points to a pointer variables is declared as *`type`*`**`. The following program illustrates the use of a pointer to another pointer.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int value = 25;
     int* ptr1;
     int** ptr2;


     ptr1 = &value;
     ptr2 = &ptr1;
     printf("%d %d %d \n", value, *ptr1, **ptr2);

     return 0;
}

25 25 25
```

4.20 Processing Run-Time (Command-Line) Parameters

The beginning of each Java main method contains the parameter:

```
public static void main(String[] parms) …   // Java
```

The beginning of each C main function contains the parameters

```
int main(int numParms, char *parms[])
```

This parameter list is frequently written as:

```
int main(int argc, char *argv[])
```

or

```
int main(int argc, char **argv)
```

The names of the variables used to define the run-time or command-line parameters is more or less irrelevant – just keep the variables meaningful.

It should be obvious that what is being passed is a pointer to an array of pointer variables, each of which points to a character string that represents one of the parameters the program was called with. For example, in addition to invoking a program with only its name,

```
myprogram
```

a program may be provided with parameters that the program is expected to process:

```
myprogram parm1 parm2 parm3
```

These parameters are supplied to the main method as parameters (an array of pointers to individual arrays of char's). The following program extracts and prints the parameters but does not perform any parsing or processing of the parameters.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int count;

     for (count=0; count<numParms; count++)
     {
        printf("%s\n", parms[count]);
     }
     printf("All done.\n");

     return 0;
}
```

```
myprogram
parm1
parm2
parm3
```

Note that the first parameter is the name of the program that was executed.

4.21 Pointers to Structures

In addition to being able to allocate memory for the basic data types, you may also use `malloc` to allocate memory for a struct. For example, the following code segment illustrates the dynamic allocation of space for a structure.

```
struct person
{
     int number;
     int age;
};

struct person *myPerson;

myPerson = (struct person*) malloc(sizeof(struct person));
```

Now the variable `myPerson` is a pointer variable to an instance of the structure `person`. Unfortunately, the notation used earlier to refer to a variable in a structure (`a.b`) can not be used with a pointer variable. However, if we dereference the pointer variable, we can then refer to the structure variable as shown below:

```
(*myPerson).number
```

The expression above is a bit clumsy and so C provides an alternative syntax that is slightly nicer:

```
myPerson->number
```

Either of the two expressions is acceptable and both expressions may be used on either side of an assignment operator.

4.22 Encapsulation

In Chapter 3, we created the following program that began a transition towards a more object-oriented style of programming in C. However, if you examine the program closely, you should notice that `person1` is assigned memory in the main function, not in the `newPerson` function. The `newPerson` function simply assigns the parameters to the existing structure and returns the modified structure. Do you remember why a `struct` that is modified must be returned (using a `return` statement) by the function that modifies it?

main.c

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     person person1;

     person1 = newPerson(person1, 25, 30);

     printPerson(person1);
     return 0;
}
```

module1.h

```
typedef struct
{
     int number;
     int age;
} person;

person newPerson(person, int, int);

void printPerson(person);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

person newPerson(person person1, int newNumber, int newAge)
{
     person1.number = newNumber;
     person1.age = newAge;
     return person1;
}

void printPerson(person person1)
{
     printf("Person1: %d %d \n", person1.number, person1.age);
}
```

Now, however, we have the ability to allocate memory for an object dynamically in the `newPerson` function.

main.c

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     Person person1;
     Person person2;
```

```
     person1 = newPerson(25, 30);
     printPerson(person1);

     person2 = newPerson(55, 55);
     printPerson(person2);

     deletePerson(person1);
     deletePerson(person2);

     printf("All done.\n");
     return 0;
}
```

module1.h

```
typedef struct person *Person;

Person newPerson(int, int);

void printPerson(Person);

void deletePerson(Person myPerson);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

struct person
{
     int number;
     int age;
};

Person newPerson(int num, int newAge)
{
     Person myPerson = (Person) malloc(sizeof(struct person));
     myPerson->number = num;
     myPerson->age = newAge;
     return myPerson;
}

void printPerson(Person myPerson)
{
     printf("Number: %d, age: %d\n", myPerson->number, myPerson->age);
}

void deletePerson(Person myPerson)
{
     free(myPerson);
}
```

The statement

```
typedef struct person *Person;
```

in the header file is interesting because it defines a pointer to the `struct` instead of defining the contents of the `struct`. (The contents of the `struct person` are defined only in the

module that requires access to them.) Thus, the statement

```
Person person1;
```

declares an "object" `person1` that is a pointer variable to the `struct` that contains the instance variables for `person1`. (There isn't an explicit name for this construct in C but referring to it as an "**inner structure**" would be appropriate.) As a result, the instance variables are hidden from the main function (encapsulated).

4.23 ArrayLists

Using the material developed in the preceding section, the following program is a slightly nicer version of the ArrayList program that was developed in Chapter 3. In this version, instead of making the structure that defines the ArrayList object a global structure, instead, we simply define a pointer to the structure.

Note that because variables of type `ArrayList` now contain pointers, the pointer variable is passed to the processing functions and the processing functions can modify the structure itself (call by reference). Also, as a result of using a pointer variable, the calling program can no longer access the contents of the `arrayList` structure since this structure is hidden in the associated module. (This is not entirely true but it is close enough for now.)

The `#define` statement is used to define a symbolic constant; the value of a symbolic constant is substituted for each occurrence of its name. Note that there is no equals sign in the definition of a symbolic constant.

`main.c`

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     ArrayList myList;

     myList = newList();

     addList(myList, 1);
     addList(myList, 2);
     addList(myList, 3);

     printList(myList);

     removeList(myList, 1);
     printList(myList);

     addList(myList, 5);
     printList(myList);
```

```
     removeList(myList, 0);
     removeList(myList, 0);
     removeList(myList, 0);
     printList(myList);

     printf("\nAll done\n");
     return 0;
}
```

module1.h

```
typedef struct arrayList *ArrayList;

ArrayList newList(void);

void addList(ArrayList, int);

int getList(ArrayList, int);

int sizeList(ArrayList);

void removeList(ArrayList, int);

void printList(ArrayList);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

#define NumEntries 5

struct arrayList
{
     int size;
     int data[NumEntries];
};

ArrayList newList()
{
     ArrayList list;
     int count;

     list = (ArrayList) malloc(sizeof(struct arrayList));
     for (count=0; count<NumEntries; count++)
     {
         list -> data[count] = 0;
     }
     list -> size = 0;
}

void addList(ArrayList list, int value)
{
     list -> data[list -> size] = value;
     list -> size++;
}

int getList(ArrayList list, int position)
{
     int entry;
```

```
     entry = list -> data[position];
     return entry;
}

int sizeList(ArrayList list)
{
     return list -> size;
}

void removeList(ArrayList list, int position)
{
     int count;

     for (count=position; count<(list -> size-1); count++)
     {
        list -> data[count] = list -> data[count+1];
     }
     list -> size--;
     list -> data[list -> size] = 0;
}

void printList(ArrayList list)
{
     int count;

     for (count=0; count<list -> size; count++)
     {
        printf("Element %d is %d\n", count, getList(list, count));
     }
     printf("\n");
}

Element 0 is 1
Element 1 is 2
Element 2 is 3

Element 0 is 1
Element 1 is 3

Element 0 is 1
Element 1 is 3
Element 2 is 5


All done
```

The program above encapsulates the data in such a way that the calling functions can not (easily) access the `arrayList` structure. But the program still has the disadvantage that the number of elements that can be stored in the `ArrayList` is fixed and can not be modified. The program below begins an improvement that will eventually permit the `ArrayList` to be expanded in size.

`module1.c`

```
#include <stdio.h>
#include "module1.h"

#define NumEntries 5
```

```
struct arrayList
{
     int maxSize;
     int size;
     int *data;
};

ArrayList newList()
{
     ArrayList list;
     int count;

     list = (ArrayList) malloc(sizeof(struct arrayList));
     list -> data = (int*) malloc(NumEntries*sizeof(int));
     for (count=0; count<NumEntries; count++)
     {
        list -> data[count] = 0;
     }
     list -> size = 0;
     list -> maxSize = NumEntries;
}

void addList(ArrayList list, int value)
{
     list -> data[list -> size] = value;
     list -> size++;
}

int getList(ArrayList list, int position)
{
     int entry;

     entry = list -> data[position];
     return entry;
}

int sizeList(ArrayList list)
{
     return list -> size;
}

void removeList(ArrayList list, int position)
{
     int count;

     for (count=position; count<(list -> size-1); count++)
     {
        list -> data[count] = list -> data[count+1];
     }
     list -> size--;
     list -> data[list -> size] = 0;
}

void printList(ArrayList list)
{
     int count;

     printf("\nContents of list:\n");
     for (count=0; count<list -> size; count++)
     {
        printf("Element %d is %d\n", count, getList(list, count));
     }
```

```
     printf("\n");
}
```

Now that the array used to store the elements in the arrayList is pointed to by a variable in the structure instead of being hard-coded in the structure, it becomes trivial to resize the array as necessary. The following method performs this processing.

```
void resizeList(ArrayList list, int newSize)
{
     int count;

     list -> data = (int*) realloc(list->data, newSize*sizeof(int));
     list -> maxSize = newSize;
     for (count=list->size; count<list->maxSize; count++)
     {
        list -> data[count] = 0;
     }
}
```

In addition, the following code must be added to the beginning of the `addList` function. (The value 10 used to increment the size of the array was chosen arbitrarily.)

```
     if (list->size >= list->maxSize)
     {
        resizeList(list, list->maxSize+10);
     }
```

Although the array list module is a good start, there are some memory management problems that must be fixed before the module could be used.

4.24 Linked Lists

Now that we can allocate memory dynamically, we can write a linked-list program that allocates memory as necessary.

`main.c`

```
#include <stdio.h>
#include "module1.h"

int main(int numParms, char *parms[])
{
     LinkedList myList;

     myList = newList();

     myList = addList(myList, 3);
     printList(myList);

     myList = addList(myList, 2);
     printList(myList);
```

```
     myList = addList(myList, 1);
     printList(myList);

     myList = removeList(myList, 1);
     printList(myList);

     myList = addList(myList, 5);
     printList(myList);

     myList = removeList(myList, 1);
     myList = removeList(myList, 1);
     myList = removeList(myList, 0);
     printList(myList);

     printf("\nAll done\n");
     return 0;
}
```

module1.h

```
typedef struct linkedList *LinkedList;

LinkedList newList(void);

LinkedList addList(LinkedList, int);

int getList(LinkedList, int);

int sizeList(LinkedList);

LinkedList removeList(LinkedList, int);

void printList(LinkedList);
```

module1.c

```
#include <stdio.h>
#include "module1.h"

struct Node
{
     int data;
     struct Node *next;
};

struct linkedList
{
     int size;
     struct Node *first;
};

LinkedList newList()
{
     LinkedList list;
     int count;

     list = (LinkedList) malloc(sizeof(struct linkedList));
     list -> first = NULL;
     list -> size = 0;
}
```

```
LinkedList addList(LinkedList list, int value)
{
     struct Node *newNode;

     newNode = (struct Node *) malloc(sizeof(struct Node));
     newNode -> data = value;
     newNode -> next = list -> first;
     list -> first = newNode;
     list -> size++;
     return list;
}

int getList(LinkedList list, int position)
{
     int entry;
     int count;
     struct Node *current;

     current = list -> first;
     for (count=0; count<position; count++)
     {
        current = current -> next;
     }
     entry = current -> data;
     return entry;
}

int sizeList(LinkedList list)
{
     return list -> size;
}

LinkedList removeList(LinkedList list, int position)
{
     int entry;
     int count;
     struct Node *current;
     struct Node *previous;

     previous = NULL;
     current = list -> first;
     for (count=0; count<position; count++)
     {
        previous = current;
        current = current -> next;
     }

     if (previous == NULL)
     {  // delete the first element in the list
        list -> first = current -> next;
     }
     else
     {  // delete an element that has another element before it
        previous -> next = current -> next;
     }

     list -> size--;
     return list;
}

void printList(LinkedList list)
{
```

```
    int count;

    if (list->size > 0)
    {
       printf("\nContents of list:\n");
       for (count=0; count<list->size; count++)
       {
          printf("Element %d is %d\n", count, getList(list, count));
       }
       printf("\n");
    }
    else
    {
       printf("\nThe list is empty.\n");
    }
}

Contents of list:
Element 0 is 3

Contents of list:
Element 0 is 2
Element 1 is 3

Contents of list:
Element 0 is 1
Element 1 is 2
Element 2 is 3

Contents of list:
Element 0 is 1
Element 1 is 3

Contents of list:
Element 0 is 5
Element 1 is 1
Element 2 is 3

The list is empty.
```

Note that the linked-list module is not yet complete. New elements are added to the beginning of the list (new elements are normally added to the end of the list). Plus, there are several memory management problems that need to be fixed.

4.25 Working with Strings

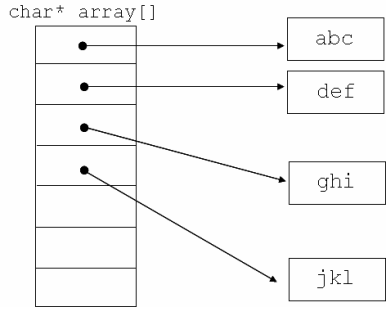
Working with strings and dynamic memory allocation requires some extra care due to the way that strings are constructed.

The following examples illustrate 4 different ways of storing a collection of strings. In the first example, the strings are stored in one large two-dimensional array. Since the array must be of a predefined size, there is wasted space at the end of each string (each row in the array) and at the end of the array (unused rows). Note that this example does not use dynamic memory allocation. The string termination character is not shown in the diagrams.


```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     const int LINE_SIZE = 1000;
     const int STRING_SIZE = 100;
     const int NUM_STRINGS = 100;

     FILE *infile;
     char strings[NUM_STRINGS][STRING_SIZE];
     char line[LINE_SIZE];
     char* result;
     int size;
     int count;

     size = 0;
     infile = fopen("in.txt", "r");
     result = fgets(line, LINE_SIZE, infile);
     while (result != NULL)
     {
        if (line[strlen(line)-1]=='\n')
        {
           line[strlen(line)-1]='\0';
        }
        if ((size>=NUM_STRINGS) || (strlen(line)>=STRING_SIZE))
        {
           fprintf(stderr, "\nRan out of memory.\n");
           exit(EXIT_FAILURE);
        }
        strcpy(strings[size], line);
        printf("<%s>\n", line);
        result = fgets(line, LINE_SIZE, infile);
        size++;
     }
     fclose(infile);

     printf("\n");
     for (count=0; count<size; count++)
     {
        printf("<%s>\n", (char*) strings[count]);
     }

     printf("\nAll done\n");
     return 0;
}
```

In the second example, memory is allocated dynamically for each string. A pointer to the

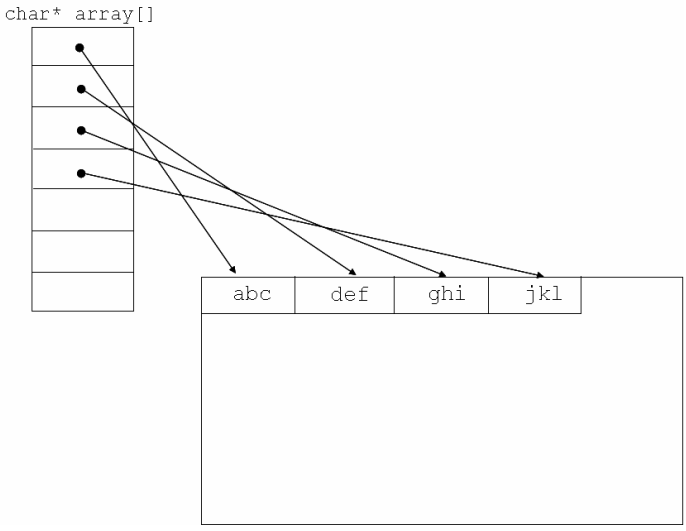
memory is stored in a `char*` array. Since the memory is allocated on a string by string basis, the amount of memory allocated for each string is just enough to store the string and its termination character. However, the array that contains the pointers to the strings is of fixed size and so there is extra space at the end of the array.




```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     const int LINE_SIZE = 1000;
     const int NUM_STRINGS = 100;

     FILE *infile;
     char* strings[NUM_STRINGS];
     char line[LINE_SIZE];
     char* result;
     int size;
     int count;

     size = 0;
     infile = fopen("in.txt", "r");
     result = fgets(line, LINE_SIZE, infile);
     while (result != NULL)
     {
        if (line[strlen(line)-1]=='\n')
        {
           line[strlen(line)-1]='\0';
        }
        if (size>=NUM_STRINGS)
        {
           fprintf(stderr, "\nRan out of memory.\n");
           exit(EXIT_FAILURE);
        }
        strings[size] = malloc((strlen(line)+1)*sizeof(char));
        if (strings[size] == NULL)
        {
           fprintf(stderr, "\nRan out of memory.\n");
           exit(EXIT_FAILURE);
        }
        strcpy(strings[size], line);
        printf("<%s>\n", line);
        result = fgets(line, LINE_SIZE, infile);
        size++;
     }
     fclose(infile);
```

```
     printf("\n");
     for (count=0; count<size; count++)
     {
        printf("<%s>\n", (char*) strings[count]);
     }

     for (count=0; count<size; count++)
     {
        free(strings[count]);
     }

     printf("\nAll done\n");
     return 0;
}
```

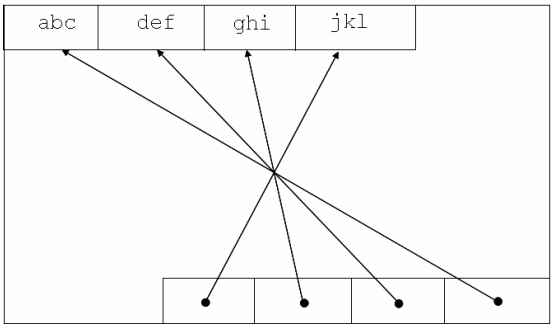
In the third example, one block of memory is allocated at the beginning of the processing and the program then allocates a portion of the big block of memory for each string. As in the previous example, the pointer to each string is stored in an array of pointers.




```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     const int LINE_SIZE = 1000;
     const int NUM_STRINGS = 100;
     const int MEM_SIZE = 2000;

     FILE *infile;
     char* strings[NUM_STRINGS];
     char line[LINE_SIZE];
     char* memoryPool;
     char* nextAvailable;
     char* lastAvailable;
     char* result;
     int size;
     int count;

     size = 0;
```

```
    memoryPool = malloc(MEM_SIZE*sizeof(char));
    nextAvailable = memoryPool;
    lastAvailable = memoryPool+MEM_SIZE-1;
    infile = fopen("in.txt", "r");
    result = fgets(line, LINE_SIZE, infile);
    while (result != NULL)
    {
       if (line[strlen(line)-1]=='\n')
       {
          line[strlen(line)-1]='\0';
       }
       if (size>=NUM_STRINGS)
       {
          fprintf(stderr, "\nRan out of memory.\n");
          free(memoryPool);
          exit(EXIT_FAILURE);
       }
       if (nextAvailable+strlen(line) > lastAvailable)
       {
          fprintf(stderr, "\nRan out of memory.\n");
          free(memoryPool);
          exit(EXIT_FAILURE);
       }
       strings[size] = nextAvailable;
       nextAvailable += (strlen(line)+1)*sizeof(char);
       strcpy(strings[size], line);
       printf("<%s>\n", line);
       result = fgets(line, LINE_SIZE, infile);
       size++;
    }
    fclose(infile);

    printf("\n");
    for (count=0; count<size; count++)
    {
       printf("<%s>\n", (char*) strings[count]);
    }

    free(memoryPool);

    printf("\nAll done\n");
    return 0;
}
```

In the final example, one block of memory is allocated at the beginning of the processing and the program then allocates a portion of the block of memory for each string and a portion of the block for each pointer. The strings are stored at the beginning of the block and the pointers are stored at the end of the block. Using this technique, as strings and pointers are added to the block, the amount of free space in the block becomes smaller and smaller until there is no longer room for the next string and its associated pointer.




```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int numParms, char *parms[])
{
     const int LINE_SIZE = 50;
     const int MEM_SIZE = 2000;

     FILE *infile;
     char line[LINE_SIZE];
     char* memoryPool;
     char* firstString;
     char* nextString;
     int* firstPtr;
     int* nextPtr;
     int* current;
     char* result;
     char* ptr;

     memoryPool = malloc(MEM_SIZE*sizeof(char));
     firstString = memoryPool;
     nextString = firstString;
     firstPtr = (int*) memoryPool;
     firstPtr += (MEM_SIZE-sizeof(int))/sizeof(int);
     nextPtr = firstPtr;

     infile = fopen("in.txt", "r");
     result = fgets(line, LINE_SIZE, infile);
     while (result != NULL)
     {
        if ((((char*)nextString)+strlen(line)) > ((char*)nextPtr))
        {
           fprintf(stderr, "\nRan out of memory\n");
           free(memoryPool);
           exit(EXIT_FAILURE);
        }
        if (line[strlen(line)-1]=='\n')
        {
           line[strlen(line)-1]='\0';
        }
        strcpy(nextString, line);
        *nextPtr = (int) nextString;
        printf("<%s>\n", line);
        nextString += strlen(line)+1;
        nextPtr -= 1;
        result = fgets(line, LINE_SIZE, infile);
     }
     fclose(infile);
```

```
    printf("\n\n");
    current = (int*) firstPtr;
    while (current > nextPtr)
    {
       ptr = (char*) *current;
       printf("<%s>\n", ptr);
       current--;
    }

    free(memoryPool);

    printf("\nAll done\n");
    return 0;
}
```

In the last two examples in this section that suballocate memory from a large block of memory, the pointers are absolute pointers, that is, they point directly at the memory address of the associated string. These pointers could also be made relative pointers (relative to the beginning of the block of memory). The pointers would then be an offset into the block of memory. To determine the actual address of a string, the address of the beginning of the block of memory would be added to the value of the offset into the block. Using this type of addressing would permit the block to be resized more easily if it became full.

Also, all four of the examples handle only insertions into the collection of strings. Other operations such as deleting a string and resizing a string are not handled but the processing for these operations could be added with some additional effort.

The examples shown in this section provide the foundation for implementing a module that maintains a collection of variable-length “objects” such as the array list module that has been developed throughout these notes.

4.26 Function Pointers

In the previous sections of this chapter, we have seen how pointers can be used to point to data values. In this section, we examine how pointers can also be used to point to functions.

Suppose that we have to write a function that performs various types of processing on an array. We create a different version of the evaluation function for each type of processing that can be performed.

```
#include <stdio.h>

int evalF1(int[], int);
int evalF2(int[], int);
int function1(int);
int function2(int);

int main(int numParms, char *parms[])
{
```

```
     int array1[] = {1, 2, 3, 4, 5};
     int result;

     result = evalF1(array1, 5);
     printf("%d\n", result);

     result = evalF2(array1, 5);
     printf("%d\n", result);

     return 0;
}

int evalF1(int array[], int size)
{
     int result;
     int count;

     result = 0;
     for (count=0; count<size; count++)
     {
        result += function1(array[count]);
     }
     return result;
}

int evalF2(int array[], int size)
{
     int result;
     int count;

     result = 0;
     for (count=0; count<size; count++)
     {
        result += function2(array[count]);
     }
     return result;
}

int function1(int value)
{
     return value;
}

int function2(int value)
{
     return value*value;
}

15
55
```

In the simple example shown above, the processing can be defined quite easily. If the number of possible functions was a bit larger, we could simplify the processing somewhat by passing a switch to the evaluation function. The switch would be used by the evaluation function to determine which lower-level function to call.

```
#include <stdio.h>

int evalF(int[], int, int);
int function1(int);
```

```
int function2(int);

int main(int numParms, char *parms[])
{
     int array1[] = {1, 2, 3, 4, 5};
     int result;

     result = evalF(array1, 5, 1);
     printf("%d\n", result);

     result = evalF(array1, 5, 2);
     printf("%d\n", result);

     return 0;
}

int evalF(int array[], int size, int whichFunction)
{
     int result;
     int count;

     result = 0;
     for (count=0; count<size; count++)
     {
        if (whichFunction == 1)
        {
           result += function1(array[count]);
        }
        else if (whichFunction == 2)
        {
           result += function2(array[count]);
        }
     }
     return result;
}

int function1(int value)
{
     return value;
}

int function2(int value)
{
     return value*value;
}

15
55
```

Instead of hard-coding the values of the switch, using an enumerated type would be more appropriate.

Suppose that we have a large number of functions that could be applied. We would not want to create a large number of condition statements in the evaluation function, one for each type of processing that can be applied. The following example illustrates how we can pass a function as parameter (actually a pointer to the function) and then invoke the function within the function that is called.

```
#include <stdio.h>

int eval(int[], int, int(*fp)(int));
int function1(int);
int function2(int);

int main(int numParms, char *parms[])
{
     int array1[] = {1, 2, 3, 4, 5};
     int (*f) (int);
     int result;

     f = function1;
     result = eval(array1, 5, *f);
     printf("%d\n", result);

     f = function2;
     result = eval(array1, 5, *f);
     printf("%d\n", result);

     return 0;
}

int eval(int array[], int size, int (*f) (int))
{
     int result;
     int count;

     result = 0;
     for (count=0; count<size; count++)
     {
        result += f(array[count]);
     }
     return result;
}


int function1(int value)
{
     return value;
}

int function2(int value)
{
     return value*value;
}

15
55
```

The program above is significantly easier to write than the previous two programs. However, we can still improve the program slightly by passing the address of the appropriate function in the argument list for `eval`.

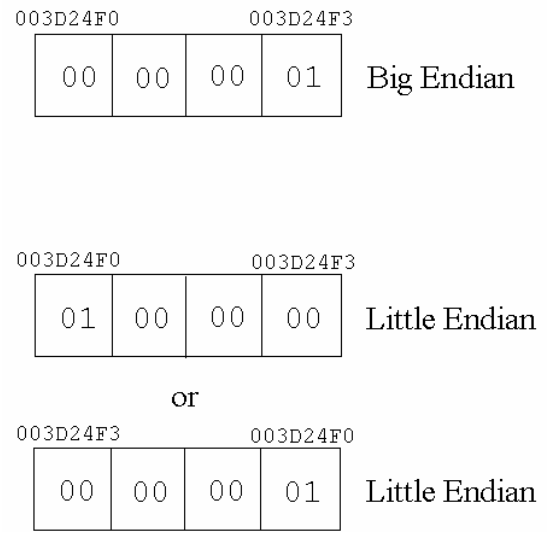
```
#include <stdio.h>

int eval(int[], int, int(*fp)(int));
int function1(int);
int function2(int);

int main(int numParms, char *parms[])
```

```
{
     int array1[] = {1, 2, 3, 4, 5};
     int result;

     result = eval(array1, 5, &function1);
     printf("%d\n", result);

     result = eval(array1, 5, &function2);
     printf("%d\n", result);

     return 0;
}

int eval(int array[], int size, int (*f) (int))
{
     int result;
     int count;

     result = 0;
     for (count=0; count<size; count++)
     {
        result += f(array[count]);
     }
     return result;
}


int function1(int value)
{
     return value;
}

int function2(int value)
{
     return value*value;
}

15
55
```

Now that we can define function pointers, we have the ability to define “complete” objects using a structure. The structure can contain not only the data (instance variables) of the object but also pointers to the functions used by the objects. These objects would have to be hand crafted but it is possible (with some additional effort) to invoke the function pointers dynamically using a table of function pointers and the associated functions. (Take a look on the web for “vtable” or virtual function table for more information.)

4.27 Big/Little Endian Memory Organization

The order in which the bytes used to represent ints, doubles, floats, pointers, etc. are stored is referred to as big-endian or little-endian memory organization. The diagram below illustrates the order in which the bytes used to store the integer value 1 are stored at the memory location `003D24F0` in the big endian and little endian representations. Note that with the little endian representation, the bytes can be viewed in two equivalent ways (the only difference is the order of the memory locations).



The following function can be used to determine which memory organization is used by a particular machine.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     int num;

     num = 1;
     if((*(char *)&num) == 1)
     {
        printf("Little-endian machine\n");
     }
     else
     {
        printf("Big-endian machine.\n");
     }
     return 0;
}

Little-endian machine   // Windows XP 32-bit machine

Little-endian machine   // x86_64 Linux system (member of CS Aviary flock)

Big-endian machine      // Sun Blade 1000 system, running Solaris 8
```

On Unix/Linux use the command `uname -a` to obtain information about the machine type.

The reason for introducing the little/big-endian memory organization is so that the output generated by the memory dump function described in the next section will make a bit more sense.

4.28 Memory Dump Function

When working with pointers, it can be helpful if you examine the contents of memory (and not just the contents of a few variables). The following function prints the contents of memory beginning at the memory location defined by the first parameter. The second parameter, s, specifies the number of bytes to be printed.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void memoryDump(void *p, int s)
{
     int *ptr;
     int *intPtr;
     int size;
     unsigned char *charPtr;
     unsigned char *charPtr2;

     printf("\n");
     printf("%10c %8c %8c %8c          0...4...8...C...\n", '0', '4', '8', 'C');
     intPtr = (int*) ((((long unsigned int) p) >> 4) << 4);
     size = (s+15)/16*4;
     for (ptr=intPtr; ptr<intPtr+size; ptr=ptr+4)
     {
        printf("%08X ", (unsigned int) ptr);

        for (charPtr2=(char*)ptr; ((char*)charPtr2)<(char*)((long int)(ptr)+16); charPtr2+=4)
        {
           printf("%02X%02X%02X%02X ",*charPtr2,*(charPtr2+1),*(charPtr2+2),*(charPtr2+3));
        }
        printf("  ");

        charPtr = (char*) ptr;
        for (charPtr2=charPtr; charPtr2<charPtr+16; charPtr2++)
        {
           printf("%c", isprint(*charPtr2) ? *charPtr2 : '.');
        }
        printf("\n");
     }
}
```

A conditional operator is used to determine (using the `isprint` function which is included in the `ctype` library) whether or not a character is printable and then print either the character or a period.

The following program declares several arrays, prints the address of the first element in each array, and then uses `memoryDump` to print the contents of some of the memory used by the program.

```
#include <stdio.h>

void memoryDump(void*, int);

int main(int numParms, char *parms[])
```

```
{
     int array1[] = {1, 2, 3, 4, 5};
     int array2[] = {6, 7};
     int array3[] = {8, 9, 10};
     char msg[] = "Hello";

     printf("array1 is at %p \n", (int*) &array1[0]);
     printf("array2 is at %p \n", (int*) &array2[0]);
     printf("array3 is at %p \n", (int*) &array3[0]);
     printf("msg    is at %p \n", (char*) &msg[0]);

     memoryDump(&msg[0], 80);

     return 0;
}
```

The output from this program is shown below. Notice that the arrays are not allocated in order. The program was run on a Windows (little-endian) machine so integer values are stored with the smallest byte used to represent an `int` value first and the largest byte last.

```
array1 is at 0022FF68
array2 is at 0022FF60
array3 is at 0022FF48
msg    is at 0022FF38

          0        4        8        C          0...4...8...C...
0022FF30 14B8C377 00000000 48656C6C 6F000000   ...w....Hello...
0022FF40 ADAEC377 864BC167 08000000 09000000   ...w.K.g........
0022FF50 0A000000 02000000 60FF2200 05EFC177   ........`."....w
0022FF60 06000000 07000000 01000000 02000000   ................
0022FF70 03000000 04000000 05000000 00304000   .............0@.
```

When the same program is run on a Sun Blade system (big-endian machine), the following results are generated. Note that the order of the bytes used to represent each `int` are reversed.

```
array1 is at ffbefb08
array2 is at ffbefb00
array3 is at ffbefaf0
msg    is at ffbefae8

          0        4        8        C          0...4...8...C...
FFBEFAE0 FF29BC20 00000000 48656C6C 6F000000   .). ....Hello...
FFBEFAF0 00000008 00000009 0000000A 00000000   ................
FFBEFB00 00000006 00000007 00000001 00000002   ................
FFBEFB10 00000003 00000004 00000005 FFBEFB9C   ................
FFBEFB20 00000005 FFBEFC00 00000000 00000000   ................
```

If the program is run on a 64-bit little-endian machine, the pointers are 8 bytes long and the information generated by memoryDump is truncated but this should not cause any confusion since the full address of the pointer is displayed in the calling program. (Compiling the program generates the message: `warning: cast from pointer to integer of different size`) The memoryDump program could be modified to generate 16 hexadecimal values instead of 8 if it is necessary to view all values.

```
array1 is at 0x7fff1df1d440
array2 is at 0x7fff1df1d430
array3 is at 0x7fff1df1d420
msg    is at 0x7fff1df1d410

         0        4        8        C          0...4...8...C...
1DF1D410 48656C6C 6F000000 1B044000 00000000   Hello.....@.....
1DF1D420 08000000 09000000 0A000000 00000000   ................
1DF1D430 06000000 07000000 C0AB21B9 3F000000   ..........!.?...
1DF1D440 01000000 02000000 03000000 04000000   ................
1DF1D450 05000000 FF7F0000 00000000 00000000   ................
```

4.29 Pointer Pitfalls

Working with pointers correctly is arguably the most difficult part of learning to program in the C language. The following are common mistakes that are made (both by beginning programmers and by experienced programmers) when manipulating memory:

- forget to assign an address to a pointer variable;
- forget to assign a value to the memory location that the pointer variable points to;
- forget to free memory that has been allocated but is no longer required (a memory leak);
- forget to assign a value of NULL to a pointer variable once the memory that it used to point to has been freed;
- set a pointer variable to point to a variable that is declared within a function (on the stack) and then use the pointer variable once the function has returned control to the calling function.

4.30 Summary

In this chapter, we examined the basics of pointer manipulation and memory management. The coverage was deliberately brief. As you delve deeper into the C language, you will find that there are many additional issues involving pointers and the use of memory.

4.31 Questions

1. How would you implement the shallow copying of an array?

2. How would you implement the deep copying of an array?

3. What happens if you pass `&array1[4]` to a function (where `array1` is declared to have at least 5 elements)?

4. When `realloc` is used to resize a chunk of memory, it is possible that the new memory location could be the same as the old memory location. What are the circumstances in

which this could be true?

5. What is the difference between a symbolic constant and a literal constant?

6. The heap contains memory that is available for dynamic allocation. Describe an algorithm that could be used by the C run-time system to manage the heap (keep track of which chunks of memory are being used and which chunks of memory are available for use.)

5 DEBUGGING

5.1 Introduction

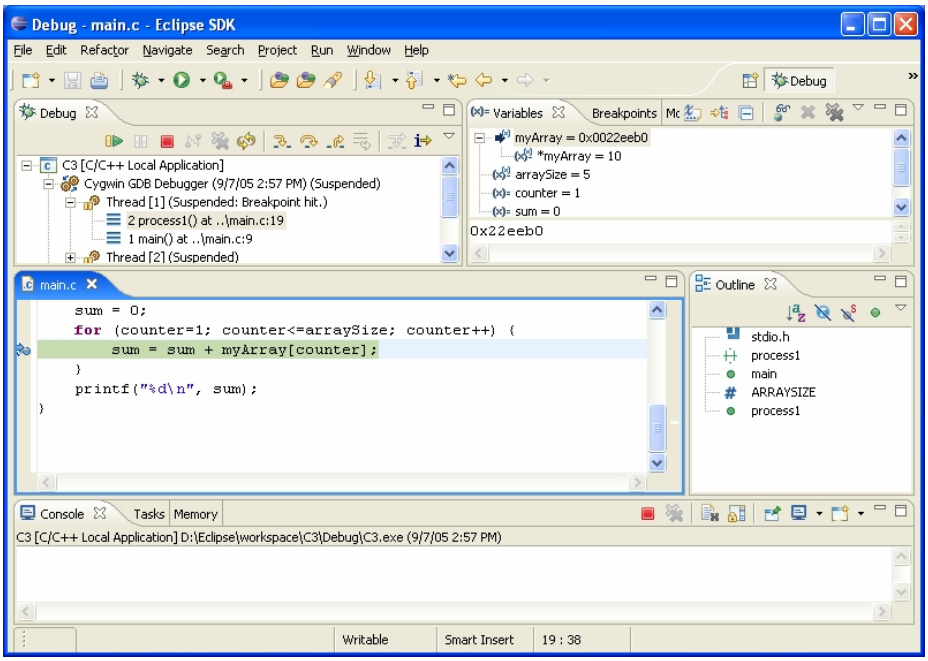
When developing C programs, it is unusual for programs to work correctly the first time. There are many subtle problems that can arise during the development of a program and while printing the contents of variables (using `printf`) and/or dumping the contents of memory (using `memoryDump`) can be helpful, it is often necessary to dig deeper into the execution of a program.

To support a more detailed examination of a program, most versions of C include a debugger which allows a program to be executed under the control of the programmer. For example, you can run the program but have the program pause when it reaches a specific statement. At that time, you can examine the contents of variables and make modifications to the values of variables. Learning to use the C debugger makes the identification of problems significantly easier than using only print statements throughout the program.

5.2 Eclipse

If you are using the Eclipse IDE, you will find working with the debugger to be much easier than using the debugger from the command line. With Eclipse, you can switch from normal C development mode to C debugger mode simply by clicking the Debug icon. You can find additional information on developing and debugging C programs in Eclipse at:

`http://www.cs.umanitoba.ca/~eclipse/Eclipse3-1.pdf`



5.3 C Debugger

In the remainder of this chapter, we assume that you are using the C debugger in a command window.

To begin, ensure that the C debugger is available on your system. Type the command `gdb –v` at the command prompt. You should receive output similar to the following if the debugger is installed.

```
C:\Ch5>gdb –v

GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".

C:\Ch5>
```

To begin, we will attempt to determine the problem with the following program. The result shown at the bottom of the program is obviously not correct. (The problem with the program should be obvious – we will use this simple program only to illustrate some of the features of the debugger.) The line number of each line in the source program has been included. We will soon see why the line numbers are important.

```
 1 #include <stdio.h>
 2
 3 int main(int numParms, char *parms[])
 4 {
 5     #define MAX_SIZE 5
 6     int array1[MAX_SIZE] = {1, 2, 3, 4, 5};
 7     int count;
 8     int result;
 9
10     result = 0;
11     for (count=1; count<=MAX_SIZE; count++)
12     {
13        result += array1[count];
14     }
15
16     printf("The result is %d\n", result);
17
18     return 0;
19 }

The result is 4206606
```

Start the debugger by typing **gdb**.

```
C:\Ch5>gdb
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".
```

The debugger prompt `(gdb)` is now displayed. Typing `help` provides a list of the categories of commands.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Typing `help` *`category`* displays the commands in a particular category.

```
(gdb) help running

Running the program.

List of commands:

attach -- Attach to a process or file outside of GDB
continue -- Continue program being debugged
detach -- Detach a process or file previously attached
finish -- Execute until selected stack frame returns
handle -- Specify how to handle a signal
info handle -- What debugger does when program gets various signals
interrupt -- Interrupt the execution of the debugged program
jump -- Continue program being debugged at specified line or address
kill -- Kill execution of program being debugged
next -- Step program
nexti -- Step one instruction
run -- Start debugged program
set args -- Set argument list to give program being debugged when it is started
set environment -- Set environment variable value to give the program
set follow-fork-mode -- Set debugger response to a program call of fork or vfork
set scheduler-locking -- Set mode for locking scheduler during execution
set step-mode -- Set mode of the step operation
show args -- Show argument list to give program being debugged when it is starte---
show follow-fork-mode -- Show debugger response to a program call of fork or vfork
show scheduler-locking -- Show mode for locking scheduler during execution
show step-mode -- Show mode of the step operation
signal -- Continue program giving it signal specified by the argument
```

```
step -- Step program until it reaches a different source line
stepi -- Step one instruction exactly
target -- Connect to a target machine or process
thread -- Use this command to switch between threads
thread apply -- Apply a command to a list of threads
apply all -- Apply a command to all threads
tty -- Set terminal for future runs of program being debugged
unset environment -- Cancel environment variable VAR for the program
until -- Execute until the program reaches a source line greater than the current

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

```
(gdb) help breakpoints

Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
catch -- Set catchpoints to catch events
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
hbreak -- Set a hardware assisted  breakpoint
ignore -- Set ignore-count of breakpoint number N to COUNT
rbreak -- Set a breakpoint for all functions matching REGEXP
rwatch -- Set a read watchpoint for an expression
tbreak -- Set a temporary breakpoint
tcatch -- Set temporary catchpoints to catch events
thbreak -- Set a temporary hardware assisted breakpoint
watch -- Set a watchpoint for an expression

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

```
(gdb) help data
Examining data.

List of commands:

call -- Call a function in the program
delete display -- Cancel some expressions to be displayed when program stops
delete mem -- Delete memory region
disable display -- Disable some expressions to be displayed when program stops
disable mem -- Disable memory region
disassemble -- Disassemble a specified section of memory
display -- Print value of expression EXP each time the program stops
enable display -- Enable some expressions to be displayed when program stops
enable mem -- Enable memory region
inspect -- Same as "print" command
mem -- Define attributes for memory region
output -- Like "print" but don't put in value history and don't print newline
print -- Print value of expression EXP
printf -- Printf "printf format string"
ptype -- Print definition of type TYPE
```

```
set -- Evaluate expression EXP and assign result to variable VAR
set variable -- Evaluate expression EXP and assign result to variable VAR
undisplay -- Cancel some expressions to be displayed when program stops
whatis -- Print data type of expression EXP
x -- Examine memory: x/FMT ADDRESS

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Typing `quit` (or `q`) exits the debugger.

```
(gdb) quit
```

To work with an existing executable file, load the file when you start the debugger.

```
gdb ch5
```

You can now run the program.

```
(gdb) run

Starting program: C:\Ch5/ch5.exe
Program exited normally.
(gdb)
```

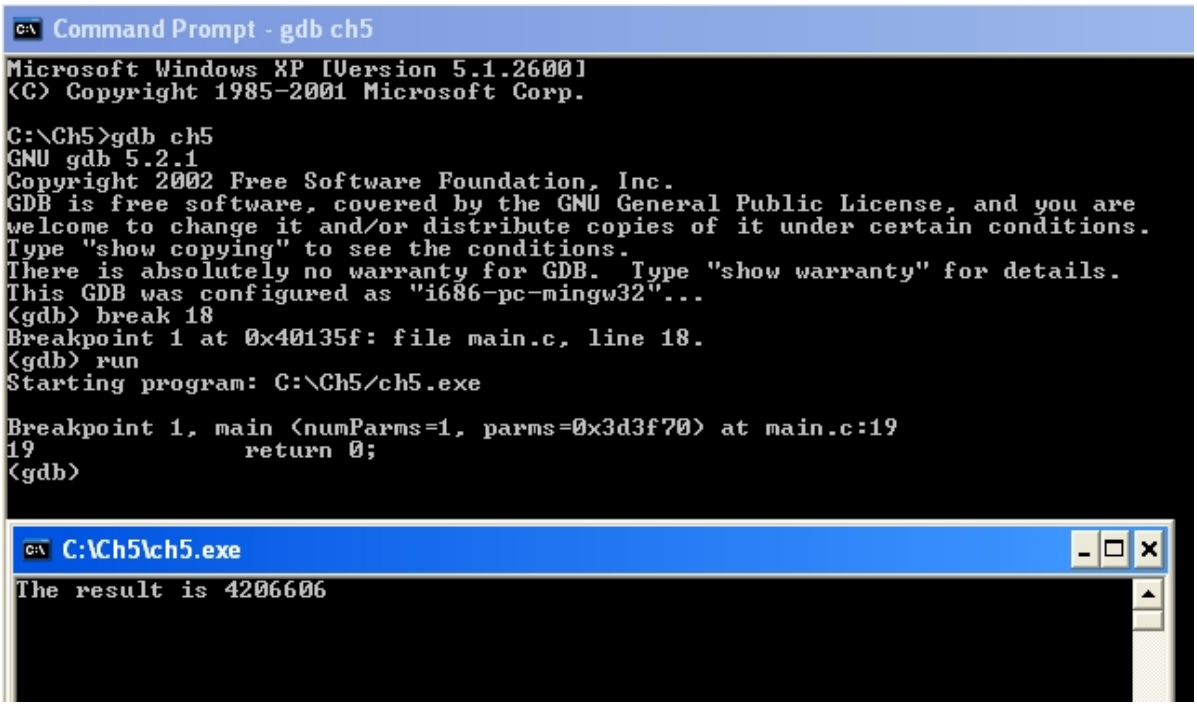
Unfortunately, the program doesn't display any output. We will see how to fix this in the next section.

5.4 Setting Breakpoints

A breakpoint is a position (line number) in the program at which the debugger temporarily pauses execution of the program. If we define a breakpoint at line 18, we can examine the status of the program at that particular point (before the instruction at the breakpoint has been executed).

```
(gdb) break 18
Breakpoint 1 at 0x40135f: file main.c, line 18.
(gdb) run
Starting program: C:\Ch5/ch5.exe

Breakpoint 1, main (numParms=1, parms=0x3d3e88) at main.c:18
18              return 0;
(gdb)
```

The debugger (at least on Windows) sends the program output to a separate console window. When the program pauses at the breakpoint, you can view the current output in this window.



Use the `continue` command to continue execution from the breakpoint.

```
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

5.5 Examining the Contents of Variables

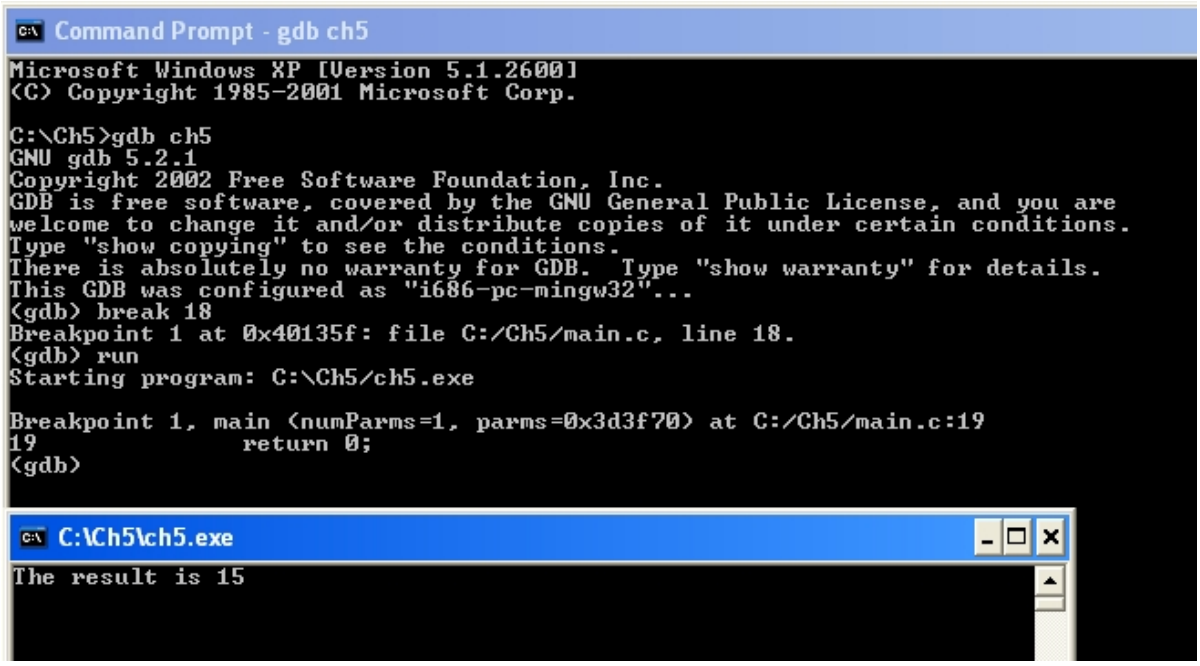
Now we are able to run the program and to pause at any line in the program. We would like to know why `result` is not being set to the correct value so we will define breakpoints at lines 13 and 18. We can then use the `print` statement to print the current contents of any variable.

```
(gdb) break 13
Breakpoint 1 at 0x401339: file main.c, line 13.
(gdb) break 18
Breakpoint 2 at 0x40135f: file main.c, line 18.
(gdb) run
Starting program: C:\Ch5/ch5.exe

Breakpoint 1, main (numParms=1, parms=0x3d3e88) at main.c:13
13                      result += array1[count];
(gdb) print result
$1 = 0
(gdb) continue
Continuing.

Breakpoint 1, main (numParms=1, parms=0x3d3e88) at main.c:13
13                      result += array1[count];
(gdb) print result
```

```
$2 = 2
(gdb)
```

As can be seen from the results of the print statements, the first time in the loop `result` is correctly set to 0 (prior to the execution of the statement `result+=array1[count]`). However, on the next iteration through the loop, `result` has a value of 2. This value is not correct (we have skipped over the first element in the array). A close examination of the program should solve the problem – the loop is defined incorrectly. If the program is modified as shown below, it should now execute correctly.

```
11   for (count=0; count<MAX_SIZE; count++)
```



As can be seen, the result is now correct.

5.6 Summary

This chapter has presented a very brief introduction to the C debugger. You should examine the commands that can be issued and experiment with the commands in order to become more familiar with the debugger.

6 DESIGN BY CONTRACT

6.1 Introduction

Beginning with this chapter, we finally start to add the code that will improve the quality of the programs that we write. In this chapter, we examine how error checking can be added to a module to ensure that the module functions correctly even in the presence of erroneous input values.

6.2 Design by Contract

Programs that ensure that certain conditions are met before the programs perform any processing are said to have been "designed by contract". The following is an excerpt from wikipedia on design by contract.

"If a routine from a class in object-oriented programming provides a certain functionality, it may:

- Impose a certain obligation to be guaranteed on entry by any client module that calls it: the routine's **precondition** — an obligation for the client, and a benefit for the supplier (the routine itself), as it frees it from having to handle cases outside of the precondition.
- Guarantee a certain property on exit: the routine's **postcondition** — an obligation for the supplier, and obviously a benefit (the main benefit of calling the routine) for the client.
- Maintain a certain property, assumed on entry and guaranteed on exit: the class **invariant**.

"The contract is the formalization of these obligations and benefits. One could summarize design by contract by the 'three questions' that the designer must repeatedly ask:

- What does it expect?
- What does it guarantee?
- What does it maintain? "

http://en.wikipedia.org/wiki/Design_by_contract

The statement above is a slightly longwinded way of saying that in any module that you write, you must ensure that any parameters passed to the module are valid (the preconditions), that the state of your processing and data structures is always valid (the invariants), and that any values returned by your module are also valid (the postconditions).

6.3 Basic Array List Module

The array list module that was developed in Chapter 4 is shown below (although this version has fixed the memory leak that existed by including a `destroyList` function). As can be seen, this module contains no error-checking and would fall apart if the user of the module made any mistakes.

```
#include <stdio.h>
#include "module1.h"

#define NumEntries 5

struct arrayList
{
     int maxSize;
     int size;
     int *data;
};

ArrayList newList()
{
     ArrayList list;
     int count;

     list = (ArrayList) malloc(sizeof(struct arrayList));
     list -> data = (int*) malloc(NumEntries*sizeof(int));
     for (count=0; count<NumEntries; count++)
     {
        list -> data[count] = 0;
     }
     list -> size = 0;
     list -> maxSize = NumEntries;
}

void addList(ArrayList list, int value)
{
     if (list->size >= list->maxSize)
     {
        resizeList(list, list->maxSize+1);
     }
     list -> data[list -> size] = value;
     list -> size++;
}

void resizeList(ArrayList list, int newSize)
{
     int count;

     list -> data = (int*) realloc(list->data, newSize*sizeof(int));
     list -> maxSize = newSize;
     for (count=list->size; count<list->maxSize; count++)
     {
        list -> data[count] = -1; // initialize new locations (just being careful)
     }
}

int getList(ArrayList list, int position)
{
     int entry;

     entry = list -> data[position];
     return entry;
}

int sizeList(ArrayList list)
{
     return list -> size;
}
```

```
void removeList(ArrayList list, int position)
{
     int count;

     for (count=position; count<(list -> size-1); count++)
     {
        list -> data[count] = list -> data[count+1];
     }
     list -> size--;
     list -> data[list -> size] = 0;
}

ArrayList destroyList(ArrayList list)
{
     free(list->data);
     free(list);

     return NULL;
}

void printList(ArrayList list)
{
     int count;

     printf("\nContents of list:\n");
     for (count=0; count<list -> size; count++)
     {
        printf("Element %d is %d\n", count, getList(list, count));
     }
     printf("\n");
}
```

6.4 Error Checking

We would like to make the array list module as fool-proof as possible. There are many opportunities for the user of this module to make mistakes and it is the responsibility of the person who develops the module to perform as much error-checking as possible.

A simple form of error-checking uses the basic if statement. With this statement, we compare a parameter with its expected values and generate an error message if the value of the parameter is not correct. However, since this is such a common sequence of actions, C provides a library (assert.h) that includes macros that assist us in writing error-checking code.

A simple form of an `assert` statement (macro) is:

```
assert(list != NULL);
```

The logical expression inside the parentheses is evaluated: if the result of the expression is true, processing continues; if the expression is false, processing terminates with an appropriate error message.

The following program shows the array list module with assert statements that attempt to

prevent the module from performing any incorrect actions.

```
#include <stdio.h>
#include <assert.h>
#include "module1.h"

#define NumEntries 5

struct arrayList
{
     int maxSize;
     int size;
     int *data;
};

ArrayList newList()
{
     ArrayList list;
     int count;

     list = (ArrayList) malloc(sizeof(struct arrayList));
     assert(list != NULL);

     assert(NumEntries > 0);
     list -> data = (int*) malloc(NumEntries*sizeof(int));
     assert(list->data != NULL);

     for (count=0; count<NumEntries; count++)
     {
        list -> data[count] = 0;
     }
     list -> size = 0;
     list -> maxSize = NumEntries;
}

void addList(ArrayList list, int value)
{
     assert(list != NULL);
     assert((0 <= list->size) && (list->size <= list->maxSize));

     if (list->size >= list->maxSize)
     {
        resizeList(list, list->maxSize+1);
     }
     list -> data[list->size] = value;
     list -> size++;
}

void resizeList(ArrayList list, int newSize)
{
     int count;

     assert(list != NULL);

     list -> data = (int*) realloc(list->data, newSize*sizeof(int));
     assert(list->data != NULL);

     list -> maxSize = newSize;
     for (count=list->size; count<list->maxSize; count++)
     {
        list -> data[count] = -1; // initialize new locations (just being careful)
     }
```

```
}

int getList(ArrayList list, int position)
{
     int entry;

     assert(list != NULL);
     assert((0 <= list->size) && (list->size <= list->maxSize));
     assert((0 <= position) && (position < list->size));

     entry = list -> data[position];
     return entry;
}

int sizeList(ArrayList list)
{
     assert(list != NULL);
     assert((0 <= list->size) && (list->size <= list->maxSize));
     return list -> size;
}

void removeList(ArrayList list, int position)
{
     int count;
     assert(list != NULL);
     assert((0 <= list->size) && (list->size <= list->maxSize));
     assert((0 <= position) && (position < list->size));

     for (count=position; count<(list -> size-1); count++)
     {
        list -> data[count] = list -> data[count+1];
     }
     list -> size--;
     list -> data[list -> size] = 0;
}

ArrayList destroyList(ArrayList list)
{
     if (list != NULL)
     {
        if (list->data != NULL)
        {
           free(list->data);
        }
        free(list);
     }

     return NULL;
}

void printList(ArrayList list)
{
     int count;

     assert(list != NULL);
     assert((0 <= list->size) && (list->size <= list->maxSize));

     printf("\nContents of list:\n");
     for (count=0; count<list -> size; count++)
     {
        printf("Element %d is %d\n", count, getList(list, count));
     }
     printf("\n");
```

```
}
```

In this module, when checking that we have a valid `list` data structure, the best we can do is ensure that the pointer to `list` is not `NULL`. If we want to add some additional protection for this data structure, we could add an additional variable (such as an `int`) to the beginning of the data structure and set this variable to a specific and unusual value. Then, in addition to ensuring that the pointer is not `NULL`, we could also ensure that the additional variable contains the correct value.

6.5 Summary

Making programs as error-proof as possible is not particularly hard work but it is work that is often avoided by the programmer who is rushing to meet a deadline. A solution to this problem is to write the error-checking code at the same time as the basic code is being written, that is, do not leave error-checking until the end of development of the program. Once you become familiar with writing appropriate error-checking code along with your basic code, you are on the way to becoming a good developer.

7 UNIT TESTS

7.1 Introduction

In the previous chapter, we examined how comprehensive error checking could be added within modules to ensure that each module is able to detect incorrect parameters or invalid situations. Performing such internal testing is one step towards providing quality software. A second step involves ensuring that a module (or small group of modules) generates the correct output for a collection of test problems. This process is referred to as unit testing. In the olden days of computing, the programmer (or a program tester) performed unit tests manually – typing values for parameters and then visually determining whether or not the function returned the correct value. Now, however, unit tests are specified in an executable form. So a unit test is defined in another module and it calls a particular function with specific parameter values and then compares the output returned by the function with the correct output to ensure that the module is functioning correctly.

Unit tests are an executable specification of what the system is supposed to do. Unit tests can (and should) be re-run every time that the system is rebuilt to ensure that no changes have been made that result in incorrect processing. By defining unit tests and running them frequently, we add additional quality to the system.

7.2 A Simple Unit Test

We will begin by writing a simple set of unit tests for the array list module that was developed in Chapter 4.

The unit test is fairly simple. To begin, an initialization function (`initSuite`) is called to initialize any variables required by the test. Then, each unit test (`addTest`, `removeTest`) is called in turn. Finally, a cleanup function (`cleanSuite`) is called to perform any final processing that is necessary.

```
#include <stdio.h>
#include <assert.h>
#include "module1.h"

int initSuite(void);
int cleanSuite(void);
void addTest(void);
void removeTest(void);

ArrayList myList;

int main(int numParms, char *parms[])
{
     int returnCode;

     if (initSuite()==0)
```

```
     {
        addTest();
        removeTest();
        cleanSuite();
        printf("Tests completed successfully.\n");
        returnCode = 0;
     }
     else
     {
        printf("System could not be initialized.\n");
        returnCode = 1;
     }

     return returnCode;
}

// The suite initialization function.

int initSuite(void)
{
     int result;

     if ((myList=newList()) != NULL)
     {
        result = 0;
     }
     else
     {
        result = 1;
     }
     return result;
}

// The suite cleanup function.

int cleanSuite(void)
{
     myList = destroyList(myList);
     return 0;
}

// Test adding elements to the list

void addTest(void)
{
     assert(myList != NULL);

     addList(myList, 1);
     assert(sizeList(myList)==1);
     assert(getList(myList,0)==1);

     addList(myList, 2);
     assert(sizeList(myList)==2);
     assert(getList(myList,0)==1);
     assert(getList(myList,1)==2);

     addList(myList, 3);
     assert(sizeList(myList)==3);
     assert(getList(myList,0)==1);
     assert(getList(myList,1)==2);
     assert(getList(myList,2)==3);
}
```

```
// Test removing elements from the list

void removeTest(void)
{
     assert(myList != NULL);

     assert(sizeList(myList)==3);

     removeList(myList, 0);
     assert(sizeList(myList)==2);
     assert(getList(myList,0)==2);
     assert(getList(myList,1)==3);

     removeList(myList, 0);
     assert(sizeList(myList)==1);
     assert(getList(myList,0)==3);

     removeList(myList, 0);
     assert(sizeList(myList)==0);
}
```

This program can be compiled using the statement:

```
gcc –ggdb unittest.c module1.c -o unittest
```

When the program is run, the following output is generated:

```
Tests completed successfully.
```

Unfortunately, there are some problems with this approach. First, as soon as a test fails, the entire system halts. Secondly, there is no information generated about the number of tests that were performed and the number of tests that were successful. This information could be generated by collecting the various statistics but we will see in the next section how a unit-testing framework can make life much easier for us.

7.3 CUnit

In the past decade (or so), various frameworks for developing unit tests have been created. One of the most popular frameworks is JUnit, a framework for writing unit tests for Java programs. JUnit has been ported to C with the result being the CUnit framework. In this section, we describe how to setup the CUnit framework. (It should be noted that the JUnit framework is more user friendly than CUnit plus JUnit has been integrated into the Eclipse system while CUnit has not yet been integrated into Eclipse.)

CUnit consists of several components that must be installed correctly before CUnit will run. To avoid path problems, the following installation instructions describe how the CUnit components can be integrated with your existing C compiler. While this is not an ideal solution, it does remove the many problems that are encountered when attempting to make the

CUnit libraries available to the C compiler without merging the CUnit libraries into the C compiler directories.

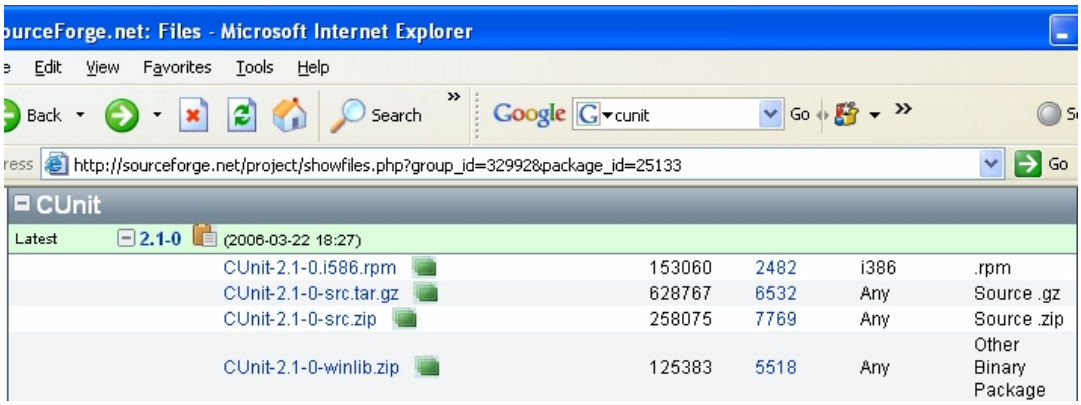
The following description specifies where the CUnit libraries are to be placed in the MinGW C compiler. If you are using a different C compiler, simply replace references to MinGW with the directory in which your C compiler is stored.

First, download CUnit from:

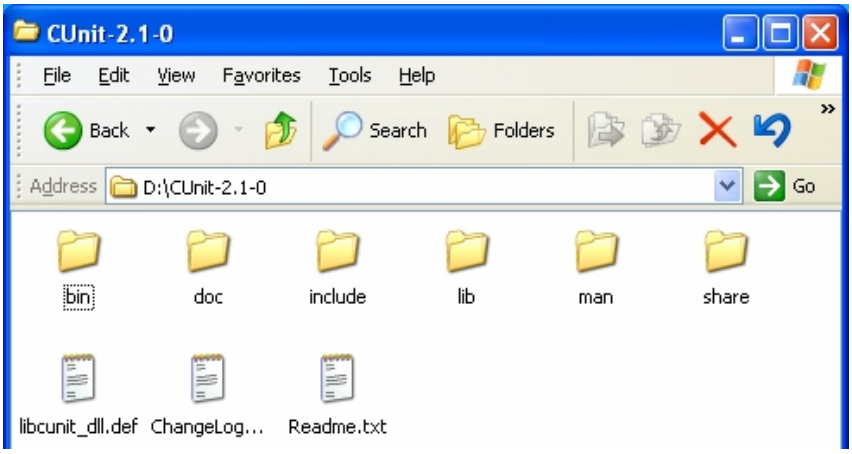
http://cunit.sourceforge.net/

http://sourceforge.net/projects/cunit/

The version that was used in these notes is CUnit-2.1-0-winlib.zip.



When the archieve is unzipped, you will find a variety of folders, as is shown below.



Now, copy the following files or folders into your C compiler:

1. Copy the folder CUnit-2.1-0/include/CUnit to MinGW/include
2. Copy the file CUnit-2.1-0/lib/libcunit_dll.a to MinGW/lib
3. Copy the file CUnit-2.1-0/bin/libcunit.dll to MinGW/bin

7.4 Creating a Unit Test

The following is a unit test that performs the same processing as the simple unit test defined earlier.

```
#include <stdio.h>
#include <string.h>
#include "module1.h"
#include "CUnit/Basic.h"
#include "CUnit/CUnit.h"
#include "CUnit/TestDB.h"
#include "CUnit/TestRun.h"
#include "CUnit/Automated.h"

int initSuite1(void);
int cleanSuite1(void);
void addTest(void);
void removeTest(void);

ArrayList myList;

/* The main() function for setting up and running the tests.
 * Returns a CUE_SUCCESS on successful running, another
 * CUnit error code on failure.
 */

int main(int numParms, char *parms[])
{
     CU_pSuite pSuite = NULL;

     /* initialize the CUnit test registry */
     if (CUE_SUCCESS != CU_initialize_registry())
     {
        return CU_get_error();
     }

     /* add a suite to the registry */
     pSuite = CU_add_suite("Suite_1", initSuite1, cleanSuite1);
     if (NULL == pSuite)
     {
        CU_cleanup_registry();
        return CU_get_error();
     }

     /* add the tests to the suite */
     /* NOTE - ORDER IS IMPORTANT - MUST TEST addTest() BEFORE removeTest() */

     if ((NULL == CU_add_test(pSuite, "test of additions", addTest)) ||
        (NULL == CU_add_test(pSuite, "test of removals", removeTest)))
     {
        CU_cleanup_registry();
        return CU_get_error();
     }

     /* Run all tests using the CUnit Basic interface */
     CU_basic_set_mode(CU_BRM_VERBOSE);
     CU_basic_run_tests();
     CU_cleanup_registry();
     return CU_get_error();
}
```

```
// The suite initialization function.

int initSuite1(void)
{
     if ((myList=newList()) == NULL)
     {
        return -1;
     }
     else
     {
        return 0;
     }
}

// The suite cleanup function.

int cleanSuite1(void)
{
     myList = destroyList(myList);
     return 0;
}

// Test adding elements to list

void addTest(void)
{
     CU_ASSERT(myList != NULL);

     addList(myList, 1);
     CU_ASSERT(sizeList(myList)==1);
     CU_ASSERT(getList(myList,0)==1);

     addList(myList, 2);
     CU_ASSERT(sizeList(myList)==2);
     CU_ASSERT(getList(myList,0)==1);
     CU_ASSERT(getList(myList,1)==2);

     addList(myList, 3);
     CU_ASSERT(sizeList(myList)==3);
     CU_ASSERT(getList(myList,0)==1);
     CU_ASSERT(getList(myList,1)==2);
     CU_ASSERT(getList(myList,2)==3);
}

// Test removing elements from list

void removeTest(void)
{
     CU_ASSERT(myList != NULL);

     CU_ASSERT(sizeList(myList)==3);

     removeList(myList, 0);
     CU_ASSERT(sizeList(myList)==2);
     CU_ASSERT(getList(myList,0)==2);
     CU_ASSERT(getList(myList,1)==3);

     removeList(myList, 0);
     CU_ASSERT(sizeList(myList)==1);
     CU_ASSERT(getList(myList,0)==3);

     removeList(myList, 0);
     CU_ASSERT(sizeList(myList)==0);
```

```
}
```

Although the main function is somewhat large and confusing, most of the code is "boiler plate", code that can be copied exactly as it is into any other unit test. The main function just calls the setup function, calls each unit test, and then calls the cleanup function.

To compile the unit test with the associated module, the following statement is used (note the addition of a library parameter):

```
gcc -ggdb unittest.c module1.c -lcunit_dll -o unittest
```

When the program is run, the following output is generated:

```
C:\Ch7>unittest


     CUnit - A Unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/


Suite: Suite_1
  Test: test of additions ... passed
  Test: test of removals ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         2       2       2       0
               asserts      15      15      15       0
```

As can be seen, CUnit generates more information than we did in the simple unit test, plus CUnit does not stop as soon as an error is encountered.

7.5 Using CUnit with Eclipse

If you want to use CUnit within the Eclipse IDE, perform the following steps:

1. Create a Standard Make C project.
2. Create the unit test source file.
3. Set the binary parser.
4. Set Make Builder **à** build on save.
5. Set Run Parms.
6. Select New Console View (or if you don't, the output will be in a new console window on the right of the screen).
7. Run the project.

Refer to the document at http://www.cs.umanitoba.ca/~eclipse/Eclipse3-1.pdf for additional information.

7.6 Summary

Writing unit tests is absolutely critical to ensuring that high-quality code is generated. Using a framework such as CUnit makes writing a large number of unit tests (and suites of unit tests) relatively straightforward. Do not skimp on unit tests!

In recent years, unit tests have taken on a higher profile due to test-driven development. With TDD, the unit tests are actually written before the basic code instead of afterwards. While this may seem like a backwards way of writing code, it turns out that writing the unit tests first provides many benefits that are not immediately obvious.

8 EXECUTION PROFILING

8.1 Introduction

When developing a system, the programmer should concentrate first on developing high-quality code – code that is correct and is able to detect abnormal situations. We have examined aspects of testing to ensure quality in previous two chapters. Once the quality has been assured, the programmer should make the code as easy to read as possible. There are various refactorings that can be used to improve the readability of code but we will not examine this topic in these notes. Finally, a system must perform at an acceptable level for the type of processing that is being carried out. For example, performing a linear search of an array of 100 elements is not likely to slow the system down significantly (unless the search is performed an extremely large number of times). Performing a linear search of 1,000,000,000 elements does slow a system down significantly.

There may be occasions when a system seems to be performing its processing more slowly than is expected. If this happens, rather than inspect the code by hand (or eye), a program called an execution profiler can be used to locate any hot spots (sections of code that are executed a significant number of times or that take a long time to execute). In this chapter, we examine one such execution profiler, GPROF.

8.2 Using GProf

In this chapter, we describe how use the GPROF system. To begin, ensure that GPROF is included on your system (it should have been included with your C compiler). Use the standard –v flag with the name of the tool to obtain information about the tool.

```
C:\Ch8>gprof -v
GNU gprof 2.13.90
Based on BSD gprof, copyright 1983 Regents of the University of California.
This program is free software.  This program has absolutely no warranty.

C:\Ch8>
```

We will test GPROF with the following program. As you can see, the program simply runs some loops a large number of times.

```
#include <stdio.h>

void run1(void);
void run2(void);

int main(int numParms, char *parms[])
{
     int count;
     int sum;
```

```
     sum = 0;
     for (count=0; count<200000000; count++)
     {
        sum += count;
     }
     printf("\nsum is %d ", sum);

     run1();
     run2();
     run1();
     run2();

     printf("\n\nAll done\n");
     return 0;
}

void run1(void)
{
     int count;
     int sum;

     sum = 0;
     for (count=0; count<300000000; count++)
     {
        sum += count;
     }
     printf("\nsum is %d ", sum);
}

void run2(void)
{
     int count;
     int sum;

     sum = 0;
     for (count=0; count<500000000; count++)
     {
        sum += count;
     }
     printf("\nsum is %d ", sum);
}
```

Once you have ensured that GPROF is available, compile your program as you normally would but include the flag `–pg` in the final step to generate profiling information.

```
gcc -ggdb –pg main.c -o ch8
```

Then, execute your program as you normally would. This step creates a file named `gmon.out` that contains the profile information.

Finally, run gprof with the `–Q` flag followed by the name of your program. This step takes the profile information from the file `gmon.out` and generates a summary of the execution profile of the program.

The following script includes the 2 execution steps (assuming that the file has already been compiled).

```
ch8
gprof -Q ch8.exe
```

The following profile was generated for the program shown above.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 54.96      1.94     1.94                             run2
 32.58      3.09     1.15                             run1
 12.46      3.53     0.44                             main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```

As can be seen from the output, approximately 55% of the execution time was spent in the `run2` function. This is what would be expected given the number of times that each of the loops is executed.

8.3 Summary

In this chapter, we have examined a simple profiling tool that provides information on the amount of time spent in each function. With the speed of current computers, worrying about hot spots is typically not necessary, but if a program does appear to be sluggish, then using a profiler is much more effective than attempting to determine the problem manually. This is particularly true if it is not your program that is causing the slowing but instead is another system that you call from your program.

9 MEMORY MANAGEMENT DEBUGGING

9.1 Introduction

When programming in C, memory management is one of the most difficult parts of a program to get correct. There are various tools available that assist in locating memory management mistakes – in this chapter, we take a look at a simple tool, `memwatch`. If you are interested, you should check out the wikipedia article on memory debugging at:

http://en.wikipedia.org/wiki/Memory_debugger

9.2 Memwatch

`memwatch` is a tool that acts as an interface to the standard C library methods, `malloc`, `calloc`, `realloc`, and `free`. When `memwatch` is used, it can detect most of the common memory management problems.

`memwatch` can be downloaded from:

http://www.linkdata.se/sourcecode.html

Unzip the files and copy the files `memwatch.c` and `memwatch.h` to your directory. Note that you must change the last line in `memwatch.c` to a comment.

9.3 Debugging a Program with Memwatch

The linked-list program that was developed in Chapter 4 contained some memory-allocation problems. We can run this program with `memwatch` and see what `memwatch` is able to determine. Note that some additional flags must be included when the files are compiled.

```
gcc –ggdb -DMEMWATCH -DMEMWATCH_STDIO main.c module1.c memwatch.c -o ch9
```

When the program is run, the console contains the following:

```
C:\Ch9\>ch9

Contents of list:
Element 0 is 3

Contents of list:
Element 0 is 2
Element 1 is 3

Contents of list:
Element 0 is 1
Element 1 is 2
Element 2 is 3
```

```
Contents of list:
Element 0 is 1
Element 1 is 3

Contents of list:
Element 0 is 5
Element 1 is 1
Element 2 is 3

The list is empty.

All done
MEMWATCH detected 5 anomalies
```

memwatch is warning us that there were 5 memory allocation problems. More detailed information is provided in the log file, memwatch.log. (The log file indicates that the problems were memory leaks.)

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Thu Sep 06 18:09:35 2007

Modes: __STDC__ 64-bit mwDWORD==(unsigned long)
mwROUNDALLOC==8 sizeof(mwData)==32 mwDataSize==32


Stopped at Thu Sep 06 18:09:35 2007

unfreed: <5> module1.c(31), 8 bytes at 003D2670      {05 00 00 00 00 00 00 00}
unfreed: <4> module1.c(31), 8 bytes at 003D2630      {01 00 00 00 B0 25 3D 00}
unfreed: <3> module1.c(31), 8 bytes at 003D25F0      {02 00 00 00 B0 25 3D 00}
unfreed: <2> module1.c(31), 8 bytes at 003D25B0      {03 00 00 00 00 00 00 00}
unfreed: <1> module1.c(22), 8 bytes at 003D2570      {00 00 00 00 00 00 00 00}

Memory usage statistics (global):
 N)umber of allocations made: 5
 L)argest memory usage      : 40
 T)otal of all alloc() calls: 40
 U)nfreed bytes totals      : 40
```

One important note about the memwatch.log file is that memwatch **appends to the end of the log file** each time that memwatch is run. In order to avoid confusion, it would be a good idea to delete the memwatch.log file each time before the program is run.

9.4 Fixing the Problems

If the memory-leak problems in the linked-list module are fixed (as shown below), memwatch will run without detecting any errors:

```
#include <stdio.h>
#include "memwatch.h"
#include "module1.h"

struct Node
{
```

```
     int data;
     struct Node *next;
};

struct linkedList
{
     int size;
     struct Node *first;
};

LinkedList newList()
{
     LinkedList list;
     int count;

     list = (LinkedList) malloc(sizeof(struct linkedList));
     list -> first = NULL;
     list -> size = 0;
}

LinkedList addList(LinkedList list, int value)
{
     struct Node *newNode;

     newNode = (struct Node *) malloc(sizeof(struct Node));
     newNode -> data = value;
     newNode -> next = list -> first;
     list -> first = newNode;
     list -> size++;
     return list;
}

int getList(LinkedList list, int position)
{
     int entry;
     int count;
     struct Node *current;

     current = list -> first;
     for (count=0; count<position; count++)
     {
        current = current -> next;
     }
     entry = current -> data;
     return entry;
}

int sizeList(LinkedList list)
{
     return list -> size;
}

LinkedList removeList(LinkedList list, int position)
{
     int entry;
     int count;
     struct Node *current;
     struct Node *previous;

     previous = NULL;
     current = list -> first;
     for (count=0; count<position; count++)
     {
```

```
         previous = current;
         current = current -> next;
      }

      if (previous == NULL)
      {  // delete the first element in the list
         list -> first = current -> next;
         free(current);
      }
      else
      {  // delete an element that has another element before it
         previous -> next = current -> next;
         free(current);
      }

      list -> size--;
      return list;
}

LinkedList destroyList(LinkedList list)
{
      int count;
      int entry;
      struct Node *current;

      if (list->size > 0)
      {
         current = list -> first;
         for (count=0; count<list->size; count++)
         {
            free(current);
            current = current -> next;
         }
      }
      free(list);

      return NULL;
}

void printList(LinkedList list)
{
      int count;

      if (list->size > 0)
      {
         printf("\nContents of list:\n");
         for (count=0; count<list->size; count++)
         {
            printf("Element %d is %d\n", count, getList(list, count));
         }
         printf("\n");
      }
      else
      {
         printf("\nThe list is empty.\n");
      }
}

Contents of list:
Element 0 is 3


Contents of list:
```

```
Element 0 is 2
Element 1 is 3


Contents of list:
Element 0 is 1
Element 1 is 2
Element 2 is 3


Contents of list:
Element 0 is 1
Element 1 is 3


Contents of list:
Element 0 is 5
Element 1 is 1
Element 2 is 3

The list is empty.

All done
```

9.5 Testing Memwatch

`memwatch` includes a file `test.c` that can be used to perform more rigorous tests than the test shown above. (Note that you must change the last line in `test.c` to a comment.) Compile the files using the statement:

```
gcc -DMEMWATCH -DMEMWATCH_STDIO test.c memwatch.c -o testmemwatch
```

When `testmemwatch` is executed, it generates the following console output:

```
C:\Ch9>testmemwatch

MEMWATCH: assert trap: test.c(69), 1==2
MEMWATCH: Abort, Retry or Ignore? i
MEMWATCH detected 9 anomalies
```

According to the console output, `memwatch` detected 9 memory-management problems.

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Thu Sep 06 15:32:21 2007

Modes: __STDC__ 64-bit mwDWORD==(unsigned long)
mwROUNDALLOC==8 sizeof(mwData)==32 mwDataSize==32

statistics: now collecting on a line basis
Hello world!
underflow: <5> test.c(62), 200 bytes alloc'd at <4> test.c(60)
assert trap: <8> test.c(69), 1==2
assert trap: <8> IGNORED - execution continues
limit: old limit = none, new limit = 1000000 bytes
grabbed: all allowed memory to no-mans-land (976 kb)
Killing byte at 003DEB00
```

```
Killing byte at 003D9CE0
Killing byte at 003D2B98
Killing byte at 003D274C
check: <8> test.c(95), checking chain alloc nomansland
check: <8> test.c(95), complete; no errors
internal: <10> test.c(105), checksum for MW-0056BDC0 is incorrect
underflow: <10> test.c(105), 0 bytes alloc'd at <9> test.c(865)
overflow: <10> test.c(105), 0 bytes alloc'd at <9> test.c(865)
internal: <10> test.c(107), no-mans-land MW-0056BDC0 is corrupted
realloc: <10> test.c(107), 0056BDE8 was freed from test.c(105)
WILD free: <11> test.c(110), unknown pointer 004012FE

Stopped at Thu Sep 06 15:32:23 2007

wild pointer: <11> no-mans-land memory hit at 003DEB00
wild pointer: <11> no-mans-land memory hit at 003D9CE0
dropped: all no-mans-land memory (976 kb)
unfreed: <3> test.c(59), 20 bytes at 003D2610       {FE FE FE FE FE FE FE FE FE FE FE FE
FE FE FE ..............}

Memory usage statistics (global):
 N)umber of allocations made: 5
 L)argest memory usage      : 12020
 T)otal of all alloc() calls: 12530
 U)nfreed bytes totals      : 12020

Memory usage statistics (detailed):
 Module/Line                                Number   Largest  Total    Unfreed
 ‰åƒì□ƒäð¸                                  0        0        0        -100
  64                                        0        0        0        -100
 test.c                                     5        12120    12530    12120
  865                                       0        0        0        0
  97                                        1        12000    12000    12000
  64                                        1        100      100      100
  60                                        1        200      200      0
  59                                        1        20       20       20
  57                                        1        210      210      0
```

9.6 Summary

Identifying and eliminating memory allocation problems is one of the more difficult parts of programming in C. Using an appropriate tool to identify memory allocation errors makes life much easier.

10 INSTALLING AND USING A C COMPILER

10.1 Introduction

In this chapter, we examine how to use a C compiler in a Unix/Linux environment and then examine how to install and use a C compiler in a Windows environment.

10.2 Using Unix/Linux

We will begin by taking a quick look at using a Unix/Linux environment for developing C programs.

First, ensure that the gcc compiler is installed by typing the following command:

```
gcc –v
```

Depending on which system you are using, the output should look something like the following:

```
kingfisher.cs.umanitoba.ca 103% gcc -v
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-
checking=release --with-system-zlib --enable-__cxa_atexit --disable-libunwind-
exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-
c++,java,fortran,ada --enable-java-awt=gtk --disable-dssi --enable-plugin --with-
java-home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre --with-cpu=generic --host=x86_64-
redhat-linux
Thread model: posix
gcc version 4.1.2 20071124 (Red Hat 4.1.2-42)
kingfisher.cs.umanitoba.ca 104%
```

Now you can use your favourite text editor to write a C program such as Hello World.

Once you have created the file, you can compile it with the following statement:

```
gcc –ggdb main.c –o hello
```

Run the program by typing the name of the executable file:

```
hello
```

The expected output is generated:

```
Hello C World!
```

If the name of the output (executable) file is not specified in the compile command, the output

file is named `a.out`. This file is executed by typing `a.out`

That is about all that you need to run a C program under Unix/Linux. In the remainder of this chapter, we will take a look at installing a C compiler on a Windows machine.

There is a difference between the way that text files (including source code files) are stored in Unix/Linux and in Windows – Windows adds an extra character at the end of each line of text. Unix/Linux uses the newline character (`'\n'`) to terminate each line of a text file while Windows uses a carriage-return character (`'\r'`) followed by a newline character to terminate each line. The C run-time system removes the carriage-return character when it is followed by a newline character so only the newline character is passed to the processing program. When writing to an output file, the C run-time system adds a carriage-return character before the newline character on Windows systems. As a result, a source file can be developed on a Windows system and then transferred to Unix/Linux and manipulated in the Unix/Linux environment; similarly, a file developed on Unix/Linux can be transferred to Windows and manipulated in the Windows environment. The Windows program TextPad provides a convenient mechanism for adding or removing the extra end of line character – just use the Save As menu item and change the File Format in the drop-down menu. Alternatively, the commands `unix2dos` and `dos2unix` can be used to convert a file from one format to the other.

When switching operating systems, the file must be recompiled before it can be executed. In the Windows environment if the name of the executable file is not specified, the executable is named `a.exe` instead of `a.out`.

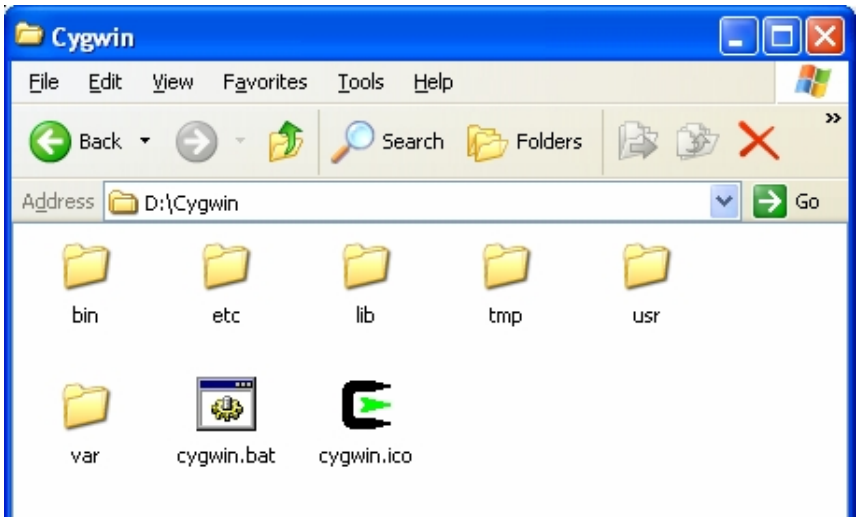
10.3 Installing a C Compiler

Before you can develop C projects, you must first install an appropriate C compiler on your system. There are several open-source compilers available for use on Windows, with most based on the Unix GCC project. We will take a quick look at 2 such systems. Both systems worked correctly with the examples that were tested.

10.3.1 Cygwin

The Cygwin (1.5.18.1) package is available at http://www.cygwin.com. When you install Cygwin, ensure that the GCC compiler tools are included. These tools are not included by default and you will have to check the packages to ensure that the tools are installed. (Expand the package DLevel and you should find the tools.) The Cygwin system requires approximately 945 MB of storage for a install that includes all of the DLevel tools. This can be reduced if you know exactly which tools you require.

Once installed, the directory structure should look something like the following:



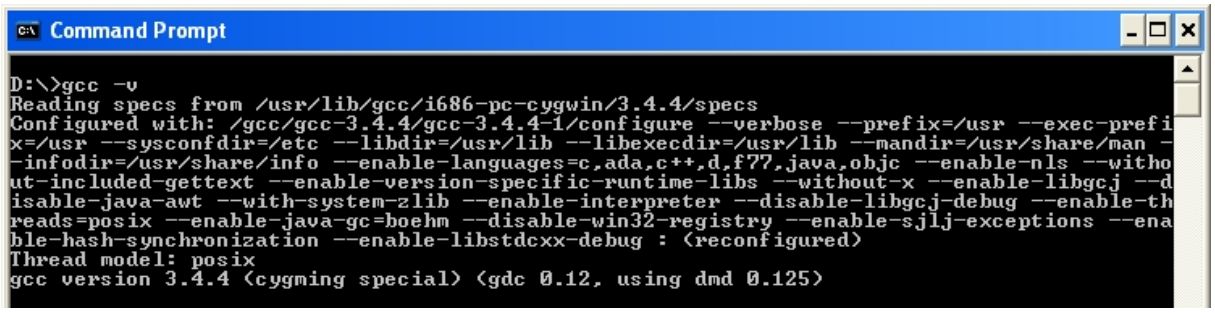
To ensure that the tools are available, first ensure that your "path" variable includes the path to the cygwin bin directory:

```
D:\cygwin\bin;
```

Test the path by typing the following command into a Windows command window:

```
gcc -v
```

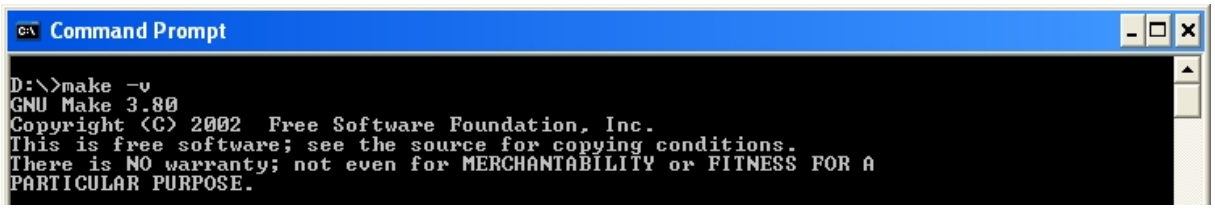
The output should be similar to the following:



Similarly, a "make" command must also be available on the system path.

```
make -v
```

The output should be similar to the following:



As long as gcc and make are available, you will be able to compile C programs.

10.3.2 MinGW

Alternatively, you can install MinGW (Minimalist GNU for Windows), a version of the gcc compiler tools. MinGW is available at http://www.mingw.org.

If you use the MinGW tools, a non-standard name for the make command is used to execute the script that compiles C programs. However, if you make a copy of the make file (named mingw32-make.exe in version 3.1), leave the copy in the same directory, and then rename the copy to "make.exe", this should eliminate the problem. The MinGW system requires approximately 46 MB of storage. Ensure that your "path" variable includes the path to the bin directory:

```
D:\mingw\gin;
```

Then test the path for the gcc compiler by typing gcc -v into a Windows command window:

Similarly, test the path for a "make" command by typing make –v into a Windows command window.

10.4 Compiling a C Source File

To compile a C source file, use the following statement:

```
gcc –ggdb main.c -o ch1
```

In this example, the file main.c is to be compiled and the output is the executable file ch1.exe. The names of both files were chosen for convenience, not for any particular reason.

One thing to be aware of when using Cygwin is that the executable (.exe file) built by Cygwin **requires** the dll cygwin1.dll. As long as Cygwin\bin is on the system path, this is not a problem but if the executable is to be run on other machines, then the dll must be available on those machines.

10.4.1 Blocking the Execution of a C Program

Any results written to the system output (console) window remain visible only while the program is active. The following instructions can be added to the end of a C program to cause the program to pause before it terminates so that any output directed to the console can be examined. The statement fflush(stdout) ensures that all output written to stdout has actually been written to the console. (This statement is not always required, it depends on the

type of system console that is being used. However, it will not cause any problems if it is included.) The `fgetc(stdin)` statement then reads one character from the console.

```
#include <stdio.h>

int main(int numParms, char *parms[])
{
     printf("Hello C World!!\n");
     fflush(stdout);
     fgetc(stdin);
     return 0;
}
```

When the program runs, the program "blocks" until the user types "enter" on the console.

An alternative is to run the program from within a Windows command window.

The other approach is create a Windows batch file that executes the program and then includes a Windows "`pause`" statement to suspend the execution of the batch file. Then, when the batch file is executed, the program will pause at the end so that the contents of the console window can be viewed.

10.5 Make Files

If there are multiple files that have to be compiled, it may be easier to group the commands into a script called a makefile and execute the script using the make command. The following is a simple make file that compiles main.c and creates the executable file `ch1.exe`. It is important to note that the lines that are indented contain a tab character at the beginning of the line, not just a bunch of spaces.

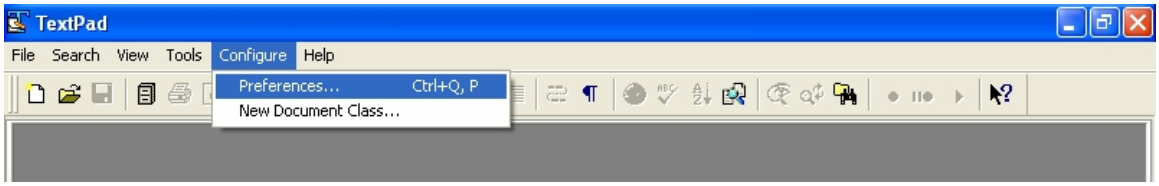
```
ch1 : main.o
     gcc -o ch1 main.o
main.o : main.c
     gcc -ggdb -c main.c
all :
     ${MAKE} ch1
clean :
     -del main.o
```

Once a standard makefile has been created, it can be executed using the statement:

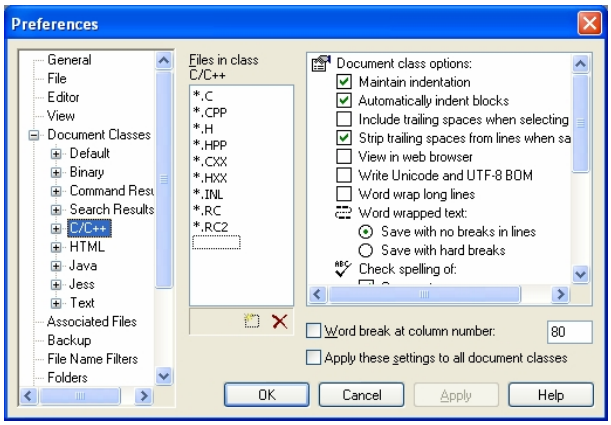
```
make –k all
```

10.6 Using the C Compiler with TextPad

In this chapter, we describe how to use a C compiler with TextPad on a Windows computer.

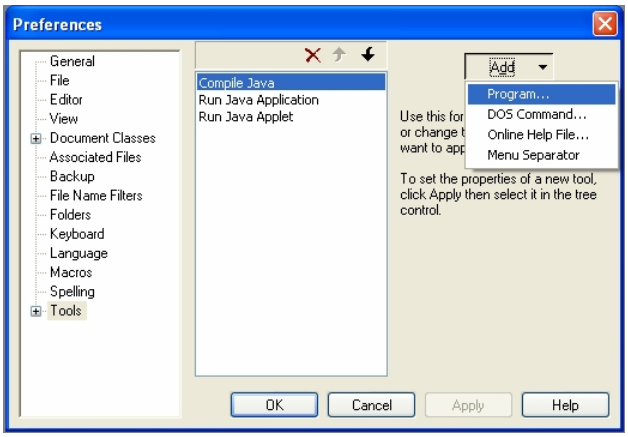


Normally, the document class for C has already been defined in TextPad, as shown in the diagram below.

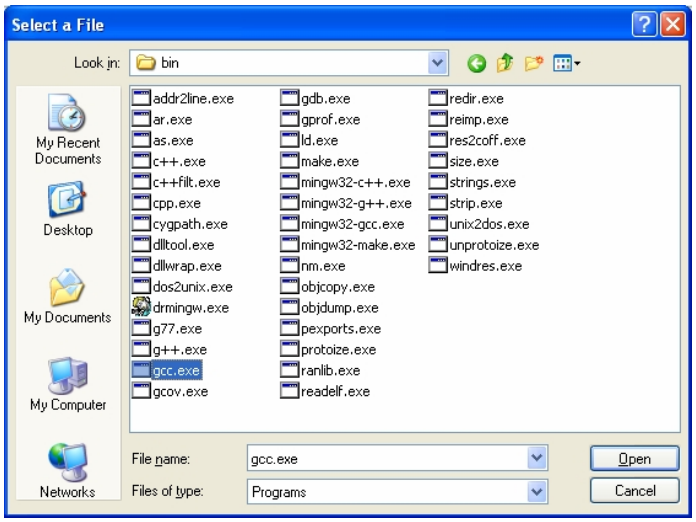


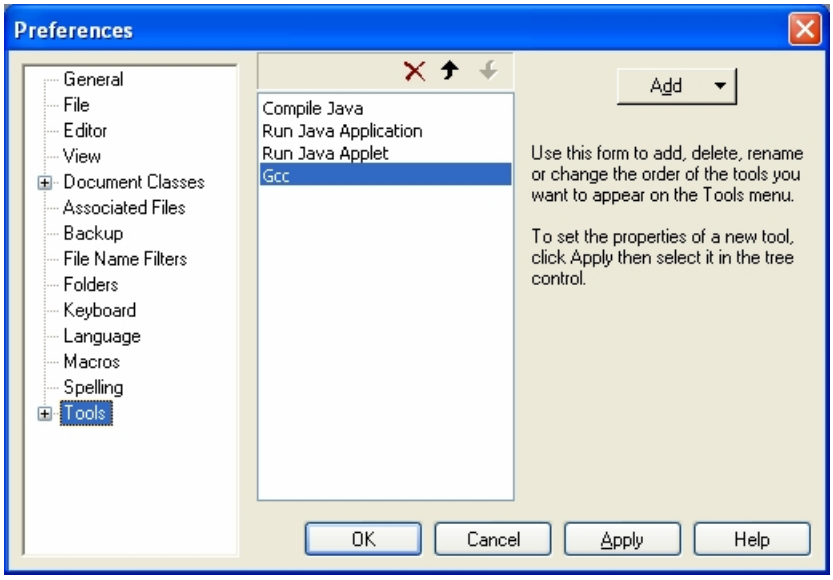
If they have not been defined, see http://www.textpad.com/add-ons/syna2g.html for information on creating the necessary document class.

Next, the tools that compile a C program and execute a C program must be defined. Select Preferences **à** Tools and then click on the Add button to select Program.

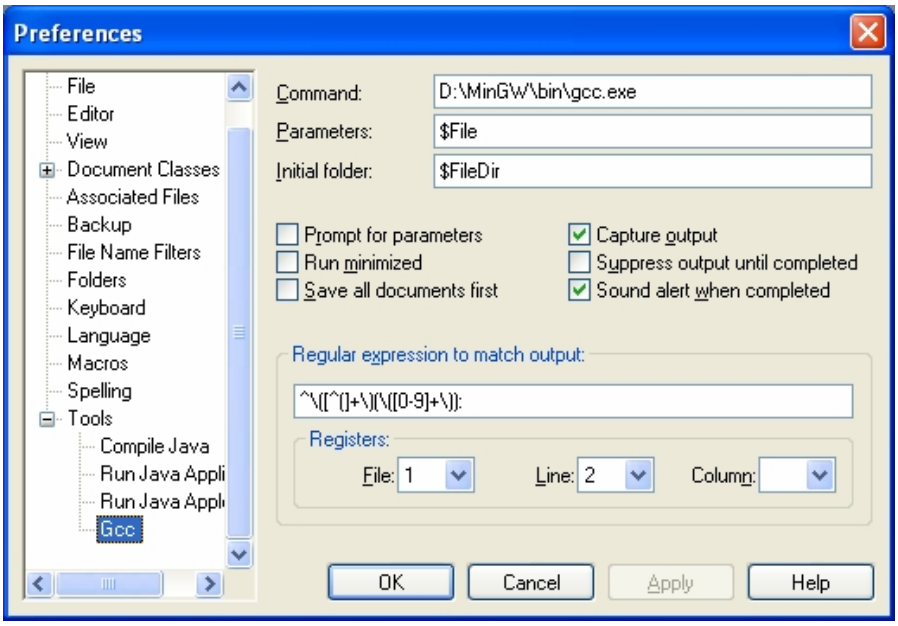


Navigate to your C compiler and select the compile command (gcc in this example).





Click Apply.



Enter the following into Parameters:

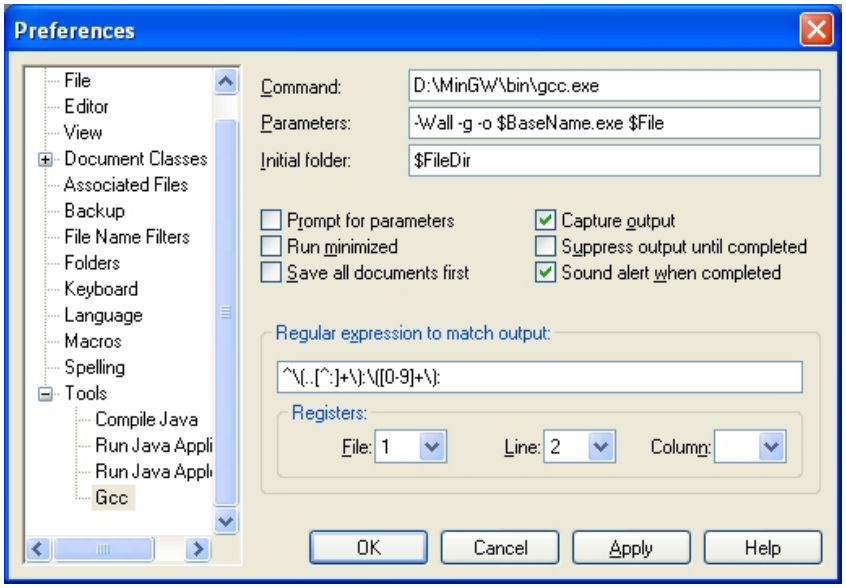
```
-Wall -g -o $BaseName.exe $File
```

Enter the following into Initial folder:

```
$FileDir
```

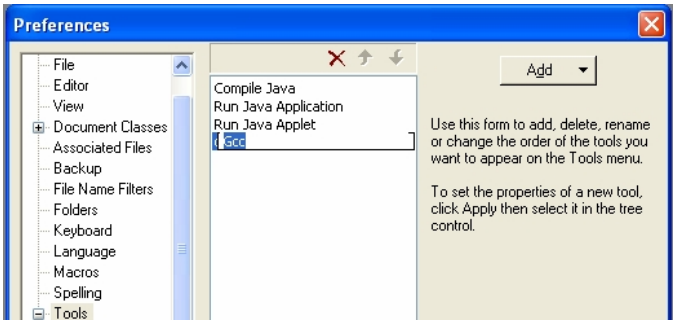
Enter the following into regular expression into the “match output” box:

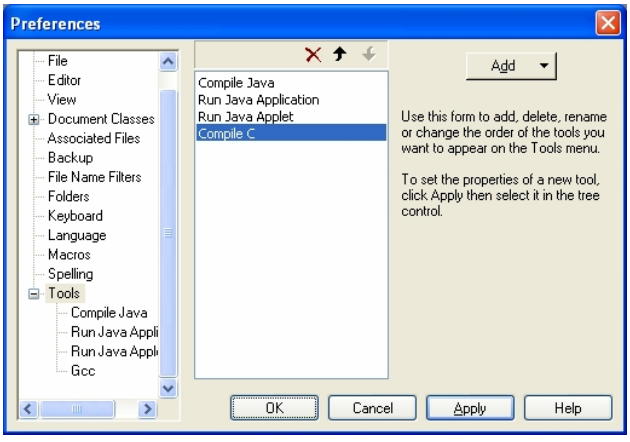
```
^\(..[^:]+\):\([0-9]+\):
```



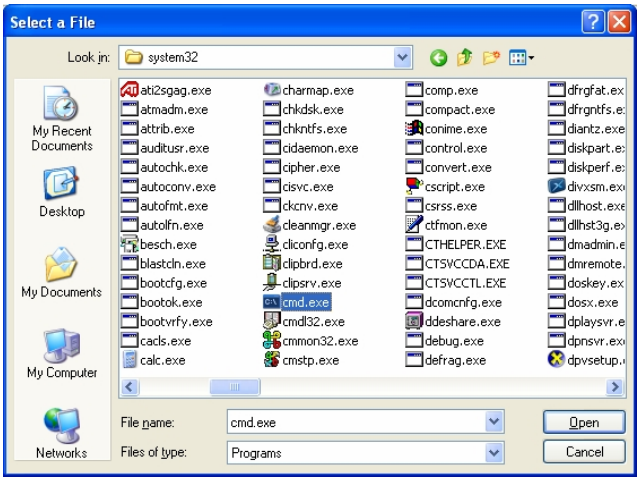
Click Apply.

It is a good idea to rename the tool to something that is more meaningful than Gcc. Double click (sometimes it takes several double clicks) on the tool name and then type the new name for the tool.

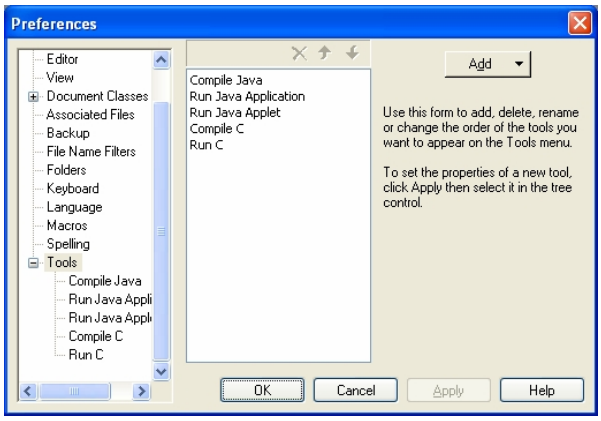




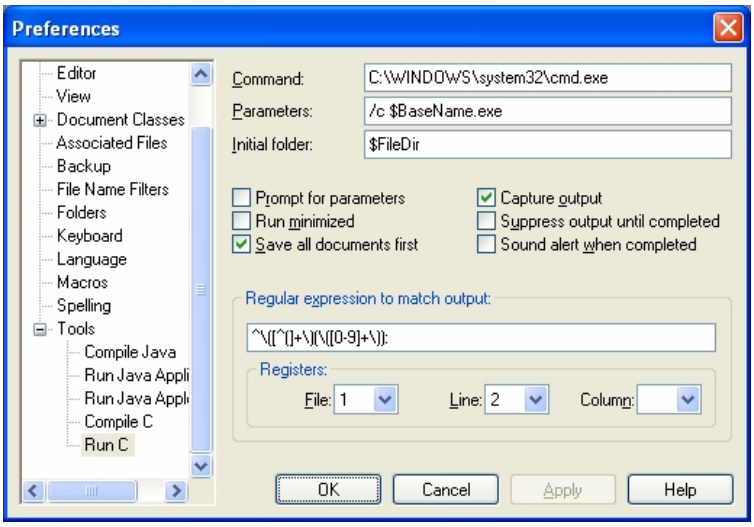
Repeat the process above to create a tool to execute the C program.



Again, rename the tool to something that is more meaningful.



The tool should be given the following parameters.

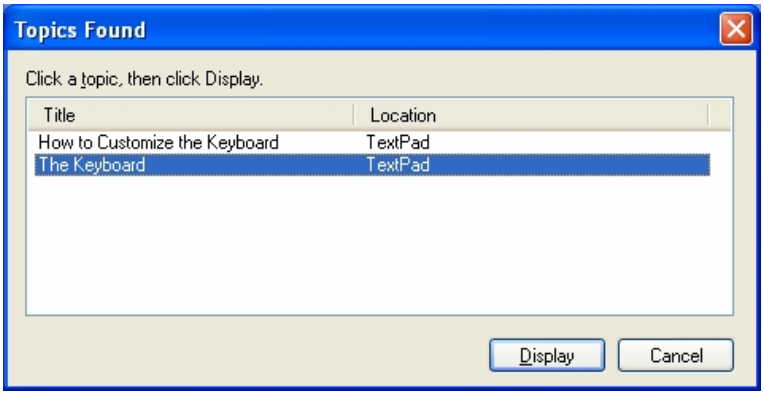


10.7 TextPad Shortcuts

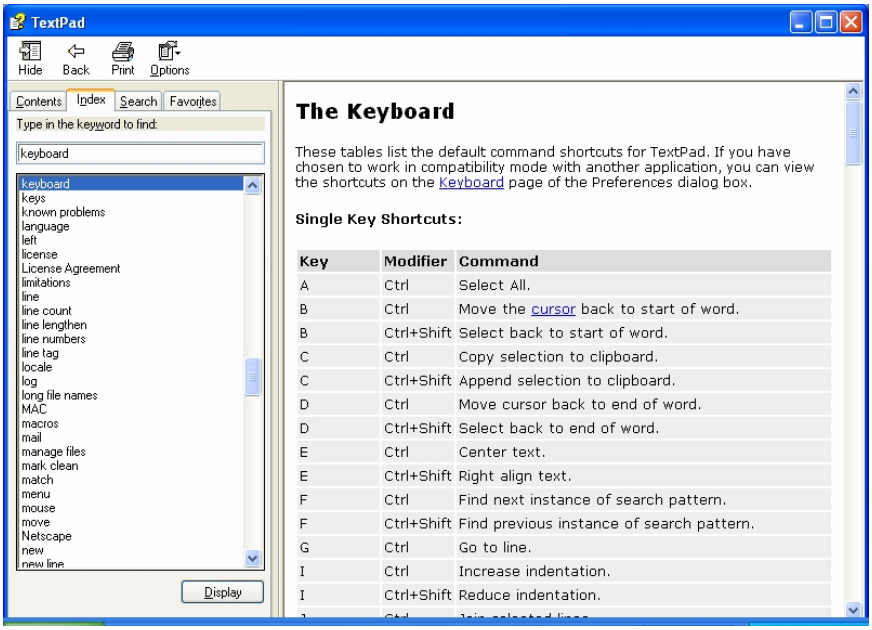
There are a few shortcuts that make life with TextPad easier.

Ctrl-F4	close current window
Ctrl-Tab	switch to next window
Ctrl-shift-Tab	switch to previous window
Ctrl-m	find matching bracket: { }, ( ), [ ], < >

To view the list of shortcuts, select Help **à** Help Topics **à** Index **à** Keyboard

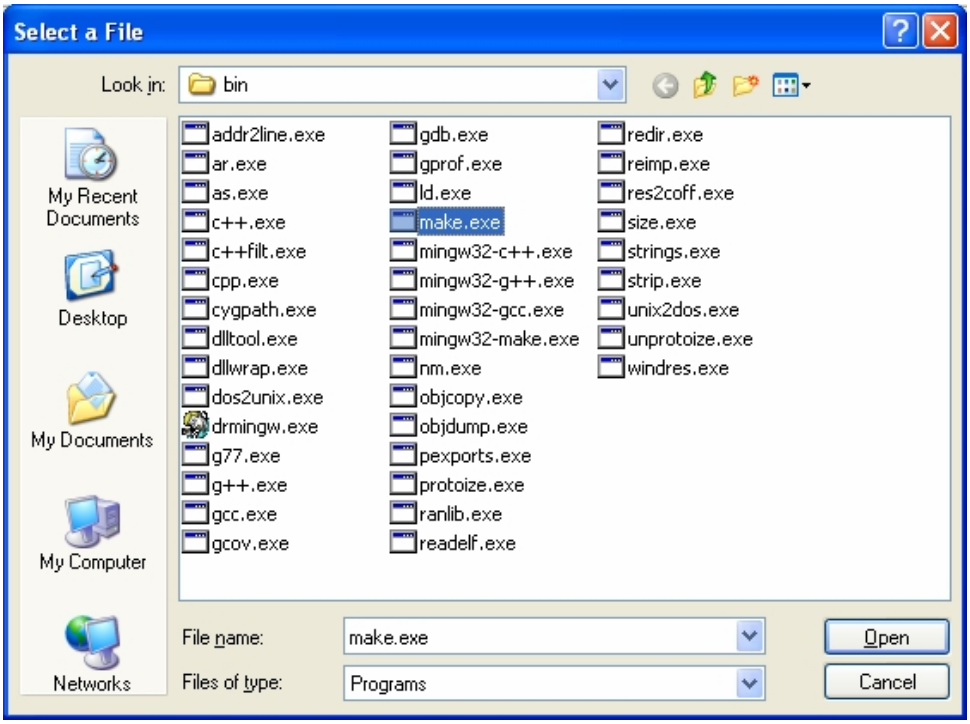


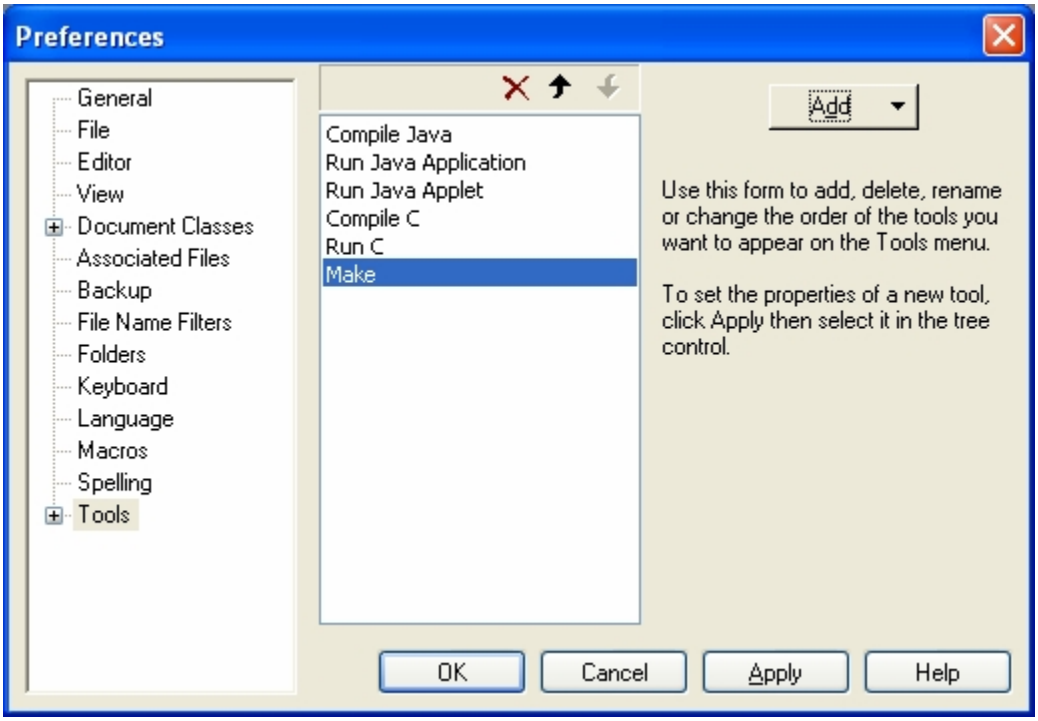
Then select The Keyboard and click Display. (Note that there is also a help topic on customizing the keyboard.)



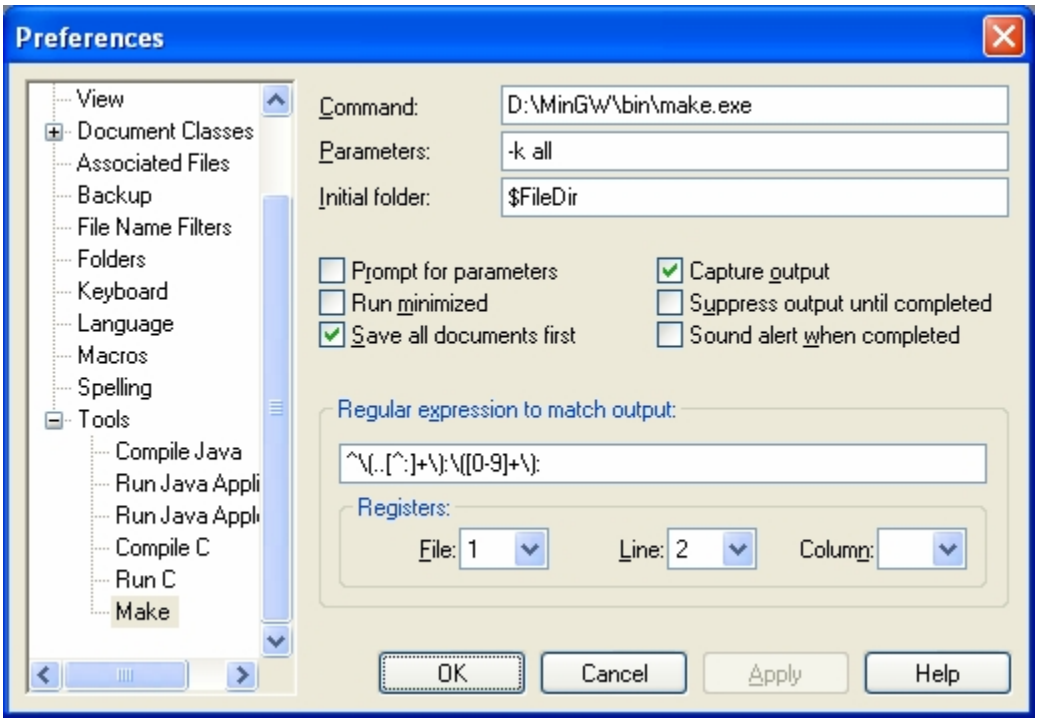
10.8 Using The Make Command

You may also execute the `make` command from within TextPad.

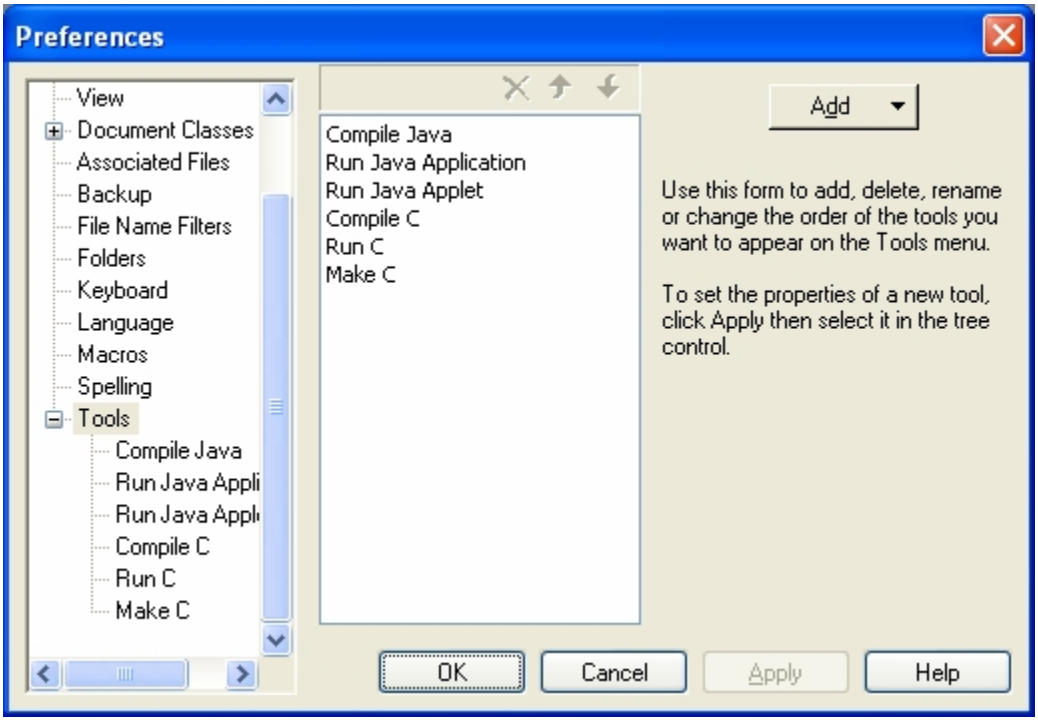




Fill in the following parameters; the regular expression is: `^\(..[^:]+\):\([0-9]+\):`



And rename the tool to “Make C”.



10.9 Using an Integrated Development Environment

Instead of using a source-code editor and a command window in a Windows or Unix/Linux environment, you could use an integrated development environment (IDE). An IDE typically provides the development tools (such as the compiler, the debugger, etc.) in one package.

The Dev-C++ IDE can be found at:

`http://www.bloodshed.net/devcpp.html`

Alternatively, the Eclipse IDE provides an excellent environment for developing not only C programs but also Java programs and a wide range of programs in other languages. A quick-start guide to using Eclipse can be found at:

`http://www.cs.umanitoba.ca/~eclipse/Eclipse3-1.pdf`

Eclipse 3.4 (Ganymede) was recently released (June 24, 2008). This release likely contains many improvements that are not reflected in the notes identified above.

10.10 Summary

Since most C compilers support the ANSI standard, it doesn't really matter which compiler is used. Using an integrated development environment will tend to make life easier once you have finished the initial learning curve.

INDEX