The C Cheat Sheet

An Introduction to Programming in C

Revision 1.0
September 2000

Andrew Sterian
Padnos School of Engineering
Grand Valley State University

TABLE OF CONTENTS

PREFACE

This document is an introduction to the C programming language. Unlike a thorough reference manual this document is limited in scope. The main goal is to provide a roadmap that can answer basic questions about the language, such as what data types are supported or what a `for` loop looks like. The beginning C programmer can use this document to get started with the language and write small-to-medium-size programs involving simple I/O, file manipulation, and arithmetic computations.

This document relies heavily on examples to teach C. Examples allow the reader to quickly grasp the concept being presented but do not allow for a thorough explanation. This is consistent with the philosophy that this document is an entry point to the language, not a reference text. When questions arise that cannot be answered here, the reader is directed to three very useful resources:

- The on-line documentation that comes with the reader's C compiler. This reference documentation thoroughly describes both the language structure and the library components.

- The classic textbook "The C Programming Language", 2nd edition, by Kernighan & Ritchie. Written by the architects of the C language, this text was published in 1988 but has endured as both a reference and as a tutorial.

- The more recent text "C: A Reference Manual", 4th edition, by Harbison & Steele. This text, as its name implies, is mostly a reference, not a tutorial, but it includes some of the latest changes in the language standard.

Finally, note that C, like spoken languages, does evolve with time. Even though it is the most mature of the three major system development languages currently in favor (C, C++, and Java) international standards committees continue to try to improve its character. For example, issues of internationalization have led to improved support for multi-byte character strings, a concept not formally part of the C language. A new proposal for the C standard has been submitted by the ISO as of December 1999. Your compiler's documentation is the last word on compliance with the most recent standards.

It is assumed that the reader is familiar with programming concepts in general and may also be familiar with the Java programming language. The core Java language design was heavily influenced by the C language and the "look and feel" of a C program at the syntactic level will be quite familiar to Java programmers. Differences in the two languages are highlighted throughout this document (also see the summary in Section 11.0).

1.0 Introduction

A C program may occupy as few as 5 lines in a single file. Here is the venerable "Hello world!" program in the file "`hello.c`"[1]:

```
#include <stdio.h>

void main(void) {
      printf("Hello world!\n");
}
```

This text file (known as the *source code*) is compiled to an executable file using a *C compiler*. For example, using the Borland "BCC32" program:

```
bcc32 -ehello.exe hello.c
```

Running the "`hello.exe`" program prints "Hello world!" on the screen (in a console window).

Java programmers may recognize the `main()` method but note that it is not embedded within a class. C does not have classes. All methods (simply known as *functions*) are written at file scope.

1.1 The `main()` Function

The `main()` function is the starting point of the program. All C programs must have a `main()` function. While this function can be written as in the example above (but see footnote 1), it is most often written with the following *prototype* (or *signature*):

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

While we may not quite understand all of the above, there are a few points to note:

- The return type of the `main()` function is an integer (type `int`). The value returned by the `main()` function is known as the *return value* of the program. Traditionally, a return value of 0 indicates that the program executed successfully while a non-zero value indicates an error condition. In many cases, we are not concerned with this return value and it is simply ignored.

- The parameters of the `main()` function (`argc` and `argv`) allow the C program to process command-line parameters. We will discuss this further in Section 7.6 after we have introduced character strings.

1. One C practitioner has declared that the author of any document which writes '`void main(void)`' "...doesn't know or can't be bothered with learning, using, and writing about the actual languages defined by their respective International Standards. It is very likely that the author will make other subtle and not-so-subtle errors in the book." Indeed, the C standard says that the `main` function must always return an integer. But this is not a document describing a standard; it's simply here to kick-start the learning process. Nonetheless, be on the lookout for subtle and not-so-subtle errors!

The `main()` method in Java has the prototype `'main(String[] args)'` which provides the program with an array of strings containing the command-line parameters. In C, an array does not know its own length so an extra parameter (`argc`) is present to indicate the number of entries in the `argv` array.

1.2 Include Files

The first line of our example program:

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

inserts the contents of a file (in this case, a file named `stdio.h`) into the current file, just as if you had cut and pasted the contents of that file into your source code. The purpose of these files (known as *include files* or *header files*) is to tell the compiler about the existence of external functions which the source code will make use of.

In this case, the file `stdio.h` defines the function `printf()` which we use to print text to the screen. Without including this file, the compiler would generate an error when it encountered the `printf()` function in the source code since this function is not a part of the core language.

The `stdio.h` file defines many other functions, all related to the "Standard I/O" component of the *standard C library*. We will discuss standard I/O in more detail in Section 7.1 and the concept of libraries in Section 8.3. While the standard C library is not part of the core C language, it is distributed with the C compiler (along with many other libraries) and is actually a part of the C language specification.

Java users may see the similarity between the `#include` statement and Java's `import` statement. Both serve the same purpose, to "pull in" external components for use by the given program.

1.3 Whitespace

The C language generally ignores whitespace (i.e., spaces, blank lines, tabs, etc.). The "Hello world!" program could have been written more succinctly (and illegibly) as:

```
#include <stdio.h>
void main(void){printf("Hello world!\n");}
```

The only cases where whitespace is significant are in preprocessor statements (such as the `#include` statement; see Section 1.4 below for more details on the preprocessor) and within character strings, like "`Hello world!\n`".

1.4 The Preprocessor

When a line in a C program begins with the octothorpe character '#', it indicates that this line is a *preprocessor directive*. The preprocessor is actually a program that runs before the C compiler itself. It serves to perform text substitutions on the source code prior to the actual compilation.

Java does not have a preprocessing step. It can be argued (quite justifiably) that Java's language features obviate the need for any text substitutions. With C however, the preprocessor can often provide useful program development support.

1.4.1 The `#define` Directive

The simplest form of preprocessor directive is the `#define` statement. As an example, consider the following program:

```
#define PI 3.1415926535

void main(void) {
      printf("The constant pi is %g\n", PI);
}
```

As the first step in the compilation process, the preprocessor looks for all occurrences of the token "`PI`" and replaces it with the token "`3.1415926535`". What the actual C compiler sees, then, is the following:

```
void main(void) {
      printf("The constant pi is %g\n", 3.1415926535);
}
```

Note that the preprocessor has simply substituted the text "`3.1415926535`" for the text "`PI`". The `#define` statement initiated this substitution. Also note that the actual line with the `#define` directive has been removed (and interpreted) by the preprocessor. The C compiler does not see these directive lines.

The `#define` directive can also perform rudimentary macro substitutions. Again, an example:

```
#define HALFOF(x) x/2

void main(void) {
      printf("Half of 10 is %d\n", HALFOF(10));
}
```

The actual C compiler sees the following source code after preprocessing:

```
void main(void) {
      printf("Half of 10 is %d\n", 10/2);
}
```

A very common use of macro substitutions is in the definition of "pseudo-functions". For example, here are some ways to compute the maximum, minimum, absolute value, and signum functions (see Section 4.8 for a description of the ternary operator '`?:`'):

```
#define max(x,y) ((x) > (y) ? (x) : (y))
#define min(x,y) ((x) < (y) ? (x) : (y))
#define abs(x) ((x) >= 0 ? (x) : -(x))
#define sgn(x) ((x) > 0 ? 1 : ((x) < 0 ? -1 : 0))
```

Macro definitions of this form can become quite unwieldy, which is most likely why the preprocessor has not survived to more recent languages like Java where better (and safer) solutions exist.

The heavy use of parentheses in the above macros is to prevent unintended side effects. This is best illustrated with an example:

```
#define SQUARED(x) x*x

void main(void) {
      int i = 5;
      printf("i+2 squared is %d\n", SQUARED(i+2));
}
```

The intention here is to print the square of `i+2` (which should be 5+2 squared, or 49). But the compiler sees the following:

```
void main(void) {
      int i = 5;
      printf("i+2 squared is %d\n", i+2*i+2);
}
```

which will print 5+2*5+2 or 17, following proper order of operations. Adding those nuisance parentheses fixes things, however:

```
#define SQUARED(x) (x)*(x)
```

now expands `SQUARED(i+2)` as `(i+2)*(i+2)` and the program works correctly.

1.4.2 Comments

The preprocessor removes all text within the comment delimiters `/*` and `*/`, just as in Java. For example:

```
/*
      This comment is removed by the preprocessor
      and is not seen by the actual compiler.
*/
void main(void) {
      printf("Hi!\n");  /* Another comment! */
}
```

While not part of the original C language definition, single-line comments introduced by `//` are supported by most modern C compilers, as with Java.

1.4.3 The `#include` Directive

We have already seen that the purpose of the `#include` directive is to insert another file into the file to be compiled. Again, it is as if you were to copy and paste the contents of the include file into the source code.

You may actually encounter two forms of the #include directive. The first has already been encountered:

```
#include    <somefile.h>
```

The second form uses quotation marks instead of angle brackets:

```
#include    “somefile.h”
```

The difference in these two forms lies in the *include file search path*. This is a list of directories that the preprocessor searches in order to find the include file. This search path should be configured when you install the compiler so you shouldn’t generally have to worry about it.

With the second form of the `#include` directive (using quotation marks), the preprocessor first looks in the same directory as the source file. If the include file is not found there, then the search path is considered.

Practically, the angle brackets form (e.g., `<stdio.h>`) is used to include system files that are part of the C compiler system, while the quotation marks form (e.g., `“mystuff.h”`) is used to include files that you write yourself and are part of your project.

1.4.4 The `#ifdef`/`#else`/`#endif` Directives

There are several directives that can be used to effect *conditional compilation*. Most often, conditional compilation is used either as a debugging tool or to compile different code for different hardware architectures. Here is an example that shows how to add debugging statements that can easily be turned on or off.

```
#include <stdio.h>

#define DEBUG 1

void main(void) {
#ifdef DEBUG
      printf(“Debugging is enabled.\n”);
#else
      printf(“Debugging is disabled.\n”);
#endif
}
```

The `#ifdef` directive tells the preprocessor to pass the subsequent lines to the compiler only if the item following `#ifdef` has been defined using a `#define` directive.

In this example, the token DEBUG has been defined to be 1, so the `#ifdef` statement succeeds. What the C compiler sees, then, is the following:

```
void main(void) {
      printf(“Debugging is enabled.\n”);
}
```

Everything following the `#else` directive is not passed to the compiler (up to the `#endif` directive).

It is easy now to switch between debugging mode and non-debugging mode. Simply comment out the line that defines DEBUG:

```
//#define DEBUG 1
```

Now, everything between `#ifdef` and `#else` is ignored, and the C compiler only sees the code between `#else` and `#endif`.

1.4.5 More Directives

There are other, less frequently used directives that you may want to investigate in a thorough C reference manual:

- `#undef` un-defines a definition created with `#define`.
- `#if` is a more general form of `#ifdef` that allows for expressions to be evaluated rather than simply testing for a label being defined or undefined.
- `#error` generates an error during compilation. It can be used to indicate that some piece of code is being compiled when it should not be.
- `#elif` can be used to augment `#ifdef`/`#else`/`#endif` to have multiple clauses in the test.
- `#pragma` is a compiler-specific directive that can be used to communicate with the compiler to, for example, enable or disable certain optimizations or warning messages.

1.4.6 Predefined Macros

The preprocessor defines some constants automatically, as if they had been defined using `#define` directives. These include the constants `__FILE__` which is the name of the source file, `__LINE__` which is the number of the current line being compiled, `__DATE__` which is the current date, and `__TIME__` which is the time that the source file was compiled. These constants can aid in debugging or in adding some version information to your program (for example, date and time of last compilation).

Here, for example, is the definition of a macro that can aid in debugging:

```
#define TRACE { printf("Executing %s line %d\n", __FILE__, __LINE__); }
```

A simple way to trace through a program's flow of execution is to sprinkle some TRACE statements at key points:

```
void func(void) {
      TRACE
      dosomething();
      TRACE
      ...
}
```

Compiler vendors generally define their own predefined constants to augment the above. For example, the BCC32 compiler always defines the constant `__BORLANDC__` to indicate the compiler vendor. Programs can use these additional constants to tailor the code based upon a specific vendor's implementation or other conditions.

2.0 Basic Data Types

C supports the following a wide variety of built-in data types. Surprisingly, more modern languages like Java have *fewer* data types. The reason is that the more data types there are, the more rules there are regarding how they mix and thus the higher the possibility of confusion. You will find that of the 8 data types in C that can store integers, you will most likely only use two on a regular basis.

2.1 Signed Integer Types

The four data types that can be used to represent integers are (from smallest to biggest):

- `char`
- `short`
- `int`
- `long`

The `char` type, as its name implies, generally stores 1-byte (8-bit) integers which can represent ASCII characters. Even though character storage is the most common use for this type, do not forget that essentially this is an integer type.

The `short`, `int`, and `long` types similarly hold integers but of varying bit widths. Most commonly, the `short` type stores 16-bit integers (2 bytes). The `int` type stores either 16-bit or 32-bit integers, and the `long` type stores anywhere from 16-bit to 64-bit integers.

The fact that the actual bit widths that correspond to these types is ***not*** specified in the original C language standard has been a vexing problem for C programmers. Each compiler vendor is free to choose whatever bit width they want. Recent enhancements to the C standard have addressed this problem by defininig known-width types such as `int16_t` which indicates a 16-bit integer.

Usually, however, the `int` type corresponds to the "native" size of an integer of the underlying computer. The early IBM PC machines (80x86 architectures) were 16-bit machines hence the compilers defined the `int` type to be 16-bit integers. With the advent of 32-bit computing (for example, the Pentium architectures) modern compilers for the PC architecture define the `int` type to be 32-bit integers. In both 16-bit and 32-bit architectures, however, compilers generally defined `long` to be 32-bit integers and `short` to be 16-bit integers.

This type of confusion has been fixed in languages like Java where the data type is strictly defined to represent a certain number of bits. In C, you must read the documentation of your vendor's C compiler to determine how many bits are represented by each type (see Section 2.5 for the details of the BCC32 compiler).

2.2 Unsigned Integer Types

All of the four types described above may have the keyword `unsigned` prepended to form integer types of the same bit width but not allowing negative numbers:

- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long`

The difference between the signed and unsigned integer types is simply whether or not we are applying the two's complement convention. With the signed types, the bits in an integer are interpreted as a two's complement number (i.e., the uppermost bit is the sign bit, etc.) With the unsigned types, the bits in an integer simply represent magnitude and there is no sign bit.

Note that Java has no unsigned types.

Unsigned types can store larger numbers (in magnitude) but, clearly, cannot represent negative numbers. For example, counting how many times a key is pressed is naturally a good fit for an unsigned data type as it makes no sense for this to be a negative quantity.

Practically, we recommend you avoid unsigned numbers unless the expanded range is truly required. Mixing signed and unsigned numbers (unavoidable when dealing with the standard C library) leads to several potential sources of errors, or at the very least lots of warning messages.

2.3 Integer Overflow

It is not an error to exceed the number of bits in an integer type as a result of some operation. The variable will simply "roll over" or wrap around to the other end of its range. For example:

```
#include <stdio.h>

void main(void) {
      unsigned val = 65530;   // maximum unsigned is 65535
      printf("val+10 is %u\n", val+10);
}
```

This code will print the value 4, since 65530+10=65540 which is too big to fit into 16 bits. All bits beyond 16 are discarded and what is left is the number 4 ($65540-2^{16}$) in the lowest 16 bits.

The same principle holds for signed integer types:

```
#include <stdio.h>

void main(void) {
      int val = 32760;  // maximum signed is 32767
      printf("val+10 is %d\n", val+10);
}
```

This code will print -32766. Note that 32760+10=32770. If you subtract 2^{16} from this quantity you obtain the wrapped-around result, -32766.

Java programmers familiar with the for loop may find it surprising that the following code is wrong:

```
#include <stdio.h>

void main(void) {
      unsigned i;

      for (i=10; i >= 0; i--) {
            printf("i is %u\n", i);
      }
}
```

While we expect this code to print the numbers descending from 10 through 0, in fact this code will print an endless stream of numbers. The problem is that the variable `i` is of `unsigned` type and unsigned types are always greater-than-or-equal-to 0. Thus, the termination condition of the for-loop (`i >= 0`) is always true. The iteration statement `i--` takes the value of `i` from 0 to 65535 rather than -1, as expected.

These forms of subtle complexities and pitfalls are most likely the reason that Java has no unsigned types. Note, however, that good compilers (including BCC32) will issue a warning indicating that the termination condition of the for-loop is always true. **MORAL**: Do not ignore compiler warnings.

2.4 Real Data Types

C has three data types that can represent real numbers:

- `float`
- `double`
- `long double`

The `float` type is used to represent single-precision numbers while the `double` type represents double-precision numbers. The `long double` type is used for even more precision. Typically, the `float` type occupies 32 bits, the `double` type occupies 64 bits, and the `long double` type occupies 80 bits, but again these bit widths are not specified in the C language standard.

2.5 BCC32 Implementation

For the BCC32 compiler, the actual sizes of the basic data types along with their allowable ranges are shown in Table 1 below.

2.6 The `void` Type

We have seen function declarations of the form:

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

Table 1: Basic Data Types for BCC32

Data Type	Bit Width	Minimum Value	Maximum Value
`char`	8	-127	127
`short`	16	-32768	32767
`int`	32	-2^{31}	$2^{31}-1$
`long`	32	-2^{31}	$2^{31}-1$
`unsigned char`	8	0	255
`unsigned short`	16	0	65535
`unsigned int`	32	0	$2^{32}-1$
`unsigned long`	32	0	$2^{32}-1$
`float`	32	-3.4×10^{38}	3.4×10^{38}
`double`	64	-1.7×10^{308}	1.7×10^{308}
`long double`	80	-1.1×10^{4932}	1.1×10^{4932}

The keyword `void` instead of a data type indicates “no type”, i.e., nothing there. Thus, a function declared as returning a `void` type in fact does not return any value. Similarly, when the argument list of a function is simply declared as `void` (as with the example above) then the function does not take any arguments.

Variables cannot be declared to be of `void` type (although pointers to `void` type are useful, but we defer this discussion until Section 6.3).

3.0 Control Flow

The structures that C implements for directing the flow of a program are:

- Selection (using `if/else/endif` and `switch/case`)
- Iteration (using `for`, `while`, and `do/while`)
- Direct (using `goto`, `break`, `continue`, `return`)

3.1 The `if/else/endif` Statements

The `if` statement only executes code if a given expression is true. For example:

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

```
void main(int argc, char *argv[]) {
      if (argc > 1) {
            printf("You passed some command-line parameters.\n");
      } else {
            printf("You passed no command-line parameters.\n");
      }
}
```

Immediately following the `if` statement must come an expression enclosed in parentheses. If the expression evaluates to true (i.e., a non-zero value) then the code following the `if` (up to the `else`) is executed, otherwise the code following the `else` is executed.

Note that the `else`-clause is optional. We can simply write, for example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      if (argc > 1) {
            printf("You passed some command-line parameters.\n");
      }
}
```

3.2 Compound Statements

The use of curly brackets above introduces the important notion of a *compound statement*. The actual syntax of the if statement is:

```
if (expression) statement;
```

That is, only one statement is allowed to follow. However, a compound statement enclosed in curly brackets is equivalent to a single statement. Thus, we are allowed to write:

```
if (expression) {
      statement;
      statement;
      statement;
}
```

Thus, the above example could have been written more concisely as:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      if (argc > 1)
            printf("You passed some command-line parameters.\n");
}
```

Only one statement followed the `if` thus we do not really need the curly brackets.

Take note, however: it is ***strongly recommended*** that you always use curly brackets, even when you only write one statement. Why? Imagine that you wanted to augment the program above to also indicate how many command-line parameters were passed. The following would be a common way for a new student of C to proceed:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      if (argc > 1)
            printf("You passed some command-line parameters.\n");
            printf("In fact, you passed %d parameters\n", argc-1);
}
```

The above code is not what was intended. Adding an extra statement requires that we include curly brackets now to create a compound statement. The correct code is:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      if (argc > 1) {
            printf("You passed some command-line parameters.\n");
            printf("In fact, you passed %d parameters\n", argc-1);
      }
}
```

Had those curly brackets been in the program already, the error would not have been introduced. This is an example of *defensive programming style*. Assume you are going to make errors in writing the code and prevent them. Always writing compound statements even when not necessary is a good thing to do. This will be even more important when writing `else-if` statements (see below).

3.3 Nested `if` Statements

It is possible to choose from several blocks of code using nested `if` statements. For example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      if (argc < 2) {
            printf("You need to enter at least 1 parameter\n");
      } else if (argc < 4) {
            printf("Everything is OK.\n");
      } else {
            printf("You entered too many parameters.\n");
      }
}
```

This is a "nested" `if` statement because there is no true `else-if` statement in C. The else-if part is actually another `if` statement nested within the `else` clause of the first. More obvious indentation and extra curly brackets will highlight this:

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

```
void main(int argc, char *argv[]) {
      if (argc < 2) {
            printf("You need to enter at least 1 parameter\n");
      } else {
            if (argc < 4) {
                  printf("Everything is OK.\n");
            } else {
                  printf("You entered too many parameters.\n");
            }
      }
}
```

This is one situation (and perhaps the only situation) where the extra curly brackets aren't really needed and just clutter up the program. The first form of the example is preferred for clarity.

3.4 The `switch/case` Statement

The `switch/case` statement is essentially equivalent to `if/else/endif` but with a few twists. Here is an example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      switch (argc) {
      case 0:
      case 1:
            printf("You need to enter at least 1 parameter\n");
            break;

      case 2:
      case 3:
            printf("Everything is OK\n");
            break;

      default:
            printf("You entered too many parameters.\n");
            break;
      }
}
```

The expression following the `switch` statement is evaluated (it must be an integer-valued expression) and the subsequent `case` statements are examined for a match. As shown in the example, multiple `case` statements may lead to the same code. The `default` clause (which is optional) is taken if none of the `case` statements match.

Why are all those `break` statements there? This is one of the twists of the `switch/case` statement. The `break` statement skips past all other cases, past the end of the whole `switch/case` code. Without a `break` statement in a `case`-clause, the program begins to execute the next `case`-clause! For example:

```
#include <stdio.h>

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

```
        switch (argc) {
        default:
              printf("argc is bigger than 1\n");
              // No "break" so fall through to next case!
        case 1:
              printf("argc is bigger than 0\n");
        }
}
```

With `argc>1`, ***both*** `printf` statements will be executed. It is good practice when taking advantage of this behavior to include a comment indicating that this is the intention, rather than making the reader wonder whether or not you simply forgot to include a `break` statement.

Many programmers claim that `switch/case` is unnecessary since `if/else/endif` has the same capabilities, plus dynamic expressions are allowed in the "case-clauses" rather than fixed integers (as required in `switch/case` clauses). It is possible, for example, to write:

```
if (argc >= 100 && argc <= 200)
```

but using `switch/case`, we would need to enumerate all 100 possible case statements:

```
case 100:
case 101:
case 102:   (etc.)
```

There is, however, one subtle benefit to `switch/case` in certain cases. Because the case-clauses are keyed to fixed integers, this allows the compiler to perform optimizations. If, for example, the case-clauses represent a "well-packed" set of integers, the compiler can generate a jump table indexed by the switch expression, which may be a much faster implementation than `if/else/endif`.

Finally, and this may be a matter of taste, `switch/case` statements are more legible and more easily modified than `if/else/endif` statements and are preferred programming style for selecting between fixed integers.

3.5 The `for` Statement

The `for` statement is a very compact form of iteration. The syntax of this statement is:

```
for (INITIALIZE; TEST; ITERATE)
      STATEMENT;
```

The sequence of operations is as follows:

- INITIALIZE
- if TEST evaluates to 0, terminate the loop
- STATEMENT
- ITERATE
- if TEST evaluates to 0, terminate the loop

- STATEMENT
- ITERATE
- if TEST evaluates to 0, terminate the loop

and so on. Recall that STATEMENT may actually be a compound statement enclosed in curly brackets.

Some things to note:

- The INITIALIZE, ITERATE, and STATEMENT components are all statements
- The TEST component is an expression
- The STATEMENT component *may never be executed* because TEST is evaluated immediately after INITIALIZE.

Here is an example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      for (i=0; i < argc; i=i+1) {
            printf("Argument %d is %s\n", i, argv[i]);
      }
}
```

The INITIALIZE statement in this case executes '`i=0`'. Then, the value of `i` is compared to `argc` and the loop does not execute if the comparison is false. Otherwise, the `printf` statement is executed and then the ITERATE statement '`i=i+1`' executes. This process repeats (i.e., TEST-STATEMENT-ITERATE) until the expression '`i < argc`' is no longer true.

Note that the declaration of a variable within the INITIALIZE statement is not allowed, as it is in Java. For example, the following is legal in Java (and C++) but illegal in C:

```
for (int i=0; i < argc; i=i+1)      // ILLEGAL in C
```

3.6 The `while` Statement

The `while` statement is a subset of the `for` statement in that it only implements a TEST expression. For example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      i=0;        // INITIALIZE
      while (i < argc) {
            printf("Argument %d is %s\n", i, argv[i]);
            i = i+1;    // ITERATE
      }
```

```
}
```

The compound statement following the `while` expression executes as long as the expression is true. As with the `for` statement, the `while` loop may never execute if the expression isn't true to begin with.

Even though it appears that the `while` statement is unnecessary, given that `for` does it all, the `while` statement is very common in most programs. In many cases, no initialization is necessary and the iteration part of the loop is embedded in the statements that are executed.

3.7 The `do/while` Statement

A slight variation on the while statement is `do/while`. Here is a variation on the example from above:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      i=0;        // INITIALIZE
      do {
            printf("Argument %d is %s\n", i, argv[i]);
            i = i+1;    // ITERATE
      } while (i < argc);
}
```

The form of the `do/while` statement is:

```
do {
      STATEMENT;
} while(TEST);
```

Note that the STATEMENT component is executed *at least once*, regardless of whether or not TEST is true. For this reason, this example is not exactly equivalent to the prior examples using `for` and `while`.

3.8 The `break` Statement

The `break` statement can be used to prematurely terminate a loop. We have seen that it can also be used to exit a `switch/case` structure.

In a loop, the `break` statement causes the loop to terminate immediately and execution to proceed with the statements following the loop. For example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      for (i=0; i < argc; i=i+1) {
            if (i >= 10) {
```

```
                        break;      //Only print the first 10 arguments
                }
                printf("Argument %d is %s\n", i, argv[i]);
        }
        // 'break' statement takes us here
}
```

The behavior for `while` loops and `do/while` loops is identical.

Note that Java's *labelled break* statements (e.g., '`break outer;`') are not allowed in C. But see the `goto` statement in Section 3.10 below which can be used to implement this feature.

3.9 The `continue` Statement

The `continue` statement proceeds to the next iteration of a loop, skipping the remaining statements. In a `for` loop, the `continue` statement causes execution to jump to the ITERATE clause of the `for` statement:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
        int i;

        for (i=0; i < argc; i=i+1) {
                // Skip over first three arguments
                if (i < 3) continue; // Jumps to 'i=i+1' above

                printf("Argument %d is %s\n", i, argv[i]);
        }
}
```

In a `while` loop, the `continue` statement returns to the top of the loop to once again evaluate the while-expression. This can lead to some unexpected errors:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
        int i;

        i = 0;
        while (i < argc) {
                if (i < 3) continue; // Jumps to test 'i < argc' above

                printf("Argument %d is %s\n", i, argv[i]);
                i = i + 1;
        }
}
```

This code doesn't work! The problem is that the iteration '`i=i+1`' is never executed since the `continue` statement jumps back to the while-expression.

In a `do/while` loop, the `continue` statement jumps to the bottom of the loop to evaluate the while-expression.

Note that Java's *labelled continue* statements (e.g., '`continue outer;`') are not allowed in C. But see the `goto` statement in Section 3.10 below which can be used to implement this feature.

3.10 The `goto` Statement

The `goto` statement can be used to jump to any other statement at the same (or lower) level of nesting. That is, you can't jump into the middle of a loop, and you can't jump from one function to another, etc. For example:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      goto ohletsjustskipit;

      for (i=0; i<argc; i=i+1) {
            printf("Argument %d is %s\n", i, argv[i]);
      }
ohletsjustskipit:       // This is a label
}
```

The `goto` statement above transfers execution to the label (declared by writing an identifier followed by a colon), skipping all statements in between.

Note that Java does not have a `goto` statement. Why? The idea of simply jumping from one statement to another is not consistent with structured programming ideals. The abuse of this statement in other languages (like BASIC) led to *spaghetti programming*, in which the flow of execution in a program would jump back and forth, making it difficult to understand and modify the program. The introduction of labelled `break/continue` statements as well as structured exception handling (`try/catch`) in Java have removed all justifiable needs for keeping `goto` in the language.

One of the main uses for the `goto` statement in C (some would say the only use) is to duplicate the functionality of labelled `break/continue` statements in Java. That is, we would often like to be able to either break or continue the non-inner-most loop. For example:

```
#include <stdio.h>

void main(void) {
      int i,j;

      for (i=0; i < 10; i=i+1) {
            for (j=0; j < 10; j=j+1) {
                  if (2*i+j > 50) {
                        goto alldone; // Like a "double break"
                  }
            }
      }
alldone:
}
```

A `break` statement instead of the `goto` above would only have terminated the innermost loop, proceeding to the '`i=i+1`' iteration statement of the outermost loop, which is not the desired behavior.

Similarly, the `goto` statement can also be used to "escape" from a deeply nested structure in the case of an error condition:

```
void main(void) {
      while (...) {
            for (...;...;...) {
                  if (...) {
                        while (...) {
                              ...
                              if (error) goto BigError;
                              ...
                        }
                  }
            }
      }
      goto AllDone;     // Finished without errors

BigError:
      // Handle errors here

AllDone:
}
```

In Java and C++, this type of escape mechanism is handled by structured exception handling using `try/catch`.

3.11 The `return` Statement

The `return` statement is used to immediately exit from a function and return to the function's caller. For functions declared as returning a value, the `return` statement must be followed by an expression indicating the return value. The compiler may cast this value to the return type as necessary. For functions declared as returning `void`, i.e., no value, the `return` statement must stand by itself.

Here is an example showing the use of the `return` statement in several locations in a subroutine.

```
long int factorial(int N) {
      if (N < 2) {
            return 1;   // 0!==1, 1!==1
      }
      return N*factorial(N-1);
}
```

In general, good programming practice dictates that there should be few, preferably just one, `return` statements in a function. This makes debugging easier as the exit points of a function are more constrained. This means less work is required to ensure that a function does not return unexpectedly during a debugging session.

4.0 Expressions and Operators

A C expression is composed of variable names, constants, function calls, and operators to combine them. We have already encountered "obvious" expressions of the form '`i < argc`'.

4.1 Basic Arithmetic Operators

The operators `+`, `-`, `/`, and `*` perform addition, subtraction, division, and multiplication, as expected. Even at this level of simplicity, things can get interesting:

- What happens when we add an `int` to a `float`?
- What happens when we write 7/4?

4.2 Promotion and Casting

The answer to the first question above is given by the promotion rules in C, which state that when two quantities are involved in an expression, the "smaller" one is converted to the type of the "bigger" one. A `float` quantity can represent integers, but integers can't represent reals. Thus, `float` is "bigger than" `int`.[1] These promotion rules also determine the type of the overall expression.

For example:

```
int i = 2;
double f = 1.0;

f = f + i; // i is promoted to double type, expression is double type
i = i + 3; // both i and 3 are ints, expression is int type
i = f + 1; // ???
```

In the expression '`f+i`', the integer variable `i` is promoted to a `double` and takes on the value 1.0. This value is then added to `f` and the result is an expression of type `double`.

In the expression '`i+3`' both operands are of integer type thus no promotion is necessary and the expression is of type `int`.

Now let's look at '`f+1`'. The constant 1 is of integer type thus must be promoted to `double` type with value 1.0. The resulting expression is of `double` type, but we are trying to store this into an integer! This type of "reverse promotion" (demotion?) can result in a loss of data (how do we represent 3.5 as an integer?) so it requires more intervention (i.e., the compiler does not perform demotion).

We need to tell the compiler our intention to demote a `double` to an `int` using an operator known as a *cast*:

```
i = (int) (f+1);  // double expression f+1 casted to int type
```

1. Note that this is an attempt to provide an intuitive explanation of promotion rules, which are in fact very clearly and thoroughly defined in reference texts. These rules, however, are lengthy and involved and it is suggested that the intuitive approach be your guide.

In this cast, the real-valued quantity ‘`f+1`’ will have its fractional part discarded and the integer part stored in the variable `i`.

The answer to the second question above (“What happens when we write 7/4?”) is that we get the integer 1. Because both 7 and 4 are integers, the operation is of integer type and integer division is performed (i.e., reals are truncated to remove the fractional part). If we expect an answer of 1.75, then we must somehow indicate that real division is to be performed. Either one of the operands must be of real type for this to happen. For example:

- 7.0/4.0
- 7/4.0
- 7.0/4

A common mistake in arithmetic computations is to forget that promotion is only performed when necessary for a specific operation, not for the overall expression. For example, suppose we wanted to express a probability as a percentage, in which we have two integers `heads` and `totalFlips`. The percentage (a real number) is given by:

```
heads/totalFlips*100.0        // This is WRONG
```

This code won’t work. Assuming `heads` is less than `totalFlips`, the integer division of `heads` by `totalFlips` will be 0. This is promoted to a 0 of type `double` prior to multiplying by 100.0 but by then it’s too late.

We want to perform *real* division of two *integers*. We can use a cast to force one of the division operands to `double` type, thus causing both division operands to be of `double` type (the second one is automatically promoted):

```
(double)heads/totalFlips*100.0      // This is RIGHT
```

Another way to do it is to force promotion to `double` type as early as possible:

```
100.0*heads/totalFlips
```

In this case, the multiplication is performed first (associativity is left-to-right) thus promoting `heads` to `double` type.

4.3 More Arithmetic Operators

The modulo operator ‘%’ computes the remainder of dividing its two integer operands. For example, ‘7 % 4’ is equal to 3.

The increment operator ‘++’ and decrement operator ‘`--`’ are shorthand ways of adding or subtracting 1 from an integer-valued variable. For example, ‘`i++`’ is equivalent to ‘`i=i+1`’ and ‘`j--`’ is equivalent to ‘`j=j-1`’.

The increment and decrement operators can be applied in both prefix and postfix form. The difference lies in what the value of the expression is (yes, the statement ‘`i++`’ is in fact an *expression*).

Let's look at an example:

```
int i,j;

i=3;
j = i++;    // j=3, i=4
```

Note that the value of the expression '`i++`' is the value of `i` ***before*** incrementing. Now consider:

```
int i,j;

i=3;
j = ++i;    // j=4, i=4
```

With the prefix form of the operator, the value of the expression '`++i`' is the value of `i` ***after*** incrementing.

4.4 Assignment Operators

It may bend your mind a bit to think of the statement '`i=3`' as an expression with '`=`' as an operator, but this is the reality. In C, just about everything is an expression (except for function/variable declarations, flow control statements, and a few other things). The "statement" '`i=3`' is actually an expression whose value is the left-hand-side (i.e., the variable `i` after the assignment is performed). Thus, the following is legal:

```
j = (i=5) + 3;    // j=8, i=5
```

Just as '`i++`' is a shortcut for '`i=i+1`', there are shortcut assignment operators, too. For example, '`f *= 5.2`' is a shortcut for '`f=f*5.2`'. The full suite of shortcut assignment operators is shown in Table 2 on page 28.

4.5 Bitwise Operators

The bitwise operators can be used to operate on individual bits of an integer rather than on the number as a whole. For example, the bitwise AND operator '&' computes the logic AND function on every bit of its operands:

```
char i,j;

i=0x0F;           // In binary, i is:     00001111
j = i & 0xF8;     // In binary, 0xF8 is:  11111000
                  // So i & 0xF8 is:      00001000
```

Thus `j` will have the value 8. The bitwise OR operator '|' computes the logic OR function while the bitwise XOR operator '^' computes the logic XOR function.

All of the above are binary operators, requiring two operands. The unary operator '~' only takes one operand and computes the one's complement. For example, '~0' yields a result that has all bits set (the number of bits depends upon the number of bits in the `int` type).

The bitwise shift operators ‘<<‘ and ‘>>’ shift all bits of the left-hand operand either left or right by the number of bits specified in the right-hand operand. For example:

```
char i = 0x80;    // In binary, i is:          10000000
i = i >> 1;       // Shift right by 1 bit:     01000000
i = i >> 5;       // Shift right by 5 bits:    00000010
i = i >> 3;       // Shift right by 3 bits:    00000000
```

The ‘<<‘ operator is fairly straightforward: all bits are shifted left and 0-bits shift in from the right.

The ‘>>’ operator is slightly more complicated in that its behavior depends upon whether we are operating on a signed or unsigned integer. For signed integers, the right-shift is arithmetic, i.e., the uppermost bit (the sign bit) is kept the same. For unsigned integers, the right-shift is logical, i.e., the uppermost bits are filled with 0’s. For example:

```
signed char i = -1;     // In binary, i is:     11111111
unsigned char j = 255;  // In binary, j is:     11111111

i = i >> 1;             // i is:          11111111
j = j >> 1;             // j is:          01111111
```

Java has the additional operator ‘>>>’ which is not present in C. This operator performs a logical right-shift (shifting in 0’s to the upper bits). Since Java does not support unsigned integers, this operator is required to support logical right-shifts in Java but is unnecessary in C.

4.6 Relational Operators

The relational operators (listed below) all return 0 if the relation is false and non-zero[1] if true:

- `<, >, <=, >=`
- `==` (equality, note double-equals)
- `!=` (non-equality)

4.7 Logical Operators

The unary logical negation operator ‘`!`’ returns non-zero if its operand is 0 and returns 0 if its operand is non-zero. Thus, the expression ‘`! (a<b)`’ is equivalent to ‘`(a>=b)`’.

The logical AND operator ‘&&’ returns non-zero if both its operands are non-zero quantities. This operator is frequently used to ensure multiple conditions are true. For example:

```
if ( (i >= 0) && (j >= 0) && (k >= 0) ) { ...
```

Similarly, the logical OR operator ‘||’ returns non-zero if either of its operands are non-zero quantities.

1. Another example where the C language specification does not specify what this non-zero value should be. In most cases, truth is represent by 1, sometimes by -1. But falsity is always 0 in C.

Note that '&&' (logical AND) and '&' (bitwise AND) are frequently confused leading to program errors. A useful convention is to define the following:

```
#define AND &&
#define OR  ||
#define NOT !
```

allowing us to write more readable (and less error-prone) statements:

```
if ( (i >= 0) AND (j >= 0) AND (k >= 0) ) { ...
```

Another aspect of the logical AND and OR operators is *deferred evaluation* (sometimes known as *lazy evaluation*). The basic concept is that if the result of the overall expression is certain at any point, the remaining components of the expression are not evaluated. Consider, for example, the assignments '`i = 5`', '`j = -5`', and '`k = 10`'. In the `if` statement above, the following would occur:

- The subexpression '`(i >= 0)`' evaluates as true, thus the overall expression *may* be true. We now have to consider the next subexpression.
- The subexpression '`(j >= 0)`' evaluates as false, thus the overall expression is *definitely* false (1 AND 0 AND 1 is false, i.e., 0).
- At this point, the whole expression's value is known and the final subexpression '`(k >= 0`') is irrelevant. Due to deferred evaluation, this expression is *not evaluated*!

Apart from saving some execution time, deferred evaluation can lead to some compact code. Consider, for example, an attempt to divide by an unknown quantity that may be 0:

```
(denom != 0) && (f = f / denom);
```

What looks like an expression (and is) is actually equivalent to:

```
if (denom != 0) {
      f = f / denom;
}
```

but is more compact. The reason this works is that if '`(denom != 0)`' fails to be true, the actual division is not performed (due to deferred evaluation) since its value is irrelevant in the logical AND computation.

4.8 The Conditional Operator

The conditional operator '`?:`' is like an embedded `if/else/endif` statement. Whereas the latter is a true statement and cannot be used in an expression, the conditional operator can. We have already seen the definition of the '`max`' macro (written here without the safety parentheses):

```
#define max(x,y) (x > y ? x : y)
```

The above definition leads to two equivalent statements:

```
z = max(x,y);     // Statement 1
if (x > y) {      // Statement 2
      z = x;
} else {
      z = y;
}
```

The syntax of the conditional operator is:

```
EXPR ? TRUE_CLAUSE : FALSE_CLAUSE
```

First, the EXPR expression is evaluated. If true, the value of the conditional operator is the result of executing the TRUE_CLAUSE. Otherwise, the value of the conditional operator is the result of executing the FALSE_CLAUSE. Note that both clauses must be “compatible” in type (again, there are a whole host of rules that govern this). Thus, the following would be illegal:

```
(i > 0) ? 5 : “Hello”              // ILLEGAL
```

We cannot have this expression return integer type in one case and a string in the other. We can, however, mix integers and reals (for example) wherein standard promotion rules would apply.

4.9 The Comma Operator

A strange operator, the comma ‘`,`’ is simply used to evaluate multiple expressions as if they were a single expression. All expressions separated by a comma are evaluated, and the value (and type) of the expression is the value (and type) of the last expression. For example:

```
code = ((i > 0) ? 5 : (printf(“Error!\n”), 0));
```

The clause following the colon is an expression containing two sub-expressions separated by a comma. The first sub-expression (the call to `printf`) is evaluated and ignored, then the second expression (simply 0) is the result.

The most common use of the comma operator is in `for` statements allowing multiple initializations, tests, or iteration statements. Consider, for example, some compact code for reversing the bits in a 32-bit integer:

```
int reverse(int i) {    // Assumes int is a 32-bit integer
      int mask;
      int rev;
      int orvalue;
      for (rev=0, mask=0x80000000, orvalue=1;
                  mask != 0;
                  mask = mask >> 1, orvalue = orvalue << 1) {
            rev = rev | ((i & mask) ? orvalue : 0);
      }
      return rev;
}
```

Note the use of the comma operator in both the initialization and iteration components of the `for` statement.

Java does not have the comma operator. This might be a good idea since the comma is also used to separate parameters in function calls. This dual use of the comma can be very confusing.

4.10 Operator Precedence and Association

There is an implied precedence of operators (for example, '`2+i*3`' evaluates '`i*3`' first, as expected) as well as an implied associativity. The associativity of an operator indicates the order in which multiple instances of the operator are evaluated. For example, '2*3*4*5' is evaluated first as '2*3', then the result multiplied by 4, then multiplied by 5. In other words, this is really '((2*3)*4)*5. This is left-to-right associativity.

For an example of right-to-left associativity, as exhibited by the assignment operator, consider the statement:

```
i = j = k = 0;
```

This statement is computed using right-to-left associativity, as follows:

```
i = (j = (k=0));
```

Table 2 below summarizes the precedence and associativity of all operators in C. This table lists operators from highest to lowest precedence. Operators in a single group have equal precedence.

Mostly, things make sense with a few exceptions. For example:

```
mask<<1 + 1
```

is actually

```
mask << (1+1)
```

since addition has higher precedence than bit shifts. Be careful and use parentheses liberally.

5.0 Program Structure

At some point, placing all of your code in the `main()` function will become unwieldy. It will then be necessary to split your code into multiple functions. Once a source file starts accumulating many functions, it too will become unwieldy. At this point it will become necessary to split your code across multiple files. In this section we discuss the structure of a C program split across multiple source files, include files, and library files.

5.1 Declaring Functions

We have already seen an example of a function, the `main()` function. Other functions can be declared in the same file as the one with the `main()` function. Let's look at an example:

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

Table 2: Precedence and Associativity of C Operators

Operators	Function	Associativity

`[]`
`()`
`.`
`->`
postfix`++`
postfix`--`
	Array dereference
	Function call
	Structure member dereference
	Structure pointer dereference
	Postincrement
	Postdecrement
		Left-to-right

`++`prefix
`--`prefix
`sizeof()`
`&`
`*`
`+`
`-`
`~`
`!`
	Preincrement
	Predecrement
	Size in bytes of given type or variable
	Address-of
	Pointer dereference
	Unary positive
	Unary negative
	One's complement
	Logical complement
		Right-to-left

`(type)`	Cast operator	Right-to-left
`*, /, %`	Multiply, divide, modulo	Left-to-right
`+, -`	Addition, subtraction	Left-to-right
`<<, >>`	Bit shifts	Left-to-right
`<, >, <=, >=`	Comparison operators	Left-to-right
`==, !=`	Comparison operators	Left-to-right
`&`	Bitwise AND	Left-to-right
`^`	Bitwise XOR	Left-to-right
`|`	Bitwise OR	Left-to-right
`&&`	Logical AND	Left-to-right
`||`	Logical OR	Left-to-right
`?:`	Conditional operator (if-then-else)	Right-to-left
`=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	Assignment operators	Right-to-left
`,`	Comma operator	Left-to-right

```
double poly(double x) {
      // Evaluate the polynomial:
      //          4x^3 - 2x^2 + x + 5
      // using Horner's rule.
      double val;

      val = 4*x-2;
      val = val*x + 1;
      val = val*x + 5;
      return val;
}

void main(void) {
      int i;

      for (i = -5; i <= 5; i++) {
            printf("(%d,%g)\n", i, poly(i));
      }
}
```

This program prints as ordered pairs the integers from -5 through 5 as well as the value of the polynomial (listed above) evaluated at these points. Note that there are two functions in this source file: `poly` and `main`. Any number of functions may be present in a source file, but the definition of a function must be known to the compiler prior to its invocation (in other words: define it before you use it).

Suppose we had swapped the two functions above, with the `main` function appearing before `poly`. The compiler would have reported an error: "Call to function 'poly' with no prototype". This is because the compiler has not encountered the definition (i.e., the prototype) for `poly` prior to its use within the `printf` statement.

There are two ways to address this problem. First, and most obvious, define a function before using it. Second, use the `extern` keyword to introduce a function's prototype before actually writing the code for it:

```
#include <stdio.h>

extern double poly(double);   // Prototype only, no code

void main(void) {
      // main code goes here
}

double poly(double x) {
      // actual code for poly goes here
}
```

The `extern` keyword indicates an "external function prototype" so that the compiler knows this function's return value and "signature" (i.e., the parameters and their types). The actual code for `poly`, of course, must match its prototype.

This use of the `extern` keyword is most common in include files, which we discuss in Section 8.2.

J Only one function with a given name may be defined. Unlike Java, C does not support overloading (i.e., two functions with the same name but different signatures).

5.2 Calling Functions

We have already seen many examples of calling functions and it should be fairly clear how this is achieved: the function's name is followed by parentheses and some number of arguments.

One issue that does arise, however, is what happens when the expected type of a function parameter doesn't match what is passed. Note in the example above that we passed an integer to the `poly` function but this function expected a `double` parameter. Is this an error? No, the standard promotion rules apply.

Note that there is an insidious behavior in the reverse direction: a mismatch in types will cause a silent demotion! If, for example, a function expects an integer but a `double` is passed, the `double` is silently (i.e., without warning) casted to an integer!

5.3 Local Variables

All variables defined inside a function are *local* to that function. This means that other functions can't access those variables, and the variables are only "alive" while the function is executing (i.e., their values don't persist from one function call to the next).

An exception to the persistence issue above are variables declared with the `static` keyword. These variables do retain their value for the duration of the program. These variables are frequently initialized and used either as counters or as flags to indicate whether the initialization of some process has occurred. For example:

```
void countme(void) {
      static int firstTime = 1;
      static int counter = 0;

      if (firstTime) {
            firstTime = 0;
            printf("This is the first time in countme\n");
      }
      printf("countme has been called %d times\n", ++counter);
}
```

Had the two variables above not been declared `static`, they would have been initialized to 1 and 0, respectively, upon every invocation of `countme`.

5.4 Global Variables

Variables declared at file scope (i.e., outside any function) can be accessed by any function in the file. Like static local variables, these *global variables* persist for the duration of the program (until changed).

Global variables can be used to communicate from one function to another but this is not recommended (that's what function parameters are for!) A good use for global variables is to use them as type-safe constants. We can always use the preprocessor to write, for example:

```
#define PI 3.1415926536
```

but this is just a text substitution. It doesn't tell the compiler we're defining a real number. More type-safe is to write (as a global variable):

```
const double PI = 3.1415926536;
```

Then, any function in this file can make use of the variable `PI` and know that it's intended to be a `double` quantity.

The `const` keyword indicates that the variable PI may never be modified. Attempting to assign to it will cause a compile-time error.

6.0 Advanced Data Types

Beyond the scalar-valued types of integers and reals, C supports arrays, structures, enumerated types, bitfields, and pointers.

6.1 Arrays

Arrays are very commonly encountered in C. They simply represent many quantities of the same type. For example:

```
int v[5];
```

represents 5 integers rather than simply 1 integer. These integers can be accessed as `v[0]`, `v[1]`, etc. up to `v[4]`. **NOTE**: Arrays are declared with the number of elements that they contain but the maximum element index is this number minus 1 (since array indices start at 0).

Note that the square brackets `[]` actually represent an operator (known as *array dereference*) which has a relative priority and associativity, as shown in Table 2 on page 28.

J Java programmers should immediately note a difference. Arrays are declared directly with their given size rather than using a two-step process of declaring a reference to an array then using the `new` operator. But note that Java's approach is really a use of pointers, as explained in Section 6.3. In addition, C arrays are not objects. They don't have methods, and most importantly, they don't know their own size (i.e., there is no `array.length` method). For this reason, C arrays are not bounds-checked. Accessing an array element beyond the end (or before the beginning) of a C array may well crash the program rather than generate an orderly exception, as with Java.

Here is an example, using arrays, of a function named `polyN` that evaluates an arbitrary-length polynomial given the coefficients of this polynomial in an array:

```
double polyN(double x, double coeffs[], int numCoeffs) {
      int i;
      double val = 0.0;

      for (i=0; i < numCoeffs; i++) {
            val = val*x + coeffs[i];
      }
      return val;
}
```

Note that array parameters in functions can not be declared with a size so that arbitrary-length arrays may be passed. However, we must also pass an extra parameter to indicate how many elements are actually in the array, since this is not a property of the array itself.

Two-dimensional (and higher-order) arrays may be declared as in this example:

```
int twod[5][3];
```

In this case (and all higher-order cases) array parameters must have all dimensions specified except possibly for the last. This makes arbitrary-size two-dimensional arrays difficult to pass as parameters (we generally just use pointers). The ability to define classes in C++ and Java that properly implement higher-order array behavior greatly improves our expressive power in this respect.

6.2 Structures

A structure represents a collection of variables bound together under one name. For example:

```
struct pixel {
      int x,y,z;
      double intensity;
      double luminance;
};
```

The two words ‘`struct pixel`’ now behave as a built-in type, with certain restrictions. For example, we may declare a variable of this type:

```
struct pixel one_pixel;
```

or a whole array of them:

```
struct pixel allPixels[128];
```

We access the members of this structure using the *member dereference* operator ‘`.`’ (see Table 2 on page 28 for this operator’s precedence and associativity). For example:

```
one_pixel.x + one_pixel.y + one_pixel.z
```

J For Java and C++ programmers, structures are simply classes without methods and with all variables public. Note, however, that C does not support assignment of structures. It is illegal to assign one structure variable to another. You must either copy each member one item at a time or perform a memory-to-memory copy (see Section 7.3).

6.3 Pointers

Ah, yes. Pointers. At last, we arrive at the most dreaded word in the lexicon of the C student. Pointers are indeed so dreaded that Java has completely done away with pointers and wrapped their functionality into the (admittedly safer) concept of *references*. C++, as a transitional step, has both pointers and references.

Pointers are important, however, because there is a close association between this data type and the underlying hardware architecture of digital computers. By distancing ourselves from the pointer concept, we may achieve fewer programming errors and fewer headaches, but we increase the distance between our code and "the bare metal." This distance, in many engineering applications involving low-level hardware interactions, should be as small as possible.

Every variable in C occupies some memory. This memory resides at a certain address. A pointer variable is capable of storing the address of another variable. That's it.

Here is an example:

```
int i=3;
int *iptr = &i;
```

The second line declares a *pointer to an integer* using the syntax '`int *iptr`'. The '`*`' symbol in a variable declaration indicates a pointer type. The type '`int`' in this declaration indicates that the variable `iptr` is a pointer to another variable that is of '`int`' type.

Simply put, we declare that `iptr` will contain the address of an integer variable.

The declaration is completed by initializing `iptr` with the quantity '`&i`'. The ampersand '`&`' is known as the *address-of operator* (see Table 2 on page 28). Once again, every variable in C occupies some memory and this memory resides at a certain address. This address can be obtained with the address-of operator.

In our example, assume that the variable `i` represents the value 3 stored at location 0x1000. Then, the variable `iptr` has the ***value*** 0x1000; the variable `i` has the ***address*** 0x1000. What is the address of the variable `iptr`? We haven't bothered to determine it, but we could always obtain it using the address-of operator: '`&iptr`'. This would give us a pointer to a pointer to an integer, of type '`int **`'.

After executing the following code:

```
*iptr = 5;
```

(read as ‘set the contents of memory whose address is stored in `iptr` to 5’) we have done the same thing as:

```
i = 5;
```

In the above example, the ‘*’ operator is the *pointer dereference operator* (see Table 2 on page 28), also known as the *contents-of operator*. It may be helpful to think of this operator as the inverse of ‘&’.

Pointers are useful for very many things. As a simple use of pointers, consider the problem of returning more than one value from a function. Suppose we wanted to compute both the sum-of-squares and sum-of-cubes of a set of numbers. It would be a shame to place these two computations in separate functions as we would be re-computing many quantities, leading to slower code. It would be more efficient to have a single function that returned both quantities. But how? One option is to declare a structure type to hold the result and return the structure. A more efficient way is as follows:

```
double sums(double values[], int howMany, double *sumOfCubes) {
      double sumOfSquares;
      int i;

      *sumOfCubes = 0;  // Dereference pointer to return value #2
      sumOfSquares = 0; // Initialize return value #1
      for (i=0; i < howMany; i++) {
            double temp = values[i]*values[i];

            sumOfSquares += temp;
            *sumOfCubes += temp*values[i];
      }
      return sumOfSquares;
}
```

Note that we are indeed returning two values, one in the conventional way (`sumOfSquares`) and another via a pointer. Here is how we may call this function:

```
double testcases[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
double squares, cubes;
squares = sums(testcases, 5, &cubes);
```

The third parameter to the `sums` function is a pointer to a `double`. This quantity must point to an existing `double` variable where we would like our sum-of-cubes result to go.

One problem with pointers is that small programming errors lead to big run-time errors. Imagine if we had written the following line in error, forgetting that `sumOfCubes` was a pointer:

```
sumOfCubes += temp*values[i];       // Should be ‘*sumOfCubes’
```

This would modify the address stored in `sumOfCubes`, not the dereferenced quantity. Worse, imagine if we had effected the same mistake at the initialization line:

```
sumOfCubes = 0;   // Intending to write '*sumOfCubes = 0;'
```

This will surely lead to a program crash later in the program as attempting to dereference the address 0 will wreak havoc. And whereas the first type of error will at least draw a compiler error ('Illegal use of floating point', since it makes no sense to increment a pointer by a real number) the second error is perfectly legal C. Not only will it lead to a run-time error but the actual error is *downstream*, i.e., it is a secondary effect and may occur very much farther in the program where it is very difficult to trace the root cause.

The moral: pointers are merciless. Simple errors in programming lead to big errors in the program. Experience and discipline in programming are the best defense.

A suggestion is to always use a pointer variable's name to indicate it is a pointer. For example, declaring the sums function as:

```
double sums(double values[], int howMany, double *sumOfCubesPtr)
```

reminds us whenever we use the variable that it truly is a pointer.

Another suggestion is to be a defensive programmer and check all function parameters prior to using them. For example:

```
#include <stdlib.h>     // Defines exit() function

double sums(double values[], int howMany, double *sumOfCubesPtr) {
      double sumOfSquares;
      int i;

      if (sumOfCubesPtr == 0) {     // How did that happen?
            printf("Whoa there!\n");
            exit(1);
      }
      ...
```

Null pointers (i.e., pointers with value 0) are a very common source of errors and catching them early can save a lot of time in debugging.[1]

A very common use of pointers is in representing strings, as discussed in Section 6.4 below.

Only a few operators can be applied to pointers. For example, it makes no sense to try to multiply or divide pointers. It is very common, however, to add or subtract quantities to pointers, or even add and subtract pointers from each other. A very interesting behavior in C is that adding an integer to a pointer adds a multiple of the pointer type, not the integer itself. For example:

```
      char *charPtr;          // Points to 1-byte quantities
      int *intPtr;            // Points to 4-byte quantities
```

1. For advanced programmers, investigate the `assert` function in the standard C library. This can be a very effective debugging aid.

```
double *doublePtr;      // Points to 8-byte quantities

charPtr++;        // charPtr is now higher by 1 address
intPtr++;         // intPtr is now higher by 4 addresses
doublePtr++;      // doublePtr is now higher by 8 addresses
```

While perhaps counterintuitive at first, this behavior is very useful for allowing pointers to "step through" arrays. For example, this function sets all the elements of a passed array to 0:

```
void setTo0(int arr[], int N) {
      int i;
      int *iptr;

      for (i=0, iptr=&arr[0]; i < N; i++, iptr++) {
            *iptr = 0;
      }
}
```

There are better ways to do this (see the `memset` function in the standard C library) but this example illustrates the point. Incrementing `iptr` by 1 does not add 1 to the address in `iptr`, it adds whatever the size of an '`int`' is, thus advancing from one element of `arr` to the next.

Note that pointers can also be used to reference structure variables. For example:

```
struct pixel {
      int x,y,z;
};
struct pixel aPixel;
struct pixel *pixelPtr;

pixelPtr = &aPixel;
```

There are two ways to access the members of a structure pointed to by a pointer variable. We can dereference the pointer and use the membership operator '`.`' as before, or use the more compact pointer membership operator '`->`'. For example, the following expressions are equivalent:

```
(*pixelPtr).y;
pixelPtr->y;
```

The form of the second expression is preferred for legibility.

6.4 Strings

In C, strings are simply arrays of `char` type that end with a 0 byte. Strings are not objects, as in Java. Most commonly, however, strings are referred to according to the address of the first character in the array, thus invoking the concept of pointers.

Here is a simple illustration of string usage in C. This program fills in each character of a string, one character at a time, then prints the whole string.

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

```
void main(void) {
      char hello[8]; // ‘hello’ is an 8-byte string
      hello[0] = ‘H’;
      hello[1] = ‘e’;
      hello[2] = ‘l’;
      hello[3] = ‘l’;
      hello[4] = ‘o’;
      hello[5] = ‘!’;
      hello[6] = ‘\n’;
      hello[7] = 0;
      printf(“%s”, hello);
}
```

We make the following observations:

- Constants of type `char` may be written in single quotes. For example, the constant `‘H’` is equivalent to writing the constant 0x48. Only one character may appear between the single quotes, with the exception of escape sequences (like ‘`\n`’ that indicates the end-of-line character.

- The last element of the `char` array ***must be 0*** in order for the array to be properly treated as a string. Omitting this 0 element (referred to as the *null byte* of the string) will lead to incorrect behavior and possibly program crashes. Remember to always declare character strings with one extra character to hold the null byte.

- A string in C is nothing more than an array of `char` type. Note that not all arrays of `char` type need be *interpreted* as strings. An array of `char` type can be used to hold a set of 8-bit values that do not have a textual interpretation.

6.4.1 String Pointers

Take a look at the `printf` statement in the above example again. It uses the name of the array variable `hello` but without any square brackets. What does this mean?

In C, writing the name of an array without square brackets is equivalent to writing the address of the array’s first element. Thus,

```
hello == & (hello[0])
```

is true. We indicated to `printf` the string we wanted to print by giving the address of the first character. We could also have written:

```
printf(“%s”, &hello[0]);
```

This is a very important concept. *In C, strings are passed from one function to another using the address of the first character, not as the whole array.*

Passing the entire array would be very wasteful. Simply passing the address of the array is much more efficient and also allows the array to be modified by any function in-place, rather than

returning a modified copy of the array. Note that this concept applies to *any* C array, not just strings: arrays are passed from one function to another using the address of the first element.

The fact that we are passing the address of a string means we are actually passing a pointer. What is the type of the expression '`& hello[0]`'? It is an address of a `char` so it is a pointer-to-`char` type, or '`char *`'. For example:

```
char *helloPtr = & hello[0];
printf("%s", helloPtr);
```

is also correct, although redundant since `helloPtr` is exactly equivalent to `hello` (without square brackets).

Let's look at another example that demonstrates the use of string pointers. This program prints each one of the characters of the string "Hello world!" on a separate line.

```
#include <stdio.h>

void main(void) {
      const char *helloPtr = "Hello world!\n";
      while (*helloPtr != 0) {
            printf("%c\n", *helloPtr);
            helloPtr++;
      }
}
```

There are some new concepts in this example. First, consider the line:

```
const char *helloPtr = "Hello world!\n";
```

We are declaring the variable `helloPtr` as storing the address of a character. Thus, we can use this variable to point to a string (array of characters). In this declaration we are also initializing the contents of this variable to point to the string "`Hello world!\n`". This string is a character array that will be stored in memory somewhere and whose address (of the first character) will be stored in `helloPtr`.

Finally, declaring `helloPtr` as a '`const char *`' type indicates that this is a pointer to memory that should not be changed (i.e., it is a pointer to *constant* characters, not writable characters). The compiler would generate an error if you attempted to modify memory through the `helloPtr` pointer.

NOTE: Writing a string in double-quotes automatically sets aside space for this string in memory. The compiler also automatically includes the null byte at the end of the string. Thus, the string "ABC" is a 4-character string. There are three characters plus a null byte. Also, the string in double-quotes is a valid C expression. The type of this expression is '`const char *`'.

Now let's look at the first statement in the `while` loop:

```
while (*helloPtr != 0) {
```

In plain English, this expression tests to make sure the pointer `helloPtr` is not the address of a character with value 0, i.e., the null byte of a string. That is, we are testing to see if we're at the end of the string. The expression '`*helloPtr`' dereferences this pointer to look at the actual character that the pointer is pointing to.

The next line:

```
printf("%c\n", *helloPtr);
```

causes `printf` to print a single character (this is what the `%c` format specifier does) followed by a newline. The character to print is '`*helloPtr`', that is, the actual character that the pointer is pointing to.

Note that `helloPtr` is of type '`const char *`'. Dereferencing this pointer with the expression '`*helloPtr`' gives us a value of type '`const char`', i.e., a single character.

Finally, the statement:

```
helloPtr++;
```

advances the pointer to point to the next character in the string. The loop terminates once this pointer points to the null byte (i.e., end of the string).

6.4.2 String Operations

Remember, strings are not objects and do not have methods. We operate on strings by calling functions that expect string pointers as arguments. Here, for example, is a function that prints each character of a string on a separate line.

```
#include <string.h>

void printChars(const char *str) {
      int i;

      for (i=0; i < strlen(str); i++) {
            printf("%c\n", str[i]);
      }
}
```

First, we must include the `string.h` include file to declare the external functions that operate on strings (these functions are in the standard C library).

The parameter of the `printChars` function is a pointer to a string, and we use the `const` keyword to indicate that we will not be modifying this string in the function.

In the `for` statement, note that we are calling the `strlen()` function and passing the string pointer parameter to it. This function computes the length of a string, i.e., how many characters are in the string *before* the null byte.

Finally, in the `printf` statement we are accessing each character of the string individually using the array dereference operator (square brackets). It may be confusing to treat a pointer variable as an array, but recall that they are virtually equivalent. In C, it is true that:

```
str[i] == *(str + i);
```

Thus, just think of the array dereference as first advancing the pointer forward by the given number of elements, then looking at the contents of memory at that address.

6.5 Enumerated Types

The `enum` keyword introduces an enumerated type. Here is an example:

```
enum direction {
      North, South, East, West
};
```

We can now declare a variable of this type:

```
enum direction currentDirection;
```

This variable can only take on one of the four constants listed in the type declaration: `North`, `South`, `East`, and `West`. The compiler automatically creates these constants and assigns integers to them, beginning with '`North=0`' and ending with '`West=3`'.

Strictly speaking, enumerated types are not necessary. We could have mimicked the above with preprocessor constants and a variable of type `int`:

```
#define North     0
#define South     1
#define East      2
#define West      3

int currentDirection;
```

The enumerated type, however, is more type-safe. For example, the statement:

```
currentDirection = 5;
```

would lead to a compile-time error (as it should) in the first example but not in the second. Using the enumerated type also prevents programming errors that arise from mis-typing the `#define` statements (suppose we mistakenly used the same number for two constants).

6.6 Bitfields

Bitfields can be very useful for accessing a set of bits in an integer rather than the whole integer itself. Most often, bitfields are used within structures, like this:

```
struct init_register {
      int regbase:4;          // Lower 4 bits
      int rambase:4;          // Next 4 bits
```

```
} iReg;
```

Writing to each structure member only modifies 4 bits of the `iReg` structure variable. For example:

```
iReg.regbase = 0xC;
iReg.rambase = 0x1;
```

These two statements together modify 8 bits of data, but only 4 bits at a time.

Bitfields can be very useful for mapping the structure of a hardware register to C variables. The example above mapped out the two 4-bit fields of the 68HC11 INIT hardware register.

6.7 Unions

Unions are very similar to structures in appearance but behave quite differently. Each member of a union occupies the same memory space as the other members. This allows the union to act as a sort of adapter to treat the same memory contents as two (or more) different types. For example, suppose we sometimes wanted to access a 16-bit hardware register as a single 16-bit `short` quantity and sometimes as two 8-bit `char` quantities. We could use the following declaration:

```
union timerReg {
      short timer;
      struct timerBytes {
            char timerLO;
            char timerHI;
      };
} theTimer;
```

The first member of this union is the 16-bit quantity `timer`. The second member of this union, *occupying the same memory as* `timer`, is the 16-bit structure named `timerBytes`, which has two 8-bit members, `timerLO` and `timerHI`. We now have two ways of writing to this hardware register:

```
theTimer.timer = 0x1234;      // One way to do it
theTimer.timerBytes.timerHI = 0x12; // Another way to do it
theTimer.timerBytes.timerLO = 0x34;
```

Note that this sort of operation is not portable across hardware architectures as it depends upon word sizes and the endianness of the platform.

7.0 Advanced Programming Concepts

7.1 Standard I/O

We have made extensive use of the function `printf` throughout our examples so far. This function belongs to a class of functions known as the *standard I/O* component (or `stdio`) of the standard C library. All of these functions operate on I/O streams declared as type '`FILE *`' (the type `FILE`, along with the `stdio` functions, are declared in the system header file `stdio.h`).

In this section, we explore some of the remaining standard I/O functions along with example usage.

7.1.1 Opening Files

The `fopen` function can be used to open a file on disk and prepare it for reading or writing. This function takes a string that indicates the file name and returns a quantity of type '`FILE *`' that can be used in other standard I/O functions. Here is an example:

```
#include <stdio.h>

void main(void) {
      FILE *file;

      file = fopen("c:\\junk.txt", "wt");
      fprintf(file, "This is junk\n");
      fclose(file);
}
```

The first parameter to `fopen` is the name of a file. It may or may not include path components (e.g., "c:\\"). If not, the file is assumed to be in the current working directory. Note that the double-backslash in the file name is used to achieve an actual single-backslash since backslashes introduce escape sequences (like `\n`) in C strings. On Unix-like systems, the regular slash '/' is the path separator (and in fact, this should work on Windows systems as well).

The second parameter to `fopen` tells this function what we want to do with the file: open it for reading, writing, both reading and writing, etc. as well as whether the file is a binary file ('b' suffix in the string) or a text file ('t' suffix in the string). The parameter "`wt`" indicates that we want to open the file for writing (overwriting any existing file by the given name) and it will be a text file.

The `fopen` function either returns a valid *file pointer* or it returns 0 to indicate an error condition. All file pointers created using `fopen` must be closed using `fclose`, as in the example above, when no longer needed.

7.1.2 Writing to File Pointers

Once a text file has been opened for writing using `fopen`, data can be written to the file using the `fprintf` function, as in the above example. Note that `fprintf` looks very much like the `printf` function we have been using, except it takes an extra parameter at the front of the parameter list: the file pointer. The format string and remaining parameters of `fprintf` are described in detail in any C reference text or help file.

Single characters may be written to files with `fprintf` and the `%c` format specifier, or with the `fputc` function (consult a reference for usage information).

Binary files are written using `fwrite` instead of `fprintf`. Consult a C reference text for the usage of this function.

7.1.3 Reading from File Pointers

There are several ways of reading data from a text file. The `fgets` function reads one text line of the file at a time. The `fscanf` function attempts to read one data value from the text file. We illustrate both of these methods with examples below.

First, let us examine a program that counts the number of lines in a text file that begin with the '#' character.

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

void main(void) {
      int numLines = 0;
      char lineBuffer[80];
      FILE *file = fopen("c:\\inputfile.txt", "rt");

      if (file==0) {
            printf("Cannot open input file\n");
            exit(1);
      }

      while (fgets(lineBuffer, 80, file)) {
            if (lineBuffer[0] == '#') {
                  numLines++;
            }
      }
      printf("%d lines began with #\n", numLines);
      fclose(file);
}
```

The `fgets` function reads one text line from the file and stores it in the string pointed to by its first argument. Space for the string must have been set aside previously, either with a character array (as in the example) or via dynamic memory allocation (see Section 7.2). The maximum number of characters to read is given by the second argument and prevents the character array from overflowing due to long lines in the input file. When the end of the file is reached, `fgets` returns 0 and the `while` loop terminates.

Now, let us examine a program that uses the `fscanf` function to read data values rather than entire lines. This program assumes that the input file contains some number of floating-point values written as text numbers, e.g., "2.345". All of the numbers are separated by whitespace (i.e., either spaces, tabs, or newlines). For each number, the program prints its value to the screen.

```
#include <stdio.h>

void main(void) {
      FILE *file;
      double value;

      file = fopen("infile.txt", "rt");
      while (fscanf(file, "%lf", &value) == 1) {
            printf("Value: %f\n", value);
```

```
        }
        if (! feof(file)) {
              printf("ERROR: Malformed floating-point value\n");
        }
  }
```

The format of the fscanf function is:

```
fscanf(FILE *, "FormatString", &var1, &var2, ...);
```

The first parameter is a file pointer, as returned by `fopen`. The second parameter is a format string (see a reference manual or help file; this format string is interpreted in essentially the same way as the one for `printf`). The remaining parameters are ***pointers to*** the variables that will hold the values to read. Note that in the example above we used the address-of operator '`&`' to provide a pointer to the variable '`value`' which will hold the floating-point quantity read from the file.

The `fscanf` function has a return value: the number of input values successfully interpreted according to the format string. In our example, there was one format specifier in the string, indicating we were expecting a single value at a time. The return value should be 1, then, to indicate that we converted some text in the file to a floating point number. If the return value is less than 1, it indicates there was an error and we should not trust the contents of the '`value`' variable.

There are two primary ways in which the `fscanf` function could fail. First, we may have reached the end of the file. In this case, the `feof` function (see example above) can be used to test for this case, which is not an error as it simply lets us know we've run out of data. Second, we may encounter something in the input file which cannot be converted to a real number. For example, encountering a string of letters would cause `fscanf` to fail, since it has been instructed to expect a floating-point number. In the example above, if `fscanf` fails to return 1 and we are not at the end of the file, we assume that the input file contained something other than a real number at the point of failure.

Single characters may be read from a file using the `fgetc` function (consult a reference for details).

Binary files are read using the `fread` function. Consult a C reference text or help file for the usage of this function.

7.1.4 The `stdin`/`stdout`/`stderr` Streams

There are three standard I/O streams that have already been opened by the time a program runs. The first, called `stdin`, is a text file pointer that can be used for reading (i.e., `fscanf`, `fgets`, etc.) It represents the user's keyboard, not a true "file". Thus, characters typed at the keyboard can be read by calling standard I/O input functions on the file pointer `stdin`. For example, here is a subroutine that expects the user to enter an integer at the keyboard:

```
#include <stdio.h>

int getInteger(void) {
```

```
        int value;

        while (1) {
              printf("Enter an integer: ");
              fflush(stdin);
              if (fscanf(stdin, "%d", &value) == 1) {
                    return value;
              }
              printf("That was not a valid integer. Try again.\n");
        }
  }
```

The purpose of the call to `fflush` is to "flush" the stream, i.e., remove anything that may be in it. This will ensure that there are no characters left hanging around from the user's previous typing.

Similarly, `stdout` is a text file pointer that can be used for writing to the user's console. You can use `fprintf` to write the file pointer `stdout`. Note that this is equivalent to using `printf` without the file pointer argument. That is, the following are equivalent:

```
fprintf(stdout, .....);
printf(.......);
```

Finally, `stderr` is similar to `stdout` but it is generally used for reporting errors, whereas `stdout` is used for printing non-error information (data, etc.)

7.2 Dynamic Memory Allocation

It is often the case that the amount of storage required for a variable is not known at compile time. For example, consider a program that must sort a file containing real numbers, but it is not known how many numbers there are in the file. The first line in the file, however, could be an integer indicating how many numbers to expect. Still, it is not possible to declare an array that can be guaranteed to hold as many real numbers as required without imposing an arbitrary limitation on the program.

The solution to these types of situations is to first determine how many elements in the array are needed (i.e., how much storage is required) then allocate space for the array based upon this number. This can be accomplished using pointers and the `malloc` set of functions in the standard C library. These functions are declared by including the `<stdlib.h>` system include file.Here is an example that reads some number of real numbers from a file where the file's first line contains the number of real numbers that follow.

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

void main(void) {
      double *reals;    // Array size not yet known
      int numReals;     // Number of numbers to read
      int i;

      FILE *file = fopen("numbers", "rt");
```

```
        if (file == 0) { errorHandling(); }

        if (fscanf(file, "%d\n", &numReals) != 1) { errorHandling(); }

        reals = (double *)malloc(numReals*sizeof(double));
        if (reals == 0) { errorHandling(); }

        for (i=0; i < numReals; i++) {
              if (fscanf(file, "%g", &reals[i]) != 1) {
                    errorHandling();
              }
        }
        fclose(file);

        ...   // Do something useful with reals array

        free(reals);
    }
```

We declared an array of unknown size by writing '`double *reals`'. Thus, we are simply declaring a pointer variable, not pointing to anything at the moment. Once we know how many real numbers we need space for, we can dynamically allocate that space using the `malloc` function. This function takes as its parameter some number of ***bytes*** and returns a pointer to the allocated memory.

The fact that `malloc` needs to know how many bytes we need to allocate means we must pass it the expression '`numReals*sizeof(double)`'. This expression multiplies the number of `double` quantities we need by the number of bytes occupied by a `double`. The C operator `sizeof()` returns the number of bytes occupied by any type, including user-defined types (see Section 7.4)[1].

The return type of `malloc` is '`void *`', meaning it is a pointer to memory of unknown type. This must then be cast to the appropriate type using the cast operator, as shown above. The `malloc` function returns 0 if it was unable to allocate the requested memory. This condition should always be checked, as in the example.

After a successful call to `malloc`, the `reals` pointer can now be treated as an array. The first element read in is `reals[0]`, then `reals[1]`, etc. all the way up to `reals[numReals-1]`.

Memory allocated with `malloc` must be returned to the operating system using the `free` function. This function takes as its parameter a pointer previously allocated using `malloc`.

J The differences between dynamic memory allocation in C and Java are great. Allocating a dynamic array of reals in Java is achieved simply by '`reals = new double[numReals]`'. Furthermore, there is no need to call a `free` function as memory is automatically reclaimed (i.e., *garbage collected*) when it is no longer used. The low-level dynamic memory allocation approach

1. Note that `sizeof()` is truly an operator, not a function call. It is an operator that is evaluated at compile time, not run time.

in C has been the cause of innumerable program errors. Calling `free` with the same pointer twice, for example, is an excellent way to crash a program.

The approach in Java makes life much simpler for the programmer. It is not without cost, however. Once again, it is a tradeoff between control over the underlying hardware versus simplicity. In C, allocated memory is freed only when the program so dictates. In Java, a garbage collector thread runs "every so often" to clean up memory. In an environment where timing is important, it may be quite inappropriate for this thread to run at certain times. More things happen behind the scenes in Java, where the programmer has less control, while C gives the programmer nearly absolute control over the runtime environment.

There are two variants of `malloc` that may also be useful. The `calloc` function also allocates memory but it ensures that the memory is zeroed-out. The `realloc` function attempts to take a pointer to previously-allocated memory and resize the memory block (either bigger or smaller).

7.3 Memory Manipulation

A set of functions in the standard C library known as the *buffer manipulation functions* allow very low-level control over a computer's memory. These functions are declared by including the system header file `<string.h>`. For example, here is an example that fills the contents of memory locations 0x100 through 0x1FF with the constant 0x55:

```
memset((void *)0x100, 0x55, 256);
```

The first parameter to `memset` is an address (type '`void *`') of unspecified type. The second parameter is the byte to use to fill the block of memory. The third parameter is the number of bytes to write beginning at the given address.

This low-level memory interface can also be used to copy memory from one location to another. This is necessary, for example, when attempting to assign one structure variable to another. Suppose we have the following declarations:

```
struct {
      int x,y,z;
} pixel1, pixel2;
```

C does not support direct assignment of one structure variable to another:

```
pixel2 = pixel1;  // ILLEGAL!
```

We can, however, achieve the same effect by simply copying the contents of memory from one variable to the other:

```
memcpy(&pixel2, &pixel1, sizeof(pixel2));       // LEGAL
```

The first parameter to `memcpy` is an address (type '`void *`') indicating the *destination* of the copy operation. The second parameter is an address indicating the source of the copy operation.

The third parameter is the number of bytes to copy. Once again we use the `sizeof` operator to return the number of bytes occupied by a variable (or type).

Note that the memory functions operate very similarly to string functions (e.g., compare the operation of `memcpy` and `strcpy`). The difference is that the string functions all expect to operate on null-terminated strings. The memory functions operate on blocks of memory that are not necessarily null-terminated (and may, in fact, contain multiple null bytes) and the length of the memory block must be specified.

A final caveat: the `memcpy` function does not work when the source and destination ranges overlap. In this case, the `memmove` function (same interface as `memcpy` but generally a little slower) must be used. For example, here is a function that removes the first element of an array and shifts the remaining elements down to take its place.

```
#include <string.h>

void removeHead(void *array, int numElements, int elementSize) {
      memmove(array,
                  (char *)array+elementSize,
                  (numElements-1)*elementSize);
}
```

The source of the memory movement is the second element of the array, i.e., the starting address of the array plus the number of bytes in 1 element. Note the expression:

```
(char *)array + elementSize
```

We cannot add integers to `void` pointers (since the size of the array type is unknown). We must cast the pointer to type '`char *`' so that adding `elementSize` to it increments the pointer by this many bytes. Recall that the expression '`array+N`' is equivalent to '`&array[N]`', but only when the type of the array is known (i.e., `array` is not a '`void *`').

7.4 Declaring New Types

C has primitive support for declaring new types. The `typedef` keyword can be used to attach a name to an existing type, which may make the program more readable. The basic usage of a `typedef` declaration is:

```
typedef known_type new_name;
```

For example, here is an attempt at improving portability by using new type names for integers with a known bit width.

```
#ifdef INTS_ARE_SHORT
typedef int       Int16;      // 16-bit integers
typedef long      Int32;      // 32-bit integers
#else // INTS_ARE_LONG
typedef short     Int16;      // 16-bit integers
typedef int       Int32;      // 32-bit integers
```

```
#endif
```

Using the type `Int16`, for example, to declare a variable guarantees that this variable occupies 16 bits, regardless of the underlying hardware platform. This assumes that the constant INTS_ARE_SHORT has been defined (using `#define`) on platforms with 16-bit integers and left undefined on 32-bit platforms.

Another common use of `typedef` is to name structures so that the `struct` keyword need not be a part of the type. For example:

```
typedef struct loc3d {
      int x,y,z;
} Location3D;
```

The known type in this case is the structure type while the new name for this type is `Location3D`. Thus, the following two declarations are identical but the second is more legible:

```
struct loc3d aPixel;
Location3D aPixel;
```

The final use for `typedef` is to build up a complex type out of simpler types. The C language supports extremely complicated types (e.g., arrays of pointers to arrays of arrays of...) but these types can be very difficult to declare properly. Suppose we wanted to declare a pointer to an array of pointers to 5-byte integer arrays. We can build it up one type at a time:

```
typedef char FiveByteArray[5];      // A 5-byte array type
typedef FiveByteArray *FiveByteArrayPtr;  // Pointer to 5-byte arrays
typedef FiveByteArrayPtr *FiveByteArrayPtrPtr;
                        // Array of pointers to 5-byte arrays
typedef FiveByteArrayPtrPtr *FinalType;
```

7.5 Pointers to Functions

An extension of the concept of the `typedef` keyword as explained in Section 7.4 is the definition of functions as types. In C, functions are *first-class objects* in that they can be stored in variables, passed as parameters, etc. Why would you want to do this?

A concept known as *generic programming* attempts to separate data structures from the functions that operate on them. For example, finding the smallest entry in an array, or a linked list, or a tree, etc. represents three different functions that implement essentially the same algorithm: check for a new minimum, get the next value in the data structure, and repeat. The fundamental *algorithm* does not depend upon whether the data is stored as an array, or a linked list, or a tree. That is, the data structure is separate from the algorithm, and the same algorithm, once developed and tested, should be reusable on a wide variety of data structures.

The C++ language implements generic programming by its use of the Standard Template Library, or STL. In C, we can use pointers to functions to try and implement some sort of generic programming. For example, one of the functions in the standard C library, `qsort`, implements a quicksort sorting algorithm that works on a wide variety of data structures (as long as they are stored as an

array). This function requires several parameters: a pointer to where the data to be sorted resides, the size of each element in the array, the number of entries in the array, and the final element, a pointer to a *compare function*.

It is in the compare function (written by the user) that the specific nature of the data structure is reflected. This function is passed two pointers, each pointing to one element of the data structure. The compare function must return -1 if the first element is "less than" the second (in some sense), 0 if they are equal, and 1 if the first element is "greater than" the second. The `qsort` function, then, uses this return value to effect the algorithm; the inner details of the data structure are abstracted out.

We can declare the type of this compare function compactly as:

```
typedef int (*CompareFunction)(void *, void *);
```

This declares the new type `CompareFunction` which is a pointer to a function taking two parameters (both `void` pointers) and returning an integer. We can now select from among multiple compare functions:

```
int func1(void *p1, void *p2) {
      ...
}
int func2(void *p1, void *p2) {
      ...
}

CompareFunction whichFunc = func1;
```

Note that simply writing the name of a function (e.g., '`func1`') is an expression of type 'pointer to function'. We can then invoke the function as follows:

```
int return_value;
return_value = (*whichFunc)(p1, p2);
```

Admittedly, this is highly esoteric and unlikely to be useful to new students of the language. We hope, however, that the power of C is becoming apparent despite its starkness and often-confusing pointer concept.

7.6 Command-Line Parameters

The prototype of the entry point of a C program is written as:

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

Let us consider how to interpret the parameters of the `main` function.

The first parameter `argc` is simple: it simply indicates the number of command-line parameters. If the compiled program is executed as follows:

```
myprogram -o stuff file
```

then `argc` would be 4. The name of the program itself (i.e., `myprogram`) is the first parameter of the argument list. Thus, a C program can discover the name by which it was invoked (a trick used by sneaky programmers to modify a program's behavior simply by creating several links of different names to the same executable).

The remaining parameters in the example above are the strings '`-o`', '`stuff`', and '`file`'. These strings, as well as the first parameter (the program name) '`myprogram`' are all available in the parameter `argv`.

The `argv` parameter is an array of string pointers. For example, `argv[0]` is of type '`char *`'. We recognize this as (possibly) a pointer to the first character of a null-terminated string. Indeed, `argv[0]` is a pointer to the string that is the first parameter (the program name). The next element in the array, `argv[1]` is also of type '`char *`' and describes another string: the second parameter ('`-o`' in the example above).

Thus, we see that the `argv` array extends from `argv[0]` through `argv[argc-1]`, with each entry being a pointer to a null-terminated string. Here is the example program from page 16 that prints each command line parameter passed to it:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      int i;

      for (i=0; i < argc; i=i+1) {
            printf("Argument %d is %s\n", i, argv[i]);
      }
}
```

J Java uses a similar prototype for the main function except that the `argv` parameter is an array of String objects. Since array sizes can be obtained in Java using the `array.length` method, the `argc` parameter is not required.

8.0 Multi-File Programs

It is possible to spread a C program across multiple source files. The compiler, however, only compiles one file at a time. Any functions declared in one source file but used in another must somehow make themselves known to the latter. In this section we discuss the standard techniques of splitting a program's functions and variables across multiple files.

8.1 Basic Concepts

Let us extend the example presented in Section 5.1 on evaluating a polynomial. Suppose we wanted to define a function `poly4` that could evaluate any 4^{th} order polynomial with arbitrary coefficients. We feel that this function is separate enough from the goings-on of the main program that we should put it in its own file. We will implement the following:

- A function called `poly4coeff` that allows the caller to set the coefficients of the polynomial. Presumably, the coefficients will be reused many times so it is more efficient to set them once rather than passing them every time.

- The function `poly4` to evaluate the polynomial with the coefficients set by `poly4coeff`.

- A variable `poly4calls` indicating how many times we've called `poly4`. This may be useful in *profiling* our code (i.e., estimating the time spent in various sections of the program).

Here is one possible implementation of the above, all as part of the file `poly4.c`:

```
int poly4calls = 0;

static double C1, C2, C3, C4;       // The coefficients

void poly4coeff(double c1, double c2, double c3, double c4) {
      C1 = c1;
      C2 = c2;
      C3 = c3;
      C4 = c4;
}

double poly4(double x) {
      double val;

      poly4calls++;

      val = C1*x + C2;
      val = val*x + C3;
      val = val*x + C4;
      return val;
}
```

What may we learn from this example?

1. The variable `poly4calls` is a global variable. We need its value to persist across multiple calls to `poly4`. We also need this variable to be accessible outside of `poly4` since other functions may want to know how many times `poly4` has been called. This means we can't make `poly4calls` a static local variable (i.e., put it inside `poly4`).

   This global variable, then, is a form of communication between two (or more) files. The `poly4.c` file modifies this variable and other functions in other files presumably inspect its value. There is nothing preventing other functions from modifying `poly4calls`, however, which would lead to program errors. The principle of information hiding or *encapsulation* in object oriented languages like C++ or Java is intended to avoid these problems (we can also do it in C using functions to hide the variable).

2. The constants C1, C2, C3, C4 are also global variables. We need these variables to persist from the call to `poly4coeff` until `poly4` is called. Note, however, that unlike `poly4calls`, these 4 constants are also declared with the `static` keyword.

   Unfortunately, `static` here means something slightly different than `static` applied to local variables. A global variable declared `static` is not accessible in functions outside the file. Only the functions defined in the file `poly4.c` can inspect or modify these variables (which is exactly what we want).

   As a brief summary of the `static` keyword, consider the following table that compares the scope (i.e., where the variable is accessible) and persistence of a variable:

Table 3: Persistence and Scope of Global and Local Variables

	`static`	**Not `static`**
Local Variables	• Function scope • Persistent	• Function scope • Not persistent
Global Variables	• File scope • Persistent	• Program scope • Persistent

   In summary, local variables always have function scope. The `static` keyword determines whether or not they are persistent. Global variables are always persistent. The `static` keyword determines whether they have program scope (i.e., accessible anywhere) or file scope.

   Finally, the `static` keyword can also be applied to functions. They behave the same as for global variables, i.e., they prevent the function from being visible outside of the file. Presumably, static functions are “helper” functions only intended for use within the file.

Now let’s use our new functions. Consider the file named `main.c` with the following contents:

```
#include <stdio.h>

// Defined in poly4.c
extern int poly4calls;
extern void poly4coeff(double c1, double c2, double c3, double c4);
extern double poly4(double x);

void main(void) {
      int i;

      poly4coeff(4.0, -2.0, 1.0, 5.0);
      for (i=-5; i<=5; i++) {
            printf(“(%d,%g)\n”, i, poly4(i));
      }
      printf(“We called poly4 %d times.\n”, poly4calls);
}
```

We can create the final program by compiling both files at the same time:

```
bcc32 -epoly4test.exe main.c poly4.c
```

Remember that any variable or function must be declared before it is used. This is why we had to include the three `extern` lines in the `main.c` file above. Note that an `extern` declaration of the constants C1 through C4 was ***not*** included in `main.c`. These constants are private to `poly4.c` and are not visible outside that file (since they were declared as static global variables).

8.2 Include Files as Interfaces

We have encountered the main principles of encapsulation in the C language: data hiding using file scope and function scope. The principles of modular program design require us to encapsulate a component of the program according to functions and variables that are visible and those that are hidden, thus minimizing interdependencies (and errors) across the program design.

An effective way to do this involves using include files to define a module's externally visible interface. The process works as follows:

- The functions and global variables that should be visible outside of the file in which they are declared represent this module's *interface*. The interface is defined by several `extern` statements placed in an include file.

- This include file is included (with the `#include` directive) wherever another file needs to reference the given module's interface.

- The module in which the functions and variables are declared also includes the include file. This isn't strictly necessary but is a very good defense against mismatches in the source code and the include file (the compiler will flag the differences).

- Functions and global variables that are private to a module are not put in the include file and are declared `static`.

For example, let us split our example program into three files now:

- The `main.c` file (same as before)
- The `poly4.c` file (same as before)
- A `poly4.h` file (all the `extern` statements go here)

Here is the new `poly4.h` file:

```
// Defined in poly4.c
extern int poly4calls;
extern void poly4coeff(double c1, double c2, double c3, double c4);
extern double poly4(double x);
```

The `main.c` file is now smaller:

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

```
#include “poly4.h”

void main(void) {
      int i;

      poly4coeff(4.0, -2.0, 1.0, 5.0);
      for (i=-5; i<=5; i++) {
            printf(“(%d,%g)\n”, i, poly4(i));
      }
      printf(“We called poly4 %d times.\n”, poly4calls);
}
```

We have gained several benefits from this approach:

- Any changes to the interface of `poly4.c` are reflected in a single file, `poly4.h`, rather than having to modify all the other files that might have used it.
- We can make changes to the implementation of `poly4.c` without changing the interface. For example, we may want to implement the `poly4` function in assembly language for maximum speed. This does not affect the interface whatsoever.

This approach to modularity and encapsulation is clearly less robust than that of classes, public/private distinctions, and inheritance, all available in Java and C++. As programs become larger and larger, the modularity mechanisms of C become unsatisfactory. Both C++ and Java are better languages for very large programs due to their better support for modularity and encapsulation.

8.3 Object Files and Linking

A complete program is created by *linking* together one or more *object files*. A single source file generates a corresponding object file, in which the ‘`.c`’ extension is replaced with ‘`.obj`’ (or ‘`.o`’ on Unix-type systems). For example, compiling ‘`hello.c`’ (successfully) generates the object file ‘`hello.obj`’.

A separate program known as the *linker* pulls together one or more object files and stitches them all together into a single executable file. The linker’s role is to resolve *external references*. A function (or variable) declared `extern` in one source file must eventually be found in an object file. If not, the linker issues an “unresolved reference” error, meaning you probably forgot to write certain functions. This error can also arise if functions or global variables are declared `static` by mistake, in which case the linker doesn’t “see” them (i.e., the have file scope, not program scope).

For example, the polynomial evaluation example program shown in Section 8.2 comprises two source files, `main.c` and `poly4.c`, which are compiled to create the object files `main.obj` and `poly4.obj`. The linker then examines the `main.obj` file and sees that it expects to use a function named `poly4`. This function, however, is not defined within the `main.obj` file, thus is an external reference. When the linker examines `poly4.obj`, it finds the definition of `poly4`, which resolves this external reference. If all external references are resolved, linking is complete and an executable file is generated.

A set of object files that may be of use in a wide variety of programs are often packaged into a *library*. A library is simply a collection of pre-compiled object files. For example, suppose you created several source files to support the evaluation of many types of polynomials. After compiling all of these source files to create object files, a *librarian* program can package these object files into a single library file, perhaps named `poly.lib`. This file can then be distributed to anyone that wishes to use your functions as part of a larger program.

C compilers include system libraries that can be used in the linking process for your program. One such library, the *standard C library* (briefly introduced in Section 9.0), contains the object code for functions such as `printf`. Linking your program with the standard C library "pulls in" the `printf` function from somewhere within the library and adds it to your executable program.

The linking process is similar to the importing of classes in a Java program. Whereas the latter occurs at run-time in a Java program, linking is a static process that must occur before you can run your program. A variant of linking, known as *dynamic linking*, is similar to the Java approach in that the required object files (packaged into *dynamic link libraries*, or DLL's in Windows environments) are not linked until the program runs. Creating programs that take advantage of dynamic linking is an esoteric process, and it doesn't help that this process is very different on Windows and Unix-type systems.

8.4 The Details of the Compilation Process

For single-file programs, the compilation process is conveniently hidden by a *driver program* that orchestrates all of the compilation steps: preprocessing, compiling, assembly, and linking. As programs become larger and are split across multiple files and libraries, the details of these steps become more important. For example, the BCC32 program is actually a driver program that invokes the following sub-programs:

- The C preprocessor (built-in to BCC32 or runable externally as CPP32.EXE)
- The C compiler (built-in to BCC32)
- The assembler (built-in to BCC32, use '`-S`' option to view assembly source)
- The linker (the ILINK32.EXE program)

The preprocessor (described briefly in Section 1.4) removes comments, handles `#include` directives, etc. and leaves another file as input to the compiler.

The compiler takes the output of the preprocessor and generates low-level assembly language code in yet another text file.

The assembler takes this text file and generates object code (see Section 8.3) which is simply several numbers that represent instructions for the computer's processor.

Finally, the linker brings together several object code files, either stand-alone or in libraries, and creates the final executable.

All of the above is hidden from the user in most cases. For example, if only the standard C library is used, the linker automatically searches it and there is no need to explicitly include it in the link-

ing process. If additional libraries need to be used, however, they must be specified to the driver program. For example, if you need to make use of a library called `serial.lib`, you can indicate this to the BCC32 program as follows:

```
bcc32 main.c serial.lib
```

The BCC32 program automatically recognizes `serial.lib` as a library (it doesn't try to compile it!) and passes it to the linker.

There can also be a mixture of source files, object files, and libraries passed to the driver program. For example:

```
bcc32 main.c file1.obj serial.lib
```

The driver program compiles `main.c` to create `main.obj`, then passes the object file `file1.obj` and `serial.lib` to the linker in order to create the final executable.

If you wish to create an object file from source code but not link, the '-c' option can be used:

```
bcc32 -c main.c
```

This will generate `main.obj` (i.e., preprocess, compile, assemble) but will not attempt to perform the link step.

9.0 The Standard C Library

The specification for the C programming language not only describes the language components but also describes a standard set of functions that must be made available by all C compiler suites. These functions comprise the *standard C library*.

As the name implies, the standard C library is a set of pre-compiled object files that can be linked in with a user's program. On Unix systems, this library is called '`libc.a`' and is traditionally supplied with the operating system (although installing a separate compiler, such as GCC, will also install a separate C library for use with that compiler). The BCC32 compiler under Windows comes with the file '`CW32.LIB`', which is the standard C library for Win32 applications.

In this section we briefly introduce the functions that comprise the C library in aggregate. A full explanation of all functions and their calling protocols is left for a reference source, such as the on-line help available with your compiler or a standard C reference text.

9.1 Assertion Checking

The include file `<assert.h>` defines the `assert()` function which expects its argument to be non-zero. If the argument is 0, the program aborts immediately. Assertions are useful for ensuring that things the programmer assumes to be true are indeed true at run-time. See Section 10.8 for an example.

9.2 Character Classification

The include file `<ctype.h>` defines several functions that test properties of ASCII characters. For example, the `isprint()` function determines whether or not its argument is a printable character (i.e., not a control character like a newline or a backspace), and the `isdigit()` function determines whether or not its argument is a decimal digit.

9.3 Error Reporting

The externally defined integer variable `errno` is often set to meaningful values by components of the standard C library to report errors. This variable is defined by including the file `<errno.h>`. In addition, this file declares several constants that describe the type of error. For example, `ENOFILE` is defined to be the constant that `errno` is set to when the `fopen` function is called to open a non-existent file. The function `perror` can be used to print a textual description of the error code.

Also see Section 10.9 for an example of error reporting using `perror`.

9.4 Buffer Manipulation

This library component has already been introduced in Section 7.3. The functions in this module are declared by including the `<string.h>` include file. The buffer manipulation functions support operations such as filling memory, copying memory blocks, and looking for bytes in a memory block.

9.5 Non-Local Jumps

The `<setjmp.h>` include file defines two functions, `setjmp()` and `longjmp()`, which can effect “non-local gotos”. These functions can cause program flow to jump back and forth between any two points in a program, not necessarily in the same function or even the same source file. These functions are most often used to effect handling of run-time errors across an entire program or to implement *coroutines*, a simulation of multi-threaded operation.

9.6 Event Signalling

The `<signal.h>` include file defines functions and constants to support the handling of events. For example, when a user presses the Ctrl+C key at the console, programs generally terminate. The reason they do so is because they receive a “signal” known as `SIGINT`. The default action for `SIGINT` is to terminate the program. Using the signal function, however, the programmer may indicate an alternative course of action to follow in response to this event.

9.7 Variable-Length Argument Lists

Have you wondered by now how come the `printf` function can take a variable number of parameters? This is because C actually supports functions which can be passed more parameters than some minimum number. The functions defined in the `<stdarg.h>` include file allow the programmer to define variable-length argument lists and write functions like `printf` that can make good use of this concept.

9.8 Miscellaneous Functions

The `<stdlib.h>` include file declares several functions that are commonly used:

- `atof, atoi, atol`: interpret strings as integers
- `strtod, strtof, strtold`: interpret strings as real numbers
- `strtol, strtoul`: interpret strings as integers (more robust than `atoi` or `atol`)
- `rand, srand`: generate pseudo-random numbers
- `abort`: immediately terminate the program
- `atexit`: define code to execute before a program terminates
- `exit`: terminate the program with proper cleanup and a return value
- `getenv`: return values from the external environment (like PATH)
- `system`: run external programs
- `bsearch, qsort`: efficient sorting and searching of arrays
- `abs, labs`: integer absolute values
- `div`: integer division and remainder in a single operation

9.9 String Handling

The `<string.h>` include file declares all of the functions that expect to operate on null-terminated character strings. Some basic string operations were already discussed in Section 6.4.2. The more common string functions are:

- `strcpy, strncpy`: copy one string to another
- `strcat, strncat`: append one string to another
- `strlen`: return the length of a string
- `strchr`: find a character within a string
- `strstr`: find a substring within a string
- `strcmp, strncmp`: compare two strings for equality
- `strtok`: split a string into tokens separated by delimiters

9.10 Time Functions

The `<time.h>` include file defines various components to support time and date functions. The `clock` function can be used to time how long a subroutine takes to execute, for example. The `time` function returns the current date and time as a variable of type `time_t`, while the `ctime` function takes the return value of the `time` function and formats it as a character string.

Consult reference documentation for additional functions and definitions in this module.

9.11 Floating-Point Math

Several mathematical functions, such as `sin`, `cos`, and `atan` are available to C programs. These functions are declared by including the `<math.h>` include file.

9.12 Standard I/O

This library component has already been introduced in Section 7.1. The functions in this module are declared by including the `<stdio.h>` include file. The standard I/O functions support buff-

ered input and output to file streams, which include both regular disk files as well as the user's console and keyboard.

10.0 Tips, Tricks, and Caveats

10.1 Infinite Loops

A loop introduced with `while` or `for` (or other looping statement) may sometimes be desirable when the exit condition is more clearly written in the middle of the loop. For example, this subroutine only returns when the correct password is entered.

```
#include <string.h>
extern const char *getString(void);
static const char *password="Swordfish";

void checkPassword(void) {
      while (1) {
            if (strcmp(getString(), password) == 0) return;
      }
}
```

Note the expression '`1`' is used as the test expression of the while loop. This "expression" always evaluates to true thus the loop never terminates, unless a `break`, `return`, or `goto` statement is encountered.

An infinite loop can also be introduced with the `for` statement:

```
for (;;)
```

10.2 Unallocated Storage

This program will crash:

```
#include <stdio.h>

void main(void) {
      int *intPtr;
      printf("Type an integer: ");
      fscanf(stdin, "%d", intPtr);
      printf("You typed: %d\n", *intPtr);
}
```

It seems to be consistent: the `fscanf` function expects a pointer to an integer in which to store the conversion result, so we provide it with `intPtr`, which is a pointer to an integer.

The problem is that we have not identified any space for this integer in memory. The variable `intPtr` points to the address of some integer, but that integer has not been *declared*. Put another way, the `intPtr` variable is not being initialized and is pointing to some random place in memory. The `fscanf` function will then store the integer it reads to this random place, causing a crash.

Put yet another way, space is created in memory to store data only by declaring non-pointer variables or dynamically (using `malloc`; see Section 7.2). ***Declaring a pointer variable does not create space to store data***. This mistake is very commonly made by new C programmers.

Using unallocated storage is even more common when using strings. The distinction between an actual array of characters, i.e., the string, and the pointer to these characters, which is how we refer to the string in function calls, is frequently confused. Here's another crash-worthy program:

```
#include <string.h>

void main(void) {
      char *myString;
      strcpy(myString, "Hello!");
}
```

Once again, we are copying characters from one "string" to another, but there is no memory space set aside for `myString`. This is a variable that points to...nothing. The correct implementation is either:

```
#include <string.h>

void main(void) {
      char myString[100];
      strcpy(myString, "Hello!");
}
```

or:

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

void main(void) {
      char *myString = (char *)malloc(100);
      strcpy(myString, "Hello!");
}
```

10.3 The Null Statement

A single semicolon is the simplest C statement; it simply means "do nothing". This can be useful as a placeholder for future code:

```
if (error) {
      ;           // Write this code later
}
```

Also, the semicolon can be used as a null body of a loop statement when all the useful work is actually done in the loop statement itself:

```
while (*(volatile char *)0x1000 == 0) /* NULL */ ;
```

This statement waits for the contents of memory location 0x1000 to be non-zero. There is nothing to be done in the body of the loop while waiting. The comment '`NULL`' is there as a form of documentation, reminding us that we intended there to be an empty loop body.

10.4 Extraneous Semicolons

A poorly-placed semicolon can lead to program bugs that are very tough to find. Consider:

```
for (i=0; i < 10; i++);
      printf("%d\n", i);
```

This looks right...but notice that extra semicolon at the end of the `for` statement. This creates a loop with a null body (see Section 10.3 above). The `printf` statement will only execute once, printing the value 10. This program fragment is perfectly legal; the compiler won't raise any errors.

10.5 `strcmp` is Backwards

Remember that in comparing two strings, the strcmp function returns 0 if they are *equal*. Thus, the following is correct:

```
if (strcmp(str1, str2) == 0) {      // They're equal
```

Some programmers find this so annoying that they define the macro:

```
#define streq(s1,s2) (strcmp((s1),(s2)) == 0)
```

and then write the more intuitive:

```
if (streq(s1,s2)) {     // They're equal
```

10.6 Unterminated Comments

Forgetting to include the final '`*/`' of a multi-line comment can lead to a compiler error many lines away from the comment, and this can be tough to find. For example:

```
/* Here starts a comment...but we didn't finish it!

void subroutine(void) {
      ...
      dosomething(); /* This comment's terminator finishes the
                        comment started at line 1 */
}
```

The C compiler will see only one line: the last. This will cause the strange error "Unexpected }". Note that the BCC32 compiler has the command-line option '`-C`' which enables "nested comments". This option will generate a much more useful error message in response to this situation.

10.7 Equality and Assignment

One of the greatest criticisms levelled at C is the use of '`==`' and '`=`' to mean totally different things. (Surprisingly, this criticism was not considered sufficient reason to change the situation in

Java.) Confusing the equality and assignment operators can be a source of very frustrating bugs. Consider the example:

```
while (error = 0) {     // OOPS!
      error = tryitagain();
}
```

The problem is that the `while`-expression should have been written '`error == 0`'. As written, the `while`-expression will assign 0 to `error` and always skip the body of the `while` loop.

The BCC32 warning message "Possibly incorrect assignment" (enabled by the '`-wpia`' command-line flag) alerts you to these types of errors.

10.8 Assertion Checking

Use the `assert` function liberally (see Section 9.1 and the reference documentation for the `assert` function). This function takes a single parameter, which it expects to be non-zero (i.e., true). If the parameter is 0, the program exits immediately. Assertion checking is one way of programming defensively. Rather than assuming that some condition is true by necessity, the assertion check verifies this assumption prior to proceeding with the program.

For example, consider a function intended to count the number of lines in a text file. This function assumes that it is passed a file pointer, as returned by a call to `fopen`. But what if this is not the case? Rather than proceed blindly with the code, we first verify our assumption.

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

int countLines(FILE *file) {
      int i;
      char line[80];

      assert(file != 0);
      for (i=0; fgets(line, 80, file); i++) /* NULL */ ;
      return i;
}
```

If it so happens that this function is called with a null pointer, the program will stop here rather than trying to proceed. The closer an error is found to its source, the easier it will be to find.

10.9 Error Checking

Programs that actively try to intercept and meaningfully describe run-time errors take longer to develop but often pay for themselves many times over in saved debugging time. Consider, for example, a program that counts the number of lines in a file. The name of the file is the first parameter of the program. The naive approach to this is as follows:

```
#include <stdio.h>

void main(int argc, char *argv[]) {
      FILE *file = fopen(argv[1], "rt");
      int lines = 0;
```

```
        char line[80];

        while (fgets(line, 80, file)) line++;
        printf("%d lines\n", line);
}
```

There are many things that could go wrong:

- The program was not called with an argument
- The argument to the program does not represent an existing file name
- An error occurred while reading the file
- The `printf` statement was unable to display the results

Any one of the above conditions will lead to incorrect behavior, possibly a crash, with little or no description of what went wrong. A more robust version of the above might be as follows:

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

int main(int argc, char *argv[]) {
        FILE *file;
        int lines = 0;
        char line[80];

        if (argc < 2) {
                fprintf(stderr, "No argument specified.\n");
                return 1;
        } else if (argc > 2) {
                fprintf(stderr, "Too many arguments specified.\n");
                return 1;
        }

        file = fopen(argv[1], "rt");
        if (file == 0) {
                perror("fopen");
                fprintf(stderr, "Unable to open '%s'\n", argv[1]);
                return 1;
        }

        while (fgets(line, 80, file)) lines++;

        // fgets could have failed either via end-of-file or
        // reading error.
        if (! feof(file)) {
                fprintf(stderr, "Error while reading input file\n");
                return 1;
        }
        printf("%d lines\n", lines);

        return 0;
}
```

A much longer program...but a much better program too. Note that we checked just about all the things that could go wrong, except for calls to `fprintf` and final call to `printf` at the end. These functions are also capable of returning error codes in case of failure, but what would we do in this case? Without an alternative means of communicating with the user, there is no point in testing for these errors.

Using subroutines to encapsulate both functionality and error checking is a good step towards managing the complexity of robust programs and also making them more readable. For example, consider the function:

```
FILE *OpenFile(const char *name, const char *mode)
```

This function could act as a *wrapper* around `fopen`, reporting any errors that arise. The main program could then simply call `OpenFile` and not worry about the error reporting:

```
if (OpenFile(argv[1], “rt”) == 0) return 1;
```

10.10 Programming Style

The question of style is perhaps difficult to define precisely. Subjectively, good style is what allows a C program to be easily read, written, and understood. This is not simply an aesthetic matter. It is possible to write some terrible code in C (and most any other language) that, unfortunately, works exactly as intended. When it is time for others to read this code (as part of a code review, perhaps) or for others to modify it, the code becomes a burden that carries with it a real penalty in productivity.

Consider, for example, the program shown below in Figure 1 that prints the value of the constant e to 3142 digits (written by Roemer B. Lievaart). While interesting, imagine if you were given the task of modifying this program so that it printed more digits. Where would you begin? The program, certainly intentionally designed to be interesting rather than useful, illustrates the idea that the source code is meant for both humans and computers.

There are some general guidelines that can lead to the development of good style:

- Be consistent. Use a style that matches existing code (if you are modifying or adding on to a program). It is difficult to read a program that uses different styles, as the style changes impair comprehension. Some of the components that should be consistent are:

  - the amount of indentation from one nesting level to the next
  - the style of curly brackets (on the same line? on new line?)
  - the naming convention of variables, functions, constants, etc. (see below)

- Use meaningful names for variables, functions, and constants. Avoid generic names like “`i`” for variables unless they are very short-lived and unimportant. For example, see if you can figure out what this subroutine does:

```
                                                                    char
                                                                _3141592654[3141
          ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
      _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
    _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
   -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
  =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415    +__3141;for                (_31415 = 3141-
           __3_1415  ;                 _31415;_31415--
           ,_3_141 ++,                 _3_1415++){_314
           +=_314<<2 ;                 _314<<=1;_314+=
          *_3_1415;_31                  =_314159+_314;
          if(!(*_31+1)                  )* _31 =_314 /
          __31415,_314                  [_3141]=_314 %
          __31415 ;* (                  _3__1415=_3_141
         )+= *_3_1415                    = *_31;while(*
         _3__1415 >=                     31415/3141 ) *
         _3__1415+= -                    10,(*--_3__1415
        )++;_314=_314                    [_3141]; if ( !
        _3_14159 && *                    _3_1415)_3_14159
        =1,__3_1415 =                    3141-_31415;}if(
        _314+(__31415                     >>1)>=__31415 )
        while ( ++ *                      _3_141==3141/314
       )*_3_141--=0                       ;}while(_3_14159
       ) ; { char *                       __3_14= “3.1415”;
      write((3,1),                        (--*__3_14,__3_14
      ),(_3_14159                          ++,++_3_14159))+
     3.1415926; }                          for ( _31415 = 1;
    _31415<3141-                           1;_31415++)write(
   31415% 314-(                            3,14),_3141592654[
  _31415    ] +                           “0123456789”,”314”
  [ 3]+1)-_314;                           puts((*_3141592654=0
,_3141592654))                             ;_314= *”3.141592”;}
```

Figure 1: C program to print 3142 digits of the constant *e*. This program does not exhibit good programming style.

```
double S(double *level, int Q) {
      int retval;
      double shrimp_on_a_plate = 0;
      for (retval=0; retval < Q; retval++) {
            shrimp_on_a_plate += level[retval]/Q;
      }
      return shrimp_on_a_plate;
}
```

Now consider the version with more meaningful variable names:

```
double mean(double *array, int numEntries) {
      int count;
```

```
                double avg = 0;
                for (count=0; count < numEntries; count++) {
                      avg += array[count]/numEntries;
                }
                return avg;
          }
```

To the compiler, they are one and the same program. To humans, the second program is much more readable. The use of meaningful names for program components leads to *self-documenting code*, meaning that few comments need to be written to explain program operation.[1]

- Develop a naming style that readily identifies functions, local variables, global variables, constants, etc. Here are some suggestions.

  Use ALL_CAPITALS for constants defined using `#define` directives.

  Use either `MixedCaps` notation or `underscore_separator` notation throughout your program.

  Distinguish between function parameters, local variables, and global variables by using single-letter prefixes. For example:

```
void SomeFunction(int pNumEntries, double *pArray) {
      int lCount;

      for (lCount = 0; lCount < pNumEntries; lCount++) {
            pArray[lCount] *= gCurrentScalingFactor;
      }
}
```

- Avoid global variables if possible, especially as a means of communication between program modules.

- As a generalization of the above, develop code that is highly modular and well encapsulated. Group logically-related functions and variables together into a single source file. In that file, only “expose” (i.e., declare non-static) the functions and global variables that truly need to be seen by other modules.

- Do not use “magic numbers”, i.e., constants that appear arbitrary. Use either `#define` statements or `const`-qualified variables to give these constants meaningful names. For example, imagine running across the following line in a program:

```
char bytes[523776];
```

  The above would certainly challenge interpretation. More meaningful is:

1. As a matter of interest, the variable name `shrimp_on_a_plate` was indeed encountered in the source code of a commercial application.

```
const int BytesPerSector = 512;
const int SectorsPerTrack = 1023;
char bytes[BytesPerSector * SectorsPerTrack];
```

- Comment well. Note, this did not say “comment a lot”. There are places where comments are appropriate, but too many comments can obscure the code. Well written, self-documenting code (see above) should not need too many comments. Good comments are ones that give the reader hints about the big picture, how a block of code does its job, issues that modifiers of the code must keep in mind, etc.

- Keep all comments up-to-date. It has been said that the only thing worse than inadequate documentation is documentation that no longer agrees with the code.

- Avoid the temptation to write highly-compact, (supposedly) highly-optimized code. It is much more important to write clear and legible code first, then to optimize the code that truly makes a difference to execution time.

- Keep all subroutines short, preferably to one screen height so you can see all of the subroutine’s code in your editor. Excessively long subroutines are hard to understand. Break out (i.e., *refactor*) logically-grouped chunks of your code into subroutines.

- Avoid excessive nesting levels. Going beyond the fourth or fifth indentation level indicates that the code is too complicated. Use additional subroutines to clarify the code.

11.0 Differences between Java and C

The C language proper can almost be considered a subset of Java. The differences between them can be very briefly summarized:

- C has no classes. All of the Java language features dealing with classes (e.g., `public`, `private`, `protected`, `interface`, `extends`, etc.) are not present. C, however, does have structures, which are essentially classes with public variables and no methods.

- The starting point of a C program is the `main` function. In Java, it is the `main` method of a class.

- C has pointers, Java does not. Java has references, C does not. In practice, references and pointers are exactly the same thing, but Java hides some of the complexity.

- Java automatically destroys objects once they are no longer needed (known as *garbage collection*). C does not. In C, you must explicitly destroy memory allocated for a dynamic object.

- In Java, you use the `import` statement to indicate you want to use library components. In C, you use the `#include` directive.

- C has the comma operator, Java does not.

- Java has the >>> operator, C does not.
- C has the `unsigned` built-in data type, Java does not.
- Java strings are objects, C strings are arrays of the `char` type and end with a 0 byte.
- C has a preprocessor, Java does not.
- Java `break` and `continue` statements can use label names, C does not allow this. However, C has the `goto` statement which is essentially equivalent. Java has no `goto` statement.