6. Introduction to the C Programming Language

Before we begin: a Quick Preview of a C Program

Although actually programming in C starts in Chapter 11 after a thorough introduction to the background of C and several necessary basics in these next few chapters, here is a “teaser” sample of a C program:

```
#include <stdio.h>

static unsigned fibo(unsigned n);

void main(void)
{
    printf("Hello World. Fibonacci(4) = %d\n", fibo(4));
}

static unsigned fibo(unsigned n);
{
    if (n <= 1)
        return 1;
    return fibo(n-1) + fibo(n-2);
}
```

(The line numbers on the left column are for explanatory purposes, and are not part of the actual program text.)

If you have followed any programming language tutorials, you might have encountered printing out “Hello World” as the first program. That practice was in fact popularized by the book “The C Programming Language” by Kernighan and Ritchie (we will hear more about them later). This version does that too, and also adds a function call to a recursive function that computes the fibonacci number for the input value of 4.

The Fibonacci sequence is the sequence 1, 1, 2, 3, 5, 8, 13, 21, ... where the next number of the sequence is the sum of the previous two numbers in the sequence. It is related to the Golden Ratio and is usually defined recursively as:

\[
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
\]

---

1 Yay! Finally! :D
2 Recursion is when you define a function based on itself, with some terminating condition to stop the recursion from running while never ending.

PRELIMINARY - ALL RIGHTS RESERVED

Copyright © 2014 ImageCraft Creations Inc. Modified Date: July 17, 2014
If you are not yet familiar with C code, this code sample may look somewhat daunting, so here’s a summary:

1. At line 1, the `#include` line is a C Preprocessor directive that directs it to textually “dump” the content of a named file (“stdio.h” in this case) into this spot in the code. The purpose here is to obtain the function prototype of the library function `printf` so that the program can print its result.

2. At line 2, a static function called `fibo` is declared (it is defined further down in the code), along with its return value type (the `unsigned` preceding the function name) and parameter type (the `unsigned n` within the parentheses).

3. At line 4, the `main` function is defined. All C programs must have a function named `main`, which is the start of the programmer’s own code.

4. At line 6, the `main` function calls the library function `printf` to print out the message “Hello World. Fibonacci(4) = 5” if the program is running correctly. For embedded systems, we will need to arrange for the output to appear somewhere. The `printf` call contains a nested function call to the `fibo` function, which in this case is being called to calculate the Fibonacci value of 4. (In the print statement, `%d` will print the integer result of the call to `fibo`.)

5. At line 9 to line 14, the `fibo` function is defined. It is a direct translation into C syntax of the recursive mathematical definition of the Fibonacci function. Lines 11 and 12 contain the terminating condition of this function (return “1” when the number `fibo` is called with is less than or equal to 1.)

Now let’s get started on discovering C from the inside, outside, and all the way “back to the Dawn of Time”...

---

3 “void” means this function has no parameters.
This Chapter
This chapter introduces the C Programming Language by presenting some history and background, followed by a description of some of the basic syntactic elements used in writing programs in C.

Additionally, the major semantic elements of the C Language can be broken down into these four parts:

- Types - the data types supported by the language
- Declarations - how to declare variables, data types, and functions
- Functions - how to define functions and their parameters
- Expressions - a list of the C operators and how to write expressions

The four chapters following this one will explore each of the above parts in detail, presenting C grammar and syntax as needed using a modified Backus-Naur Form (BNF⁴; see later in this chapter) with explanatory text. The goal is that these five chapters may serve both as tutorials and reference in the future.

---
⁴ Not to be confused with the social term "BFF", or “Best Friends Forever”. :p
C Standards

The definitive technical description of this language is the “ISO C Standard.” You can find that on
the web by using search terms, but in order to actually see the description of the Standard, you
must pay a fee to the Standards Organization\textsuperscript{5}. Luckily, there are also various FAQs,
summaries, and plenty of helpful information on the web and in printed literature that you can
peruse for free.

There have been three major releases of the C Standard. Summarizing, with emphasis on
language differences (standard library differences are omitted):

- ISO C89/C90\textsuperscript{6} - the first C Standard; 1989. This mostly codified existing practices and
  resolved some of the differences between compiler implementations.
- C99 (ratified in 1999) - This release added inline functions, interspersed declarations and
  statements, variable-length arrays, and // as a single line comment delimiter.
- C11 (2011) - This release added new keywords and macros for alignment,
  thread-local-storage, and anonymous struct / union.

Most embedded C compilers implement C89/C99. This book describes C as implemented by
the ImageCraft JumpStarter C compilers, which implement C89 with some C99/C11 features,
including // comment and anonymous struct / union. Additional C99 features are also in line to be
added.\textsuperscript{7}

\textsuperscript{5} Almost as if they want to encourage people to use a non-standard C instead. Short-sighted, much?
\textsuperscript{6} Originally it was ANSI C89, an American standard. When it was adopted by the International Standards
  Organization (ISO), it became ISO C90, but it’s the same standard.
\textsuperscript{7} Other implementation choices that are not applicable will not be discussed here; for example, ImageCraft
  JumpStarter compilers do not support multi-byte characters or non-US locales.
Classes of Programming Languages

C is a *procedural language*. Procedural languages are a subset of a class called *imperative programming languages*. In an imperative language, a source statement specifies changes in the program state (e.g. add x to y). A procedural language is an imperative language that groups sets of statements into routines (sometimes also call procedures or functions ⁸). Referring back to the stored-program machines described in Chapter 2, <2. Fundamentals of CPU Architecture>, it is clear that support for procedural languages was a major goal in the invention of stored-program machines.

For contrast, the other major programming language class is *declarative languages*, where a program describes what needs to be done, but not the “internal steps” on how it is done. *Declarative languages* include the subsets *Functional languages* and *Logic Programming languages*. *Functional languages* include pure LISP and Haskell. In a pure functional language, a program has no program states (i.e. no assignments and therefore no mutable objects), and computations are viewed as evaluations of mathematical functions. Prolog is an example of a *Logic programming language*. In Prolog, you write logic statements to describe the problems to be solved. Declarative languages feature powerful concepts suitable for certain specific types of programming, but it is beyond the scope of this book to explore them further.

The essence of programming in a *procedural language* involves defining *data structures* in which to store the program states, and writing statements to implement *algorithms* which use and (as necessary) modify the data. The classic text book “data structures + algorithms = programs” by Niklaus Wirth is a concise introduction to the approach, and is still relevant today even though the book was written in 1976. Data structures and introduction to algorithms are also covered in <Advanced Chapters> in this book.

The term *object-oriented programming* (OOP) refers to an extension of procedural languages where objects and message passing are used instead of traditional data structures and function calls. Most modern programming languages (which are mainly procedural languages) such as C++, C#, Java, Python etc. all incorporate object-oriented features.

While standard C does not have direct language support for object-oriented programming support, one can add the basic features of OOP to C programs by using function pointers defined within *struct* definitions. This will also be covered in <Advanced Chapters>.

---

⁸ “Routine” is a generic term. Some computer languages differentiate between a “procedure” and a “function”: a procedure might have side effects or program state changes inside the routine, while a function does not modify any program states. In C, however, no such distinctions are made, and “routine”, “procedure”, and “function” are synonymous. This book generally prefers the term “function”.

PRELIMINARY - ALL RIGHTS RESERVED

Copyright © 2014 ImageCraft Creations Inc. Modified Date: July 17, 2014
A Brief History of C

In the 1960s, Bell Labs, the research arm of American Telephone and Telegraph\(^9\), was engaged in some of the most important computer science research of the time. After working on a mainframe operating system project called “Multics” (which Bell Labs finally pulled out of), Ken Thompson, a researcher at Bell Labs, decided to implement the best ideas of Multics using Assembly Language for Digital Equipment Corp’s PDP-7 minicomputer.

At that time, paper tape - not even punch cards! - was the standard type of program storage unit. The code was first developed on a GE-635, then the output paper tapes were carried over to the PDP-7 for processing, until enough of the system was working on the PDP-7 to enable native development (although still in Assembly).

Ken Thompson then began working on a compiler for a language he called B, the name being attributed either to the language BCPL (to which B bears some similarities) or Bon, an earlier language which Thompson had written for Multics\(^10\). He then rewrote part of the as-yet unnamed new OS in B.

In 1970, the Bell Labs researchers managed to convince Bell Labs management to procure a PDP-11, a much more powerful machine, in order to implement a “text processing system.” Along the way, the name “Unix” was adopted for the new operating system by Brian W. Kernighan, attributed either to being a play-on-words of “Multics”, or even possibly a tongue-in-cheek reference to “eunuchs”. Meanwhile, Dennis M. Ritchie, while working with B on the PDP-11 Unix, decided to add some needed features to B, and picking the next letter from BCPL, he called this new language C. Using a procedure called “bootstrapping”, he first wrote a prototype C compiler in B, after which he rewrote the compiler in C itself, adding more features at each iteration as needed.

By 1973 and 1974, C was sufficiently mature enough that Unix and its utilities were entirely rewritten in C. Remarkably, this version of C still bears a great resemblance to even the latest Standard C, a testament to how well the original C language was designed. During the same period, C was retargeted to the Honeywell 635 and IBM 360/370, and through that experience, C adopted features and encouraged practices that have improved the portability of C programs, which in turn, contributes to the success of the language today.

Kernighan and Ritchie published the book “The C Programming Language” in 1978, arguably considered “the C Bible” even to this day (a “version 2” was published in 1988). During the same time period, Steve Johnson wrote a reference C compiler called the Portable C Compiler. Unix and C were ported to the Interdata 8/32, and then to the VAX-11 in the late 70s, Due to some

---

\(^9\) The original AT&T, with a monopoly on phone service, is the foremother (“Ma Bell”) of the current 2000s AT&T cell phone carrier

\(^10\) B and BCPL have similar operators to C, but they have no concept of data types.
anti-monopoly agreement with the US government, the Unix source was given to the University of California Berkeley, which they (mainly a programmer named Bill Joy, who went on to co-found the legendary company Sun Microsystems) modified to become BSD Unix. The various AT&T Unix and BSD Unix and C versions influenced an MIT hacker named Richard Stallman (around 1984) to work on “free” versions of the compiler and operating system, which eventually grew to become the set of GNU software, including the GCC suite of (non-standard C) compilers. The GNU efforts in turn influenced a young Finnish student named Linus Torvald (in 1992) to write a “toy” operating system that eventually became Linux.

Meanwhile, back in the early 1980s, commercial C compilers started to appear for chip targets such as Motorola 68K, Intel 8086, and even small microcontrollers such as the Motorola 68HC11 and Zilog Z80. P.J. Plauger, a researcher at Bell Labs, left to found Whitesmiths Ltd. and produced one of the first commercial C compilers outside of Bell Labs with wholly independently-developed compilers.

In the 1980s, C compiler companies practically blossomed like daisies after a spring rain. Borland’s Turbo C blew the market open with its low price, but eventually the Windows compiler market coalesced to mainly just Microsoft Visual C (later Visual Studio). The Mac market was dominated by Think C, then Code Warrior C, until Mac changed to the Intel x86 processors in the late 90s.

C became the “lingua franca” programming language for controllers from small embedded systems to large mainframes. The Bell Labs researchers eventually followed Unix/C with Plan 9 and then the commercial Inferno OS and the Limbo language, but they did not achieve the earlier successes of Unix and C.

The timely popularity of many computer languages has often been fickle. After the computer industry collectively embraced C/C++ in the 90s, its attention now seems poised to fragment again, into Go (Google), Swift (Apple), C#, Java (Android), Objective C (Mac OS X and iOS, although with Apple working on Swift, the writing is on the wall for Objective C), and a smattering of scripting / interpreting languages for the web such as PHP, Python, Ruby, and Perl.

For Embedded Systems though, C is STILL the most popular high-level programming language, even for the 32-bit segment. As long as price and performance are the prime driving factor in the Embedded space, C’s future is still assured.
Order of Translation

A C compiler consists of multiple programs that transform the C source files from one format to another. Starting with a .c source file, and moving from one row to the next, ImageCraft’s JumpStarter C for Cortex-M compiler goes through these stages:

<table>
<thead>
<tr>
<th>Input File Type</th>
<th>Program that Processes the File</th>
<th>Output File Type</th>
<th>Output File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>C source file</td>
<td>C Preprocessor icppw.exe</td>
<td>C source file without # directives</td>
<td>.i</td>
</tr>
<tr>
<td>Preprocessed C source file (also known as a translation unit)</td>
<td>C Compiler Proper icomcortex.exe</td>
<td>Cortex-M assembly language file(^{11})</td>
<td>.s</td>
</tr>
<tr>
<td>Cortex-M assembly</td>
<td>Assembler iascortex.exe</td>
<td>Object file</td>
<td>.o</td>
</tr>
<tr>
<td>Object files and library files</td>
<td>Linker ilinkcortex.exe</td>
<td>Output “executable”</td>
<td>.bin .hex</td>
</tr>
</tbody>
</table>

Some compilers eliminate the “assembler” step with the compiler proper generating object files directly, but this breakdown reflects the behavior of most C compilers.

In an embedded system, the output executable is loaded onto the target device. For example, on the Cortex-M3 device, you might use a JTAG pod to download the program to the target flash memory.

C Preprocessor

A C source line that starts with the # symbol is a preprocessor directive, and it is processed by the C preprocessor (see <XYZ>). Certain text in the source file might be textually replaced by other content if it is affected by such directives. For example:

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

replaces the line with the contents of the file “stdio.h”. By convention, files that are to be #include’d by C source files use a .h extension.

The C preprocessor is described in its own chapter at <XYZ>.

C Compiler Proper

The C compiler proper translates C into assembly language.

\(^{11}\) For a quick Assembly Language description, see Chapter 2.
Assembler
The assembler translates an assembly language file into an object file.

Linker
The linker combines any number of object files with the required library files to form the final output executable (this is the file the target machine actually runs!)

Source File Structure
Typically, a program is written as a set of functions. Related functions are by convention grouped into the same source file. Thus, a project usually contains a set of source files. **NOTE: the C compiler proper only “sees” one source file at a time.**

To communicate information about one source file (for example, what functions are available in the source file), a header file is usually used. The header file is inserted into a source file using the `#include` directive (described above).

IDE and IDE Project File List
A “compiler program” usually comes packaged with an Integrated Development Environment (IDE). This is the GUI (Graphical User Interface) program that the users use to manage their programming projects. ImageCraft’s JumpStarter C’s GUI is a program called “CodeBlocks” which is an easy-to-use interface to the underlying compiler programs. CodeBlocks and the rest of the JumpStarter C compiler programs are described later in Chapter <XYZ>.

It is important to note that the CodeBlocks (or any) IDE is separate from the C compiler itself. That is, while it understands C constructs (i.e. for symbol browsing and syntax highlighting), it is not part of the traditional program translation chain, and the compiler proper still only sees one source file at a time.
The Elements of a C Source File

While the only official ordering requirement in C is that an object must be declared before it is referenced, a C source file is generally divided into the following lexical ordered sections. The C language does not dictate these divisions, but this ordering guideline provides a logical method to achieve more maintainable source files:

- `#include` C preprocessor directives
- `#define` C preprocessor macros
- Global variable declarations
- Function declarations
- Function definitions.

A function definition contains the executable statements. We will explore them later.
Indentations and White Spaces
Unlike some languages, indentations and white spaces have no effect on the meaning of a C program. Carriage returns affect C preprocessor macros, but not the C syntax itself.
Character Sets
The base character set for Standard C is ASCII: basically, the English alphabetic characters and digits and all the punctuation that one may find on a standard U.S. keyboard. Standard C also has the *locales* concept, where non-ASCII (see <1. Fundamentals of Computer Arithmetic>) characters can be used. (This book does not discuss locales much, as the ImageCraft compiler currently only supports the English locale.)

The following sets of ASCII characters are accepted:

<table>
<thead>
<tr>
<th>Character Set</th>
<th>Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>letters</td>
<td>A to Z, and a-z</td>
</tr>
<tr>
<td>digits</td>
<td>0 to 9</td>
</tr>
<tr>
<td>underscore</td>
<td>_</td>
</tr>
<tr>
<td>punctuation marks</td>
<td>! &quot; # % &amp; ' ( ) + , . / : &lt; = &gt; ? [ ] { } ~</td>
</tr>
<tr>
<td>non-printable graphics (used inside string literals or character constants)</td>
<td>space, BEL - bell, BS - back space, FF - form feed (go to top of page), NL - newline (cursor down), CR - carriage return (go to start of line), HT - horizontal tab, VT - vertical tab</td>
</tr>
</tbody>
</table>

Escape Sequences
You can use escape sequences to represent characters that are otherwise cannot be written in a character constant or a string literal. For example, you may want to include a double quote " as part of a string literal (which is written inside a set of double quotes). An escape sequence is also the only way to specify the non-visible graphic characters above. An escape sequence starts with the backslash \ followed by a character or sequence of digits:

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>BEL</td>
<td>\a</td>
</tr>
</tbody>
</table>
A numeric escape sequence is a \ followed by a sequence of 0-7. The value is the sequence of digits in base 8. For example, \123 is octal 123, or the decimal value 83. Since the sequence is in octal, the maximum number of valid digits is 3, with the maximum value of \777 or 511 (in decimal), the largest integer that can be fit into a byte.

A hexadecimal escape sequence is a \x followed by a sequence of 0-9, a-f, and A-F. The value is the sequence in base 16 with a or A equal to 10, b or B equal to 11, and so on. For example, \123 is hexadecimal 123.

An invalid base character will terminate the sequence, e.g. “\128” is 2 bytes, the escape sequence of \12 (or 10 in decimal), and the character 8. Since ‘8’ is not a valid octal digit, it ends the escape sequence.

Diacritical Characters
Another example of using escape sequences is for writing the occasional diacritical character in a string literal. For example, “álcançar” may not be correctly compiled by the compiler, but you can write “álcan\120ar” instead. Note that you should use the octal sequence \200 and not the otherwise equivalent hex sequence \x80, as the compiler would interpret the following ‘a’ as part of the hex sequence.

---

12 See Modified BNF form below for meanings of { } and +.
Modified Backus-Naur Form (BNF)
BNF is a notation for describing the grammar, or syntax, of computer languages. This book uses a variant of BNF to describe C syntax. BNF definition statements are written in the following form:

\[
\text{nonterminal} = \text{expression};
\]

Examples:
- \(\text{digits} = (0..9)^+;\)
- \(\text{alphabets} = (a..z | A..Z)^+;\)
- \(\text{float} = \text{digits} [ \, \{ . \} \, \text{digits} ];\)

Expressions
A BNF expression is recursively defined as:
- nonterminal
- terminal
- sequence
- expression*  
- expression+
- ( expression )
- [ expression | expression ]
- ( expression | expression )
- ?..?

Each of these are explained below.

A nonterminal is the name of an element / construct in the grammar, for example, \(\text{digits}\). For a BNF grammar to be complete, all nonterminals must have a corresponding definition statement.

Examples:
- \(\text{digits}\)
- \(\text{alphabets}\)
- \(\text{float}\)

A terminal is a word that appears in the language. A terminal is written as characters inside a set of single or double quotes. This book uses the convention that single letter terminal is enclosed by single quotes and multi-letter terminal is enclosed by double quotes.

Examples:
- "int"
A sequence is a list of expressions. The listed expressions appear in order of the sequence.

Example:

```c
void
```

digits . digits

The * in expression* is a special symbol meaning that the preceding expression appears zero or more times.

Example:

```c
"hello"*
```

digits

this matches:
- (nothing)
- "hello"
- "hello" "hello"
- "hello" "hello" "hello"
  ...

The + in expression+ is a special symbol meaning that the preceding expression appears one or more times.

Example:

```c
"hello"+
```

digits

this matches:
- "hello"
- "hello" "hello"
- "hello" "hello" "hello"
  ...

The ( ) in ( expression ) is a special symbol grouping an expression, mostly useful for applying the * or + iteration special symbol, or to specify alternation (see below).

Example:

```c
("hello" "world")+
```

digits

this matches:
- "hello" "world"
- "hello" "world" "hello" "world"
“hello” “world” “hello” “world” “hello” “world”
...
...

The [ ] in [ expression ] is a special symbol grouping an expression. It differs from the ( ) special symbol in that the expression is optional.

Examples:
[“hello”]

this matches:
- (nothing)
“hello”

The | in [ expression | expression ] or ( expression | expression ) is a special symbol specifying an alternation. It must be inside either the [ ] or ( ) special symbol. It means that either the left expression or the right expression may appear. Multiple | may appear inside the bracketed symbols.

Examples:
[“hello” | “world”]

this matches:
- (nothing)
“hello”
“world”

------------------------
(“hello” | “world” | “xyz” )

this matches:
“hello”
“world”
“xyz”

The ?..? is a special symbol specifying a range. It specifies any terminal character within the range. A range must be either alphabetic characters of the same case or digits in ascending order.

Examples:

13 While [ expression ]+ does mean that the expression may appear once or more times, for consistency sake, it should be written as ( expression )+
0..9 matches the digits 0 to 9
a..z matches the lower case letters ‘a’ to ‘z’
A..D matches the upper case letters ‘A’ to ‘D’
C Tokens and Operators
The C compiler parses a preprocessed C source file into a stream of C tokens, collectively known as a translation unit. In compiler-speak, this process is known as “lexical analysis”. C is case sensitive, e.g. “case” is different from “Case”. In the descriptions below, when used in an integer or floating point constant, any use of an alphabetic character, e.g. ‘a’, implicitly allows the use of the capitalized version, e.g. ‘A’. In all other uses, however, case matters.

This section describes the syntax only. The semantics (i.e. the meaning) of the language are explained in the next chapters.

A valid C token belongs to one of these classes:

- Keywords - these are words that have special meanings in the C language:

  auto   break   case   char   const
  continue default do double else
  enum extern float for goto
  if int long register return
  short signed sizeof static struct
  switch typedef union unsigned void
  volatile while

  The ImageCraft compilers also add the following extended keywords:

  __flash   __firstarg
  __typecode

- Names or identifiers
  An identifier starts with either a letter or an underscore, followed by letters, digits, or underscores. An identifier must NOT match a C keyword listed above.

  To be fully conformant with Standard C (i.e., in case there is an ancient compiler out there), a C external identifier length should not exceed some limited small amount \(14\) , but most if not all modern compilers - JumpStarter C included - accept identifier lengths of at least 128 characters or even longer. While most programmers would not exceed the 31 character limit, having a longer limit available is useful for programs that emit C programs as output and use machine-generated long identifier names in the output.

  \[
  \text{identifier} = (\text{letter} \mid \_') [\text{letter} \mid \_ \mid \text{digit}]^*;
  \]

  Examples:

---

\(14\) It was pitifully short at 6 characters in C89 (hence you have names like “v1”, “v2”, “v3” etc.); later C Standards extended it to 31 characters.
foo
ThisIsCalledCamelCase
this_is_a_long_name
_314159267
Int ← this is OK (differs from “int” due to capital ‘I’
but is not a recommended practice
x ← a popular variable name, but usage should be discouraged

• Integer constants
An integer constant is either a binary, decimal, octal, or a hexadecimal. Binary is a
common extension that ImageCraft compilers support, and binary numbers are denoted
by the prefix “0b”. Octal numbers are denoted by a ‘0’ prefix and hexadecimals are
denoted by a “0x” prefix. ‘a’ through ‘f’ are valid hexadecimal digits with ‘a’ representing 10
and ‘b’ for 11 and so on\textsuperscript{15}.

An integer constant by default has the signed int data type. You may add a ‘u’ suffix to
indicate unsigned data type, or an ‘l’ to indicate long data type. You may combine ‘u’ and
‘l’ together.

\int\_const = (\text{bin} | \text{dec} | \text{oct} | \text{hex}) [\text{int}\_\text{suffix}];

\text{bin} = "0b" (0|1)*;
\text{dec} = (1-9)[\text{digit}]*;
\text{oct} = 0[\text{digit}]*;
\text{hex} = "0x"[\text{digit} | \text{a}_\text{to}_f]+;

\text{int}\_\text{suffix} = ('u' | 'ul' | 'l' | 'll' | 'ull');

Examples:
10
65000u
1L
42u1

• Floating point constants
A floating point constant is in either decimal or scientific notation. By default, a floating
point constant has the double\textsuperscript{16} data type, unless it is followed by the ‘f’ suffix, then it has
the float data type. The optional ‘l’ suffix means that the constant has the long double data
type, which is the same as the double data type under ImageCraft compilers.

\textsuperscript{15} For more detail, see Chapter 1.
\textsuperscript{16} See Chapter 7 for full descriptions of data types.

PRELIMINARY - ALL RIGHTS RESERVED

Copyright © 2014 ImageCraft Creations Inc. Modified Date: July 17, 2014
`float const = digit* '.' [digit]* [e_notation] [float_suffix];`

`e_notation = 'e' ['+' | '-' ] digit*;`

`float_suffix = ('f' | 'l');`

Examples:
3.14159
1f
0.31415926e1

- **Character constants**
  A character constant represents a byte value with a character or escape sequence code, e.g. ‘a’.

  `char const = "" (character | escape_sequence) "";`

  An escape sequence is a character code starting the the backslash character \, e.g. `\", \b` (see above for escape sequence), `\x123` etc.

  Examples:
  ‘a’
  ‘\200’
  ‘\b’

- **String literals**
  A string literal is a sequence of characters enclosed in double quotes, e.g. “hello world”
  The characters are laid out in memory in increasing addresses. A C string literal has an
  implicit ending zero (0) as the last byte. For example, “abc” occupies 4 bytes, ‘a’ ‘b’ ‘c’
  and 0.

  `string = "" (c_character_set | escape_sequence)+ "";`

  Examples:
  “Hello World”
  “Embedded Zero \0 Inside a String. Watch out when using strlen”
  “\200 Nonprintables: \b\g”

- **Wide character string literals**
  You write a *wide character* string literal by prefixing the string with the character L, e.g.

  L”Hello World”

- **Operators and punctuation**
An operator specifies a mathematical, or logical, or computer operation. In source code, some multi-character operators must have other text located between them, e.g. “[5]” while others do not, e.g. “&&”. The set of valid C operators and punctuation are:

Mathematical:
+  -  *  /  %  ++  --

Shift
<<  >>

Assignment
=  +=  -=  *=  /=  %=  ^=  <<=  >>=

Bitwise or logical AND and OR
&  |  ^  ~

Boolean and comparison
&&  ||  !  ==  !=  <  >  <=  >=  ?:?

Miscellaneous
[ ] ( ) *  .  ->  ,  sizeof

Punctuation
{}  …  ;  :