
# Programming in C

## A Simple Introduction

Version 1.0

**Frank Vahid**
University of California, Riverside

**Tony Givargis**
University of California, Irvine

UniWorld Publishing, Lake Forest, California

Version 1.0, January 2012.

Formatted for electronic publication.



ISBN 978-0-9829626-3-3 (e-book)

UniWorld Publishing

www.programmingembeddedsystems.com
info@programmingembeddedsystems.com

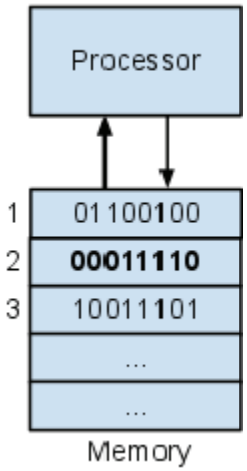
# Contents

# Chapter 1 -- Introduction

## C language background


1940s computer
(U.S. Army photo)

A ***processor***, the main component in a computer, is an electronic device that executes a list of instructions called a ***program***. Writing those instructions is called *programming*. In the 1940s, people wrote programs using ***machine instructions*** consisting of 0s and 1s, by flipping switches up or down on the front of a computer. For example, the machine instruction "0001 11 10" might instruct the processor to add ("0001") the data stored in memory location 2 ("10") to the data stored in location 3 ("11"). ***Memory*** is a component that accompanies a processor and that can store information, such as instructions, in numbered locations, as in the figure on the right. Information is stored using bits (short for "binary digit"), each bit being either 0 or 1. Each memory location in the figure is 8 bits wide. A processor might support a few hundred instruction types, some for moving data from one memory location to another, for operating (adding, subtracting, etc.) on data, and for telling the processor to jump to some other instruction in the list rather than executing the next instruction. In the 1950s and 1960s, people began programming using a more-readable representation of machine instructions known as ***assembly instructions***, where each instruction was represented using text, and another program called an ***assembler*** automatically translates those textual instructions to machine instructions. For example, a human might write the assembly instruction "ADD R3 R2", which an assembler would convert to the machine instruction "0001 11 10", as in the figure at memory location 2. In the 1960s and 1970s, ***high-level languages*** became popular to support programming using formulas or algorithms, such as the FORTAN (for "Formula Translator") or ALGOL (for "Algorithmic Language") languages, which were more closely related to how humans thought than were machine or assembly instructions. Another program called a ***compiler*** automatically translates high-level language programs into assembly instructions. For example, a human might write "X = Y + 2*Z", which a compiler would convert to perhaps five assembly instructions, one of which might be "ADD R3 R2".




In 1978, Brian Kernighan and Dennis Ritchie at AT&T Bell Labs published a book describing a new high-level language with the simple name ***C***, being named after another language called *B* (whose name came from a language called *BCPL*). C was the dominant programming language in the 1980s and 1990s. In desktop computer programming in the 1990s and 2000s, C was largely replaced by C-inspired languages like C++ and C# and other languages like Java, due to those languages supporting an object-oriented programming approach conducive to creating larger programs.

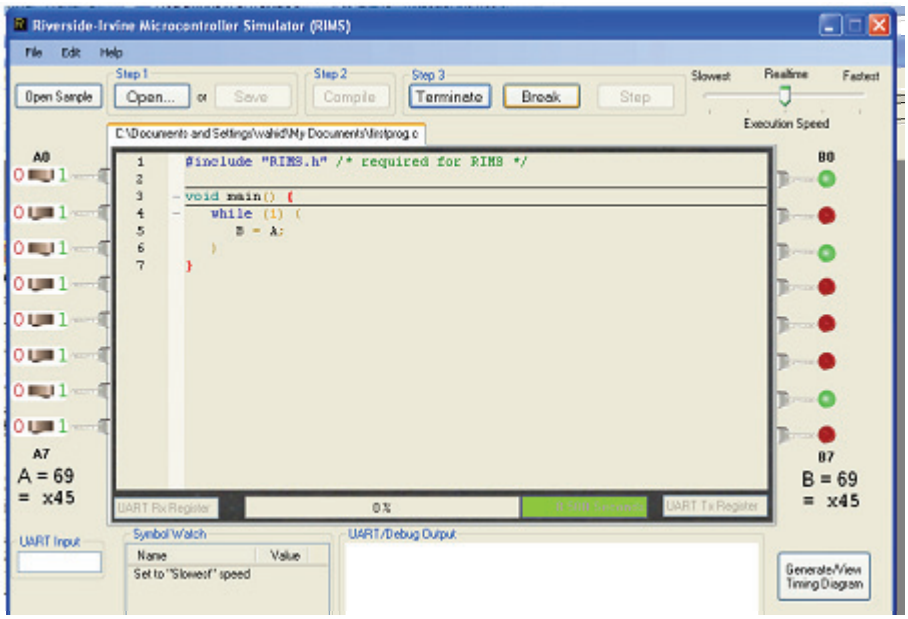
C is still a popular language for programming processors in embedded systems. An *embedded system* is a computing system, including a processor, embedded within larger electronic devices such as automobiles or medical equipment (as opposed to within desktop computers). Hundreds of times more embedded systems exist than desktop computers, and those embedded systems processors must be programmed. C is often preferred over object-oriented languages for programming embedded systems due C compilers typically generating size- and performance-efficient assembly programs. Such efficiency comes from C's simplicity. Efficiency is important in embedded systems because their processors are commonly low cost, meaning limited memory for instructions and data and fewer instructions executed per second; efficiency enables good use of such processors.

C is also commonly used for programming operating systems, compilers, and scientific computations. Several newer languages were derived from C and thus share many features with C.

Furthermore, C is still popular as a first language for learning programming. People may become better programmers when first learning to program in a language that requires some knowledge of processor and memory aspects, before learning more advanced languages that hide more processor and memory aspects behind powerful language constructs. Furthermore, learning C first can improve later learning of C-inspired languages like C++, C# (used for Microsoft Windows application programming), Objective-C (used for Apple Mac/iPhone/iPad programming), and others.

This book teaches programming, using a common subset of C. The book does not describe all features of C, which can be found in C language manuals/books and online. The book uses a concise writing approach to enhance learning.

## A first program in RIMS




To aid the learning of C programming, this book uses a processor known as ***RIM***, short for Riverside-Irvine Microcontroller. A microcontroller is single-chip integrated circuit (IC) commonly used in embedded systems, typically being small, low-cost, and low-power, and consisting of a simple processor, memory, certain peripherals, and input and output pins that a program can read or write. RIM has 8 input pins (A0-A7, collectively called A) and 8 output pins (B0-B7, collectively called B); each pin can have the value 0 or 1, representing a low voltage near 0 or a higher voltage typically near 3 V, respectively. ***RIMS***

(short for RIM Simulator, available at www.programmingembeddedsystems.com) is a graphical simulator that depicts a virtual RIM IC, with each input pin connected to a virtual switch that can be set to provide a 0 or 1 value to the input pin, and with each output pin connected to a virtual LED (or light) that is red when the output pin is 0 and green when 1. A person can write a C program inside the virtual RIM IC and run the program, moving switches to change inputs to the program, and observing the program's outputs via the LED colors.

Try: Run RIMS. Enter the following C program inside the virtual RIM IC.

```
#include "RIMS.h" /* required for RIMS */

void main() {
   while (1) {
      B = A;
   }
}
```

Press "Save" and name the file "test1.c". The ".c" suffix is commonly used for C files. Press "Compile" to translate to RIM assembly instructions (which aren't visible). Press "Run" to execute the program. Click any switch on the left to change it from 0 to 1, and notice that the running program, which copies the input pins. values to the output pins (via "B = A;"), causes the corresponding LED on the right to turn green. Click various switches multiple times and observe the LEDs. When done, press "Terminate".

For subsequent examples that ask you to run a C program, the above "Save", "Compile", and "Run" steps should be followed.

In the above C program, '#include "RIMS.h"' incorporates some items needed by RIMS, such as definitions for inputs A0-A7 and A and outputs B0-B7 and B. More on "#include" later.

The text between "/*" and "*/" is called a ***comment***. The compiler skips comments. A programmer may optionally use comments to explain program parts to people. A comment may span multiple lines, as in:

```
/* This comment spans
   two lines */
```

A ***line*** is the term used for a text row in a program.

*A C program starts by executing a function called* ***main***. A function is a list of statements. In the above program's "void main()" line, "()" means the function has no parameters and "void" means the function returns nothing; more on these items later. The main function's statements appear between the first "{" and the last "}", known as ***opening*** and ***closing brackets***, respectively. Such curly brackets are used to denote a list of statements.

The main function's first statement is "while (1)", which means to repeatedly execute the statement's

sub-statements. The sub-statements are contained between the subsequent opening and closing brackets. Such a "while" (1) statement is commonplace in a C program's main function, especially in embedded systems. More on while statements later.

The program's unique behavior comes next, consisting of one statement "B = A;" which assigns the value of input pins A to the output pins B. More on *assignment statements* later. The ***semicolon ";"*** indicates the end of the statement.

New lines and extra spaces (more than one space) between words, each known as ***whitespace***, are not relevant in C functions. For example, the above main function could have been written as:

```
void    main(  ) {  while (1) {  B       = A;  }}
```

but such code would be considered poor style. The term ***code*** commonly refers to the textual representation of a program, and programming is thus sometimes called *coding*.

## Sequential execution of statements

The following program has multiple assignment statements. Each statement is executed sequentially, meaning one at a time.

```
#include "RIMS.h"

void main() {
   while (1) {
      B0 = A0;
      B1 = A1;
      B2 = A2;
      B3 = A3;
      B4 = A4;
      B5 = A5;
      B6 = A6;
      B7 = A7;
   }
}
```

The program first assigns input A0's value to B0. The program then assigns A1's value to B1. And so on.

Try: Run the above program. The program's behavior should appear identical to the earlier program having just "B = A;".

While the running program appears to execute all the statements simultaneously, in fact the program executes the statements sequentially. A processor has a clock input, where a ***clock*** is a signal that "ticks" at a particular frequency such as 1 MHz, meaning 1 megahertz or 1 million ticks per second. A typical processor can execute approximately one machine instruction every clock tick, meaning one machine instruction every microsecond for a 1 MHz clock.

In RIMS, pressing "Break" stops the RIM processor's clock. A triangle appears next to the C statement where the processor was stopped. Then, pressing "Step" generates a single clock tick, causing one machine instruction to execute. *Because each C statement may have been compiled to several machine instructions, you may have to press "Step" several times before the current C statement finishes executing and the triangle moves to the next C statement.* Breaking and then stepping gives you control of the RIM processor's clock such that you can observe each C statement's execution.

Try: Run the above program in RIMS and set all input switches to 0, then press "Break" to stop the RIM processor. Notice the triangle that appears. Press "Step" until the triangle moves to the next statement. Do so until the triangle is on the last assignment statement, then change all the switch positions from 0 to 1, and notice (by the LED colors) that the output pin values do not change, because the processor is stopped. Now press "Step" several times to generate processor clock ticks and notice that the triangle jumps back (because of the "while" statement) to the statement "B0 = A0;" and notice that B0 finally changes after that statement is executed. Step more and notice that B1 changes after the "B1 = A1;" statement executes. Continue stepping until all output values have changed to 1.

A common misconception among new programmers is that an assignment statement somehow permanently connects the items on the left and right sides of the "=" operator. Such permanent connection matches the use of "=" in mathematics, indicating that the left and right sides are the equivalent. However, in C, "=" implies no such connection or equivalency. Rather, when the assignment statement is executed, the *current value* of the statement's right side is simply *copied* (or *assigned*) to the item on the left side. Later changes of the item on the right side (like above when the A values were changed by moving the switches) would have no impact on the item on the left side, until perhaps a later assignment statement was executed. An arrow pointing to the left might be a better symbol than "=" to convey the meaning of assignment, and thus some other languages use "<=" or ":=" to represent assignment rather than just "=" as in C.

## Syntax errors and compiler messages

Sometimes a program contains an error. One kind of error is a ***syntax error***, wherein the programmer wrote code that is not legitimate C code. For example, the following code is missing the semicolon at the end of the "B = A" statement:

```
#include "RIMS.h"

void main() {
   while (1) {
      B = A
   }
}
```

Try: Save and then compile the above program. Notice that an error message appears in RIMS' bottom-right window, indicating a syntax error at a particular line number: found '}', expecting ';' Correct the error and compile again.

Note that the line number may be a few lines past where the actual syntax error occurred; the compiler may not notice the problem until reaching a later line, as in the above example where the compiler does not notice the missing semicolon until reaching the "}" on the subsequent line.

Another kind of error is a ***runtime error***, wherein a program's syntax is correct but the running program does not behave as the programmer intended. An example might be if a programmer accidentally typed "B = 1;" rather than "B = A;" -- the program would compile fine, but would not behave as intended.

*For brevity, subsequent examples may omit the '#include "RIMS.h"' line, but remember that the line is needed at the top of any RIMS program.*

# Chapter 2 -- Variables and Assignment Statements

## Variables, data types, and constants

A ***variable*** stores a data item in memory. RIMS.h predefines some variables like A and B, but most variables are defined by the programmer. Below is a program using a programmer-defined variable.

```
unsigned char x;

void main() {
   while (1) {
      x = A;
      B = x;
   }
}
```

The program defines a new variable named "x". The program assigns the value of A to x, and then assigns the value of x to B.

Try: Run the above program (don't forget to add to the top the line: #include "RIMS.h"). Set all input pins to 0, and press "Break". Press "Add symbols" under the "Symbol watch" section, and find and select "x" from the symbol list. Step until the triangle is at the line "x = A;". Change some input pins to 1s. Step until the triangle moves one line, and note that x's value changes, but B does not yet change; A's value has been stored in variable x. Step until the triangle moves again, and note that B is now updated with the value that was stored in x.

A variable definition must indicate the size of memory that a compiler should allocate for a variable, by preceding the variable name with words indicating the variable's *data type* as follows:

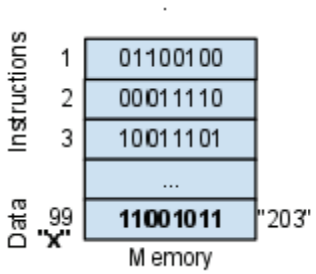
| | |
|---|---|
| ***unsigned char:*** | 8 bits (decimal range: 0 to 255) |
| ***unsigned short:*** | 16 bits (decimal range: 0 to 65,535) |
| ***unsigned long:*** | 32 bits (decimal range: 0 to 4,294,967,295) |

A variable is commonly used to store a decimal number like 203, also known as an *integer*. Because a memory only contains bits (0s and 1s), a processor stores numbers in base 2, known as ***binary*** numbers. In base 10, known as *decimal* numbers, each digit must be 0-9 and each digit's place is weighed by increasing powers of 10. So the decimal number 203 represents $2*10^2 + 0*10^1 + 3*10^0 = 2*100 + 0*10 + 3*1 = 200 + 0 + 3 = 203$. In base 2, each digit must be 0-1 and each digit's place is weighed by increasing powers of 2. So the binary number 1101 represents $1*2^3 + 1*2^2 + + 0*2^1 + 1*2\text{^} = 1*8 + 1*4 + 0*2 + 1*1 = 8 + 4 + 0 + 1 = 13$ (in base 10). A compiler translates decimal numbers into binary numbers. The decimal number 203 would be converted to the binary number 11001011 meaning $1*128 + 1*64 + 0*32 + 0*16 + 1*8 + 0*4 + 1*2 + 1*1 = 203$. Based on the bits for each size, the above three variable sizes can

represent the shown decimal ranges.

Unfortunately, C lacks a data type for representing a single bit. For example, RIMS' A0 variable can only be 0 or 1 and thus requires only a single bit of storage. In the absence of a single bit data type, programmers declare a single-bit variable as "unsigned char" but take care to only assign a 0 or 1 to that variable.

In summary, a variable declaration causes locations in memory to be allocated for the variable, as in the figure to the right for unsigned char x, for which the compiler has allocated memory location 99. (For a memory with 8 bits per location, the 16 or 32-bit variable types would require two or four locations).




An assignment statement may assign a variable the value of a ***constant***, which is a programmer-specified value. Below is a program using the *integer constant* "203":

```
unsigned char x;

void main() {
   while (1) {
      x = 203;
      B = x;
   }
}
```

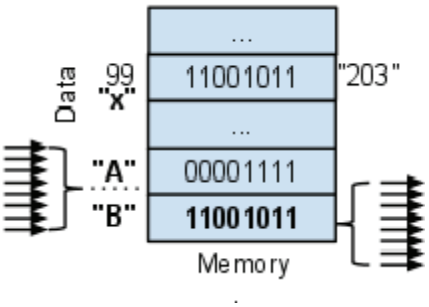
An assignment statement updates the contents of the variable's memory location, as in the figure.

RIMS shows the decimal values of input A and output B below the respective pins.

Try: Run the above program. Note that B gets assigned 203 (in binary).

The programmer must take care not to assign a variable with too large a number for the variable's size, else the program results may be incorrect. Assigning a value larger than a variable's size is called ***overflow***.

Try: Modify the above program to use the constant 260, and run the program. Because 260 is outside the 0-255 range of an unsigned char, note that program results are incorrect due to overflow, namely B gets a value other than 260.




In RIM, the special variables A and B can now be better understood -- they refer to special memory locations connected to RIM's 8 inputs and 8 outputs, respectively, as in the figure to the right. Thus, the above program assigned location x with 203, and then assigns location B to x's value, so B gets value 203 also.

Sometimes a variable should be able to store both positive and negative numbers. Such variables are defined as follows:

***signed char:*** 8 bits (decimal range: -128 to 127)
***signed short:*** 16 bits (decimal range: -32,768 to 32,767)
***signed long:*** 32 bits (decimal range: -2,147,483,648 to 2,147,483,647)

C allows for "signed" to be omitted, but for clarity we recommend that such shorthand be avoided. C also has an "***int***" type, but its width is not defined and thus we recommend avoiding that type's use. short and long types may optionally have the word "int" after short or long, but that word is superfluous so we omit it.

In addition to storing integer numbers, variables are sometimes used to store floating-point numbers like 1.05 or 6.02x10^23. Thus, two additional variable data types are:

***float:*** 32 bits (typical) (range: $3.4 \times 10^{-38}$ to $3.4 * 10^{38}$)
***double:*** 64 bits (typical) (range: $1.7 \times 10^{-308}$ to $1.7 * 10^{308}$)

The above ranges are typical, but C (unfortunately) does not actually define the number of bits for float and double types. A programmer should thus check the manual of their compiler. *RIMS does not presently support float or double.*

Above, the words "unsigned", "signed", "char", "int", "short", "long", "float", and "double" are ***reserved words*** in the language and thus cannot be used as a programmer-defined name. Reserved words are also known as ***keywords***.

Sometimes a variable should store a character like the letter 'm' or the symbol '%'. The compiler translates each possible character to a unique 8-bit number as defined by ASCII (See Wikipedia: ASCII). For example, 'm' is 01101101, or 109 in decimal. A *character constant* is written using single quotes as follows:

```
unsigned char y;

void main() {
   while (1) {
      y = 'm';
      B = y;
   }
}
```

Try: Run the above program. Add "y" to the symbols being watched ("Break", "Add symbols", then "Continue"). Note that y gets assigned 109, as does output B.

For debugging purposes (i.e., finding problems in a program or just seeing what a program is doing), RIMS.h includes the function ***putc***, which can be used as follows:

```
unsigned char y;

void main() {
   while (1) {
      y = 'm';
      B = y;
      putc(y);
   }
}
```

Invoking the function by name as above is referred to as a *function call*. The programmer places a variable of "char" type between the function call's parentheses, and the function will then print (or "put") the variable's character in RIMS' debug output. In the function name "putc", the "c" stands for "character". As a sidenote, the name "char" in the type "unsigned char" is due to programmers usually using that type to store characters rather than numbers, since that type's decimal range is so small. Nevertheless, that type is still just a number.

RIMS.h defines two other functions for printing: ***puti***(parm) to print an integer parameter, and ***puts***(parm) to print a string parameter (a string is a sequence of characters). For putc, puti, and puts, the parameter need not be a variable, but could be a constant like 'm', 7, or "Hello", respectively; note that a character is surrounded by single quotes but a string by double quotes. The special sequence '\n' is treated as a single character and causes printing to proceed to the next output line, so putc('\n') would proceed to the next output line, and puts("Hello\n") would print Hello and also proceed to the next output line.

Try: Run the above program. Notice that the letter 'm' gets printed in RIMS's debug output window each time the putc line gets executed. Press "Break" and then press "Step" multiple times to see 'm' get printed each time. Terminate, add the statement:

```
putc('\n');
```

after the putc(y) line, compile, and run, and notice that each 'm' is now printed on its own line.

A variable's name is called an identifier. An ***identifier*** consists of letters (a-z, A-Z, _) and digits (0-9) and must starts with a letter. Note that "_", called an *underscore*, is considered to be a letter. Upper and lower case letters differ. The following are valid identifiers: c, cat, Cat, n1m1, short1, and _hello. Note that cat and Cat are different identifiers. The following are invalid identifiers: 42c (doesn't start with a letter), hi there (has a space), cat$ (has a symbol other than a letter or digit), and short (uses a reserved word). Identifiers are also used to name functions and other items in addition to variables. .

Multiple variable names may be introduced in a single declaration, as in: "unsigned char c, d;" This option allows a programmer to group related variables, but should be used sparingly, usually only when the variables are somehow related. A ***semicolon ";"*** indicates the end of a declaration.

Variable declarations may appear outside of a function (as above), or inside a function after the opening bracket but before the function's statements. Variables declared outside functions are known as ***global variables***[1]. Variables declared inside a function are known as ***local variables***[2] and can only be accessed by statements within that function. *RIMS presently can only watch global variables, which is why "y" was defined above as global.* Variables should be declared locally whenever possible to keep them close to the statements that use them and to prevent multiple functions from conflicting use of a single variable. Below is a locally declared variable:

```
void main() {
   unsigned char y;
   while (1) {
      y = 'm';
      B = y;
   }
}
```

## Assignment statements and arithmetic expressions

An ***assignment statement*** such as "B = A;" or "c = 203;", upon being executed, copies the current value of the item on the ='s right side to the variable on the ='s left side. The "=" is known as the ***assignment operator***. An assignment statement has the form:

variable = expression;

An ***expression*** is a combination of items like variables, constants, and operators, that results in a value. In "B = A;", the expression is just the variable A, with the result being the current value of A. In "c = 203;", the expression is just the constant "203", which is thus the resulting value. Expressions commonly include ***arithmetic operators***:

"+" for ***addition***,
"-" for ***subtraction***,
"*" for ***multiplication***,
"/" for ***division***, and
"***%***" for ***remainder*** (modulus).

For example, the following program outputs the value of the input A plus 5 via the expression "A + 5":

[1] C refers to such variables as *external*, but *global* is the more common name in practice.
[2] C refers to such variables as *automatic*, but *local* is the more common name in practice.

```
void main() {
   while (1) {
      B = A + 5;
   }
}
```

Try: Run the above program. Move the input switches, and note (by observing the decimal numbers below the input and output pins) that the output value is equal to the input value plus 5. Next, set all inputs switches to 1s to represent the value 255, and note that the program's output is incorrect because the 8 output bits have a range of 0-255 and cannot represent a number larger than 255.

Parentheses may be used in expressions, as in: "B = (2 * A) - (A + 5);"

Expressions are evaluated using ***precedence rules***. Items within parentheses are evaluated first. Next to be evaluated are *, /, and %, having equal precedence. Finally come + and - with equal precedence. Thus, B = 3 + 2 * A is evaluated by first computing 2 * A, followed by computing 3 plus that result, because * has higher precedence than +. If more than one operator of equal precedence could be evaluated, evaluation occurs left to right. Thus, "B = A * 2 / 3;" evaluates as if it were written "B = (A * 2) / 3;" *Good practice involves using parentheses to explicitly represent the order of operations rather than relying on precedence rules*, thus writing "B = 3 + (2 * A);" rather than "B = 3 + 2 * A;", except when the order is unimportant as in "B = 3 + x + A;"

Try: Rewrite the following expressions by introducing parentheses to make the evaluation order explicit: "B = 4 + A * 2 - 5;" and "B = A * 3 + A / 2;" (Answers: : "B = (4 + (A * 2)) - 5;" and "B = (A * 3) + (A / 2);")

Below is a program that converts an input Celsius value to an output Fahrenheit value using an expression, per the equation Fah = ((9*Cel)/5) + 32:

```
void main() {
   while (1) {
      B = ((9 * A) / 5) + 32;
   }
}
```

Try: Run the above program. Set the input switches to all 0s, and observe that the output is 32. Set the input switches to one hundred, or "01100100" in binary (A7 is 0, A6 is 1, A5 is 1, etc.), and observe that the output is 212. Try other inputs also and observe and verify the output.

Because B and A are integer data types ("unsigned char" in particular), fractional numbers are ignored during execution. Thus, 9 / 4 will not evaluate to 2.25 but rather to just 2. Using the remainder operator, 9 % 4 will evaluate to 1.

Try: Create and run a program that sets B = 9 / 4, and note that B is just 2. Modify the program to set B = 9 % 4, and note that B is 1. Modify the program to set B to various values involving fractions, such as to 0.9 or 4.8, and notice that the fractional part is ignored.

## Variable initialization and constants

A variable declaration may assign an initial value to the variable, known as ***initialization***, as follows:

```
unsigned char x = 0;
```

Such initialization has the same behavior as:

```
unsigned char x;
x = 0;
```

*Good practice involves always initializing all variables, using one of the two methods above.*

Constants are common in programs. For example, a program in a car's computer may compare input, representing speed, to a value like 90. Rather then using constants throughout the program, such as the statement "if (A <= 90)", better practice uses a special form of an initialized variable whose value cannot be changed after being declared, known as a ***constant variable***, as follows:

```
const unsigned char maxSpeed = 90;
```

The variable has been initialized to 90. Furthermore, the const keyword indicates that the program cannot change the variable's value, so a statement later in the program like "maxSpeed = 95;" will cause a compiler error. The constant variable can be read in statements, as in "if (A <= maxSpeed)"; note that the descriptive name makes the statement's intent easier to understand than "A <= 90". Furthermore, using a constant variable makes changes of the constant value easy, by changing a single value as in "const unsigned char maxspeed = 95;" rather than searching throughout the program for all instances of the constant "90" and changing each to "95" (which is especially problematic if "90" is used for other purposes in the program too).

## Style for whitespace and brackets

Different programmers have different styles for the use of whitespace and brackets. We will follow the common style used in the programs above, wherein:

- Whitespace
  - Each statement usually appears on its own line
  - Blank lines rarely appear within a function's statements
  - A single space (not more) separates items
  - Sub-statements are indented 3 spaces from their parent statement, and tabs are not used
- Brackets
  - For an item with sub-statements, the opening bracket appears at the end of the item's line, as in "while (1) {" in the above programs, to reduce total lines for readability, and the closing bracket appears on its own line aligned under the item's start
  - Although brackets are optional when a statement has only one sub-statement, brackets are *always used* anyways for clarity

# Chapter 3 -- Branching using if-else statements

## If-else

Thus far, the statements inserted in our program's main "while (1)" loop all execute each time through the loop. Sometimes we want a statement to execute only if some condition is satisfied. The following program sets B0 to 1 on the condition that A is greater than 32, else the program sets B0 to 0:

```
void main() {
   while (1) {
      if (A > 32) {
         B0 = 1;
      }
      else {
         B0 = 0;
      }
   }
}
```

Try: Run the above program[3]. Set input A's switches to various positions, and note that B0's LED turns green only when the A's decimal value is greater than 32. Press "Break", set A's switches to some value less than 32, and then step repeatedly; note that the statement "B0 = 1;" gets executed, but not the statement "B0 = 0;" does not. Now set A's switches to some value greater than 32, and step again, noting this time "B0 = 0;" gets executed, but "B0 = 1;" does not.

***Branching*** in a program involves using a construct that directs execution to one list of statements (one branch) or another list of statements based on the value of an expression. C uses an "if-else" statement as one type of branching. The statement has the form:

```
if (expression) {
  /* sub-statements to execute when expression evaluated to true */
}
else {
  /* sub-statements to execute when expression evaluated to false */
}
/* statements here will execute after the if-else */
```

That statement has two branches, the "if" sub-statements and the "else" sub-statements. If the expression evaluates to true, the if sub-statements execute. Otherwise, the else sub-statements execute. After either situation, execution proceeds with statements below the if-else.

[3] Remember that all programs in RIMS must begin with the line: #include "RIMS.h"

## Relational and equality operators

If-else expressions use operators beyond the earlier arithmetic operators, as follows; the first four are known as ***relational operators***, and the last two as ***equality operators***:

"**<**" for ***less-than***,
"**>**" for ***greater-than***,
"**<=**" for ***less-than-or-equal***,
"**>=**" for ***greater-than-or-equal***,

"**==**" for ***equal***, and
"**!=**" for ***not equal***.

Try: Write a program that sets B to all 1s (i.e., B = 255) only when A is all 1s, otherwise setting B to all 0s (i.e., B = 0). Run the program and test different settings for A. (Answer hint: Use the above if-else structure with the expression "A == 255").

## If-else variations

Commonly, multiple branches are required, in which case if-else statements can be cascaded, as follows:

```
void main() {
   while (1) {
      if (A <= 100) {
         B0 = 1;
         B1 = 0;
         B2 = 0;
      }
      else if (A <= 200) {
         B0 = 0;
         B1 = 1;
         B2 = 0;
      }
      else {
         B0 = 0;
         B1 = 0;
         B2 = 1;
      }
   }
}
```

If the first expression evaluates to true, that branch's sub-statements are executed and the program then jumps to the bottom. If the first expression is false, only then is the second if statement executed and

hence the second expression evaluated, and if true, that branch's sub-statements are executed and the program then jumps to the bottom. If false, then the third branch's sub-statements are executed. Any number of such cascaded if-else statements is allowed.

Try: Run the above program, change A's switches to different values, and observe the output LEDs. Use the step functionality to observe that only one of the three branches' will have its statements executed.

The "else" part of if-else is optional. For example, the following program counts the number of input switches among A0, A1, and A2 that are set to 1, and sets B to the counted number:

```
void main() {
   unsigned char count;
   while (1) {
      count = 0;
      if (A0 == 1) {
         count = count + 1;
      }
      if (A1 == 1) {
         count = count + 1;
      }
      if (A2 == 1) {
         count = count + 1;
      }
      B = count;
   }
}
```

Note that the program uses three distinct if statements. All three "count = count + 1;" sub-statements could thus potentially be executed. In contrast, for cascaded if-else statements, only one branch could execute.

Try: Run the above program, change A0, A1, and A2, and observe B. Step through the program to observe the branches that are executed when A0 is 1, A1 is 0, and A2 is 1, noting that B becomes 2. Now modify the program by putting "else" before the latter two "if" statements, and step again to observe the undesired behavior wherein only the first branch's sub-statements are executed for those values of A0, A1, and A2.

Consider a program related to an automobile's automatic transmission. Input A represents the car's current speed. The program selects gear 1, 2, 3, or 4, and outputs the selected gear on B, as shown:

```
void main(){
   B = 1; /* Need some initial value */
   while (1) {
      if (A <= 15) {
         B = 1;
      }
      else if (A <= 28) {
         B = 2;
```

```
        }
        else if (A <= 45) {
           B = 3;
        }
        else { /* must be > 45 */
           B = 4;
        }
    }
}
```

More operators are available for expressions, namely ***logical operators*** that treat operands as being true or false, and evaluate to true or false. Such operators include:

"&&" for ***logical AND***, evaluating to true when both of its operands are true,
"||" for ***logical OR***, evaluating to true if at least one of its two operands are true,
"!" for ***logical NOT***, evaluating to true when its single operand is false

For example, suppose x currently has the value 99. Then the expression "(x > 50) && (x < 100)" would evaluate to true, because both operands (the item on the left and the item on the right) are true. On the other hand, the expression "(x > 50) && (x < 75)" would evaluate to false, because both operands aren't true, namely the second operand isn't true. The expression "(x > 50) || (x < 75)" would evaluate to true when x is 99 because at least one of the operands is true, namely the first operand is true. The expression "!(x > 50)" would evaluate to false, while "!(x < 75)" would evaluate to true. Expressions using logical operators are known as ***logical expressions***.

Parentheses in logical expressions are used similarly as in arithmetic expressions, with items within the innermost parentheses evaluated first.

Try: For x having a current value of 25, determine whether each expression evaluates to true or false:

```
x != 50
!(x == 50)
(x > 5) && (x < 50) && (x != 25)
(x > 5) && ( (x != 25) || (x != 50) )
```

You can test your answers by writing a program that sets x=25 and that uses an if-else statement that sets B0=1 if the expression is true and sets B0=0 otherwise.

## True and false in C

C doesn't actually have *true* and *false* data values. Instead, a logical expression that evaluates to true returns the number 1, and a logical expression that evaluates to false returns the number 0. Furthermore, for a logical operand, an operand with a zero value is considered as false, and an operand with a non-zero value (not necessarily just 1) is considered true.

Thus, in "if (A0 == 1)", the expression will evaluate to 1 if A0 is 1, and evaluate to 0 otherwise. As such,

the statement could have been written simply as "if (A0)", because A0 is either 1 or 0 and thus evaluates to the same value as A0 == 1. C programmers use such a simplification extensively. For example, an expression detecting that all of A0-A2 are 1s could be written simply as "if (A0 && A1 && A2)", rather than as "if ( (A0 == 1) && (A1 == 1) && (A2 == 1) )".

The following program demonstrates such an expression, implementing an automotive system that illuminates a warning light (by setting B0 to 1) if a key is in the ignition (detected by A0 being 1) and a person is in the driver's seat (A1 is 1) and the seat belt is not fastened (A2 is 0):

```
void main() {
   while (1) {
      if (A0 && A1 && !A2) {
         B0 = 1;
      }
      else {
         B0 = 0;
      }
   }
}
```

## Common error: Using assignment in a branch expression

*A warning: Perhaps the most common programming error in C is to use in a branch expression the assignment operator "=" when actually intending to use the equality operator "==".* In C, using the assignment operator in an if-else expression is *not* necessarily an error, instead executing as an assignment statement. Sometimes such execution results in a variable being unexpectedly updated, and then results in an incorrect branch. For example, consider the following erroneous program:

```
unsigned char x;

void main() {
   while (1) {
      x = A;
      if (x = 32) {  // Oops
         B0 = 1;
      }
      else {
         B0 = 0;
      }
   }
}
```

The intent was to set B0 to 1 only when the input A is 32. However, the program uses "=" rather than "==" in the if expression. When the if expression is evaluated, *x will be assigned the value 32*, and the expression will then evaluate to 32. Because 32 is non-zero, *the expression is considered true*, and thus the first branch is always taken, regardless of the value of A.

Try: Run the above program. Use "Add symbols" to watch the value of x. Step through the program and notice that regardless of the value of A, x gets set to 32 when the if statement is executed, and the first branch is always taken, thus setting B0 to 1 always. Now fix the above program by replacing "x = 32" with "x == 32", and observe the correct program behavior.

## Compiler error: Missing bracket

A common programming error is to omit a bracket, as in the following code:

```
void main() {
   while (1) {
      if (A0 && A1 && !A2) {
         B0 = 1;
      }
      else
         B0 = 0;
      } /* matches with while's opening bracket */
   } /* matches with main's opening bracket */
} /* The compiler asks: What's this bracket doing here? */
```

Notice that the else part is missing an opening bracket.

Try: Compile the above code. Notice that the compiler generates an error, but the line number is several lines below where the missing bracket's line and the error does not say "missing bracket."

Missing brackets commonly cause such unhelpful compiler error messages. In the above program, the missing bracket is itself not an error, because the brackets are not required for the else part (though we will always use brackets as good practice). The closing bracket below the else part thus closes the "while (1) {" opening bracket; the indentation of the bracket is irrelevant to the compiler, which ignores whitespace. The next closing bracket matches with main's opening bracket. To the compiler, the program so far is correct, and the main function is complete. Then, the compiler reaches the line with the last closing bracket, and does not recognize the bracket's purpose as there are no unmatched opening brackets above, so the compiler generates an error on that line.

*Warning: A very common C programming error stems from not always using brackets with if-else.*
Consider the following if statement, which is valid C code because brackets are optional:

```
if (A0)
   B0 = 1;
else
   B0 = 0;
```

Suppose the programmer later adds a statement as follows:

```
if (A0)
   B0 = 1;
else
```

```
x = 0;
B0 = 0;
```

The indentation makes the intent obvious, namely that both "x = 0;" and "B0 = 0;" are intended to be sub-statements of the else part, but the compiler ignores whitespace and thus ignores the indentation. Instead, only "x = 0;" is a sub-statement of the else part. "B0 = 0;" is not part of the if-else statement, and thus executes after the if-else statement regardless of the value of A0. Thus, B0 will always be set to 0. Note that the above is not a syntax error; the program compiles fine, but then execution results will be incorrect. Debugging such an error is hard. To avoid such errors, *we use a style that uses brackets for any statement that has sub-statements*, even if only a single sub-statement exists.

The cascaded if-else structure shown earlier is an exception to such style. In particular, nearly all programmers choose the style:

```
if (...) {
}
else if (...) {
}
else if (...) {
}
```

over the style:

```
if (...) {
}
else {
   if (...) {
   }
   else {
      if (...) {
      }
   }
}
```

The latter is less readable and less intuitive.

# Chapter 4: Loop statements and arrays

## While loop

Sometimes a list of statements should be executed multiple times. For example, programs in the previous chapter used a "while (1) { }" statement, as in:

```
void main() {
   while (1) {
      B = A;
   }
}
```

The "while (1) { }" statement ensured that the statements within the curly brackets, namely the single statement "B = A;", would be re-executed as long as the expression "1" evaluates to true. Because "1" is considered true, the statements within the curly brackets execute over and over again.

***Looping*** in a program involves using a construct that causes a list of statements to be executed repeatedly subject to some expression. C uses a "while" statement as one type of looping. The statement has the form:

```
while (expression) {
  /* Sub-statements to execute if the expression evaluated to true;
     known as the "loop body" */
}
/* Statements here will execute if the expression evaluated to false
```

and is known as a *while loop*. Upon execution of the while statement, the expression is evaluated. If true, execution proceeds to the sub-statements inside the while statement's curly brackets, known as the ***loop body***. Upon reaching the end of the loop body, execution *loops back to the while statement again*, akin to a runner on an oval track going from the start line around the track and ending at the start line again. The expression is again evaluated, and if true, execution again proceeds to the loop body. But if the expression evaluates to false, execution instead proceeds to below the while statement's closing bracket. Each execution of the loop body is called an ***iteration***.

Note that once execution proceeds into the loop body, execution continues until reaching the end of the loop body *even if the while expression changes to false midway through the loop body's statements*.

The "while (1) { }" loop expression used in earlier-introduced programs always evaluates to true and is thus called an ***infinite loop***. An infinite loop is common in the main function to cause the program to repeat forever, or more precisely for as long as the processor is running (especially in embedded systems),

but rarely appears anywhere else in a program.

Instead, many loops evaluate some condition to determine whether or not to continue looping. For example, the following program waits until A0 is 1 (using "while (A0 != 1)"), then sets B to a random number (as determined by a call to the "rand()" function from RIMS.h"), then waits until A0 is 0 (using "while (A0 != 0)"), and then repeats:

```
void main()
{
   while (1) {
      while (A0 != 1) {
      }
      B = rand();
      while (A0 != 0) {
      }
   }
}
```

Try: Run the above program[4]. Set A0 to 1, and note that B's value changes. Set A0 back to 0, then back to 1 again, and note that B changes again. Break and step to observe the looping.

The following program implements a guessing game, wherein the program randomly chooses a secret number between 1 and 100 (inclusive), and the user tries to guess the number by setting A. If the guess is too small, the program sets B0 to 1. If too large, B7 is 1. If correct, the program sets B to all 1s. The user can then restart the game by setting A to 0.

```
void main(){
   unsigned char num;
   while (1) {
      B = 0; /* set B to all 0s */
      num = (rand() % 100) + 1; /* 1 <= num <= 100 */
      while (A != num) {
         if (A < num) {
            B0 = 1; B7 = 0;
         }
         else if (A > num) {
            B0 = 0; B7 = 1;
         }
      }
      /* A must now equal num */
      B = 255; /* Set B to all 1s */
      while (A != 0) { /* A==0: User ready for next turn */
      }
   }
}
```

[4] Remember that all programs in RIMS must begin with the line: #include "RIMS.h"

The program initializes B to 0, and then sets num to a random number. The "rand() % 100" result will be a number between 0 and 99 (inclusive); the " + 1" ensures that num will always be between 1 and 100 (inclusive). The program then loops as long as A doesn't equal num, and that loop's body sets B0 and B7 appropriately. When A does equal num, that loop finishes and thus B is set to all 1s (note that 255 is "11111111" in binary). Another loop then waits until the user sets A to 0, after which the main "while (1)" loop repeats, causing the game to repeat.

Commonly, a loop should iterate a specific number of times. A variable is used to control the number of iterations. To iterate N times using a loop variable i, the program structure is:

```
/* Iterating N times using a loop variable */
i = 1;
while (i <= N) {
   /* Particular statements go here */
   i = i + 1;
}
```

For example, the following program sets B to 2^A, using a loop to compute the result. The program also uses puts/puti functions for our benefit to illustrate the loop behavior during execution.

```
void main() {
   unsigned long result;
   unsigned char i;
   while (1) {
      result = 1;
      i = 1;
      while (i <= A) {
         result = result * 2;
         puts("  Iteration: "); puti(i); puts("\n");
         i = i + 1;
      }
      puts("Result: "); puti(result);  puts("\n\n");
      B = result;
   }
}
```

Try: Run the above program. Set A to 2 and note from the printed output that the inner while loop body executes twice, and the result is 4, which appears on B. Set A to 0 and note that the inner while loop body does not execute, and the result is 1. Set A to 15 and note that the result is 32768, though B does not show that value because B's 8 bits cannot represent numbers larger than 255.

Try: Write a program that sets B to the sum of the first A numbers. For example, if A is 3, then B should be 1 + 2 + 3 = 6. Hint: only minor modifications are needed to the above 2^A program's statements.

*Warning: A common error is to forget to include the loop variable update (e.g., "i = i + 1;") at the end of the loop, causing an infinite loop.*

*Another warning: A common error is to use the assignment operator "=" rather than the equality operator "==" in a loop expression, causing an infinite loop, as in:*

```
while (x = 32) {
   // loop statements
}
```

The programmer intended to write "x == 32" but instead wrote "x = 32", which always evaluates to true (and also assigns x to 32).

Loops are commonly used in conjunction with arrays, which are now introduced.

## Arrays

A variable declaration can be modified to declare an ***array*** (meaning an ordered list) of elements of a given data type, as follows:

```
unsigned char voltages[5];
```

The square brackets denotes an array, and the "5" indicates the array's size, meaning the number of elements. So the variable "voltages" consists of 5 elements of type unsigned char.

In subsequent statements, each array element can be accessed as voltages[0], voltages[1], voltages[2], voltages[3], and voltages[4], as in:

```
voltages[4] = 6;
voltages[3] = voltages[4] * 2; // voltages[3] will be 12
```

In such use, the number within the square brackets is called the element's *index*. For an array of size N, the elements' indices are 0 to N-1.

An array variable can be initialized using the following syntax:

```
unsigned char voltages[5] = {0, 24, 17, 12, 6};
```

The following example uses input A as the index into the voltages array, outputting the corresponding element value on B:

```
void main(){
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   while(1) {
      B = voltages[A];
   }
}
```

The program could be useful in a desk fan with a switch connected to A, with 0 meaning Off, 1 Very Fast,

2 Fast, 3 Medium, and 4 Slow. B sets the input voltage of a DC motor that controls the rotation speed of the fan, B=0 meaning no rotation, and higher B numbers meaning faster rotation.

Try: Run the above program. Set A to all 0s and note that B is 0. Set A to 1 (A0=1, the rest 0s) and note that B is 24. Set A to 2 (A1=1, A0=0) and note that B is 17. Repeat for A being 3 (A1=1, A0=1) and for A being 4 (A2=1, A1=0, A0=0).

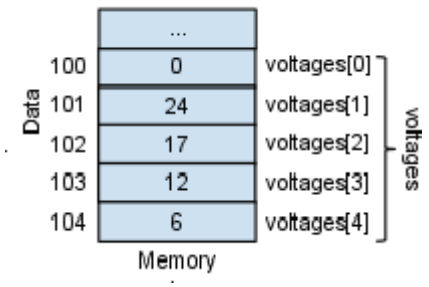
*Warning: A common error in C involves using an invalid index for an array.* Note that a size N array has elements with indices 0 to N-1, and *NOT 1 to N*. For example, voltages[5] is invalid for the above array, whose valid indices are 0 to 4. The compiler may be using the location in memory referred to by voltages[5] for another purpose. "x = voltages[5];" might thus set x to some unpredictable value. Even worse, "voltages[5] = 99;" might write 99 to a word in memory that was being used for something else. Using an invalid array index is unfortunately *not* a compiler error (a compiler cannot usually detect such invalid index values; the program has to be run to encounter such a value), and may cause bizarre errors in a running program that can be very hard to debug.

Try: Run the above program again. Set A to 5 (A2=1, A1=0, A0=1) and notice that RIMS generates an error message and terminates the program. Be aware that not all tools will generate a runtime error; some will simply continue running, but with mystifyingly-incorrect behavior.

The above program can be modified to prevent invalid array index values:

```
void main(){
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   while(1) {
      if (A < 5) {
         B = voltages[A];
      }
      else { // A is too large
         B = 0;
      }
   }
}
```

In memory, a compiler allocates a sequence of memory locations for an array. In the figure below, the compiler has allocated memory locations 100-104 for the above-declared voltages array.



## Loops and arrays together

Loops and arrays are commonly used together. The following program uses a loop to print each element of the above array:

```
void main(){
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   unsigned char i;
   while(1) {
      i = 0;
      while (i < 5) {
         puti(voltages[i]); putc(' ');
         i = i + 1;
      }
      putc('\n');
   }
}
```

Notice that, because the loop variable is also being used as the array index, the variable is set to range from 0 to 4, rather than from 1 to 5. Use of such a range starting at 0 is common. Furthermore, the loop variable is compared using "i < 5" rather than "i <= 4" so that the same number 5 is used in the array size declaration and the loop expression. Even better is to use a constant variable in the loop expression, as in:

```
   const unsigned char numVoltages = 5;
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   ...
      while (i < numVoltages) {
         ...
```

Note: Ideally, the array declaration would also use "numVoltages" rather than "5", i.e., "unsigned char voltages[numVoltages] ...". Unfortunately, C does not allow a constant variable, but rather only a constant expression like "5", in an array declaration.

Try: Modify the above program to compute and output on B the average value of the voltages array elements. Hint: Use a "result" variable as in earlier examples.

*Warning (repeated from earlier): A common error in C involves using an invalid index for an array*. A common source of an invalid index stems from writing an incorrect loop expression, such as writing the expression "i <= 5" rather than "i < 5" above.

The following program demonstrates how computation goals can be achieved using a combination of variables, if-else, and loops. The program finds the largest element value in the given array:

```
void main(){
   const unsigned char numVoltages = 5;
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   unsigned char highest;
   unsigned char i;
   while(1) {
      highest = voltages[0];
      i = 0;
      while (i < numVoltages) {
         if (voltages[i] > highest) {
            highest = voltages[i];
         }
         i = i + 1;
      }
      B = highest;
   }
}
```

The program visits each array element, updating highest if the element's value is larger than the largest seen so far (as indicated by highest's current value). Note that highest was initialized to voltages[0]; it could have been initialized to any array element's value. After visiting every element, highest contains the largest value in the array, and is thus assigned to B.

Visiting each array element is called ***traversing*** an array. Computations involving array traversals are a common and fundamental part of programming, and thus should be thoroughly practiced.

Try: Modify the above program (the "voltages" program) to output on B the lowest value found rather than the highest.

Try: Modify the voltages program to output on B the index (0, 1, 2, 3, or 4) where the highest value was found.

Try: Modify the voltages program to set any voltage greater than 15 to 16 (thus modifying the array), and then to print the array. The printed result should be:
0 16 16 12 6

Try: Modify the voltages program to repeatedly print the array in reverse, i.e., the printed result should be:
6 12 17 24 0

Sometimes a loop variable is used to traverse an array, but the loop should be exited before visiting every element. The ***break*** statement forces an exit from a loop regardless of the loop expression. Recall the earlier automotive transmission example that chose among gears 1-4 based on speed input A. Rather than using if-else statements, the following program uses an array and loop, involving a break statement:

```
void main(){
   unsigned char gearSpeeds[3] = {15, 28, 45};
   unsigned char i;
   B = 1; /* Need some initial value */
   while (1){
      i = 0;
      /* Find index where A <= that gear's speed */
      while (i < 3) {
         if (A <= gearSpeeds[i]) {
            break; /* Exit this loop */
         }
         i = i + 1;
      }
      B = i + 1; // Output the selected gear
   }
}
```

The maximum speed for each gear is stored in the array, so gear 1's maximum speed is 15, gear 2's is 28, and gear 3's is 45; gear 4 has no listed maximum speed. The inner while loop begins visiting each element in the array, and if the current speed is less than that element's value, the loop is exited via the break statement because the proper gear has been found. For example, if the current speed is 12, the loop breaks when i=0 (because 12<=15). If the current speed is 33, the loop breaks when i=2 (because 33<=45). If the current speed is 65, the loop will not break but rather will complete with i=3. Because the indices start at 0 while the gears start at 1, B is assigned the found index plus 1.

Although the while loop version of the automotive transmission example may seem to have little benefit over the earlier if-else version, the loop version begins to show greater benefit for larger arrays, such as an array with 50 values; the loop version would be almost identical to above, while the if-else version would have 50 branches.

## For loop

Iterating a specific number of times is so common in programs that C supports a second loop type for that purpose. A ***for loop*** statement collects the loop variable initialization, loop expression, and loop variable update all at the top of the loop. Thus, to iterate 5 times, instead of the while loop:

```
i = 1;
while (i <= N) {
   /* Statements go here */
   i = i + 1;
}
```

a for loop *having exactly the same functionality* is written as:

```
for (i = 1; i <= N; i = i + 1) {
   /* Statements go here */
}
```

The for loop has the advantage of making the number of loop iterations more obvious, and reducing the likelihood of making the mistake of forgetting to update the loop variable at the bottom of the loop. The following program is a rewrite of the earlier voltages program that used a while loop:

```
void main(){
   const unsigned char numVoltages = 5;
   unsigned char voltages[5] = {0, 24, 17, 12, 6};
   unsigned char highest;
   unsigned char i;
   while(1) {
      highest = voltages[0];
      for (i = 0; i < numVoltages; i = i + 1) {
         if (voltages[i] > highest) {
            highest= voltages[i];
         }
      }
      B = highest;
   }
}
```

Try: Modify the above voltages program to print the values of the array, using a for loop.

Try: Modify the voltages program to output on B the index (0, 1, 2, 3, or 4) where the highest value was found, using a for loop.

C includes an ***increment operator "++"*** for the common occurrence of updating a variable by adding 1, namely "`i = i + 1`" can be rewritten simply as "`i++`". The operator commonly appears in for loops, as in:

```
      for (i=0; i < numVoltages; i++) {...
```

A ***decrement operator "--"*** also exists, such that "i = i - 1" can be replaced by "i--".

Try: Modify the voltages program to repeatedly print the array in reverse, i.e., the printed result should be:
6 12 17 24 0
Use a for loop and the decrement operator.

Note: The increment and decrement operators can appear in postfix ("i++" or "i--") or prefix ("++i" or "--i") form. The distinction is relevant mainly when the operator is used within an expression, as in "if (x < i++)"; the prefix form increments the variable first, then uses the incremented value in the expression, while the postfix form uses the current value of the variable in the expression first, and then increments the variable. As we do not condone use of the increment/decrement operators in expressions, we do not discuss the distinction further.

# Chapter 5: Functions

## Function definitions and function calls

A function is a list of statements, such that invoking the function's name in a program causes the function's statements to execute. For example, the function named "puti" is defined in RIMS.h, and invoking "puti(x);" executes statements that causes the integer value of x to be printed. Likewise, the function "rand" is defined in RIMS.h, and invoking "x = rand();" executes statements that causes a (psuedo-)random number to be returned.

A programmer commonly defines new functions so that code is more understandable. Recall the earlier program that counted the number of 1s on A0, A1, and A2. The program can be rewritten using a newly-defined function named CountOnes:

```
// function definition
unsigned char CountOnes(unsigned char x,
                        unsigned char y, unsigned char z) {
   unsigned char count;
   count = 0;
   if (x == 1) {
      count = count + 1;
   }
   if (y == 1) {
      count = count + 1;
   }
   if (z == 1) {
      count = count + 1;
   }
   return count;
}

void main() {
   while (1) {
      B = CountOnes(A0, A1, A2); // function call
   }
}
```

Note how the main code is more understandable than the previous version, clearly and simply stating that B is set to the number of 1s on A0, A1, and A2.

A programmer provides a ***function definition*** to define a function's contents, having the general form:

```
return-type function-name(parameter declarations) {
  declarations
  statements
}
```

Function-name must be a valid C identifier, such as "CountOnes". The ***parameter declarations*** are a comma-separated list of values that must be passed to the function, such as three unsigned chars x, y, and z for CountOnes. The return-type indicates the data type that the function returns. The function may contain local declarations, such as "unsigned char count;" above, and then a list of statements that carry out the function's behavior, typically computing something based on the parameters' values. The last statement is typically a "return expression" statement that provides the result of the function's computation.

Given a function definition, a function operates as follows. A program may invoke a function, referred to as a ***function call***, by using the function's name in an expression. The call must include in parentheses a list of values, known as ***arguments***, for the function parameters, as in "B = CountOnes(A0, A1, A2);" where the values of A0, A1, and A2 are the three arguments. The function carries out its computation using those values, referring to those values by the function's parameter names (x, y, and z above) in the function's statements. The function returns a single value via a return statement, and the function call evaluates to that returned value. Thus, if A0, A1, and A2 had the values of 1, 0, and 1, then the function CountOnes parameters x, y, and z would have values 1, 0, and 1. The function's statements would result in the variable count having a value of 2, which the function would return. In main, B would thus be set to 2.

Try: Run the above program involving the CountOnes function[5]. Press "Break" and then repeatedly press "Step." Notice how the function call causes execution to jump from the main function to the CountOnes function, then returning back to the main function. Change the switch positions on A0, A1, and A2 to see different values on B after each function call completes.

Try: Rewrite the following program by defining a new function CelsToFahr having a single parameter F of type unsigned char and a return value of unsigned char, and by calling that function in main.

```
void main() {
   while (1) {
      B = ((9 * A) / 5) + 32; // Celsius to Fahrenheit conversion
   }
}
```

## Parameters and return values

The number of arguments passed to a function must exactly equal the number of parameters in the function's definition. Thus, for the CountOnes function as defined above having three parameters, "CountOnes(A1, A2, A3, A4)" or "CountOnes(A1, A2)" will generate compiler errors.

Each argument passed to a function may be an expression. Thus, "CountOnes(A1, 0, A3&A4)" is a valid call to the function.

[5] Remember that all programs in RIMS must begin with the line: #include "RIMS.h"

A function may have no parameters, in which case the function definition will simply have nothing between its parentheses:

```
return-type function-name( ) {
...
```

Calling such a function involves parentheses with no arguments, as in "x = rand( );".

A function (with or without parameters) need not have a return value, in which case the function should be defined having a return-type of "void", and the return statement should have no expression. For example, the following function just prints "Hello there everybody" when called and has no return value; a call to that function is also shown in main:

```
void PrintHello() {
   puts("Hello there everybody\n");
   return;
}

void main() {
   while (1) {
      PrintHello();
   }
}
```

Good practice involves always returning from a function using a single return statement at the bottom of the function, even though a function could be written with multiple return statements, and even though a void function could be written without a return statement (thus returning when the end of the function is reached).

A parameter's type may be an array type, as in the following:

```
void milesDrivenPrint(unsigned short milesList[],
                      unsigned char  milesListSize)
{
   unsigned short year = 1990;
   unsigned char i;
   for (i=0; i<milesListSize; i++) {
      puts("Year: "); puti(year+i);
      puts("   Miles: "); puti(milesList[i]); puts(" trillion miles\n");
   }
   puts("\n");
}

void main(){
   // When A is 255, prints U.S. annual miles driven
   unsigned char  milesDrivenSize = 20;
   unsigned short milesDriven[20] = // in trillions
```

```
     { 2144, 2172, 2247, 2296, 2358, 2423, 2486, 2562, 2632, 2691,//1990-99
       2747, 2781, 2855, 2890, 2962, 2989, 3014, 3029, 2973, 2979};//2000-09
   while(1) {
      if (A == 255) {
         milesDrivenPrint(milesDriven, milesDrivenSize);
      }
   }
}
```

When A is 255 (all 1s), the program prints the number of U.S. annual miles driven. The miles numbers are initially stored in array milesDriven. The program uses function milesDrivenPrint to carry out the printing. In the function definition, the parameter named "milesList" is an array type as indicated by the brackets, i.e., "milesList[]". The size of the array is not specified in such a parameter declaration. In the function call in main, the array argument is written merely as "milesDriven" without any brackets. In contrast to a function's parameters, a function's return type may *not* be an array type.

Normally, a programming rule is that a function's parameters should only be read and should not be written. Although a parameter is essentially a new local variable with an initial value (from the argument) and thus could be written by a function's statements (e.g., the above milesDrivenPrint function could include a statement like "milesListSize = milesListSize - 1;"), functions that write to their parameters are harder to understand. Instead, local variables should be defined for local storage needs. (Note that writing to a parameter does not affect the value of a variable passed as the argument to that parameter, e.g., in the above program, writing to milesListSize would not change the value of milesDrivenSize, because milesListSize is essentially a new local variable).

However, array parameters are an exception to the rule about not writing to parameters. Unlike non-array parameters, an array parameter is not a new local copy of the array argument, but rather is the same array as the argument. Commonly, a function is defined to manipulate the contents of an array. For example, the following function ACopyToArray copies global variables A0-A7 into its 8-element array parameter, which is array AA declared in main:

```
void ACopyToArray(unsigned char Ary[]) {
   Ary[0]=A0; Ary[1]=A1; Ary[2]=A2; Ary[3]=A3;
   Ary[4]=A4; Ary[5]=A5; Ary[6]=A6; Ary[7]=A7;
   return;
}

void AryPrint(unsigned char Ary[], unsigned char ArySize) {
   unsigned char i;
   for (i=0; i<ArySize; i++) {
      puti(Ary[(ArySize-1)-i]); // (ArySize-1)-i: print in reverse
      putc(' ');
   }
   putc('\n');
   return;
}
```

```
void main(){
   const unsigned char AASize = 8;
   unsigned char AA[8];
   while(1) {
      ACopyToArray(AA);
      AryPrint(AA, AASize);
   }
}
```

Try: Run the above program and observe that the printed output reflects the values of A7-A0. Next, add to the above program a new function definition named AryCountOnes, having the same two parameters as AryPrint and with a return type of unsigned char, that returns the number of array elements equal to 1. Update main to also set B to that number.

Try: Add to the above program a new function definition named AryComplement, having the same two parameters as AryPrint and with a return type of void, that replaces 1s by 0s and 0s by 1s in its array parameter. Update the main function to repeatedly print both the original and complemented array.

## Scope

The ***scope***, meaning visibility to parts of the program, of a function parameter's and a function's local declarations is limited to the function's own declarations and statements. For example, in the above CountOnes function, the scope of x, y, and z, and of count, is limited to within the braces of the CountOnes function. The main function cannot access x, y, z, or count.

Try: Modify the CountOnes example's main function by adding a puti statement as shown. Compile the program and notice the undefined identifier 'count' error that results.

```
//CountOnes definition from earlier goes here...

void main() {
   while (1) {
      B = CountOnes(A0, A1, A2);
      puti(count); // ADD this statement
   }
}
```

On the other hand, a function can access global variables, such as global variables A and B (defined in RIMS.h), as in the following function definition:

```
void SetBToAllOnes() {
   B = 255;
   return;
}
```

To be accessible to a function, the global variable must have been defined somewhere above the function in the C file. A global variable's scope is all code below the variable's definition.

If a function's local parameter or variable has the same name as a global variable, the local item takes priority and thus blocks the global variable from being seen within the function.

Try: Run the following program and note that RIM output B is set to all 1s. Next, remove the "//" from the front of the "unsigned char B" line and run the program again, noting that RIM output B is not set to 1s.

```
void SetBToAllOnes() {
   //unsigned char B; // This declaration blocks out the global B
   B = 255;
   return;
}

void main() {
   while (1) {
      SetBToAllOnes();
   }
}
```

The potential for confusion between local and global variables is one reason why use of global variables should be kept to a minimum.

The scope of a function itself is all code in the C file, whether above or below the function definition. Commonly, a programmer defines new functions *below* the main function, so that the main function is the first function in the C file and thus immediately visible to a person reading the code. In this case,, the programmer should add a ***function declaration*** at the top of the file, which declares the return-type, function-name, and parameters of a function but excludes the body of the function, thus making the function's features known and suggesting that the function is defined further down. Below is an example. (Don't overlook that the function declaration has a semicolon after the parentheses).

```
void SetBToAllOnes(); // known as the function DECLARATION

void main() {
   while (1) {
      SetBToAllOnes();
   }
}

void SetBToAllOnes() { // known as the function DEFINITION
   B = 255;
   return;
}
```

If the function declaration's parameters and return type don't match the function definition, or if the function definition is missing, the compiler generates an error.

Try: Modify the SetBToAllOnes function definition (the definition appears after main) to return an unsigned char, compile, and note the error messages. Next, delete the SetBToAllOnes definition, compile, and note the error message.

## Typedef

A type definition, or ***typedef***, is a construct that allows a programmer to define a new name for a type, so that name can then be used for variable declarations or for function parameter and return types, as in the following:

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
```

These new type names can improve code readability by reducing the amount of text, which can be especially useful in parameter lists, for example::

```
void milesDrivenPrint(unsigned short milesList[],
                      unsigned char  milesListSize)
```

could be replaced by:

```
void milesDrivenPrint(ushort milesList[], uchar milesListSize)
```

## Creating good functions

In addition to functions leading to code that is more understandable, functions can also lead to code having less redundancy, as in the following example:

```
//CountOnes definition from earlier goes here...

void main() {
   while (1) {
      B =   CountOnes(A0, A1, A2)
          + CountOnes(A3, A4, A5)
          + CountOnes(A6, A7, 0);
   }
}
```

The CountOnes function is called from three places with different arguments at each place, yet the statements that count the 1s appear only once in the code (namely, in the CountOnes function definition). Note that the last call needed only two arguments counted but had to pass a third argument of 0, because the number of arguments must match the number of parameters in the function definition.

The skill of decomposing a program's behavior into a good set of functions is a fundamental part of programming that helps characterize a good programmer. Each function should have easily-recognizable behavior, and the behavior of the main function (and any function that calls other functions) should be easily understandable via the sequence of function calls. As an analogy, the main behavior of "Starting a

car" can be described as a sequence of function calls like "Buckle seatbelt," "Adjust mirrors," "Place key in ignition," and "Turn key" -- note that each function itself consists of more detailed operations as in "Buckle seatbelt" actually consisting of "Hold belt clip", "Pull belt clip across lap," and "Insert belt clip into belt buckle until hearing a click." "Buckle seatbelt" is a good function definition because its meaning is clear to most people, whereas a coarser function definition like "GetReady" for both the seatbelt and mirrors may not be as clear, while finer-grained functions like "Hold belt clip" are distracting from the purpose of the "Starting a car" function. .

As an example of creating good functions, consider the following "Miles driven statistics program" code that outputs the highest and lowest values from the milesDriven array as well as the average value:

```
//Original miles driven stats program, hard to understand, needs functions
typedef unsigned char  uchar;
typedef unsigned short ushort;

void main(){
   uchar milesDrivenSize = 20;
   ushort milesDriven[20] = // In trillions
     { 2144, 2172, 2247, 2296, 2358, 2423, 2486, 2562, 2632, 2691,//1990-99
       2747, 2781, 2855, 2890, 2962, 2989, 3014, 3029, 2973, 2979};//2000-09
   ushort highest, lowest, sum, avg;
   uchar  i;
   while(1) {
      highest = milesDriven[0];
      lowest = milesDriven[0];
      sum = 0;
      for (i=0; i < milesDrivenSize; i++) {
         if (milesDriven[i] > highest) {
            highest = milesDriven[i];
         }
         if (milesDriven[i] < lowest) {
            lowest = milesDriven[i];
         }
         sum = sum + milesDriven[i];
      }
      avg = sum / milesDrivenSize;
      puts("Highest:   "); puti(highest); puts("\n");
      puts("Lowest:    "); puti(lowest);  puts("\n");
      puts("Average:   "); puti(avg);     puts("\n\n");
   }
}
```

Such code is a bit of a mess, as the three behaviors of finding the highest, the lowest, and the average values are intermixed, hampering understandability. The main code would be more understandable if it were written using functions, as follows:

```
//More understandable miles driven statistics program using functions
typedef unsigned char  uchar;
typedef unsigned short ushort;
```

```
ushort MilesDrivenGetHighest(ushort milesDriven[], uchar milesDrivenSize);
ushort MilesDrivenGetLowest (ushort milesDriven[], uchar milesDrivenSize);
ushort MilesDrivenGetAvg    (ushort milesDriven[], uchar milesDrivenSize);

// more-understandable main function for the miles driven statistics program
void main(){
   uchar  milesDrivenSize = 20;
   ushort milesDriven[20] = // in trillions
     { 2144, 2172, 2247, 2296, 2358, 2423, 2486, 2562, 2632, 2691,//1990-99
       2747, 2781, 2855, 2890, 2962, 2989, 3014, 3029, 2973, 2979};//2000-09
   ushort highest, lowest, sum, avg;
   while(1) {
      highest = MilesDrivenGetHighest(milesDriven, milesDrivenSize );
      lowest  = MilesDrivenGetLowest (milesDriven, milesDrivenSize );
      avg     = MilesDrivenGetAvg    (milesDriven, milesDrivenSize );
      puts("Highest:  "); puti(highest); puts("\n");
      puts("Lowest:   "); puti(lowest);  puts("\n");
      puts("Average:  "); puti(avg);     puts("\n\n");
   }
}

ushort MilesDrivenGetHighest(ushort milesDriven[], uchar milesDrivenSize )
{
   ushort highest;
   uchar i;
   highest = milesDriven[0];
   for (i=0; i<milesDrivenSize; i++) {
      if (milesDriven[i] > highest) {
         highest = milesDriven[i];
      }
   }
   return(highest);
}

ushort MilesDrivenGetLowest (ushort milesDriven[], uchar milesDrivenSize )
{
   return(0); // FIXME Finish this function. 9/1/11 Alan S.
}

ushort MilesDrivenGetAvg    (ushort milesDriven[], uchar milesDrivenSize )
{
   return(0); // FIXME Finish this function. 9/1/11 Alan S.
}
```

Try: Finish writing the above program by completing the function definitions. .

Beyond enhancing understandability, good functions can improve the code's robustness by encouraging careful definition and testing of each function. For example, a programmer might check the MilesDrivenGetHighest function for a variety of array element values. Furthermore, a programmer might

notice that, because the variable highest is initialized with milesDriven[0], the for loop could actually begin with i=1. However, such internal details can be hidden from the main function -- a feature known as ***information hiding*** -- thus not distracting someone trying to understand main's behavior.

The above program created temporary function definitions to be completed, commonly called ***function stubs***. Programmers commonly first decide how to decompose a main function into sub-functions, create stubs as above, and then complete each function one at a time, testing each function thoroughly before writing the next function. Common practice is to include the comment "FIXME" next to any code that needs to be fixed or completed, with the comment also describing what needs to be done, and perhaps the date and name/initials of the programmer that wrote that comment.

One might believe that the original miles driven statistics program is better because it is more ***efficient***, meaning the results are achieved using fewer computations, in this case because the array is only traversed once, rather than being traversed three times in the three-functions version of the program. While the original program is indeed more efficient, that does not necessarily make the program better. In practice, *understandability is often a more important measure of "better" than is efficiency*. Understandability enhances correctness, and eases program maintenance over the many years and the variety of programmers of a program's lifetime. As a general rule, programmers should strive first to make their programs as *understandable* as possible. Then, only if inefficiency is a problem should programmers seek ways to make their programs more efficient, while still trying to maintain understanability.

Likewise, one might believe the original program is better because it has less code, but again, less code does not necessarily mean the code is more understandable and hence does not mean the code is better.

# Chapter 6: Structures and pointers

## Structures

Commonly, two variables are closely related and should be grouped. Consider a driving statistics program that has information on total miles driven *and* on DUI[6] deaths for years 1990-2009 in the U.S. If the user sets input A to 0, the information for 1990 is printed, A=1 prints 1991's info, A=2 prints 1992's info, and so on, with A=19 printing 2009. Below is one version of such a program, using two arrays, without grouping of related variables:

```
// Initial driving statistics program, with yearl-info spread across 2 arrays
typedef unsigned char  uchar;
typedef unsigned short ushort;

void main(){
   uchar  yearsInfoSize = 20;
   ushort milesDriven[20] = // in billions
     { 2144, 2172, 2247, 2296, 2358, 2423, 2486, 2562, 2632, 2691,
       2747, 2781, 2855, 2890, 2962, 2989, 3014, 3029, 2973, 2979 };
   uchar duiDeathsSize = 20;
   uchar duiDeaths[20] =  // in thousands
     { 18, 16, 14, 14, 13, 13, 13, 13, 13, 13,   // 1990-1999
       13, 13, 13, 13, 13, 14, 13, 13, 12, 11 }; // 2000-2009
   ushort baseyear = 1990;
   uchar  i;
   while(1) {
      i = A;
      if (i < yearsInfoSize) { // Ensure valid index
         puts("Year: "); puti(baseyear+i);  puts("\n");
         puts("   "); puti(milesDriven[i]); puts(" billion miles driven\n");
         puts("   "); puti(duiDeaths[i]);   puts(" thousand DUI deaths\n");
      }
      while (A == i) {}; // Print once, then wait for A to change
   }
}
```

Try: Run the above program, changing A to values ranging from 0 to 19, noting the printed info.[7]

Note that information for a given year such as 1990 is spread across two arrays within the program. Instead, a ***struct*** construct coupled with typedef allows grouping of information into a new composite type:

```
   typedef struct {
      ushort milesDriven; // in trillions
```

[6] DUI: Driving under the influence of alcohol or other drugs

[7] Remember that all programs in RIMS must begin with the line: #include "RIMS.h"

```
      uchar  duiDeaths;   // in thousands
   } yearInfo;
```

A variable could then be declared of type yearInfo:

```
   yearInfo curryear;
```

That variable has two members that can be written or read:

```
   curryear.milesDriven = 2144;
   curryear.duiDeaths   = 18;
   puti(curryear.milesDriven);
   ...
```

Even more usefully, an array with yearInfo elements can be declared:

```
   yearInfo yearsInfo[20];
```

Each array element has members, accessible as follows:

```
   i = 0;
   yearsInfo[i].milesDriven = 2144;
   yearsInfo[i].duiDeaths   = 18;
   puti(yearsInfo[i].milesDriven);
   ...
```

An array of struct elements can be initialized as follows:

```
   yearInfo yearsInfo[3] = { {2144, 18}, {2172, 16}, {2247, 14} };
```

Using an array of structs, the driving statistics program can be improved as follows:

```
// A better driving statistics program, with yearly-info grouped in a struct
typedef unsigned char  uchar;
typedef unsigned short ushort;

typedef struct {
   ushort milesDriven; // in trillions
   uchar  duiDeaths;   // in thousands
} yearInfo;

void main(){
   uchar yearsInfoSize = 20;
   yearInfo yearsInfo[20] = { // {miles driven, dui deaths}
      {2144, 18}, {2172, 16}, {2247, 14}, {2296, 14}, {2358, 13}, //1990-1994
      {2423, 13}, {2486, 13}, {2562, 13}, {2632, 13}, {2691, 13}, //1995-1999
      {2747, 13}, {2781, 13}, {2855, 13}, {2890, 13}, {2962, 13}, //2000-2004
      {2989, 14}, {3014, 13}, {3029, 13}, {2973, 12}, {2979, 11}};//2005-2009
   ushort baseyear = 1990;
   uchar i;
   while(1) {
      i = A;
      if (i < yearsInfoSize) { // Ensure valid index
         puts("Year: "); puti(baseyear+i);  puts("\n");
         puts("   "); puti(yearsInfo[i].milesDriven);
```

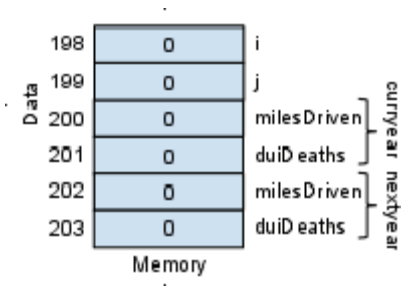
```
            puts(" billion miles driven\n");
            puts("   "); puti(yearsInfo[i].duiDeaths);
            puts(" thousand DUI deaths\n");
         }
         while (A == i) {}; // Print once, then wait for A to change
      }
}
```

Note: A struct declaration need not be coupled with typedef as above, but we recommend such coupling, and thus don't discuss standalone struct declarations.

In memory, a struct occupies memory locations akin to having declared the members as separate variables, as follows:

```
typedef struct {
   ushort milesDriven; // in trillions
   uchar  duiDeaths;   // in thousands
} yearInfo;

uchar    i, j;
yearInfo currYear;
yearInfo nextYear;
```




(Note: The above figure neglects issues with memory location width; a single variable whose type is larger than the memory location width may occupy multiple memory locations).

Assigning one struct type to another struct type causes corresponding members to be assigned. For example, for the above declarations, the following statement:

```
nextyear = curryear;
```

would have the same effect as

```
nextyear.milesDriven = curryear.milesDriven;
nextyear.duiDeaths   = curryear.dueDeaths;
```

A function parameter may be a struct type. For example, the above printing could be carried out by a function:

```
...
```

```
void yearInfoPrint(yearInfo thisyear) {
   puts("   "); puti(thisyear.milesDriven);
   puts(" billion miles driven\n");
   puts("   "); puti(thisyear.duiDeaths);
   puts(" thousand DUI deaths\n");
}

void main() {
   ...
      if (i < yearsInfoSize) { // Ensure valid index
         puts("Year: "); puti(baseyear+i); puts("\n");
         yearInfoPrint(yearsInfo[i]);
      }
   ...
```

As with other function parameters, a struct parameter is a local copy of the function call's argument, so the argument's members are copied to the parameter's members. (Thus, the above function performing "thisyear.milesDriven = 0;" would have no effect on the yearsInfo array's structs). Likewise, a function return type may be a struct type.

Sometimes we do want a function to change a struct's members, similar to how sometimes we want a function to change an array's elements. Pointers can be used for that purpose, as now discussed.

## Pointers

For most data types, a variable name refers to the contents of the variable's memory location. In the above memory figure, "i" refers to the contents of location 198 (which is 0 in the figure), and "curryear.milesDriven" refers to the contents of location 200 (also 0). However, sometimes we instead want to refer not to the *contents* of the variable's memory location, but to the *address* of the variable's memory location. The address of variable "i" is 198. Placing the "&" symbol before a variable name refers to the variable's address, not contents. Thus, for the above "i" is 0, but "&i" is 198 (per the memory figure). The "&" symbol is called "ampersand," so "&i" would be spoken as "ampersand i."

A common use of a variable's address is in the situation when a function should change the value of the arguments passed to the function. Such a situation arises, for example, when a function should "return" more than one value, whereas we know a function can't return more than one value. Instead, two parameters can be declared as variable addresses rather than variable contents. Placing the "*" symbol before a parameter name indicates that the parameter is a variable's address, not content. A parameter (or variable) that is an address is called a ***pointer,*** because it "points to" a memory location such as location 199, rather than referring to the location's contents. The pointer's memory contents can then be accessed by again placing a "*" symbol before the parameter name within the function (known as *dereferencing* the pointer). The "*" symbol is called an "asterisk" or more commonly "star". For example, consider a function Nibbles that converts a single 8-bit number into two numbers high and low, high being the high 4 bits and low being the low 4 bits (4 bits is sometimes called a *nibble*). For example, the number 31 is

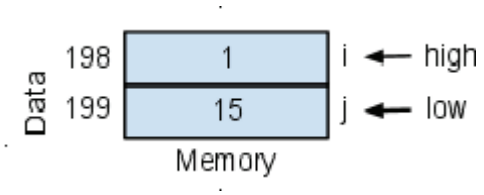
00011111 in binary. Function Nibbles should set high to 1 (0001) and low to 15 (1111). The function can be defined and called as follows:

```
typedef unsigned char uchar;

// high and low parameters are declared as POINTERS to uchars
void Nibbles(uchar num, uchar *high, uchar *low) {
   *high = num >> 4; // Note the "*" before high
   *low  = num & 0x0F; // Note the "*" before low
   return;
}

uchar i, j;
void main(){
   uchar num;
   while(1) {
      num = A;
      Nibbles(num, &i, &j); // Passing i/j ADDRESSES
      puts("High nibble: "); puti(i);
      puts("  Low nibble: "); puti(j); puts("\n");
      while (num == A) {} // Print once; wait for change
   }
}
```

(We omit explanation of the "num >> 4" and "num & 0x0F" expressions that compute the high nibble and low nibble). The function call is "Nibbles(num, &i, &j);", which passes the addresses of i and j (e.g., 198 and 199 from the earlier memory figure) as the last two arguments. The corresponding function parameters are "uchar *high" and "uchar *low", meaning they are pointers. In the function, the computed high nibble is assigned to "*high" and the computed low nibble to "*low" (note the asterisks), which cause the contents of locations 198 and 199 to be updated, as in the memory figure below for A=31 (high=1, low=15).




Try: Run the above program, set A to 31 (00011111) and observe the correct printed output. Press Break, change A, add symbols i and j to be watched, step repeatedly, and notice that i and j change when the high and low assignments occur within Nibbles, before the function returns, because "*high" and "*low" refer to the contents of i's and j's memory locations.

Try: Remove the "*" from "*high = …" in the Nibbles function and compile; note the compiler error, indicating that a number cannot be assigned to a pointer.

Sometimes a struct argument should be modified by a function. Just passing the struct causes the argument to be copied to the function's local parameter, as for most other parameters (except arrays).

Instead, if the function should change the struct, then the struct's *address* should be passed. In the above figure with curryear and nextyear, "&curryear" would refer to address 200, and "&nextyear" to address 202. Below is an example of modifying a struct in a function using pointers:

```
typedef unsigned char  uchar;
typedef unsigned short ushort;

typedef struct {
   ushort milesDriven; // in trillions
   uchar  duiDeaths;   // in thousands
} yearInfo;

void yearInfoInit(yearInfo *curryear, ushort miles, uchar duis) {
   (*curryear).milesDriven = miles;
   (*curryear).duiDeaths = duis;
   return;
}

void main(){
   uchar yearsInfoSize = 3;
   yearInfo yearsInfo[3];
   uchar i;
   ushort baseyear = 1990;
   yearInfoInit(&yearsInfo[0], 2144, 18);
   yearInfoInit(&yearsInfo[1], 2172, 16);
   yearInfoInit(&yearsInfo[2], 2247, 14);
   while(1) {
      i = A;
      if (i < yearsInfoSize) { // Ensure valid index
         puts("Year: "); puti(baseyear+i);  puts("\n");
         puts("   "); puti(yearsInfo[i].milesDriven);
         puts(" billion miles driven\n");
         puts("   "); puti(yearsInfo[i].duiDeaths);
         puts(" thousand DUI deaths\n");
      }
      while (A == i) {}; // Print once, then wait for A to change
   }
}
```

Struct pointers are so common that C has a shorthand notation wherein:

```
   *curryear.milesDriven = miles;
```

can instead be written as:

```
   curryear->milesDriven = miles;
```

That arrow-like notation is very commonly used.

# Chapter 7: More topics

## Preprocessor directives: #include and #define

C compilers utilize a preprocessor that basically prepares a C file for compilation. One simple preprocessing task is the replacement of all comments by whitespace. Another preprocessing task involves the "#include" directive, as follows. If a file "types.h" exists and has the contents:

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
```

and if a file prog1.c has the following contents with a #include directive:

```
#include "types.h"
void main(){
   uchar num;
   while(1) {
      num = A;
      B = num;
   }
}
```

then the preprocessor replaces those contents by:

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
void main(){
   uchar num;
   while(1) {
      num = A;
      B = num;
   }
}
```

Note that the actual prog1.c file is not modified; rather, the contents passed to the compiler are replaced as above. The "#" symbol indicates a directive for the preprocessor. The "#include" directive must be followed by a file name, as in: #include "types.h". The filename may be enclosed in double-quotes, indicating the file is in the same directory/folder as the prog1.c file (or may be enclosed in angle brackets <filename>, indicating the file is in the compiler's standard directory). That file's contents will be included where the directive appears, as in the above example. Convention is to name such files with a ".h" suffix, "h" standing for "header" since include directives usually appear at the top or "head" of a ".c" file.

(Note: RIMS currently looks for include files in the "bin" folder where the RI toolkit was installed, typically under "C:\\Program Files\RI Toolkit\<RI toolkit version>\bin").

Another directive is "#define". Although we discourage this directive's use, the directive is so common in existing programs that we describe it here. An example is:

```
#define MAXITEMS 100
void main(){
   while(1) {
      if (A < MAXITEMS) {
         ...
   }
}
```

The preprocessor replaces all occurrences of MAXITEMS by 100 in the code. The syntax for the directive is: *#define identifier code*. Convention is that the identifier is in all capital letters. The preprocessor detects instances of the identifier below the directive (including in other #define directive code) and replaces the identifier by the specified code. Another form of #define allows for parameters and is sometimes used instead of defining a function. We strongly discourage both uses of #define. The former form can usually be replaced by a const variable declaration. The latter form can be replaced by function definitions.

## Type casting

A C program may have variables of different types, such as signed or unsigned, and char, short , or int. Care must be taken when assigning a variable of one type to a variable of another. For instance, assigning a signed variable to an unsigned variable may cause unexpected behavior. Consider the following example.

```
signed   short x;
unsigned short y;
x = 1;
y = x;    // OK: y is now 1
...
x = -1;
y = x;    // Problem: y will be 65535
```

x = -1 will store 1111111111111111 (the 16-bit two's complement representation of -1) into the 16-bit variable x. x = y will copy those 1s to the 16-bit variable y, but because y is unsigned, that value will be interpreted as 65535, e.g., printing x and y will output -1 and 65535, respectively. No reasonable programmer would actually intend such behavior.

A distinction exists between a legal program, as defined by the C language, and a correct program. The above code segment is legal and will compile without error (though there may be warnings), but the behavior may not be what a programmer expects and thus may execute incorrect behavior.

***Type casting*** is a programming mechanism to convert a variable of one type to a variable of a different type. The above was an instance of casting a signed variable to an unsigned variable. An unsigned variable can also be cast to a signed variable.

The C language allows assigning an integer variable of smaller size to an integer variable of larger size, and vice-versa. C's size casting mechanism is illustrated in the following example.

```
unsigned char  c = 9;
unsigned short i = c;  // Implicit casting
c = (char)i;  // Explicit casting (required to avoid warnings,
              // because i is larger than c)
```

Wikipedia: Type Conversion

In C, casting an integer to another integer of larger size is straightforward. The value remains the same, but its bit representation is widened. The compiler preserves the sign of the original value by filling the new leftmost bits with ones, if the value is negative, or with zeros, if the value is positive. When converting to an unsigned integer, the value is always positive, so the new bits are always filled with zeroes. Casting from smaller to larger can be done using implicit casting as shown above.

When an integer value is cast to another integer of narrower size, explicit casting is required, and the excess bits on the left are discarded. Explicit casting is done by indicating the new type in parenthesis just before the larger item, as shown above. Care must be taken to ensure the effect of discarding excess bits is acceptable for a given program.

In C, an expression that contains a mix of signed and unsigned variables yields a signed value, as shown below.

```
signed   short x, y;
unsigned short z;
...
x = y + z; // (y + z) is signed
```

An expression that contains a mix of variables of different sizes yields a value that is equal to the largest variable in the expression, such as shown below.

```
unsigned short x, y;
unsigned char  z;
...
x = y + z; // (y + z) is 16 bits
```

Type casts are the source of many bugs. Consider the following program[8]:

```
unsigned char Square(unsigned char x) {
   return(x*x);
}

void main(){
```

[8] Remember that all programs in RIMS must begin with the line: #include “RIMS.h”

```
   unsigned short a = 15;
   unsigned short b = 20;
   unsigned short c;
   while(1){
        c = Square(a);
        puti(a); puts(" squared is "); puti(c); puts("\n");
        c = Square(b);
        puti(b); puts(" squared is "); puti(c); puts("\n");
   }
}
```

Running the program in RIMS yields the following output:

15 squared is 225
20 squared is 144
…

The first output line is correct but the second clearly is not. What happened is that a, b, and c are declared as unsigned short, but the argument and return values of function Square are declared as unsigned char. The shorts are silently type cast to chars, whose maximum value is 255. 20 squared is 400, which is too large for a char and thus overflows, resulting in the "garbage" value of 144, which is then returned and cast back into a short. The programmer must pay much attention to variable types to avoid such bugs.

## Branching with a switch statement

Sometimes branches are intended to execute unique sub-statements depending on the particular value of a variable (or expression), such as in the following:

```
void main(){
   while(1){
      if (A == 0) {
         B = 0xFF;
      }
      else if (A == 1) {
         B = 0xAA;
      }
      else if (A == 2) {
         B = 0x55;
      }
      else {
         B = 0;
      }
   }
}
```

C has another branching construct called ***switch*** providing a more explicit means to execute the proper sub-statements according to the value of a variable (or expression). The above can be rewritten as:

```
void main(){
   while(1){
      switch(A) {
         case 0:
            B = 0xFF;
            break;
         case 1:
            B = 0xAA;
            break;
         case 2:
            B = 0x55;
            break;
         default:
            B = 0x00;
      }
   }
}
```

If the switch expression is 0, execution jumps directly to case 0, causing statements following the colon to execute, until reaching a break statement, which jumps to the end of the switch statement. If no case matches the switch expression, execution jumps to the 'default' case.

Try: Run the above program in RIMS. Press Break and then use Step to observe how execution jumps directly to the appropriate case based on A. Repeat for A=0, A=1, and A=4.

Sometimes multiple cases should have the same statements executed, achieved as follows:

```
      switch(A) {
         case 0:
            B = 0xFF;
            break;
         case 1:
            B = 0xAA;
            break;
         case 2:
         case 3:
            B = 0x55;
            break;
         default:
            B = 0x00;
      }
```

Above, if A is 2, execution jump to case 2, but then "falls through" to case 3 because there is no "break" statement. Thus, each of cases 2 or 3 causes the statement "B = 0x55" to execute.

Forgetting to include an intended "break" statement is a common source of bugs.

The general form of a switch statement is:

```
switch (expression) {
  case const-expr: statements
  case const-expr: statements
  default: statements
}
```

The "default" case is optional, but always including a default is good practice. The cases can occur in any order, not just ascending order as in the above examples, but ordering is good practice. The const-expr should be an integer constant (or a constant expression like "1+2", but such form is rare).

## Branching with a conditional operator

The following code pattern:

```
if (expression1) {
   x = expression2;
}
else {
   x = expression3;
}
```

is so common that C provides a shorthand version using the ***?:*** ternary operator, known as a ***conditional expression:***

```
x = (expression1) ? expression2 : expression3;
```

The items on the right side of the "=" form the conditional expression. Conditional expressions can appear anywhere than an expression can appear (so not just on the right side of an assignment statement). .

The following code, which blinks four output LEDs in sequence, provides an example:

```
unsigned char LedsTick(unsigned char leds) {
   /* if (leds == 8) {
      leds = 1;
   }
   else {
      leds = leds * 2;
   } */

   // The above code can be replaced by the one line below:
   leds = (leds == 8) ? 1 : leds * 2;

   B = leds;
   return leds;
}


void main(){
```

```
   unsigned char x = 1;
   while(1){
      x = LedsTick(x);
   }
}
```

## Static variables in a function

Normally, a variable declared inside a function is temporary -- the variable is created (i.e., allocated a place in memory) when the function is called, and then destroyed when the function returns. However, sometimes we wish for the variable to maintain its value between calls. Prepending the keyword ***static*** to a variable declaration in a function causes that variable to be created when the program begins executing and persisting throughout execution. Consider the following example:

```
void LedsTick() {
   static unsigned char leds = 1;
   leds = (leds == 8) ? 1 : leds * 2;
   B = leds;
}

void main(){
   while(1){
      LedsTick();
   }
}
```

The "leds" variable is declared static and initialized to 1; the initialization only occurs once at the beginning of the program's execution, and not each time the function is called. Each call to the LedsTick function modifies the current leds variable value based on its previous value. The previous value is available because the leds variable is declared static and thus retains its value.

Try: Run the above program in RIMS and notice how the LEDs blink in sequence (you may wish to move RIMS' slider to "Slowest"). Next, remove the word "static", compile, and run again. Notice that a single LED stays lit, because each time the LedsTick function is called, the leds variable is newly created and initialized.

Compared to a global variable, a static variable in a function has the advantage of only being accessible within the function, and thus is preferred if no other functions need access to that variable. Generally, though, use of global variables and static variables in functions should be kept to a minimum, as they tend to hide actual behavior and thus could make programs harder to understand.

# Book and Author Info

This book, Programming in C, differs from most books, tutorials, and websites introducing C, introducing material via core programming concepts such as the concept of a variable, an expression, an if-then-else statement, a while loop, a subroutine, etc. -- concepts found in most programming languages -- rather than dwelling on the language of C itself. The book introduces preferred C constructs and usage styles, bypassing many C constructs and usage styles stemming from C's creation before establishment of modern programming language discipline. The book uses simple realistic examples throughout.

The book was initially created to provide an introduction to C programming as background for programming embedded systems. Many examples utilize a microcontroller with input and output pins. This approach may in fact be a simpler starting point for beginning programmers. However, the book is not limited to embedded systems. A second book, Programming Embedded Systems (Vahid/Givargis, Uniworld, www.programmingembeddedsystems.com), teaches higher-level programming concepts related to disciplined time-oriented programming of embedded systems.

The book represents a modern approach to creating learning content. Its conciseness enables a complete read of key concepts, with reference information today available online. It is intended for electronic publication, being read on various devices, or being printed by the end user without excessive paper use. As such, the book passes on highly polished formatting and figures, in favor of content that is viewable on various devices, and is amenable to update and revision on a faster cycle than traditional book "editions," leading to a book that is more modern, and has a far lower purchase price.

The bok also uses a modern approach by stressing active learning, coming with tools that enable a reader to put learned concepts into practice. The Riverside-Irvine Microcontroller Simulator (RIMS) includes complete C capture, compilation, simulation, and debug for microcontroller programming, all in a single extremely easy-to-use graphical interface -- it is most likely the simplest tool for learning C programming, both in terms of tool installation and use. The book is thus well-suited for traditional in-person course as well as for online courses. The RI tools presently run on Microsoft Windows platforms (XP/Vista/7)..

The book and tools are based upon work supported by the U.S. National Science Foundation's CCLI (Course Curriculum and Laboratory Improvement) program under grant number DUE-0836905. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, and Associate Director of the Center for Embedded Systems at UC Irvine. He received his bachelor’s degree in electrical engineering from the University of Illinois at Urbana-Champaign, and his master’s and doctoral degrees in computer science from the University of California, Irvine. He has worked for Hewlett Packard and AMCC, and has consulted for Motorola, NEC, Atmel, Cardinal Health,

and several other engineering firms. He is the inventor on three U.S. patents and has published over 150 research papers. He is author of "Digital Design, with RTL Design, VHDL, and Verilog" (J. Wiley and Sons, 2011, 2nd ed), books on VHDL and Verilog, and co-author of "Embedded Systems Design: A Unified Hardware/Software Introduction" (J. Wiley and Sons, 2001), and "Specification and Design of Embedded Systems" (Prentice-Hall, 1994 -- perhaps the first published book on embedded systems). He helped establish the Embedded Systems Week conference and has chaired the CODES and ISSS symposia. He established UCR's Computer Engineering program, and has received several UCR teaching awards. His current research focuses on embedded systems design and programming for medical devices and for home/office monitoring and control. http://www.cs.ucr.edu/~vahid.

Tony Givargis is a Professor of Computer Science at the University of California, Irvine, and a member of the Center for Embedded Computer Systems at UC Irvine. He received his B.S. and Ph.D. in Computer Science from the University of California, Riverside in 1997 and 2001 respectively. His research focuses on Realtime Operating System (RTOS) synthesis, high-confidence embedded software, serializing compilers, and code transformation techniques for efficient software to hardware migration. He is coauthor of a textbook entitled "Embedded System Design: A Unified Hardware/Software Introduction," has published over 70 peer reviewed articles in the general areas of embedded system research, and is co-inventor on 8 patents. His passion for teaching has earned him the UC Irvine Information & Computer Science Dean's Award for the Excellence in Undergraduate Teaching in 2010, the UC Irvine Excellence in Teaching Award in 2003, and the UC Irvine Chancellor's Award for Excellence in Fostering Undergraduate Research, in 2005. http://www.ics.uci.edu/~givargis. He received the 2011 Terman award from ASEE for outstanding contributions to engineering education including authoring of a high-impact textbook.

See www.programmingembeddedsystems.com for further information, to obtain the most recent version of this book, or to obtain the RI tools. Send comments, questions, and corrections to info@programmingembeddedsystems.com.

This book was created using Google Docs.