Embedded Systems

A Review of ANSI C and Considerations for Embedded C Programming

Review of ANSI C Topics

• Basic features of C
  – C fundamentals
  – Basic data types
  – Expressions
  – Selection statements
  – Loops
  – Arrays (to be covered later)
  – Functions (to be covered later)
  – Pointers (to be covered later)
  – Program organization
C Fundamentals

• C is a “low-level” high-level language
  – Provides direct access to machine-level concepts
    • Bytes, addresses, low-level operators
    • Useful when directly programming system hardware

• C is a small language
  – This is good from an embedded systems perspective

• C is a permissive language
  – Most C compilers assume you know what you are doing
  – Promotes a wide degree of latitude compared to other languages
  – Does not provide detailed error checking mandated by many languages
    • Much like assembly language in this respect

C Strengths

• Efficiency
  – Originally intended for applications where assembly language had been used
  – Execute quickly and in a limited amount of memory

• Portability
  – Embedded systems to supercomputers
  – Small compilers, easy to write, widely available

• Power
  – Rich set of data types and operators

• Flexibility
  – Originally intended for systems programming
  – Few restrictions on use of its features

• Standard library
  – Many functions supported (as needed)

• Integration with UNIX
C Weaknesses

- C programs can be error prone
  - C’s flexibility makes it error prone
  - Many (logical) errors are not detected by the compiler
  - Most errors are not detected until runtime
    - Promotes need for good debugging practices and skills
- C programs can be difficult to understand
  - Terse syntax
  - Obfuscated C
- C programs can be difficult to modify
  - Large programs, if not designed with maintenance in mind, can be difficult to change
  - Limited features for the division of a large program (mainly routines)

Compiling and Linking

- Creating a machine executable usually involves three steps
  - Preprocessing – processes commands that begin with # (known as directives). Can add things to a program and make modifications
  - Compiling – translates code into machine language, producing object code (files)
  - Linking – links user code with any other code (library functions, other user code, etc.) to produce an executable
- Integrated Development Environments (IDEs) combine the above with an editor and possible a debugger
Form of a Simple Program

\[ \text{directives} \]
\[ \text{int main(\text{void})} \]
\[ \{ \]
\[ \text{statements} \]
\[ \} \]

- Example directive
  - \#include <stdio.h>
    - Include the information in stdio.h in the compilation process
    - Header files such as this contain some information about some part(s) of one or more library functions

Statements and Functions

- Statement – a command to be executed
  - Terminated with a semicolon
  - return 0; is an example of a statement
- Functions – like procedures (or subroutines) in other programming languages
  - A series of statements grouped together and given a name
    - May or may not return a value
- Grouped into two categories
  - Those written by the programmer (user functions)
  - Those provided as a part of the C implementation (library function)
Comments

• Take one of two forms
  – Multi-line comments
    • Begin with /* and end with */
      – /* This is a valid comment */
    • All text between those delimiters are considered part of the comment
    • Comments should not be nested
  – Single line comments (introduced in C99 standard)
    • Begin with // and extend to the end of the line
      – // This is a valid comment

Variable Declarations

• Every variable must have a **type** (defines an intended use)
• C data types include:
  – char – a single byte capable of holding a single character
  – int – an integer, typically reflecting the natural size of integers on the target processor
  – float – single precision floating point
  – double – double precision floating point
• Modifiers to the basic data types usually include **unsigned**, **long**, **short**
  – unsigned int would refer to an integer to be treated by the compiler as an unsigned quantity
• Individual compilers may support additional data types defined for the compiler/user
  – These are usually derived from the basic data types and are supported for program readability
Variable Assignments

- Variable declarations should come before assignment directives

```c
#include <stdio.h>
int i;
int main(void)
{
    // declaration and
    // initial assignment
    int j=0;
    i=j;
}
```

Basic Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Unary</th>
<th>Binary</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>unary plus</td>
<td>additive</td>
<td>Highest</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>/ division</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>% remainder</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator Precedence</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>+ - (unary)</td>
</tr>
<tr>
<td></td>
<td>* / %</td>
</tr>
<tr>
<td>Lowest</td>
<td>+ - (binary)</td>
</tr>
</tbody>
</table>
Compound Operators

- C supports compound operators for both manipulating a value and assigning the result to a variable.
- For example, `i=i+2;` can be written as `i+=2;`
- Other compound operators include:
  - `-=`
  - `*=`
  - `/=`
  - `%=`
- For example, `i=i/j;` can be written as `i/=j;`
- Increment operators (`++` or `--`) are also supported:
  - `i++;`
  - `++i;`
  - `i--;`
  - `--i;`

Operator Precedence

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Name</th>
<th>Symbol(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>increment (postfix)</td>
<td><code>++</code></td>
</tr>
<tr>
<td></td>
<td>decrement (postfix)</td>
<td><code>--</code></td>
</tr>
<tr>
<td>2</td>
<td>increment (prefix)</td>
<td><code>++</code></td>
</tr>
<tr>
<td></td>
<td>decrement (prefix)</td>
<td><code>--</code></td>
</tr>
<tr>
<td></td>
<td>unary plus</td>
<td><code>+</code></td>
</tr>
<tr>
<td></td>
<td>unary minus</td>
<td><code>-</code></td>
</tr>
<tr>
<td>3</td>
<td>multiplicative</td>
<td><code>* / %</code></td>
</tr>
<tr>
<td>4</td>
<td>additive</td>
<td><code>+ -</code></td>
</tr>
<tr>
<td>5</td>
<td>assignment</td>
<td><code>*= /= %= += -=</code></td>
</tr>
</tbody>
</table>
Selection Statements

- C has relatively few statements
- In addition to the return and expression/assignment statements, most C statements can be grouped into one of three categories
  - **Selection statements** – the *if* and *switch* statements allow a program to select a particular execution path
  - **Iteration statements** – the *while*, *do*, and *for* statements support iteration (looping)
  - **Jump statements** – the *break*, *continue*, and *goto* statements cause an unconditional jump to some other location in the program (*return* falls into this category as well)

Logical Expressions

- Logical expressions are used in selection statements to determine an operation to be performed
- Test operators (relational, equality, and logical) are commonly used in formulating logical expressions

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The if Statement

• The *if* statement allows for selection of program flow alternatives
• Allows for multiple statement execution
• Includes *else if* and *else* clauses
• Maybe nested as necessary

General form:
```c
if (expression) {
    statements;
}
else if (expression) {
    statements;
}
else {
    statements;
}
```

Conditional Expressions

• The *if* statement allows for performing an action based on a condition
• A C *conditional operator* allows for producing a value depending on the value of a condition
• General form:
  ```c
  expr1 ? expr2 : expr3
  ```
• Operation: If *expr1* is true then return *expr2*, else *expr3*
• Example:
  ```c
  int i=1, j=2, k;
  k = i > j ? i : j; // k=2
  k = ( i >= 0 ? i : 0) + j; // k=3
  ```

• One common use
• Instead of:
  ```c
  if (i>j)
      return i;
  else
      return j;
  ```
• write the following:
  ```c
  return i > j ? i : j;
  ```
The switch Statement

- The **switch** statement is useful in comparing an expression against a series of values (preferable to cascaded **if** statements)
- General form:
  ```
  switch (expression) {
    case constant-expression : statements 
    ...
    case constant-expression : statements
    default : statements
  }
  ```

The switch Statement (continued)

- **Controlling expression** – the **switch** keyword must be followed by an integer expression (a character is also allowed)
- **Case label** – each case begins with a label of the form **case constant-expression**:
  - The constant expression cannot contain variables or function calls
- **Statements** – one or more statements follow the label (no braces are required). The last statement is usually **break**
- A **default** case is usually the last case, but is not required
The while Statement

- The **while** statement provides the most basic looping functionality in C
- General form:
  - While (expression) statement
- Example
  ```c
  while (i<n) // controlling expression
  i*=2; // loop body
  ```
- Infinite loops can be created using syntax similar to
  - While (1) { statements }
  - Completely interrupt driven embedded systems will often have a main function similar to this, particularly if there is no operating system and no multitasking

The do Statement

- The **do** statement is closely related to the **while** statement (essentially a **while** statement whose expression is evaluated after each execution of the loop body
  - The loop body will execute at least once
- General form:
  - Do statement while (expression);
- Example
  ```c
  i=10;
  do {
      printf("T minus %d and counting\n", i--);
  }
  while (i>0);
  ```
The for Statement

- The **for** statement is probably the most widely used (and flexible) of all the C looping statements
  - Most useful when there is some “counting” variable associated with a loop
- General form:
  - `for (expr1 ; expr2 ; expr3) statement;`
  - `expr1` is usually an initial assignment statement, `expr2` is a test condition for loop termination, and `expr3` is an increment (or decrement) statement affecting a variable in `expr1`
- Example
  ```c
  for ( i=10 ; i>0 ; i-- ) {
    printf("T minus %d and counting\n", i);
  }
  ```

The break and continue Statements

- The **break** statement has already been used in the **switch** statement, but it can also be used to jump out of (i.e. break execution of) a **while**, **do**, or **for** loop
- The **continue** is similar to the **break** but it does not terminate the loop. Rather, control is transferred to the end of the loop.
- Example:
  ```c
  n=0, sum=0;
  while (n<10) {
    scanf("%d", &i);
    if (i==0) continue;
    sum += i;
    n++;
    // continue jumps to here
  }
  ```
Embedded Programming Considerations

• Most programming for embedded systems involves a high degree of interaction with hardware components
• Programming features of particular interest will be:
  – Polled and interrupt driven code
    • Depending on the hardware to be controlled
  – Efficient C code in terms of:
    • Size (code and data)
    • Execution time (performance and power utilization)
  – Integration with assembly language code as necessary
    • C calling assembly, assembly calling C
    • Sharing variables between C and assembly
  – Efficient memory usage, functions and function calls
    • How are the stack and heap memories used
    • How can heterogeneous memory availability be exploited