Lesson 26

Objectives: -

- A Note on Standards
- Programming: C Examples

Programming: C Examples

A Note on Standards

This section describes what the ISO C Standard is, and why you should care about it.

Introduction

The C programming language was created by Dennis Ritchie and Brian Kernighan in the early 1970s. It quickly gained popularity because it was small, efficient, and its code could be easily ported between different computer systems. For many years, the standard reference manual was Kernighan & Ritchie's book, "The C Programming Language".

However, despite this reference, compiler authors were free to implement the language however they chose, resolve any ambiguities in any way they chose, and provide any additional services they chose. The result was that code written to work on one system would often not work, or at least not work as expected, on another system, without significant modifications.

To remedy this, "in 1983 the American National Standards Institute (ANSI) established a committee whose goal was to produce 'an unambiguous and machine-independent definition of the language C,' whilst still retaining its spirit" ("The C Programming Language", second edition).

In 1990, ISO (the International Organisation for Standardization) and IEC (the International Electrotechnical Commission) issued ISO/IEC9899, an international standard "(specifiying) the form and (establishing) the interpretation of programs expressed in the programming language C". This was superseded in 1999 by a new standard from the same organisations. Unfortunately, since ANSI and ISO make their living through selling standards, this document is not legally available for free download.

The language described by this standard is known interchangeably as "ANSI C", "ISO C", and "standard C". The language existing before the standard, and described in Kernighan & Ritchie's first book is often known as "K&R C".

In short, the ISO C Standard describes what the C programming language is, how to use it correctly, what it can do, and what it can't do. It is a very important document, and every C programmer should be at least familiar with it, if not expert in every detail. The Usenet newsgroup comp.lang.c is an excellent place to learn correct, standard C programming. You should read the FAQ before posting there (a link to this is available in the links section of this site).

Why have a standard?

According to the abstract in the latest C standard:

> [The Standard's] purpose is to promote portability, reliability, maintainability, and efficient execution of C language

programs on a variety of computer systems.

Each of these factors is discussed below:

Portability

refers to the ability of a program to compile without problems, and execute as expected, on a wide variety of different computer systems.

Reliability

refers to the ability to be sure that your program will work as expected on a wide variety of computer systems, without having to worry about whether different compilers will interpret the same code in different ways.

Maintainability

refers to the fact that when written according to a prescribed set of rules, a program can be extended, debugged, checked and maintained by any person who is familiar with those rules. This avoids having to train new staff in a closed, proprietary language or dialect, regardless of the program or computer system being used.

Efficient execution

refers to the fact that when the behaviour of a given code snippet is well known, it can be relied on to exhibit that behaviour on a wide variety of computer systems. Therefore, code that is efficient on one platform will be efficient on another.

Thus, the key is in the phrase *a variety of computer systems*. The situation is analagous to many other things which have national or international standards. For instance, when you buy a light bulb, as long as you buy the right type, you don't have to worry about whether it will slightly too small, or slightly too large for the light fittings in your house, because they are made according to a predetermined standard size. Equally, when you write a C program according to the rules laid down in the Standard, you can be sure that it will compile and execute as expected, no matter which computer system you use (providing the computer system itself has a conforming implementation).

For DOS and MS Windows users in particular, portability is often seen as somewhat unnecessary,

perhaps because DOS and Windows are themselves closed, proprietary operating systems. In the UNIX world however, there is a large variety of systems which look and feel similar, but which are different enough to cause problems. Before the Standard, porting applications between these systems could be a long a troublesome process. After the introduction of the Standard, designers had a terms of reference by which they could build development tools which provide a consistent and reliable environment which, provided developers themselves stick to the rules, allow programs to be written which will build and run on any system with little or no modifications needed.

In short, if you write your code according to the Standard, it will:

- be easier for you to port it to other systems;
- be easier for other people to use your code;
- provide you with a common reference point by which to read about, discuss with others, and find assistance in C language programming.

What are the drawbacks?

Because the Standard aims at producing code which will work unaltered on a variety of computer systems, there is a necessary element of lowest common denominator; only those facilities which will be supported on all these systems can be specified.

In particular, the Standard does not specify how to create or manipulate computer graphics in any way. This is because:

- different computer systems deal with graphics in wildly different ways. Some systems on which C programs are expected to run do not implement graphics at all (e.g. checkout tills, microwave ovens, text-only dumb terminals
- the field of computer graphics is still developing at a rapid pace; any attempt to standardise it would

soon be obsolete, would not be adhered to, and may even inhibit such development

Other facilities that the Standard does not deal with include:

- sound and other multimedia
- direct hardware access, such as serial port control, mouse, printer or modem control
- operating system specific file system facilities, such as obtaining file sizes and permissions, creating and searching directories, and using pipes
- low level terminal control such as clearing the screen (imagine trying to clear the screen on a system whose only output is to a ticker tape device), positioning the cursor at a given point, detecting keypresses, turning off terminal echo, and changing text colour
- networking, using TCP/IP or any other protocol
- thread or process control
- rebooting or switching off the system

This is not meant to be an exhaustive list.

So what good is the Standard?

A common, and perfectly reasonable question, is that if the facilities specified by the Standard are so limited, what good is it at all?

The short answer is that some common code is better than none. Almost all non-trivial applications will have to use non standard code. However, a surprisingly large amount of code can be written using pure standard C. If these parts of the application can be segregated from the rest, then only the non standard parts need be ported, rather than recoding the entire program.

Also, if you take another look at the list above, you will notice that many of the unsupported facilities deal with input and output (including storage). Often, the meat of

the program can be written using standard C, and only the parts that get input from the user, and produce output, need be written using nonstandard extensions (although this is not always the case). A common object oriented programming paradigm is that the implementation should be separate from the interface. Separating what your program does from how it interacts with the user will also make your program easier to improve and extend, as you will not need to completely redesign the program just to make a change to the interface, or vice versa. This practice will also most likely result in a better designed and more well thought out application.

POSIX, and other standards

The situation is not as bleak as it may appear above. Although the ISO Standard described is the only formal standard applicable to the C programming language, there are a number of other "standards" of varying degrees of formality which can make porting applications dealing with facilities outside the scope of the Standard less painful.

POSIX is a standard dealing with programming and system-tool interfaces issued by the Institute of Electrical and Electronic Engineers (IEEE). As well as C library functions dealing with operating system calls, threads, file system management, signals, and process control, it also deals with more user oriented facilities such as the shell, command line utilities and so on. It was modelled on facilities already common in UNIX; Linux is a fully POSIX compliant operating system.

Other "standards" include:

- The X Window System provides a standard set of services for providing a graphical user interface under UNIX and other operating systems
- The Berkeley Sockets API provides a standard set of services for TCP/IP network programming.

Microsoft's Winsock is a Windows-ized port of this API
- curses is a terminal independent library for low level console control, available on UNIX and other platforms

C Examples

If you are new to C, please read a note on standards before continuing.

- Fundamental
- Mathematical
- Utilities
- CGI
- Financial
- Games & puzzles
- Xlib
- Microsoft Windows
- ncurses
- Berkeley Sockets API & TCP/IP

Fundamental

This section shows examples of basic C syntax and usage, including fundamental areas such as console and file input/output, arrays and pointers, functions, and structures. It also covers some of the more esoteric areas such as pointers to functions, and functions with variable numbers of arguments.

Some complete sample applications are also given, as a demonstration.

- Application: Hello, world!
- Console IO
- Arrays

- Strings
- Pointers
- Retrieving command line arguments
- Application: Number guesser
- File IO
- Structures
- Application: Address book
- Pointers to functions
- Variable length argument lists

Mathematical

This section shows algorithms to solve a variety of mathematical problems.

- Prime numbers
- Fibonacci numbers
- Permutations (with performance statistics)

Utilities

This section shows some more complex and complete utilities written with C. Concepts such as string handling, expression parsing and modifying the program's behaviour with command line arguments are explored.

- C to HTML
- Foreign language phrase tester
- A Simple Calculator
- Equation solver
- Web server logfile analyser

CGI

This section shows the fundamentals of CGI programming. It examines issues such as getting and sending data from/to the server, parsing URL-encoded input, and getting around the stateless nature of HTTP using hidden form fields and cookies.

- Important note on these programs
- FormEcho
- Hangman
- Calculator
- Cookies

Financial

This section shows some financial applications.

- Option price calculator

Games & Puzzles

This section shows some examples of algorithms useful to solve popular puzzles, and providing an "intelligent" computer opponent.

- Towers of Hanoi
- Maze Solver
- OXO

Xlib

This section shows some basic Xlib examples. Xlib is the lowest level C interface to programming for the X window system, the standard UNIX graphical user interface.

- "Hello, World!", X Window System style

Microsoft Windows

This section explains the basics of using the Win32 API to program for the Microsoft Windows operating system.

- "Hello, world!", Win32 style
- Basics:
  - Mapping modes, scrolling & zooming
  - Mouse & keyboard interaction
  - Dialog boxes
  - Menus, context menus and toolbars
  - Child windows and controls
  - Common dialogs and controls
  - Comprehensive Example
- Bitmaps
- Winsock

---

ncurses

This section shows the basics of using the ncurses library. ncurses is a "CRT screen handling and optimisation package" In English, it provides screen, window and terminal input operations. ANSI C provides no mechanisms for accurately manipulating terminal I/O (e.g. clearing the screen, getting a single character from the user, positioning text at a certain screen location, changing colours). ncurses provides the resources to do these things. It is a UNIX library, but has also been ported to other platforms, including DOS. A version of

ncurses is available with the DJGPP DOS compiler (see the links page).

-

1. Hello, world!

Function

This program is just about the simplest one you can write in C which actually does something visible. Traditionally the first program written in a new language (as it shows the basic, most simple format of a complete, working program), it simply outputs "Hello, world!" to the screen.

```
#include <stdio.h>




int main() {

    printf("Hello, world!\n");

    return 0;

}
```

Retrieving Command Line Arguments

Function

This program demonstrates how to access and retrieve the arguments passed via the command line. It simply echoes each command line argument back to standard out.

Programming Issues

`int argc` contains the number of command line arguments. This is required to be non-negative, but most environments set the first argument to the name of the program, or its path, so in practice `argc` is usually at least 1.

`char *argv[]` is an array of pointers to type `char`, and points to the actual command line arguments themselves. Thus, `argv[0]` points to the first command line argument, and `argv[argc-1]` points to the last one.

Note that all command line arguments are supplied to your program as strings. If you need numerical arguments, then you will have to convert them using functions such as `strtol` and `strtod` (see the Number Guesser application for an example of this). See the Towers of Hanoi example for a demonstration of how to receive and parse command line switches, or options.

Usage

Experiment with supplying different command line arguments, e.g.:

```
cmdline
cmdline 1 2 3
cmdline hello world
cmdline "hello world"
cmdline what does "this do?"
cmdline 'what does "this do?"'
cmdline 'ls'
cmdline `ls`
```

2.cmdline.c

```
/*

  CMDLINE.C
  =========
  (c) Paul Griffiths 1999
  Email: mail@paulgriffiths.net

  Demonstrates how to retrieve command line
arguments

*/


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

int main(int argc, char *argv[]) {
    int n = 0;

    while ( n < argc ) {
     printf("Command line argument %d is %s\n",
(n+1), argv[n]);
     ++n;
    }

    return EXIT_SUCCESS;
}
```

Number Guesser

Function

This is a simple guessing game. The computer picks a random number, and the user has to guess what it is in as few goes as possible.

By default, the computer picks a number between 1 and 100, but the user can specify a different maximum number by specifying it as the command line (see **usage** below).

Programming Issues

The following issues are involved:

- Choosing a random number. The statement `rand() % num + 1;` will pick a random number between 1 and n. Note the call to `srand()` near the beginning; this is necessary or the computer will pick the same set of "random" numbers every time the program is run.
- Getting input from the user. A simpler way would be to call `scanf()` which would work fine, providing the user entered numbers when they should. If the user enters something else (e.g. a character) then a "matching failure" occurs. `scanf()` is notoriously bad at coping with matching failures, so we implement a different method:
    1. Call `fgets()` to retrieve an entire string from stdin
    2. Call `sscanf()` on that string to retrieve the number

    Thus, if the user does not enter input we expect, we can just get a new string from the input stream without worrying about extraneous characters.

- Retrieving the maximum number the user specifies from the command line. The program simply checks whether the user entered one argument, and if they did, it tries to convert it to a valid number.

Usage

Run the program by typing:

```
guesser
```

for a number between 1 and 100. To specify a different maximum number, use:

```
guesser 20
```

where the computer will pick a number between 1 and 20 (or whatever number you specify

3.guesser.c

```
/*

  GUESSER.C
  =========
  (c) Paul Griffiths 1999
  Email: mail@paulgriffiths.net

  Simple number guessing game

*/


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>


/*  Global macro  */

#define BUFFER_SIZE             (100)


/*  Forward function declarations  */

void ParseCmdLine(int argc, char *argv[], int
*max);


/*  main() function  */

int main(int argc, char *argv[]) {
    int guess, target, goes = 0, max = 100;
    char buffer[BUFFER_SIZE];

    /*  Get maximum number from command line  */

    ParseCmdLine(argc, argv, &max);


    /*  Seed random number generator  */

    srand( (unsigned) time(NULL) );


    /*  Output welcome message  */

    printf("Number Guesser\n");
    printf("==============\n\n");
```

```
    /*  Loop until user wants to quit  */

    do {
     printf("I'm thinking of a number between 1
and %d.\n", max);
     printf("Can you guess what it is?\n\n");


     /*  Pick a random number to guess  */

     target = rand() % max + 1;


     /*  Loop until correct answer  */

     do {

         /*  Get a guess from the user  */

         printf("Enter your guess: ");
         fgets(buffer, BUFFER_SIZE - 1, stdin);
         while ( !sscanf(buffer, "%d", &guess) )
{
             printf("Enter a number! ");
             fgets(buffer, BUFFER_SIZE - 1,
stdin);
         }


         /*  Test whether user guessed the right
number  */

         if ( guess < target )
             printf("Too low! ");
         else if ( guess > target )
             printf("Too high! ");
         ++goes;

     } while ( guess != target );


     /*  User has guessed the right number  */

     printf("Well done, it was %d!\n", target);
     printf("You took %d goes.\n\n", goes);

     printf("Another go? (y/n) ");
     fgets(buffer, BUFFER_SIZE - 1, stdin);

    } while ( buffer[0] == 'y' || buffer[0] ==
'Y' );

    printf("Goodbye!\n");
    return EXIT_SUCCESS;
}
```

```
/*  Gets maximum number from command line  */

void ParseCmdLine(int argc, char *argv[], int
*max) {
    char *endptr = NULL;

    if ( argc == 2 ) {

      /*  Convert first command line argument
          to a number and store it in 'max'    */

      *max = strtol(argv[1], &endptr, 0);

      if ( *endptr ) {

          /*  Command line argument was not
numeric  */

          printf("You did not enter a valid
maximum number.\n");
          exit(EXIT_FAILURE);
      }
      else if ( *max < 2 ) {

          /*  No point guessing a number between 1
and 1  */

          printf("Maximum number must be at least
two!\n");
          exit(EXIT_FAILURE);
      }
    }
}
```

Questions: -

1. Try above program and check o/p