RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Strings and Debugging Techniques in C

"Code softly and carry a big debugger."

-- anon.

Text strings

- There is no explicit data type in C for a *string*
- C defines a string to be:
  an ASCII-NUL ('\0') terminated sequence of char so...

'H'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'

is just an array of char … but …

'H'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\0'

is an array of char which is also (i.e. *contains*) a string

Strings – I/O

- The following are equivalent:

```
char s1[4] = {'a','b','c','\0'};
char s2[] = {'a','b','c','\0'};
char s3[] = "abc";
```

- Note the string constant "abc" has the length 4, since the last character is the nul character '\0'

- You can also have:

```
char *p = "abc";
```

Strings – I/O

- As we have seen, strings can be printed using the %s format with *printf*

```
char s[] = "Hal";
printf("My name is %s\n",s);
```

Strings – I/O

- Similarly, strings can be read using *scanf*

```
char s[5];
printf("Enter your name: ");
scanf("%s",s);
printf("Hello %s\n",s);
```

- The maximum number of char read can be restricted with:

```
scanf("%4s",s);
```

Strings – I/O

- If similarly, strings can be read using *scanf*

```
char s[5];
printf("Enter your name: ");
scanf("%s",s);
printf("Hello %s\n",s);
```

- The maximum number of char read can be restricted with:

```
scanf("%4s",s);
```

Strings – I/O

- Function scanf is useful when input is in a fixed format (e.g. data files) but can be awkward to use if input needs validation:

```
char s[5];
int num;
printf("Enter integer: ");
scanf(%d",&num);
printf("Enter name: ");
scanf("%4s,s);
printf("num %d, name %s\n",num,s);
```

- What if 123.0 is entered as the integer input?

Preview: fgets

- Function *fgets* can be used to read a string up to a length of *num - 1*, from a stream (such as standard input, *stdin*), into an array buffer *str*
  - We cover this in more detail next week

```
fgets(str, num, stdin);
```

- It will also stop reading once a newline character or EOF (end-of-file) is encountered. If newline is read, it is added to the buffer. A nul character is added to the end of the read characters

String functions

There is an extensive library of functions within C for manipulating strings (see <string.h>)

```
char *strcpy(char *destn, char *source);
```

copies source to destn

```
int strcmp(char *s1, char *s2);
```

compares s1 with s2

```
char *strcat(char *s1, char *s2);
```

concatenates s2 on to s1

most of these depend on their parameters (arrays of char) being terminated with an ASCII-NUL ('\0')

… and many others … see also Standard Library Function examples …

String functions

There is an extensive library of functions within C for manipulating strings (see <string.h>)

```
char *strncpy(char *destn, char *source, size_t n);
```

copies up to n char from source to destn

```
int strncmp(char *s1, char *s2, size_t n);
```

compares up to n char of s1 with s2

```
char *strncat(char *s1, char *s2, size_t n);
```

concatenates first n char of s2 on to s1

strtok

- *strtok* breaks a string into a series of tokens, using a set of delimiters to determine where a token ends
- The string to be tokenised is passed to *strtok* on the first time it is called. Subsequent calls to *strok* are passed NULL
- The delimiter is a string of one or more delimiter characters

strtok

```
#include <stdio.h>
#include <string.h>
...
char line[23] = "03/04/2000,Suzan Smith";
char *token;
token = strtok(line, "/, ");
while (token != NULL){
      printf(":%s:\n",token);
      token = strtok(NULL,"/, ");
}
```

strol

- *strtol* converts a string to a long int

```
long int strtol(char *str, char **endptr, int base);
```

- base is the number system (e.g. base *10*)

- If an error occurs, *endptr* will point to the first character that failed the conversion, otherwise endptr will be NUL char

Extended example

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

int main(void){

        char line[12];
        char *token, *endPtr, *result;
        long num;

        printf("Enter birthdate in format dd/mm/yyyy\n");
        result = fgets(line,sizeof(line),stdin);
        if (result == NULL)
                printf("Failed to read a line\n");
```

Extended example (cont.)

```
        else {
                if (line[strlen(line)-1] == '\n')
                        line[strlen(line)-1] = '\0';
                token = strtok(line,"/");
                while (token != NULL){
                        num = strtol(token,&endPtr,10);
                        if (token == endPtr || *endPtr != '\0')
                                printf(":%s: invalid\n",token);
                        else
                                printf("Number is %ld\n",num);
                        token = strtok(NULL,"/");
                }
        }
        return 0;
}
```

Debugging and Testing

- Coding your algorithm is not the end of the software development process; next comes testing and debugging.
- You can very dramatically reduce the complexity of debugging by *coding-testing-debugging* incrementally (i.e. just a “few” lines of code at a time)
- Always, always indent your program consistently. If you change the program, change the indentation to match. It's worth the trouble. Debugging is aided enormously by easily readable code.
- Take the time to understand the error before you try to fix it. Remember the language rules and the software requirements before fixing bugs.

Debugging and Testing (cont'd)

- When your program compiles without error, you are still not likely to be done.
- You must test your programs carefully in order to find bugs.
- Your program must work correctly. The goal is not to get the right output for just the few sample inputs that you may have been provided with, but to get the right output for all valid inputs. When presented with invalid input, your program should respond with appropriate error messages and/or directions, and certainly should not "crash".
- Debugging is a skill that, like other aspects of programming, requires practice to master. So don't be afraid of bugs. Learn from them.

Debugging and Testing (cont'd)

- One essential debugging skill is the ability to read source code and to follow it the way the computer does (ie. mechanically, step-by-step) to see what it really does. This is hard.
- Often, it's a problem just to find the part of the program that contains the error. Most programming environments come with a debugger, which is a program that can help you find bugs.
- Older (but often as effective) approaches to debugging include inserting (temporarily) debugging statements into your program.
- Remember: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong!

Debugging Techniques

- Incrementally code-test-debug just a few lines at a time
  - start with say 6 lines at a time, this can be increased with experience.
- Use test-drivers to test functions and/or modules in isolation from the rest of the main program.
- Use function stubs for incomplete functions.
- Use program checkers such as lint, lclint, …
- Use debuggers, where available, for features including breakpoints, single-step execute, watch variables, …
- Use dynamic memory usage checkers, where available, for features including memory leak detection, buffer overrun detection, …

Kernighan & Pike on Debugging

- “Another effective [debugging] technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind. I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmers as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them to the bear before they could speak to a human counselor.”

  --"The Practice of Programming" by Brian W Kernighan & Rob Pike

A Debugging tool – gdb

- gdb is a "symbolic" debugger which means that you can examine your program during execution using the symbols (variable and function names etc) of your source code.

- These symbols usually do not get saved by compilation so you need to compile your program with the `-g` option in order for symbols to be saved in your executable file.

```
gcc -g myprog.c -o myprog
```

- This will compile source file `myprog.c` into executable file `myprog` including in file `myprog` the symbols (names) from `myprog.c`.

- Once you've compiled (without errors) your source code in this way, you can then run it using gdb with the command:

```
gdb myprog
```

A Debugging tool – gdb (cont'd)

- If your program failed with a core dump then you can use gdb to examine the core dump (e.g. where it was and variable values etc. when it crashed) by:

```
gdb myprog core
```

- After you start up gdb you can issue commands to gdb to control, monitor and modify your program's execution. There are a large number of commands available. Here are some as example (the abbreviation for each command is shown in square brackets):

```
[l]ist          – display source code
[br]eak 'function' | linenum
```

  – sets a breakpoint at 'function' or at linenum of your source code

  – when the program reaches this breakpoint, execution is suspended and you can examine your program using other gdb commands

  e.g.

```
br myFunc       – sets breakpoint at function 'myFunc'
br 120          – sets breakpoint at line no. 120
```

Valgrind – A memory debugger

- Valgrind is a memory debugger available on the coreteaching servers.
- In order to get the full information out of valgrind, you will once again need to use the `-g` compiler flag with `gcc`.
- A common problem in programs developed in C is use of variables without having initialised them.
- Students are often sure that they have been initialised but in most situations, if your program is showing random behaviour, it is likely due to a failure to initialise memory correctly.

Valgrind (Continued)

- Have a look at the file `buggy.c`.
- Let's test the program and see what's wrong:

```
valgrind ./buggy
```

- We get the following output:

```
==35093== Use of uninitialised value of size 8
==35093==    at 0x4005DC: main (buggy.c:24)
```

Splint (Secure Programming Lint)

- Splint is a static code validator.
- It looks for classic errors and unsafe practices and warns you about such things in your code.
- Some of the things it checks for are:
  - Unused variables
  - Dereferences of NULL pointers
  - Dangerous aliasing – aliasing is when a pointer is assigned to another pointer and changing the second pointer's contents might have unexpected consequences
  - Unreachable code or infinite loops.