Strings and Multiple Source Files

Comp-206 : Introduction to Software Systems
Lecture 15

Alexandre Denault
Computer Science
McGill University
Fall 2006
As you have probably noticed by now, C does not have a string primitive.

This means that traditional operators, such as = or == cannot be used with strings.

- string1 == string2 will compare the pointer addresses, not the content of the string.

C provided functions in the <string.h> library to manipulate and compare string.
The strlen functions returns the length of a string, not including the terminating null character.

- size_t strlen(const char *s);
Comparing Strings

- To compare two strings, use the `strcmp` function.
  - `int strcmp(const char *s, const char *t)`
- The function does a char by char comparison:
  - If both strings are equal, the function returns a 0.
  - The function returns a negative number if `s > t`
  - The function returns a positive number if `t > s`
- The `strncmp` function can also be used if only the first `n` characters of a string need to be compared.
  - `int strncmp(const char *s, const char *t, size_t n)`
The `strcat` function appends the `src` string to the `dest` string.

- `char *strcat(char *dest, const char *src);`

Once both strings have been concatenated, a terminating null character is added.

- The original null character at the end of `dest` is overwritten.

The strings may not overlap.

The `dest` string must have enough space for the result.

If only `n` characters from `src` need to be concatenated, the `strncat` function should be used instead.

- `char *strncat(char *dest, const char *src, size_t n);`
■ The `strcpy()` function copies the string pointed to by `src` to the array pointed to by `dest`.
  - `char *strcpy(char *dest, const char *src);`
■ The terminating `\0` character is also copied.
■ The strings may not overlap.
■ The destination string `dest` must be large enough to receive the copy.
■ In only `n` characters need to be copied, the `strncpy` function should be used instead.
  - `char *strncpy(char *dest, const char *src, size_t n);`
■ Note that if no null byte among the first `n` bytes of `src`, the result will not be null-terminated.
Small example application to manage a book list.

Striking similarity to assignment 2.

For modules:

- Library : Structure to hold collection of books.
- File : Utilities to save/load books.
- Main : Runs the application.
As mentioned before, functions and types in C need to be defined before they can be used.

For functions, we can solve this problem by declaring the functions before we use them (function prototype).

But what happens when we want to use a function defined inside another file?

C allows us to include a file inside another file.
  * We will take a look at this latter in the lecture.
When programming in C, code is usually separated into two types of files:
- Header files (\texttt{.h})
- Code files (\texttt{.c})

All the preprocessor commands, type declarations, global variable declaration and function prototypes are usually stored in the header file.
- Header files have the \texttt{.h} extension.

The actual code is stored in the code file.
- Code files have the \texttt{.c} extension.
#if !defined(BOOK)
#define BOOK

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

typedef struct {
    char * title;
    char * author;
    int pages;
} book;

book * createBook(char* title, char* author, int pages);
void printBook(book* myBook);
void unallocateBook(book* myBook);

#endif
#include "book.h"

book * createBook(char* title, char* author, int pages) {

    book* newBook;

    newBook = (book *)malloc(sizeof(book));
    newBook->title = (char *)malloc(strlen(title)+1);
    newBook->author = (char *)malloc(strlen(title)+1);
    newBook->pages = pages;
    strcpy(newBook->title, title);
    strcpy(newBook->author, author);

    return newBook;
}

void printBook(book* myBook) {
    ...
}
#include "book.h"
#include "library.h"
#include "file.h"

int main (int argc, char **argv) {

    library* mylibrary = createLibrary(20);

    loadLibrary("lib.txt", mylibrary);

    addBookToLibrary(mylibrary, createBook("Lotr", "Tolkien", 300));
    addBookToLibrary(mylibrary, createBook("Harry Potter", "Rowing", 50));
    addBookToLibrary(mylibrary, createBook("C Prog", "Kerning", 100));

    printLibrary(mylibrary);

...
The preprocessor processes a file before it is compiled.

- It removes comments from source files.
- It executes preprocessor commands (#define, #include).

Preprocessor commands (or directives) are most often found in the beginning of the source file.
■ Any instance of the #include directive is replace by the content of the filename attached to the directive.

■ #include directive come in two formats:
  • #include <filename>
  • #include “filename”

■ When using the include statement with the < >, the preprocessor searches for filename in the library directories of the operating system.

■ When using the include statement with the “ “, the preprocessor searches for filename in the same directory as the source file.
- `#define` directives are used to define symbolic names or values.
- Similar to constant variables, `#define` can be used to insert hard-coded values.
- However, `#define` commands are executed by the preprocessor, before the code is compiled.
- The directive will replace the defined keyword by the defined value.
  - `#define STEP 20`
  - In this case, all occurrence of the string `STEP` will be replaced by the number 20
#define vs const

- #define is more efficient
  - Const is a variable and requires memory.
  - #define is a text replacement and requires no additional memory

- const is safer
  - Since the constant is a variable, the compiler can safely type check it.
  - #define can have some weird interaction (next slide).

- My rule of thumb: unless you have a specific reason for using #define, use const.

- Good reasons for using #define include:
  - Memory is a concern
  - Performance is a concern
  - You need the constant in another preprocessor command.
Dangers of define

```c
#define C1 10/5
float const C2 = 10/5;

float C3 = 23.0 / C1;  // C3 = 11
float C4 = 23.0 / C2;  // C4 = 11.5
```

- Why? Because C3 gets preprocessed to
  ```c
  float C3 = 23.0 / 10/5;
  ```
- Because of the presence of 10 and 5, we get integer division.
- This example is pretty simple, but in large applications (hundreds of source files), this can be difficult to find.
Macros exploit the substitution power of `#define` directive to embed small functions in the code.

Macros have the following syntax:

- `#define name function`

A common example of macro is the maximum value macro.

- `#define max(A, B) ((A) > (B) ? (A) : (B))`

Not that macros are even more dangerous than `#define` statements.

- By themselves, they are usually ok.
- However, if you start mixing them, you might get some unexpected behavior.
It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing.

This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The two most common uses of conditional inclusions are:

- Making sure that a header is included only once.
- Adapting code to different OS.
Mixed an matched headers

- filea.c
- fileb.h
- filec.h
- filea.c
- fileb.c
- filec.c
To make sure that the contents of a file are included only once, the contents of the file are surrounded with a conditional like this:

```c
#if !defined(UNIQUE_KEYWORD)
#define UNIQUE_KEYWORD
/* contents of header go here */
#endif
```

Note that the defined variable should be unique to the file (unless you know exactly what you are doing).
Introduction to Make

- Make is an automated build utility.
- It automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- Although all our examples will be based on C programming, Make can be used with any language.
- These slides are based on the excellent Make Tutorial at http://theory.uwinnipeg.ca/gnu/make/make_toc.html
Make gets its instruction for “Makefile” file.

It's a collection of rules and instruction which explain how to compile your program.

The first rule in your make file is your default rule.

If you make a mistake building your rule, your application will not compile properly.

You just need to type “make” at the command to run make. This will run the default rule.

```make
make
```

- makefile is the default instruction file and is automatically used.
- To execute a specific action, specify that action as an argument.

```make
make clean
```
Anatomy of a Rule

```
target ... : dependencies ...
   command
   ...
```

- **Target**: either the name of the file you want to compile or the name of an action you want to perform.
- **Dependencies**: name of files needed to execute the rule.
- **Command**: Action that needs to be carried out.
  - A rule can have more than one command, one on each line.
  - You **need** to put a tab character at the beginning of every command line!
Example makefile

librarydemo : main.o library.o file.o book.o
    gcc -Wall -g -ggdb -o library main.o library.o file.o book.o

main.o : main.c file.h library.h book.h
    gcc -c -Wall -g -ggdb main.c

file.o : file.c file.h library.h book.h
    gcc -c -Wall -g -ggdb file.c

library.o : library.c library.h book.h
    gcc -c -Wall -g -ggdb library.c

book.o : book.c book.h
    gcc -c -Wall -g -ggdb book.c

clean :
    rm -f library main.o library.o file.o book.o
Using Variables

objects = main.o library.o file.o book.o
coptions = -Wall -g -ggdb

librarydemo : ${objects}
    gcc ${coptions} -o library ${objects}

main.o : main.c file.h library.h book.h
    gcc -c ${coptions} main.c

file.o : file.c file.h library.h book.h
    gcc -c ${coptions} file.c

library.o : library.c library.h book.h
    gcc -c ${coptions} library.c

book.o : book.c book.h
    gcc -c ${coptions} book.c

clean :
    rm -f library ${objects}
Implicit rule

- Make has an implicit rule for updating a ".o" file from a correspondingly named ".c" file.
- Although I don't recommend you using it, I'm showing them to because will see them used frequently.

objects = main.o library.o file.o book.o

diskdemo : ${objects}
    gcc ${coptions} -o library ${objects}

main.o : file.h library.h book.h
file.o : file.h library.h book.h
library.o : library.h book.h
book.o : book.h

clean :
    rm -f library ${objects}