The standard library and the preprocessor

Lecture 9
CS 113 – Fall 2007

Announcements

- Assignment 3 posted, due next Friday
  - Do two of the three problems
- Quiz on Friday
  - Written quiz, ~30 minutes

Lecture outline

- Assignment 2, Problem 3
- Command line arguments
- The C standard library
  - File I/O
- The preprocessor

Command-line arguments

- Command line arguments

```
C:\> myprogram -x file1.txt file2.txt
```

- Command line arguments are passed as parameters to `main()`
  - Full prototype of main: `int main(int argc, char *argv[])`
  - `argc` contains the *number* of command line arguments
  - `argv` is an array of strings, one per command line argument
  - e.g. for the command line above:
    - `argc = 4`
    - `argv[0]: "myprogram"`
    - `argv[1]: "-x"`
    - `argv[2]: "file1.txt"`
    - `argv[3]: "file2.txt"`

Command line arguments example

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int j;

  for(j=0; j<argc; j++)
    printf("%d %s\n", j, argv[j]);

  return 0;
}
```

```
C:\> program 12 -x test.txt
0 program
1 12
2 -x
3 test.text
```


The C standard library

- Library contains many useful functions
  - Caution: Most functions are standardized, but there are differences between compilers and architectures.
- Main library headers
  - stdio.h : input/output
  - string.h : string manipulation
  - stdlib.h : memory/process management, conversions, etc.
  - time.h : time and date manipulation
  - math.h : mathematical routines
  - and others…

stdio.h

- Console I/O
  - printf, scanf, gets, puts
- File manipulation
  - fopen, fclose, remove, rename
- Text file I/O
  - fprintf, fscanf, fgets, fputs
- Binary file I/O
  - fread, fwrite, fgetc
- String I/O
  - sprintf, sscanf

string.h

- Concatenation
  - strcat, strncat
- Comparison
  - strcmp, strncmp
- Searching
  - strchr, strrchr, strstr
- Copying
  - strcpy, strncpy, memcpy, memmove
- Length
  - strlen

stdlib.h

- Type conversion
  - atoi, atol, atof
- Random number generation
  - rand, srand
- Memory allocation
  - malloc, free
- Searching, sorting
  - qsort, bsearch
- Process control
  - exit, system

math.h

- Absolute value
  - abs, fabs
- Exponentiation, logarithms
  - pow, log, log10
- Trig functions
  - cos, sin, tan, sinh, cosh, etc.
  - acos, asin, atan
- Rounding
  - floor, ceil
- Square root
  - sqrt

time.h

- Reading current time and date
  - time
- Time comparison
  - difftime
- Time conversion
  - gmtime, mktime, asctime

File I/O in C

1. Open the file using `fopen()`
   - `fopen()` returns a file handle that you use to perform I/O
2. Perform any I/O operations
   - Most of the input and output routines we've seen have a version for file I/O: `fprintf()`, `fscanf()`, `fgets()`, `fputs()`, etc.
   - Routines expect a file handle as the first parameter
3. Close the file using `fclose()`
   - Frees memory associated with the file handle
   - Writes may not actually occur until file is closed

Opening a file

- When you open a file, you specify a filename and a *file mode*
- The file mode specifies how you want to access the file
  - Give a string indicating the mode
  - For reading: `"r"`
  - For writing: `"w"`
  - For append: `"a"`
- Return value is a file handle of type `FILE *`
  - `FILE` is a struct defined in `stdio.h`
  - You pass it to subsequent file I/O routines

File I/O example

```
#include<stdio.h>

int main(int argc, char *argv[])
{
  FILE *file1, *file2;
  char buf[1024];

  file1 = fopen(argv[1], "r");
  file2 = fopen("out.txt", "w");

  while(!feof(file1)) {
    fgets(buf, 1024, file1);
    fprintf(file2, buf);
  }

  fclose(file1);
  fclose(file2);
  return 0;
}
```


File I/O error handling

- Check the return values of functions to detect errors
  - `fopen()` : returns zero on failure
  - `fclose()` : returns non-zero on failure
  - `fgets()` : returns zero on failure
  - `fputs()` : returns non-zero on failure
  - `fprintf`, `fscanf()` : returns negative on failure
- Global variable `errno` stores which error occurred
  - e.g. file not found, disk full, etc.
  - Call `perror()` to print a helpful error message automatically

Error handling example

```
#include<stdio.h>

int main(int argc, char *argv[])
{
  FILE *file1, *file2;
  char buf[1024];

  file1 = fopen(argv[1], "r");
  file2 = fopen("out.txt", "w");
  if(!file1 || !file2) goto error;

  while(!feof(file1)) {
    if(!fgets(buf, 1024, file1)) goto error;
    if(fprintf(file2, buf) < 0) goto error;
  }

  if(fclose(file1) || fclose(file2)) goto error;
  return 0;
error:
  perror(""); // automatically prints correct error message
  return 1;
}
```


The Preprocessor

- The *preprocessor* is the first stage of the C compiler
- The preprocessor looks for lines that begin with `#`
  - These are called *preprocessor directives*
  - e.g. `#include`, `#define`, `#ifdef`, etc.

#include

- `#include` instructs the preprocessor to insert the *entire contents* of another file into the current file
- Two forms
  - #include "header.h"
    - Look for header.h in the same directory as the source file
    - Typically used for custom header files you've written
  - #include<header.h>
    - Look for header.h in the compiler's *include path*
    - Typically used for standard library header files

Multiple source files

- C lets you break up a program into several source files
  - Unlike Java or Matlab, it doesn't *require* you to do this
  - How you break up the program is up to you
- To do this,
  - Put the code for a function in exactly one file
  - Put a function prototype in any file that uses the function

Example: multiple source files

main.c

```
#include <stdio.h>

int power(int base, int exp);
int cube(int base);

int main(void)
{
  printf("%d\n", power(5, 3));
  printf("%d\n", cube(10));
  return 0;
}
```

power.c

```
int power( int base, int exp )
{
  int i, p = 1;
  for( i = 1; i <= exp; i++ )
    p *= base;
  return p;
}
```

cube.c

```
int power(int base, int exp);

int cube(int base)
{
  return power(base, 3);
}
```


Header files

- A C header file contains function prototypes
  - But does *not* contain implementation code
  - Files have extension `.h` by convention
- e.g. a snippet of `stdio.h`:

```
void     perror(char *);
int      printf(char *, ...);
int      putc(int, FILE *);
int      putchar(int);
int      puts(char *);
int      remove(char *);
int      rename (char *, char *);
void     rewind(FILE *);
int      scanf(char *, ...);
void     setbuf(FILE *, char *);
int      setvbuf(FILE *, char *, int, size_t);
int      sprintf(char *, char *, ...);
int      sscanf(char *, char *, ...);
```


Example: header files

mymath.h

```
int power( int base, int exp );
int cube( int base );
```

main.c

```
#include <stdio.h>
#include "mymath.h"

int main(void)
{
  printf("%d\n", power(5, 3));
  printf("%d\n", cube(10));
  return 0;
}
```

power.c

```
int power( int base, int exp )
{
  int i, p = 1;
  for( i = 1; i <= exp; i++ )
    p *= base;
  return p;
}
```

cube.c

```
#include "mymath.h"

int cube(int base)
{
  return power(base, 3);
}
```


#define

- `#define` creates *macros*
  - Syntax `#define macro_name macro_value`
  - The preprocessor replaces `macro_name` with `macro_value` everywhere it appears in the program
  - Typically used for defining constants

```
#define CENT_TO_INCHES 2.54
#define SIZE 10

int main() {
  double inches[SIZE], centimeters[SIZE];
  int j;
  /* read in inches ... */

  for(j=0; j<SIZE; j++)
    centimeters[j] = CENT_TO_INCHES * inches[j];

  return 0;
}
```


#define

- Advantages to macros
  - Makes it easier to change constants
  - Makes code easier to read
  - No overhead associated with a variable
- A macro is not the same as a variable
  - The preprocessor literally does a search-and-replace in your source code, before it is seen by the rest of the compiler

Macros with arguments

- You can `#define` macros with parameters
  - Looks like a function call, but it isn't
  - Like functions, macros make code easier to read
  - But don't have the overhead associated with a function call
  - Compare:

Macro

```
#define sum_sqrs(a, b)  a*a + b*b

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(num1, 5.0);
  return 0;
}
```

Function

```
double sum_sqrs(double a, double b) {
  return a*a + b*b;
}

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(num1, 5.0);
  return 0;
}
```


Macros with arguments

- Preprocessor literally does a search-and-replace operation before the code is compiled

Before processing

```
#include <stdio.h>
#define sum_sqrs(a, b)  a*a + b*b

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(num1, 5.0);
  return 0;
}
```

After preprocessing

```
/* ... contents of stdio.h here ...*/

int main() {
  double num1 = 3.0, ssq;

  ssq = num1*num1 + 5.0*5.0;
  return 0;
}
```


Macro pitfalls

- "Search-and-replace" can be problematic
  - It's easy to write macros that have unintended consequences

Before processing

```
#include <stdio.h>
#define sum_sqrs(a, b)  a*a + b*b

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(num1+3, 5.0);
  return 0;
}
```

After preprocessing

Macro pitfalls

- "Search-and-replace" can be problematic
  - It's easy to write macros that have unintended consequences

Before processing

```
#include <stdio.h>
#define sum_sqrs(a, b) (a)*(a)+(b)*(b)

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(num1+3, 5.0);
  return 0;
}
```

After preprocessing

```
/* ... contents of stdio.h here ...*/

int main() {
 double num1 = 3.0, ssq;

 ssq = (num1+3)*(num1+3) + (5.0)*(5.0);
 return 0;
}
```


Macro pitfalls

- "Search-and-replace" can be problematic
  - It's easy to write macros that have unintended consequences

Before processing

```
#include <stdio.h>
#define sum_sqrs(a, b) (a)*(a)+(b)*(b)

int main() {
  double num1 = 3.0, ssq;

  ssq = sum_sqrs(rand(), 5.0);
  return 0;
}
```

After preprocessing

More preprocessor features

- The preprocessor has many other features
  - check K&R for details