Intermediate Programming

Day 6

Outline

- Exercise 5
- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

Exercise 5

- Copy reverse string
- Add a null terminator

count1.c

```
int main()
{
	…
	// TODO: set the value of rev_comp[rci] for every valid index
	for( rci=0 ; rci<dna_len ; rci++ )
	{
		switch( dna[ dna_len-1-rci ] )
		{
			case 'A': rev_comp[rci] = 'T' ; break;
			case 'T': rev_comp[rci] = 'A' ; break;
			case 'C': rev_comp[rci] = 'G' ; break;
			case 'G': rev_comp[rci] = 'C' ; break;
			default:
				fprintf( stderr , "[ERROR] Bad character: %c\n" ,
					dna[ dna_len-1-rci] );
		}
	}

	// TODO: add the null character to the end of rev_comp
	rev_comp[dna_len] = 0;
	…
}
```

Exercise 5

- Count occurrences of digit, whitespace, and alphabet characters

count2.c

```
int main()
{
	…
	// TODO: count alphabetical, digit and whitespace characters.
	// Optional challenge: instead of using isalpha, isdigit and
	// isspace, use relational operators and your knowledge of the
	// characters' ASCII values: http://www.asciitable.com

	for( int i=0 ; i<text_len ; i++ )
	{
		if( text[i]>='0' && text[i]<='9' ) num_digits++;
		if( text[i]>='A' && text[i]<='Z' ) num_alpha++;
		if( text[i]>='a' && text[i]<='z' ) num_alpha++;
		if( text[i]==' ' || text[i]=='\t' ) num_space++;
		if( text[i]=='\n' || text[i]=='\r' ) num_space++;
	}
	…
}
```

Exercise 5

- Count occurrences of every character

count3.c

```
int main()
{
        …
        // TODO A: with a single loop, count the # occurrences of
        //        each ascii character
        // HINT: use each char of the text as an offset into the
        //       ascii_count array, then update using increment (++)

        for( int i=0 ; i<text_len ; i++ ) ascii_count[ text[i] ]++;
        …
}
```

Exercise 5

- Find the top two most frequently occurring characters

count3.c

```
int main()
{
	…
	// TODO B: With a single loop find the most frequent and
	//		second-most-frequent characters in the text.
	//		Store most frequent character and its frequency
	//		in top_char and top_freq.
	//		Store second-most-frequent character and its
	//		frequency in next_char and next_freq.
	for( int i=0 ; i<256 ; i++ )
	{
		if( ascii_count[i]>top_freq )
		{
			next_freq = top_freq , next_char = top_char;
			top_freq = ascii_count[i] , top_char = i;
		}
		else if( ascii_count[i]>next_freq )
			next_freq = ascii_count[i] , next_char = i;
	}
	…
}
```

Outline

- Exercise 5
- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

File I/O

- To read to / write from the command line, we use the commands
  - `int printf( const char format_str[] , … );`
  - `int scanf( const char format_str[] , … );`

- These are special instances of more general functions:
  - `int printf( format_str[] , … ) = fprintf( stdout , format_str , … );`
  - `int scanf( format_str[] , … ) = fscanf( stdin , format_str , … );`

- `stdout` and `stdin` are instances of file-handles

File-handles

- Different operating systems store data in different ways
- To avoid having to tailor code to the OS, C supports *file-handles*
  - These are abstract representations of objects we can read from / write to
    - Files on disk
    - Command line
    - Sockets across a network
    - etc.

File-handles

- When working with file handles we:
  1. Create a file handle
  2. Access the file's contents
  3. Close the handle

```
#include <stdio.h>
int main( void )
{
	FILE *fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , “hello\n” );
	fclose( fp );
	return 0;
}
```

File-handles (opening)

FILE *fopen(const char file_name[] , const char mode[]);

```
#include <stdio.h>
int main( void )
{
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp )
    {
        fprintf( stderr , … );
        return 1;
    }
    fprintf( fp , “hello\n” );
    fclose( fp );
    return 0;
}
```

File-handles (opening)

FILE *fopen(const char file_name[] , const char mode[]);

- Input:
  - The name of the file

```
#include <stdio.h>
int main( void )
{
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp )
    {
        fprintf( stderr , … );
        return 1;
    }
    fprintf( fp , "hello\n" );
    fclose( fp );
    return 0;
}
```

File-handles (opening)

```
FILE *fopen( const char file_name[] , const char mode[] );
```

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string composed of characters indicating access intent
    - ‘r’: read
    - ‘w’: write
    - ‘a’: append
    - ‘b’: binary*

```
#include <stdio.h>
int main( void )
{
	FILE *fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , “hello\n” );
	fclose( fp );
	return 0;
}
```

*More on binary file I/O later

File-handles (opening)

<u>FILE *</u>fopen(const char file_name[] , const char mode[]);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle*

```
#include <stdio.h>
int main( void )
{
	FILE *fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , … );
		return 1;
	}
	fprintf( fp , “hello\n” );
	fclose( fp );
	return 0;
}
```

*More on pointers soon

File-handles (opening)

FILE *fopen(const char file_name[] , const char mode[]);

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle
    - The function returns NULL (zero) if the system couldn't open the file
      - reading: file doesn't exist
      - reading: file/directory isn't ours
      - writing: the file is already open
      - writing: file/directory isn't ours

```
#include <stdio.h>
int main( void )
{
    FILE *fp = fopen( "foo.txt" , "w" );
    if( !fp )
    {
        fprintf( stderr , … );
        return 1;
    }
    fprintf( fp , "hello\n" );
    fclose( fp );
    return 0;
}
```

File-handles (opening)

```
FILE *fopen( const char file_name[] , const char mode[] );
```

- Input:
  - The name of the file
  - The mode in which to open the file
    This is a string of characters indicating intent
- Output:
  - A pointer to a file-handle
    - The function returns NULL (zero) if the system couldn't open the file

    ⇒ Check to make sure the command succeeded

```
#include <stdio.h>
int main( void )
{
	FILE *fp = fopen( "foo.txt" , "w" );
	if( !fp )
	{
		fprintf( stderr , ... );
		return 1;
	}
	fprintf( fp , “hello\n” );
	fclose( fp );
	return 0;
}
```

File-handles (accessing)

- Commands for reading from / writing to a file
  - Writing:
    - int fprintf(FILE *fp , const char format_str[] , ...);
      - Writes a formatted string to the specified file-handle
      - Returns the number of characters written (a negative value if the write failed)

```
#include <stdio.h>
int main( void )
{
	FILE* fp = fopen( "foo.txt" , "w" );
	if( !fp ) …
	fprintf( fp , “hello\n” );
	fclose( fp );
	return 0;
}
```

File-handles (accessing)

- Commands for reading from / writing to a file
  - Reading:
    - int fscanf(FILE *fp , const char format_str[] , ...);
      - Reads a formatted string from the specified file-handle
      - Returns the number of variables successfully set

```
#include <stdio.h>
int main( void )
{
	char word[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) ...
	while( fscanf( fp , "%s" , word )==1 )
		printf( "Read: %s\n" , word );
	fclose( fp );
	return 0;
}
```

File-handles (accessing)

- Commands for reading from / writing to a file
  - Reading:
    - int fscanf(FILE *fp , const char format_str[] , ...);
      - Reads a formatted string from the specified file-handle
      - Returns the number of variables successfully set

```
#include <stdio.h>
int main( void )
{
	char word[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) ...
	while( fscanf( fp , "%s" , word )==1 )
		printf( "Read: %s\n" , word );
	fclose( fp );
	return 0;
```

[NOTE] This function could be unsafe as we might read in a string longer than word.

File-handles (closing)

```
int fclose( FILE *fp );
```

- Input:
  - The file-handle
- Output:
  - Returns 0 if the file was successfully closed (EOF* if it wasn't)

```
#include <stdio.h>
int main( void )
{
	char word[512];
	FILE* fp = fopen( "foo.txt" , "r" );
	if( !fp ) …
	while( fscanf( fp , "%s" , word )==1 )
		printf( "Read: %s\n" , word );
	fclose( fp );
	return 0;
}
```

*EOF is an int, typically with value -1.

File-handles (testing)

```
int feof( FILE *fp );
```

- Input:
  - The file-handle
- Output:
  - Returns non-zero (true) if we have read to the end of the file.

```
int ferror( FILE *fp );
```

- Input:
  - The file-handle
- Output:
  - Returns non-zero (true) if the file is in an error state

Aside

- In addition to `scanf` and `fscanf`, C also defines `sscanf`:

  ```
  int sscanf( const char str[] , const char format_str[] , ... );
  ```

- Instead of scanning content from the command prompt or a file, it tries to can it from a C string (the first argument).

- This may be preferrable to using functions `atoi` and `atof` since the return value lets you know if the integer/float was parsed correctly. (The functions `atoi` and `atof` return zero if they fail, but do not inform you of the failure.)

stdin, stdout, and stderr

- C defines three file-handles:
  - standard input (stdin): the command prompt, for reading
  - standard output (stdout): the command prompt, for writing
  - standard error (stderr): the command prompt, for writing error messages

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out
This is not an error message
This is an error message
>>
```

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

- These are separate file-handles! (e.g. You can redirect them separately)

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out > foo.txt
This is an error message
>>
```

stdin, stdout, and stderr

stdout and stderr are both file-handles that allow writing to the command prompt

- These are separate file-handles! (e.g. You can redirect them separately)

```
#include <stdio.h>
int main( void )
{
	fprintf( stdout, "This is not an error message\n" );
	fprintf( stderr , "This is an error message\n" );
	return 0;
}
```

```
>> ./a.out > foo.txt
This is an error message
>> more foo.txt
This is not an error message
>>
```

Outline

- Exercise 5
- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

assert

- Although your code compiles and runs, it doesn't mean that it does the right thing.
- Sometimes you would like to verify (sanity check) that the code does the right thing.

```
#include <stdio.h>

int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {
        d = a[i+1]-a[i];

        printf( "%d\n" , d );
    }
    return 0;
}
```

assert

- Although your code compiles and runs, it doesn't mean that it does the right thing.
- Sometimes you would like to verify (sanity check) that the code does the right thing.
- C allows you to "assert" that a desired behavior is preserved.
  - Include the `assert.h` header file
  - assert the validity of a test
    - If the argument is true, nothing happens
    - Otherwise, the code aborts and a core dump file is generated

```c
#include <stdio.h>
#include <assert.h>
int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {
        d = a[i+1]-a[i];
        assert( d>=0 );
        printf( "%d\n" , d );
    }
    return 0;
}
```

assert

- Although your code compiles and runs, it doesn't mean that it does the right thing.
- Sometimes you would like to verify (sanity check) that the code does the right thing.
- C allows you to "assert" that a desired behavior is preserved.
  - Include the `assert.h` header file
  - `assert` the validity of a test
    - If the argument
    - Otherwise, the

```
#include <stdio.h>
#include <assert.h>
int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {
        d = a[i+1]-a[i];
        assert( d>=0 );
        printf( "%d\n" , d );
    }
    return 0;
}
```

```
>> ./a.out
a.out: foo.c:15: main: Assertion `d>=0' failed.Abort (core dumped)
>>
```

assert

- assert is defined as a macro*
  - ✓ Once we are convinced that the code is correct, we can disable all assert statements so they are not evaluated.
    - This can make the code execute more efficiently.

```
#include <stdio.h>
#include <assert.h>
int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {
        d = a[i+1]-a[i];
        assert( d>=0 );
        printf( "%d\n" , d );
    }
    return 0;
}
```

*more on this later (maybe)

assert

- *assert* is defined as a macro*
  - ✓ Once we are convinced that the code is correct, we can disable all *assert* statements so they are not evaluated.
    - This can make the code execute more efficiently.
  - ✗ If the assert statement sets in addition to testing, the setting will not be performed either.
    (Similar issues happen with short-circuiting if we set in the second predicate.)

```
#include <stdio.h>
#include <assert.h>
int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {

        assert( (d = a[i+1]-a[i])>=0 );
        printf( "%d\n" , d );
    }
    return 0;
}
```

*more on this later (maybe)

assert

- *assert* is defined as a macro*
  - ✓ Once we are convinced that the code is correct, we can disable all *assert* statements so they are not evaluated.
    - This can make the code execute more efficiently.
  - ✗ If the assert statement sets in addition to testing, the setting will not be performed either.

You should use assert to sanity check your code.
⇒ If your code is correct, the *assert* should never be triggered.

You should not use it to handle malformed user input:
- Failing to open a file for reading.
- Failing to convert a string to a number
- Etc.

```
#include <stdio.h>
#include <assert.h>
int main( void )
{
    int a[] = { 11 , 7 , 9 , 5 , 8 , 4 , 2 };
    int d , sz = sizeof( a ) / sizeof( int );
    // Sort the integers (poorly)
    …

    // Print the differences
    for( int i=0 ; i<sz-1 ; i++ )
    {

        assert( (d = a[i+1]-a[i])>=0 );
        printf( "%d\n" , d );
    }
    return 0;
}
```

*more on this later (maybe)

Outline

- Exercise 5
- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

Functions

- A function takes multiple arguments and returns (at most) one value

```
int foo( char c , int i )
{
        return i;
}
```

Functions

- A function takes multiple arguments and returns (at most) one value

```
int foo( char c , int i )
{
        return i;
}
```

  - The function name

Functions

- A function takes multiple arguments and returns (at most) one value

```
int foo( char c , int i )
{
        return i;
}
```

  - The function name
  - The return type (could be `void` if nothing is returned, needs to be stated explicitly)

Functions

- A function takes multiple arguments and returns (at most) one value

```
int foo( char c , int i )
{
        return i;
}
```

  - The function name
  - The return type (could be `void` if nothing is returned, needs to be stated explicitly)
  - The list of argument types

Functions

- A function takes multiple arguments and returns (at most) one value

```
int foo( char c , int i )
{
        return i;
}
```

  - The function name
  - The return type (could be `void` if nothing is returned, needs to be stated explicitly)
  - The list of argument types
  - The function body
    - Needs to be in braces, even if the function is just one command
    - Needs to return something of the type it promised to return

Functions

- We’ve seen that `string.h` provides a number of useful functions for processing strings:
  - `size_t strlen( const char str[] ){ … }`
    - Returns the length of a string
  - `char *strcpy( char destination[] , const char source[] ){ … }`
    - Copies the source string into the destination
  - `char *strcat( char destination[] , const char source[] ){ … }`
    - Concatenates the source string to the destination
  - etc.

Functions

- Similarly `math.h` provides a number of useful functions for processing numbers:
  - double sqrt(double x)
    - Returns the square-root, $\sqrt{x}$
  - double exp(double x)
    - Returns the exponential, e^x
  - double pow(double x , double y)
    - Returns the exponential of the base, x^y
  - double cos(double x)
    - Returns the cosine of an angle (in radians)
  - double ceil(double x)
    - Returns the ceiling of a number, $\lceil x \rceil$
  - etc.

Functions

- Similarly `math.h` provides a num
  numbers:
  - double sqrt(double x)
    - Returns the square-root, $\sqrt{x}$
  - double exp(double x)
    - Returns the exponential, e^x
  - double pow(double x , double y
    - Returns the exponential of the base, x
  - double cos(double x)
    - Returns the cosine of an angle (in radian
  - double ceil(double x)
    - Returns the ceiling of a number, $\lceil x \rceil$
  - etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main( void )
{
    char str[16];
    printf( "Enter a number: " );
    if( scanf( " %s" , str )!=1 )
        printf( "Failed to read in number\n" );
    else
        printf( "Sqrt( %f ) = %f\n" ,
            atof(str) , sqrt( atof( str ) ) );
    return 0;
}
```

```
>> gcc temp.c -std=c99 -pedantic -Wall -Wextra
/tmp/cclJmVjw.o: In function `main':
temp.c:(.text+0x3a): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
>>
```

To access the math functionality, need to include the math library (add "`-lm`" at compile time).

Functions

- Similarly `math.h` provides a num
  numbers:
  - double sqrt(double x)
    - Returns the square-root, $\sqrt{x}$
  - double exp(double x)
    - Returns the exponential, e^x
  - double pow(double x , double y
    - Returns the exponential of the base, x
  - double cos(double x)
    - Returns the cosine of an angle (in radian
  - double ceil(double x)
    - Returns the ceiling of a number, $\lceil x \rceil$
  - etc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main( void )
{
    char str[16];
    printf( "Enter a number: " );
    if( scanf( " %s" , str )!=1 )
        printf( "Failed to read in number\n" );
    else
        printf( "Sqrt( %f ) = %f\n" ,
            atof(str) , sqrt( atof( str ) ) );
    return 0;
}
```

```
>> gcc temp.c -std=c99 -pedantic -Wall –Wextra -lm
>> ./a.out
Enter a number: 12345
Sqrt( 12345.000000 ) = 111.108056
>>
```

Functions

You can also write your own:

- (For now) define the function before **main**

```
#include <stdio.h>
#include <stdlib.h>
double CelsiusToFarenheit( double c ){ return c * 1.8 + 32.; }
int main( void )
{
	char str[16];
	printf( "Enter a temperature in Celsius: " );
	if( scanf( " %s" , str )!=1 ) printf( "Failed to read temperature\n" );
	else printf( "%f -> %f\n" , atof(str) , CelsiusToFarenheit( atof( str ) ) );
	return 0;
}
```

Functions

Factoring your code into functions, instead of putting everything in `main`, has major advantages:

- Keeps you concentrating on smaller problems
- Makes code more readable
- Helps with testing
  - Can test *functions* one by one
  - Tests are easy to write; call function with certain inputs, assert something about return value
- Easier to collaborate
  - "I'll write functions X and Y, you write everything else assuming you have X and Y."

Functions

Argument values in C are *passed by value*

⇒ The function sees a **copy** of the value passed in as an argument

⇒ Changes made to the argument within the function will not be seen when the function returns.

```c
#include <stdio.h>
void increment( int i ) { i += 1; }
int main( void )
{
    int i = 1;
    printf( "i = %d\n" , i );
    increment( i );
    printf( "i = %d\n" , i );
    return 0;
}
```

```
>> gcc temp.c -std=c99 -pedantic -Wall –Wextra
>> ./a.out
i = 1
i = 1
>>
```

Functions

- A function can return (at most) one value:

```
double exp( double exponent )
```

- What happens if we want the function to return two values?
  - E.g. Divide two integers and return both the quotient and the remainder.

Outline

- Exercise 5
- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

Command line arguments

One way to get input to an executable is to prompt the user and read it in using `scanf`.

But we can also pass arguments directly to the `main` function.

- These will necessarily be strings
- We need to let the `main` function know how many were specified.

Command line arguments

```
int main( void ){…}
```

$$\Downarrow$$

```
int main( int argc , char *argv[] ){ … }
```

<u>Input</u>:

Command line arguments

```
int main( void ){…}
⇓
int main( int argc , char *argv[] ){ … }
```

Input:

- The first argument gives the number of command line arguments provided. *The executable name is always the first command line argument*.

Command line arguments

```
int main( void ){…}
⇓
int main( int argc , char *argv[] ){ … }
```

Input:

- The first argument gives the number of command line arguments provided. *The executable name is always the first command line argument*.
- The second argument is an array of strings, corresponding to the different command line arguments.

Command line arguments

int main(v

⇓

int main(int argc , c

```c
#include <stdio.h>
int main( int argc , char *argv[] )
{
    for( int i=0 ; i<argc ; i++ )
        printf( "%d] %s\n" , i , argv[i] );
    return 0;
}
```

Input:

- The first argument gives the number c
  The executable name is always
- The second argument is an arra
  command line arguments.

```
>> ./a.out all the other slim shadys are just imitating
0] ./a.out
1] all
2] the
3] other
4] slim
5] shadys
6] are
7] just
8] imitating
>>
```

Outline

- File I/O
- Assertions
- Writing functions
- Command line arguments
- Review questions

Review questions

1. Is `fprintf( stdout, "xxx" )` the same as `printf( "xxx" )`?

Yes

Review questions

2. When should we use assertions instead of an `if` statement?

When sanity testing a conditional that should never be true

Review questions

3. What will happen if you pass an `int` variable to a function that takes a `double` as its parameter?
   What will happen if a `double` is passed to an `int` parameter?

The `int` will be converted to a `double` without any loss of information.

The `double` will be rounded/quantized to an `int`, which could cause loss of information.

Review questions

4. What is “pass by value”?

When the invoked function sees a copy of the variable, not the original

Review questions

5. How do you change the `main` function so that it can accept command-line arguments?

```
int main( int argc , char *argv[] )
```

or

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

Exercise 6

- Website -> Course Materials -> Exercise 6