Welcome to...

CS113: Introduction to C

Instructor: Erik Sherwood

E-mail: wes28@cs.cornell.edu

Course Website:

http://www.cs.cornell.edu/courses/cs113/2006sp/

The website is linked to from the courses page of the CS department.

Assignments, lecture slides, etc. will be posted on the course website.

Registration Details

The course meets from January 23 - February 17, MWF, 12:20 - 1:10.

The add/drop deadline is January 30. THIS IS SOON!

Administration

- Lecture slides and readings are posted to the course website, as are assignments, announcements, etc.

- Check the website regularly because that's where I'll post stuff like corrections, hints for homeworks, etc.

- Class runs for four weeks, three meetings a week.

- Office hours: Wednesday, 2:30-4 (tentative), or by appointment

- Prerequisite: CS100 or equivalent experience (e.g., at a different college or in an advanced high school course)

- All auditors welcome

- Course goal: to cover all major features of the C programming language, to the extent that students can subsequently learn about any features not discussed by reading a standard reference such as "The C Programming Language" by Kernighan and Ritchie.

Assignments, etc.

- There will be three programming assignments, due Mondays by 5 p.m. The last assignment will be longer and count for more; you will have more time to complete it.
  - Turn in a printout of your program (no sample data necessary), AND e-mail the source code (the .c files) to wes28@cs.cornell.edu
  - DO NOT just e-mail me your source code. YOU MUST turn in a printout of your source code at Friday's class or in my mailbox in Rhodes 657.
- There is one non-programming assignment, due immediately.
- We'll have one quiz.
- Please come to class! We'll cover new material every time, and there's a lot to get through in four weeks. If you can't make a lecture, let me know and I'll tell you if you missed anything big.
- Please let me know if you anticipate that you'll have problems attending lectures or turning assignments in on time.

More Course Info.

- Grades on Assignments: check plus (exceptional), check (acceptable), check minus, X (insufficient effort)
  - Turning in a program that compiles and is neatly formatted can help you avoid an X. (But this is not necessarily sufficient!)

- Grading for course: S/U only.
  - You are guaranteed an "S" if all assignments completed with a grade of "check".
  - If you get an "X" on two or more assignments, do poorly on the quiz, and have attendance problems, you'll probably get a "U".
  - In-between cases will be handled at my discretion. I'll let you know if there's a problem though, so you shouldn't worry.

- Two textbooks: "Practical C Programming" by Steve Oualline and K&R. Both are recommended; neither is strictly necessary, especially if you have high-speed Web access so you can look stuff up.

- You may use any C compiler. (I recommend gcc, which comes with UNIX/Linux/MacOS X.) Information for using CodeWarrior is on the website.

A Note on Collaboration

- Collaboration: You may discuss ideas on a high level with others, but all code must be your own. You should understand everything that you turn in.

- You can help another student debug his or her code, but you should not write any code for anyone else, and no one else should write code for you.

- Information on the Code of Academic Integrity is available on the website. In a self-selecting, non-competitive course like this, the potential payoffs from cheating are extremely low, even though the risks involved are fairly high.

"A language that doesn't affect the way you think about programming is not worth knowing."

– Alan Perlis

Why Learn C?

Learning C will help you to master programming concepts that higher-level languages like Java don't require you to worry about, such as:

- Pointers: how do you know where your objects are stored in memory?
- Function invocations: what happens when you call a function?
- Dynamic memory allocation: where does the memory for a new object come from?

These concepts all revolve around the issue of *memory*. C forces you to understand how your programs deal with run-time memory – because if you don't, you'll regularly make programming errors that can crash your whole program in nasty ways.

"C is quirky, flawed, and an enormous success."

– Dennis Ritchie

Meet the C programming language

- More about C:
  - Very good for writing *fast* code, especially code that needs to have very explicit control over how memory is used.
  - Good for writing programs that do system-level tasks (e.g., drivers, operating systems, etc.)
  - "Least common denominator": good building block for learning other languages. Subset of C++, similar to Java.
  - Portable - compilers available for most any platform!
- ANSI C standard - aim for ANSI C compliance.
- C is almost always compiled to machine code. (Contrast with Java.)

The canonical "first C program"

```
#include <stdio.h>

void main() {
   printf( "Hello, world!\n" );
}
```

Notice:

- Every program you'll write for this class will "include" the standard functions functions in "stdio.h". Functions like `printf` (which is in stdio.h) aren't part of the core C language, so you need to include them explicitly.

- All C programs must have a `main()` function; this is the first function invoked. The program terminates when this function terminates.

- `printf` is a function that prints formatted output

- `void` indicates that the program itself returns no value. Don't worry about this too much now...

- But: note that some compilers insist on the declaration `int main()`, in which case the statement `return 0;` should be added to end of program, or anywhere else where you want the program to end. (Returning "0" indicates that the program ended successfully.)

Another example

```
#include <stdio.h>

void main() {
   int x = 1, y;
   int sum;
   y = 3;
   sum = x + y; /* evaluates right hand side,
                   places value in variable sum */
   printf( "%d plus %d is %d\n", x, y, sum );
}
```

Some comments on comments and keywords

- Comments
  - Any string of symbols placed between the delimiters /* and */.
  - Can span multiple lines
  - Can't be nested (according to ANSI C standard)! Be careful.
    - Some development environments have an option that allows one to nest comments.
    - A curiosity: one can actually write a program that detects whether or not comments are nested or not!
  - Example: /* /* /* Hi, I'm a comment */
- Keywords
  - Reserved words that cannot be used as variable names
  - Examples: break, if, else, do, for, while, int, void (exhaustive list in K&R, p192)
  - Can be used within comments

Reading integers from standard input

```
#include <stdio.h>

void main() {
   int x, y;
   int product;

   printf( "Enter an integer: " );
   scanf( "%d", &x );
   printf( "Enter another integer: " );
   scanf( "%d", &y );

   product = x * y;

   printf( "%d times %d is %d\n", x, y, product );
}
```

- scanf is like printf, except it reads from standard input instead of writing to it

- scanf is a dangerous, bad function, but you can use it for now, in this limited way, to read integers

- `&x` is a *reference* to the variable `x` rather than the *value* of `x` itself. You need to pass a reference so that scanf can modify the variable `x`. (Much more on this later in the course – don't worry about it now.)

Summing the numbers 1 through 10.

```
#include <stdio.h>

void main() {
   int i = 1, sum = 0;

   while( i <= 10 ) {
      sum = sum + i;  /* shortcut:  sum += i;  */
      i = i + 1;      /* shortcut:  i++;       */
   }

   printf( "The sum is %d\n", sum );
}
```

- Note that the function is split into two parts: the variable declarations (and sometimes initializations), and the rest of the function.

- All C functions require variable declarations at the start of the function.

- This is different from, say, Java.

Summing the numbers 1 through 10.

```
#include <stdio.h>

void main() {
   int i = 1, sum = 0;

   for( i = 1; i <= 10; i++ ) {
      sum = sum + i;
   }

   printf( "The sum is %d\n", sum );
}
```

General form of a for loop:

```
for( initial-stmt; condition; iteration-stmt )
   body-stmt;
```

What happens?

1. Initialization is performed.

2. Condition is checked; if false, loop terminates. Otherwise...

3. Body is performed, followed by the iteration statement.

4. Then, the condition is checked again, and so forth.

Equality testing.

```
#include <stdio.h>

void main() {
   int a, b;

   printf( "Enter a number: " );
   scanf( "%d", &a );
   printf( "Enter another: " );
   scanf( "%d", &b );

   if( a == b ) {
      printf( "They're equal!\n" );
   }
   else {
      printf( "They're not equal.\n" );
   }
}
```

Note:

- Double equals used to compare ints
- "else" portion of "if" optional

Bracing Styles

Four widely used bracing styles:

- 1TBS: One True Bracing Style - used by K & R, and my personal preference

```
for( j = 0; j < 10; j++ ) {
    printf( "%d", j );
}
```

- Allman

```
for( j = 0; j < 10; j++ )
{
    printf( "%d", j );
}
```

- Whitesmith

```
for( j = 0; j < 10; j++ )
    {
    printf( "%d", j );
    }
```

- GNU

```
for( j = 0; j < 10; j++ )
    {
        printf( "%d", j );
    }
```

Most important rule of style: *Be consistent.*

Exponentiation

```
void main() {
    int base, exponent, result;
    printf( "Enter the base:" );
    scanf( "%d", &base );
    printf( "Enter the exponent:" );
    scanf( "%d", &exponent );

    for( result = 1; exponent > 0; exponent-- ) {
        result *= base;
    }
    printf( "%d\n", result );
}
```

Exponentiation, the sequel

```
void main() {
    int base, exponent, result = 1;
    printf( "Enter the base:" );
    scanf( "%d", &base );
    printf( "Enter the exponent:" );
    scanf( "%d", &exponent );

    while( exponent > 0 ) {
        if( exponent % 2 == 1 ) result *= base;
        base *= base;
        exponent /= 2;
    }
    printf( "%d\n", result );
}
```

Note:

- Trickier: maintain invariant that *result times base to the exponent power* is the desired value
- Faster algorithm