ANSI C Prelab

C Programming for the C++ Student

C++ is (originally) a superset of the C language. That is, it has as its basis everything that is in C plus the additional features that make it different. The name, C++, is meant to indicate that it is the next step in the evolution of the C language.

Most of what you've learned to do in this course is directly relevant to coding in plain C. The main areas to consider if you must write in C are:

- header files
- input/output
- reference parameters (including arrays)
- strings
- commenting style

Other topics that differ, which are covered in CSC 250, are:

- pointers
- dynamic memory allocation
- no classes or inheritance
- no functions in structs
- no function overloading

Header Files

C uses header files (#include < … >) that end in ".h". In C++, many of the standard C headers are renamed with a "c" prefix and the ".h" is omitted, as in cstdlib or cmath. Some of the more commonly used headers, and their C++ equivalent, are:

C file	C++ file
stdio.h	iostream
math.h	cmath
stdlib.h	cstdlib
string.h	cstring

Input/Output

These are supported in the stdio.h library. The basic functions are printf( ) for display and scanf( ) for input. Both work by use of a format string that may indicate the type of data to be output or input. Additionally, the printf( ) format string can contain the text you want the data to be embedded within. Some programmers find this easier to set up output that mixes text and variables than the C++ style. You will find the use of the '\n' (newline) character embedded in text strings to be common (there is no equivalent to the cout << endl usage of C++.)

```
printf( "Display some text\n" );        cout << "Display some text" << endl;
```

Format codes that can be embedded in the output string serve as placeholders and format indicators for the variables. The variables or values that replace them at run time follow the format string as function parameters, separated by commas. Some of the common formats are:

%d	integer value, signed
%u	integer value, unsigned
%f	floating point value (float)
%lf	long floating point value (double)
%c	character
%s	string (text with more than one character)

Numeric output can be formatted in ways similar to the use of cout.precision and the setw( ) by embedding **`width.precision`** values between the % and the letter in the format code.

```
int x = 10;
double p = 19.54678;
```

C version

```
printf( "An integer: %d   --  A double: %6.3lf \n\n", x, p );
```

C++ version

```
cout.setf(ios::fixed | ios::showpoint);
cout.precision(3);
cout << "An integer: " <<  x << "  -- A double: " <<  setw( 6 )
<< p << endl << endl;
```

Input with the scanf( ) function is similar to the output form, without the need to put any text in the format string besides the formatting codes. The variable(s) to receive the input must be prefaced with the "address of" operator, the ampersand (&). Multiple inputs can be done in a single call to the function.

```
int x;
double p;
```

C version

```
scanf( "%d", &x );
scanf( "%d%lf", &x, &p );
```

C++ version

```
cin >> x;
cin >> x >> p;
```

Note: to eliminate the warnings about scanf being deprecated (planned to be deleted), insert the following before the #include lines:

```
#define _CRT_SECURE_NO_DEPRECATE
```

Functions

Functions in C are generally written the same as in C++, with the exception of parameters being passed by reference. To pass by reference in C, you use pointers, which are not covered in CSC 150, or in this document.

C also uses function prototypes to provide the compiler information about the functions' interfaces before they are actually defined in the code.

Arrays

Arrays are created and accessed the same in C. You declare an array giving it a size enclosed in square brackets ([ ]), and use the brackets to access an element at a given index (0 through size-1). The method by which you define functions that take arrays as parameters will be a bit different. The formal parameter declared for an array will be written with an asterisk preceding the name of the array, or the name of the array followed by [] (same as you do in C++). When you pass an array to the function, you simply pass the array name, as you do in C++. When accessing elements of the array in the function, simply use the index in brackets as you do in C++. So, the only visible difference is in the function definition. The function prototype and function header are the same.

C version

```
/*This function finds the integer average of an array of int's.
Note that the array parameter is indicated with the asterisk
(thus it's a reference parameter), but inside the function
elements are accessed with the [index] as in C++. */

int arr_avg( int *ar, int size )   /* note the asterisk */
{
    int sum = 0;
    int i;

    for( i = 0; i < size; i++ )
        sum += ar[i];

    return sum / size;  /*return only integer value average */
}
```

When calling this function, the argument for the array parameter is simply the array name (no brackets, no use of preceding ampersand.) Sample usage:

```
int data[6] = { 2, 3, 5, 7, 1, 9 );
int average;
average = arr_avg ( data, 6 );
```

C++ version

```
int arr_avg( int ar[], int size ) //only array param is different
{
    int sum = 0;
    int i;

    for( i = 0; i < size; i++ )
        sum += ar[i];

    return sum / size;  /*return only integer value average */
}
```

Sample usage:

```
int data[6] = { 2, 3, 5, 7, 1, 9 );
int average;
average = arr_avg ( data, 6 );
```

Comments

As you note in the example above, C uses only the `/* ... */` style. The `//` usage is a C++ addition. You may find it supported in some compilers (Visual C++ did not complain), but don't count on it.

Read through the two sample files: c.c and c_funcs.c before coming to the ANSI C Lab.

Also note that every chapter in your text has a section at the end describing the C way of doing the actions in that chapter.