Introduction to Cryptography (5:00-9:00)

- Here is an encrypted message: Or fher gb qevax Ibhe binygvar!
- It has been encoded using a cipher. A cipher is an algorithm that takes some input text (plaing text) and produces output text (cipher text). Its purpose is to hide the message.
- This message has been encoded by rotating every letter 13 places (A→N, B→O, etc.)
- What does it say? Be sure to drink your ovaltine.
- This is just like using one of those decoder rings that children use to send secret messages to their friends. (In other words, it's not a very good way to conceal data.)

From C to Other Languages (9:00-13:00)

- Many of the things we learn in C are concepts that can be carried over to other languages.
- For instance, if we wanted to write good old "hello, world" in C++, it would look like this:

```
#include <iostream>

using  namespace std;

int
main(int argc, char * argv[])
{
    cout << "hello, world!\n" << endl;
}
```

- Check your sourcecode handout for the same program written in Java, Perl, and other languages
- After a course like this, you'll be able to do a lot of programming, but you'll have to teach yourself some languages.

Bugs (13:00-17:00)

- According to legend, Grace Hopper discovered the first "bug" in a program when she pulled a dead moth out of a computer.
- Today, we use the term "bug" to refer to any sort of problem that causes our programs to run in a way other than intended
- Take a look at buggy1.c. This program is supposed to print 10 asterisks. What is the problem?

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

    for (i = 0; i <= 10; i++)
        printf("*");
}
```

- It prints 11 asterisks because i runs from 0 to 10, which is 11 integers. Conventionally, we would change this to `for(i=0; i<10; i++)`
- Take a look at buggy2.c. This program is supposed to print 10 asterisks all on the same line.

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

    for (i = 0; i <= 10; i++)
        printf("*");
        printf("\n");
}
```

- Instead, we get all the asterisks on the same line. Why?
- We are missing curly braces. If you want to associate multiple statements with a for loop, you have to put them inside curly braces. Only if there is a single line can you omit them.

Integer and Character Casting (17:00-27:30)

- We know that chars are associated with ints and vice versa. By means of ASCII, the computer has an integer corresponding to every character on the keyboard.
- What if we want to know which integer is associated with a particular integer?
- We can use a cast.
- Last week, we used a cast to make an int into a float to force floating point division.
- Again, we will use casting to change a type, but this time between the types int and char.
- So we can say

```
int i = (int) 'A';
char c = (char) 65;
```

- Let's take a look at ascii1.c

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

    /* display mapping for uppercase letters */
    for (i = 65; i < 65 + 26; i++)
        printf("%c: %d\n", (char) i, i);

    /* separate uppercase from lowercase */
    printf("\n");

    /* display mapping for lowercase letters */
    for (i = 97; i < 97 + 26; i++)
        printf("%c: %d\n", (char) i, i);
}
```

- In this program, we iterate from 65 up to 91 and for each number , print out the associated character, (char) i, followed by the integer itself, i. This prints out a list of the capital letters with their associated numbers (A:65, B: 66, …, Z:90). Then we do the same from 97 to 122, which is the lowercase letters.
- Now let's take a look at ascii2.c.

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

```
{
    int i;

    /* display mapping for uppercase letters */
    for (i = 65; i < 65 + 26; i++)
        printf("%c  %d    %3d  %c\n", (char) i, i, i + 32,
               (char) (i + 32));
}
```

- This time, we make use of the fact that the integer corresponding to any given lowercase letter is exactly 32 more than the integer corresponding to the same letter in uppercase. Therefore, we needn't cycle through 97-122 separately. When we want the uppercase letter, we just add 32 to the letter representing the uppercase letter.
- We also make use of the width aspect of format strings. We know that some numbers will be 2 digits and some 3. To make things line up perfectly, we tell each number to take up three places.
- In ascii3.c, we iterate over the letters themselves.

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

    /* display mapping for uppercase letters */
    for (c = 'A'; c <= 'Z'; c = (char) ((int) c + 1))
        printf("%c: %d\n", c, (int) c);
}
```

- We initialize char c to A. In the update, we cast c to an int, add 1, and then cast it back to a char. We repeat until c is Z.
- Actually, we don't always need to be this explicit with casting because the compiler can sometimes figure it out without us telling it to (implicitly). For instance, if we simply added 1 to char c, it would know that we meant to add 1 to the int value and put the result back in c.
- Suppose we watned to implement the game Battleship. The following code, battleship.c, prints out the gameboard:

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

    /* print top row of numbers */
    printf("\n   ");
    for (i = 1; i <= 10; i++)
        printf("%d  ", i);
    printf("\n");

    /* print rows of holes, with letters in leftmost column */
    for (i = 0; i < 10; i++)
    {
        printf("%c  ", 'A' + i);
        for (j = 1; j <= 10; j++)
```

```
            printf("o  ");
        printf("\n");
    }
    printf("\n");
}
```

- The first for loop prints the numbers from 1 to 10 across the top of the screen
- The second for loop prints 10 separate lines (we know because it ends with printf("\n"). On each line, we print a char that is i letters past A (where i goes from 0 to 9) followed by 10 o's.

Functions (27:30-40:00)

- Think about the song "99 Botles of Beer on the Wall."
- If we wanted to print out the lyrics to this song, we would make use of a loop.
- Examine the following code, beer1.c:

```
int
main(int argc, char * argv[])
{
    int i, n;

    /* ask user for number */
    printf("How many bottles will there be? ");
    n = GetInt();

    /* exit upon invalid input */
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    /* sing the annoying song */
    printf("\n");
    for (i = n; i > 0; i--)
    {
        printf("%d bottle(s) of beer on the wall,\n", i);
        printf("%d bottle(s) of beer,\n", i);
        printf("Take one down, pass it around,\n");
        printf("%d bottle(s) of beer on the wall.\n\n", i - 1);
    }

    /* exit when song is over */
    printf("Wow, that's annoying.\n");
    return 0;
}
```

- First, we get input from the user and, if given a negative number, print an error message and return 1.
- Recall that main has a return value of int. If all goes well, it returns 0. If it exits with a problem it returns 1. 1 is an error code that signals to someone running the program that the program exited abnormally.

- Then we loop from n *down* to 0, decrementing i by one every time.
- We can improve this program using hierarchical decomposition. This just means breaking down a large problem into smaller problems.
- For example, one major piece of this program is the part that prints out the chorus for a given value of n. We can factor this out and put it into its own function. When we want to print the chorus, we simply call the function.
- Look at beer4.c:

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

    /* ask user for number */
    printf("How many bottles will there be? ");
    n = GetInt();

    /* exit upon invalid input */
    if (n < 1)
    {
        printf("Sorry, that makes no sense.\n");
        return 1;
    }

    /* sing the annoying song */
    printf("\n");
    while (n)
        chorus(n--);

    /* exit when song is over */
    printf("Wow, that's annoying.\n");
    return 0;
}
```

- Now, we just call the function chorus() repeatedly as long as n is nonzero.
- In the line while (n) we make use of the fact that, when considered as a Boolean, 0 is false, and everything else is true. This statement will be false only when 0 is reached.
- Notice also that we decrement n within the call to chorus itself. This tells the computer to perform chorus with the current value of n, *then* reduce it by one.
- This is the same as:

```
while(n)
{
   chorus(n);
   n = n-1;
}
```

- Now look at the function void chorus (int b)

```
void
chorus(int b)
```

```
{
    string s1, s2;

    /* use proper grammar */
    s1 = (b == 1) ? "bottle" : "bottles";
    s2 = (b == 2) ? "bottle" : "bottles";

    /* sing verses */
    printf("%d %s of beer on the wall,\n", b, s1);
    printf("%d %s of beer,\n", b, s1);
    printf("Take one down, pass it around,\n");
    printf("%d %s of beer on the wall.\n\n", b - 1, s2);
}
```

- This is a *void* function because it does not return anything. That is, it produces no output that can be assigned to a variable. It only has "side effects" (printing text to the screen)
- Within the function, we now refer to the number of bottles as b. We don't call it n because we might call this function in many different contexts with many different variables.
- The function simply makes a local copy of whatever number it is passed and calls it b. Throughout the function, it refers only to b when it wants to deal with the number of bottles.
- In this function we have also fixed the bottle/bottles grammar issue in this line:

```
s1 = (b == 1) ? "bottle" : "bottles";
```

- This line is the combination of an if statement and a variable assignment. It takes the if statement "If b=1, bottle. Else, bottles" and assigns the result to s1.
- Notice that we need two strings, s1 and s2, because the first three lines of the chorus refer to b bottles while the last one refers to b-1 bottles.