# Let's Talk About C

# Topics

- Key differences with C++
- Command-line compiling
- Basic I/O: Output
- Basic I/O: Input
- Dynamic Memory Allocation
- File I/O:  Writing
- File I/O:  Reading

# Key Differences With C++

# Key Differences With C++

- No objects, C is not an OOP language ( no cin, cout )
- No name spaces:

```
using namespace std;
```

doesn't compile

# Key Differences With C++

- No classes – closest thing is structs
- Dynamic memory allocation not built in: no new, delete key words. Must use functions (*malloc*, *free*).
- Everything is done in functions, pretty much.

# Key Differences With C++

- All functions are stored in header files, with .h extension.
- For example, to use the stdio library,

```
#include “stdio.h”
```

- No bool data type:  booleans are represented as integers.  0 = false, !0 = true

# Key Differences With C++

- File I/O is different – no objects, no ifstream, ofstream, fstream.
- All strings are C-strings, no string objects. Reminder: C-Strings are arrays of characters terminated by '\0'.
- Minor syntactical differences. ( no defining variables in for loop headers, for example )

# Command-line Compiling

# Command-Line Compiling

- To compile your source code, use the gcc command:

```
gcc source.c -o name
```

<u>gcc</u> : compiler command
<u>source.c</u> : the source code file to compile
<u>-o</u> : switch to specify executable name
<u>name</u>: the name of the executable

# Command-Line Compiling

- To run the executable after it's been compiled, go to the directory it's in and type:

```
./name
```

where name is the name of the executable

# Basic I/O

# Displaying Output

- There are no objects in C, thus no cin and cout.
- Most everything is done through functions.
- One function we can use to display output is *printf()*.
- *printf()* is defined within the stdio.h header file.

# Displaying Output

- Consider this example printf function call:

```
printf("Hello, world!\n");
```

- In this example, we've passed a single argument to the printf function, the string literal

  "Hello, World!\n".

- This statement causes Hello, World! to appear on the screen.

# Displaying Output

- C supports escape sequences, such as \n, \t, etc.
- What if we wanted to display the contents of variables? Consider:

```
int i = 10;
char name[] = “Timmy”;
printf(“I'm %s. I'm %d years old\n”, name, i );
```

# Displaying Output

- In this example, we have passed three arguments to the *printf()* function:

  “I'm %s. I'm %d years old\n” - a string literal
  name – a c-string
  i – an integer

- In the string literal, there are special characters → %s and %d.
- These are known as <u>format specifiers</u>.

# Displaying Output

- Format specifiers serve as “place holders” for arguments that follow the string literal where they are found.
- These place holders are replaced with the values inside the following arguments within the string.
- So, the output of the *printf()* statement would be:

I'm Timmy. I'm 10 years old.

# Displaying Output

- There are many types of format specifiers:
  %s – string of characters
  %d, %i – integers
  %f – floats
  %c – characters
- There are many more, these are just a few.
- A C++ analogy of the previous example:

```
cout << “I'm “ <<  name << “. I'm “ << i << “ years old.\n”;
```

# Reading Input

- To read input from the keyboard ( stdin ), we need another function: *scanf()*.
- *scanf()* works in a similar way to *printf()*, except the format specifiers specify what types of data is being read and the arguments after the string argument are the memory locations where they are stored.
- The arguments must be memory addresses.

# Reading Input

- Consider the following example:

```
int a, b;
char c;
printf(“Enter an expression: “);
scanf(“%d %c %d”, &a, &c, &b );
```

# Reading Input

- In this example, %d, %c, %d are the format specifiers. They specify the position within the input where data will be read from.
- &a, &c, &b are the memory locations where the input will be stored.
- Relative positions matter!
- A user who enters: 10 + 20 will have 10 stored in the a variable, '+' stored in the c, and 20 stored in b.

# Reading Input

This is very similar to something like:

```
cin >> a >> c >> b;
```

in C++.

# fgets

- Another option is the fgets function.
- fgets allows us to read in an entire line of input sort of like how getline() does in C++.
- Like getline(), fgets can be used to read from the keyboard or a file.

# fgets

- With fgets, you need to specify three things:
  - memory location you want to write to
  - the number of bytes to read, and
  - the *file descriptor* you are reading from ( more on file descriptors in a bit ).

# fgets

- The null terminator is automatically appended to the string of characters fgets reads.
- returns the array it read on success or it will return null on failure.
- *stdin* specifies input should be read from the keyboard.

# fgets – a sample program

```c
/* fgets example */

#include <stdio.h>

int main()
{
  char mystring[100];

  printf("Type something: " );

  if( fgets( mystring, 100, stdin ) != NULL )
    printf( "You typed: %s\n", mystring );

}

```

# Processing Command-line Arguments

# Processing Command-line Arguments

- Command-line arguments are arguments passed to a command when it is invoked.
  - For example :

    ./foo hello world

- Here, foo is a program being run from the command line.

# Processing Command-line Arguments

- The foo program is being passed three command-line arguments : foo, hello and world.
- Unix considers there to be 3 total arguments in this command.
- The command itself is considered an argument, in this case, foo.

# Processing Command-line Arguments

- The number of arguments and the arguments themselves are stored in special parameters within int main:

```
int main( int ac, char* av[] )
```

- Unix stores the number of arguments in ac, and the arguments themselves within av.

# Processing Command-line Arguments

- We can use these parameters, within our programs:

<u>argument count</u>

<u>arguments</u>

```c

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

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

    return 0;
}
```

# Processing Command-line Arguments

```
hank@netbook: ~/Desktop/CSUEB/CS3560/Intro To C
hank@netbook:~/Desktop/CSUEB/CS3560/Intro To C$ ./foo Hello World
./foo
Hello
World
hank@netbook:~/Desktop/CSUEB/CS3560/Intro To C$
```

# Processing Command-line Arguments

- Each element in argv is a pointer to each argument.
- The last element in argv is null terminated.
- Arguments are stored as C-Strings.  There are NO objects in C, and thus no string objects.

# Processing Command-line Arguments

- Another example ( displays the arguments backwards ) :

```c
#include "stdio.h"

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


    while( argc-- )
        printf("%s\n", argv[argc] );

    return 0;
}
```

# Processing Command-line Arguments

- Review the following from CS2360 Gaddis textbook :
  - CH09 – Pointers
  - CH10 – Characters, C-Strings, and More About the string Class

# Dynamic Memory Allocation

# Dynamic Memory Allocation

- In C++, we have the built in operators *new* and *delete* to allocate and delete dynamic memory.
- Don't have them in C, we have to use functions.
- To dynamically allocate memory in C, use the *malloc()* function.
- To free the memory, use the *free()* function.

# Dynamic Memory Allocation

- *malloc()* and *free()* are found in the stdlib.h header file.
- *malloc()* accepts an integer as an argument, the number of bytes to dynamically allocate.
- returns the memory address of the allocated memory as a void*.
- void* is a "generic" pointer, so we need to cast to the data data type of the memory we want to allocate

# Dynamic Memory Allocation

- Just like in C++, it's the programmer's responsibility to manage memory.
- Use the *free()* function to free dynamic memory.
- Accepts the memory address of the dynamically allocated memory as it's argument.
- Return type is void.

# Dynamic Memory Allocation

```c
#include "stdlib.h"     // for malloc, free
#include "stdio.h"      // for printf

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

    int* array = NULL;  // pointer to our new array

    // dynamically allocate the array of 5 elements
    // 5 elements * 4 bytes for an int = 20 bytes to allocate
    array = (int*)malloc( 20 );

    array[0] = 5;
    array[1] = 10;
    array[2] = array[0] + array[1];

    printf("%d\n", array[2] );

    free(array);    // free the memory

    return 0;
}
```

# Additional References

- You'll probably visit these often:
- cplusplus.com: **http://www.cplusplus.com/reference/**
- tutorialspoint.com: **http://www.tutorialspoint.com/c_standard_library/**
- cprogramming.com: **http://www.cprogramming.com/**