Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.
Announcements

• Homework 1 is now due Wednesday
Outline

• Functions
• Pointers and memory
• Command line arguments
• Variants of printf
• scanf and fgets
Functions

• A function takes multiple arguments and returns (at most) one value
  
  ```c
  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)
Functions

• A function takes multiple arguments and returns (at most) one value
  \[
  \text{int } \text{foo(} \text{char } c, \text{ int } i \text{)}
  \]
  \[
  \{
  \text{ return } i; \}
  \]

• The function name
• The return type (could be \text{void} if nothing is returned)
• 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)
• 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)
• The list of argument types
• The function body
  • Needs to be in braces, even if the function is just one command
Pointers and memory

• A function can return (at most) one value:
  char c = getchar();
• What happens if we want the function to return two values?

Memory:
• Every variable we use resides somewhere in memory
• We can pass the memory location of a variable to the function, and then the function can set the value in that location
• The variable containing the memory address is called a pointer
Pointers and memory

• A pointer to a variable of type “\(T\)” has type “\(T\) *
• char * a pointer to a variable of type char
• int * a pointer to a variable of type int
• float * a pointer to a variable of type float
• etc.

```c
#include <stdio.h>

void foo( int* i1 , int* i2) {
}

int main( void )
{
    int i1 = 1 , i2 = 2;
    int* i1_ptr = NULL;
    int* i2_ptr = NULL;
    foo( i1_ptr , i2_ptr );

    return 0;
}
```
Pointers and memory

- A pointer to a variable of type “T” has type “T *”
- The address of a variable is obtained using the ampersand (&) operator

```c
#include <stdio.h>

void foo( int * i1 , int * i2) {

}

int main( void ) {
    int i1 = 1 , i2 = 2;
    int * i1_ptr = &i1;
    int * i2_ptr = &i2;
    foo( i1_ptr , i2_ptr );

    return 0;
}
```
Pointers and memory

- A pointer to a variable of type “\texttt{T}” has type “\texttt{T *}”
- The address of a variable is obtained using the ampersand (\&) operator
- A pointer is dereferenced (its data is accessed) using the star (*) operator

```c
#include <stdio.h>

void foo( int * i1 , int * i2) {
    (*i1) += 1;
    (*i2) -= 2;
}

int main( void ) {
    int i1 = 1 , i2 = 2;
    int * i1_ptr = &i1;
    int * i2_ptr = &i2;
    foo( i1_ptr , i2_ptr );
    printf( "\%d \%d\n" , i1 , i2 );
    return 0;
}
```

```
./a.out
2 0
>>
```
Pointers and memory

• A pointer to a variable of type $T$ has type $T$ *
• The address of a variable is obtained using the ampersand (&) operator
• A pointer is dereferenced (its data is accessed) using the star (*) operator

Note:
• A variable that is an array is essentially a pointer. (e.g. char* is the type of a string)

```c
#include <stdio.h>
int main(void)
{
    char str1[] = "hello world";
    char * str2 = str1;
    printf("%s\n", str1);
    printf("%s\n", str2);
    return 0;
}
```

>> ./a.out
hello world
hello world
>>
Pointers and memory

- A pointer to a variable of type \( T \) has type \( T \ast \)
- The address of a variable is obtained using the ampersand (\&) operator
- A pointer is dereferenced (its data is accessed) using the star (*) operator

**Note:**
- A variable that is an array is essentially a pointer. (e.g. \texttt{char *} is the type of a string)
- "Essentially" because some information is lost when an array is represented as a pointer

```
#include <stdio.h>
int main(void)
{
    char str1[] = "hello world";
    char * str2 = str1;
    printf("%d\n", sizeof( str1 ));
    printf("%d\n", sizeof( str2 ));
    return 0;
}
```

```
>> ./a.out
12
8
>>
```
Command line arguments

• When you type:
  
  mkdir cs120

  you’re running the program mkdir and passing an argument cs120

• Your compiled C programs can take arguments similarly

```c
#include <stdio.h>
int main( int argc , const char * argv[] )
{

    return 0;

}
```
Command line arguments

• When you type:
  
  **mkdir cs120**

  you’re running the program `mkdir` and passing an argument `cs120`

• Your compiled C programs can take arguments similarly

```c
#include <stdio.h>
int main( int argc , const char * argv[] )
{
    printf( "Command: %s\n" , argv[0] );
    printf( "Arguments: %d\n" , argc-1 );
    for( int i=1 ; i<argc ; i++ ) printf( "Arg[%d]: %s\n" , i , argv[i] );
    return 0;
}
```

```
>> ./a.out hello
Command: ./a.out
Arg[1]: hello
>>
```
Command line arguments

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

This `main` breaks the command line into sub-strings separated by white space, including the executable name, and provides them to the function.

- `int argc`: The number of sub-strings
- `const char * argv[]`: the sub-strings themselves*
  - `argv[]`: `argv` is an array...
  - `char * argv[]`: `argv` is an array of `char` pointers...
  - `const char * argv[]`: `argv` is an array of `char` pointers that are constant
  - Since a `char *` is a string, this is an array of (unmodifiable) strings

*For more details, see the Clockwise/Spiral rule (http://c-faq.com/decl/spiral.anderson.html)
Writing output: `printf`

Last time:
- We saw how to use the `printf` function to write to the command prompt:
  ```c
  int printf( const char * format_str, ... )
  ```
Writing output: `printf`

- More generally, we can use the same type of syntax to write to a file:
  
  ```c
  int fprintf( FILE * file, const char * format_str, ... )
  ```

- The command prompt is a special “file” called `stdout`:
  
  ```c
  printf( format_str, ... ) = fprintf( stdout, format_str, ... )
  ```

- We can also write to a string using the `sprintf` function:
  
  ```c
  int sprintf( char * str, const char * format_str, ... )
  ```
Reading input

Last time:

• We saw how to use the `getchar` function to read user input
  • This was cumbersome because we read one character at a time
Reading input: `scanf`

```c
int scanf( const char * format_str , ... );
```

`scanf` can be used to read in strings from the command line

- It is the opposite of `printf`:
  - Instead of writing a formatted string to the command line, it reads a formatted string from the command line
  - The variables after the format string need to be pointers

```c
#include <stdio.h>
int main( void )
{
    int i;
    printf( "Please enter an integer: " );
    scanf("%d", &i);
    printf( "You entered: %d\n" , i );
    return 0;
}
```
Reading input: `scanf`

```c
#include <stdio.h>
int main( void )
{
    int i;
    printf( "Please enter an integer: " );
    scanf( "%d" , &i );
    printf( "You entered: %d\n" , i );
    return 0;
}
```

**scanf** can be used to read in strings from the command line

- It reads the characters from the command prompt and tries to match them to the characters in the first string (whitespace is ignored).
- If it encounters a special character it tries to convert the next word on the command line into the appropriate type and sets the associated pointer
  - `%d`: the next word should be an `int`
  - `%f`: the next word should be a `float`
  - `%s`: the next word should be a string
  - etc.
Reading input: `scanf`

```c
#include <stdio.h>
int main( void )
{
    int i;
    printf( "Please enter an integer: " );
    if( scanf( "%d" , &i )!=1 ) printf( "Failed to read input\n" );
    else
        printf( "You entered: %d\n" , i );
    return 0;
}
```
Reading input: `scanf`

```c
#include <stdio.h>
int main( void )
{
    int age;
    char name[128];
    printf( "Please enter your name and age: " );
    scanf( "%s %d" , name , &age );
    printf( "Your age is %d\n" , age );
    return 0;
}
```
Reading input: \texttt{fgets + scanf}

\begin{verbatim}
char * fgets( char * str, int num, FILE * file );
\end{verbatim}

1. Read from the command prompt using \texttt{fgets} (with \texttt{file} set to \texttt{stdin})
   • Reads up to [ENTER] but terminates sooner if \texttt{num-1} characters have been read
   • It adds a null-terminator at the end of the string
   • If it gets to a new-line, that will be part of the string
   • It returns \texttt{str} or \texttt{NULL} if the reading failed
   • To read from the command line, we set \texttt{file} to \texttt{stdin}

2. Use \texttt{scanf} to separate out the parts of the string
   • Like \texttt{printf}, this a variant of \texttt{scanf} that reads from an input string
Reading input: fgets + sscanf

```c
#include <stdio.h>
#include <string.h>
int main( void )
{
    char line[128] , name[128];
    int age;
    printf( "Please enter your age and name: " );
    fgets( line, sizeof(line), stdin );
    printf( "You entered: %s\n" , line );
    if( strlen( line)==sizeof(line)-1 )
        printf( "Input is too big\n" );
    else if( sscanf( line, "%d %s" , &age , name )!=2 )
        printf( "Failed to parse input\n" );
    else
        printf( "%d / %s\n" , age , name );
    return 0;
}
```

Q: What happened to the new line?
In-Class Exercises

• On Piazza, find Resources section, then click Resources tab
• Scroll down to section for this course section
• Find link for Exercise 2-2 and follow it
• Follow the instructions; raise your hand if you get stuck
• Make sure you check in with a course staff member sometime during this session