CSC 4356 / ME 4573 Interactive Computer Graphics

C Language Refresher

C

All of you are programmers.

Few of you are C programmers.

These notes *briefly* list those aspects of C that may differ from what you already know.

See Wikibooks or K&R for more.

Variables and Types

C is a *statically*-typed, *imperative* language. Variables and their types must be declared before use.

```
double pi = 3.14159265358979;
float  x  = 0.5;
char   c  = 'a';
int    a  = 42;
int    b;
```

The values of variables change, but not the types.

```
a = a * 10;
b = a + 1;
```

Structure and Array Types

Structures are groups of values behaving as a single variable.

```
struct city
{
    int   population;
    float latitude;
    float longitude;
};
```

Arrays are contiguous blocks of values, all of the some type.

```
int n[10];
```

```
struct city capitals[50];
```

Pointer

A C *pointer* is an address in RAM.

```
int i  = 10;     // i is a normal integer variable.
int *p = &d;     // p refers to the location of i.
```

The & operator gives “the address of” a variable

```
i  = 42;
*p = 42;
```

The * operator gives “the variable at” an address.

Pointers

A C *string* is a pointer to a zero-terminated array of chars.

```
char *filename = "teapot.obj"
```

Pointers and arrays are interchangeable.

```
int a[10];
int *p = a;

a[5] = 12;
p[5] = 12;
```

Functions

Functions are declared with arguments and return type.

```
int average(int a, int b)
{
    return (a + b) / 2;
}
```

Calling this function…

```
x = average(2, 8);
```

Conditionals

```
if (value == 0)
{
    answer = "false";
}
else
{
    answer = "true";
}
```

Remember that *assignment* uses =, while *equality* testing uses ==.

Looping

For-loop syntax includes initialization; test; and step.

```
int n = 0;
int i;

for (i = 0; i < 10; i++)
{
   n = n + i;
}
```

The Main Function

All C programs begin execution with the `main` function. It receives command line arguments as an array of strings.

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

    return 0;
}
```

It returns zero to indicate success.

Command Line Arguments

For example, this command line…

```
./myprogram teapot.obj 32
```

… invokes `main` with these arguments:

`argc` = 3 `argv` =

0	`"./myprogram"`
1	`"teapot.obj"`
2	`"32"`

Function Declaration

A C function must be *declared* before it may be *used*.

```c
int average(int a, int b);

int compute()
{
    return average(2, 8);
}

int average(int a, int b)
{
    return (a + b) / 2;
}
```

However, function *definition* counts as *declaration*.

```c
int average(int a, int b)
{
    return (a + b) / 2;
}

int compute()
{
    return average(2, 8);
}
```

For this reason, C programs are often structured in a bottom-up fashion, with `main` at the very end.

Declaration becomes necessary when code is *defined* in one module…

```
int average(int a, int b)
{
    return (a + b) / 2;
}
```

… and *used* in another module.

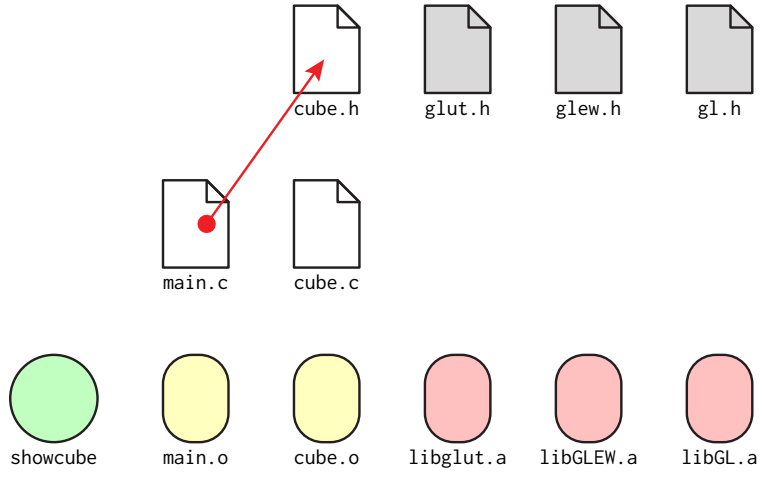
```
int average(int a, int b);

int compute()
{
    return average(2, 8);
}
```

This declaration is usually handled by a header.

```
#include "average.h"

int compute()
{
    return average(2, 8);
}
```



Recall the relationship discussed in the course notes on Project Management.

Memory Management

The C library provides *manual* memory management.

```
#include <stdlib.h>
```

Allocate with `malloc`.

```
int *p;
p = (int *) malloc(10 * sizeof (int));
```

And release with `free`.

```
free(p);
```

These two functions both allocate 10 integers.

```c
void func()
{
    int a[10];

    // ...

}
```

- On the stack.
- Auto-free.
- Static size.

```c
void func()
{
    int *p;
    p = (int *) malloc(10 * sizeof (int));
    // ...
    free(p);
}
```

- On the heap.
- Manually free.
- Dynamic size.

Basic Output

The `printf` function performs basic formatted output.

```
char *s = "World";
int   x = 42;

printf("Hello, %s!\n", s);
printf("The answer is %d.\n", x);
```

This produces:

```
Hello, World!
The answer is 42.
```

File I/O

Access to files on disk is provided via *file pointers*.

```
#include <stdio.h>
FILE *fp;
```

These may be opened for either reading or writing.

```
fp = fopen("myfile.txt", "r"); // or
fp = fopen("myfile.txt", "w");
```

Be sure to close the file when finished.

```
fclose(fp);
```

Text File I/O

`fprintf` performs basic file output just list `printf`.

```
int x = 42;
fprintf(fp, "The answer is %d.\n", x);
```

`fscanf` does just the opposite, basic file input.

```
int x;
fscanf(fp, "%d", &x);
```

Note, `fscanf` takes a *pointer* to the variable taking the value.

Binary File I/O

`fwrite` and `fread` perform output and input of *binary* data.

```
FILE *fp = fopen("output.dat", "wb");
```

For example, to write five 32-bit integers in $5 \times 4 = 20$ bytes.

```
int n[5] = { 1, 2, 3, 4, 5 };
fwrite(n, sizeof (int), 5, fp);
```

To read the same…

```
int n[5];
fread(n, sizeof (int), 5, fp);
```

Mathematics

The C math library provides floating point functions, including trigonometry.

```
#include <math.h>

double radius = sqrt(pow(x, 2.0) + pow(y, 2.0));
double angle  = atan2(y, x);

x = cos(angle) * radius;
y = sin(angle) * radius;
```