CS2141 – Software Development using C/C++

C++ vs. C

Differences from C++

- No classes, no operator overloading, no templates, and no references
- Local variables must all be declared at the start of a function before any other code
- Stdio rather than iostream
- Malloc and free instead of new and delete
- Older compilers only support `/*` ... `*/` style comments (`//` is okay with newer ones)

stdio

- **`stdio`** is the widely known and available C input/output library
- Not object-oriented, nor as extendable or adaptable as **`iostream`**
  - Uses a fixed set of formatting directives
  - Cannot be extended to work with user-defined types
- There is no type checking, so using the wrong formatting directive can cause problems
- To use stdio, include **`<stdio.h>`**

Print to terminal

- The function **`printf`** prints text to **`stdout`**:

```
printf( “Cows go moo.\n” );
printf( “Pigs go oink.\n” );
```

- Conversion characters are used to do formatting when printing values:

```
int a = 3;
double d = 2.8;
printf( “i is %d\n”, i );  // %d for ints
printf( “d is %lf\n”, d ); // %lf for doubles
char * s = “Quack quack!”;
printf( “%s\n”, s );       // %s for strings
```

Print to terminal cont.

- Common conversion characters:

```
%d   integer decimal value
%o   integer printed as octal
%x   integer printed as hex
%c   integer printed as a character
%u   unsigned integer decimal
%f   floating point value
%g   floating point value exponential notation
%e   same as %g but shorter
%s   null terminated string
%%   percent sign
```

- Try '**`man 3 printf`**' for more information

Read from terminal

- The **`scanf`** function formats values as they are read in from **`stdin`**
  - Uses same conversion characters as **`printf`**
  - Arguments must be pointers rather than values

```
int i;
float j;
scanf( "%d %f", &i, &f );
```

- **`fgets`** can be used to read an entire line of text:

```
char buffer[200];
fgets( buffer, 200, stdin );
```

File I/O

- Files are opened using the **`fopen`** function
  - Takes a filename and a mode. Some modes are:
    `"r"` Open the file for reading
    `"w"` Open the file for writing
  - Returns a **`FILE *`** pointer, or **`NULL`** if unsuccessful
- Use **`fclose`** to close a file

```
FILE * f = fopen( "secretplans.dat", "r" );
if( f == NULL )
   printf( "Could not open file\n" );
else
   fclose( f );
```

File I/O cont.

- **`fprintf`** and **`fscanf`** work like **`printf`** and **`scanf`**, but have a file pointer parameter:

```
int i;
int codes[5];

FILE * f = fopen( “codes.ts”, “r” );
for( i = 0; i < 5; ++i )
fscanf( f, “%d”, &codes[i] );
fclose( f );

f = fopen( “spyreport.txt”, “a” );  // “a” = append
for( i = 0; i < 5; ++i )
fprintf( f, “%d\n”, codes[i] );
fclose( f );
```

Structures

- Structures are like class definitions, but with only public data fields:

```
struct Fish {
   float pos[2];  // x-y position of the fish
   int age;       // age of the fish
   char * name;   // name of the fish
};

Fish bubbles;
bubbles.pos[0] = 1.2;
bubbles.pos[1] = 3.6;
bubbles.age = 5;
bubbles.name = “Bubbles”;
```

Unions

- A **`union`** is similar to a **`struct`**, but defines fields that share the same memory location

```
union Node {            // Can hold...
   int i;               // an int OR
   double d;            // a double OR
   struct Fish * f;     // a pointer to a Fish
};
```

- Only one field can be used at a time
- In C++ this is replaced by polymorphism

Memory management

- In C, the **`malloc`** and **`free`** functions are used to allocate and free heap-resident memory
  - **`malloc`** takes the number of bytes to allocate
  - **`free`** takes a pointer

```
// Allocate an array of Fish
struct Fish * tank;
int fishCount = 50;
tank = (struct Fish *)malloc(
      sizeof( struct Fish ) * fishCount );

swim( tank );  // Let the fish swim around

free( tank );
```