Intro to C++

Lecture 8

Binary File I/O, structures, Classes I

File I/O

- Previously, we looked at ***sequential*** file I/O (Input/Output) where we proceeded through a file, gathering each piece of data after the prior piece.
- The files we looked at were also ***ASCII*** files, which are human-readable in an editor such as Notepad. Each character takes up 8 bits of memory.
- ***Binary*** files are not easily human-readable. Data is stored in these files in the same way that it is stored in RAM (e.g. a long would take up 32 bits and it would be stored with the most-significant bit as the negative flag.

File I/O

- To emphasize the difference, let’s try storing the value `unsigned char num = 65;` in each type of file, then opening it in Notepad:
  - 65
  - A
- In the first case (ASCII), the value ’65’ is displayed because we stored ’65’ as an ASCII value. In the file, two bytes are stored, the first which is the ASCII representation of the number ‘6’, and the second which is the ASCII representation of the number ‘5’.
- In the Binary file, only a single byte with the value ’65’ is written to the file. This is interpreted by Notepad as ‘A’ because 65 is the ASCII value for the letter ‘A’.

Binary Files

- Binary files are often read ***non-sequentially***. For instance, we might only want to obtain a single value from the file, and it would be senseless to read in ALL of the data to only retrieve one byte.
- Thus, we can tell the computer to read data from different positions in the file; however, because the hard disk is slow at reading, we should not skip around too often (it is faster to read data in order than in random order).

Opening the File

```
ifstream inFile (“data.dat", ios::in | ios::binary);
```

- Note that this is exactly how we opened our ASCII files, except this time we appended an extra parameter. The ‘|’ (pipe, bitwise-OR) allows us to apply both conditions to the file at once.
- ios::in allows us to open the file for reading.
- ios::binary allows us to treat the file as non-ASCII.

Reading from the File

```
char buffer[100];
ifstream inFile ("data.dat", ios::in | ios::binary);

if (!inFile.read (buffer, 100))
{
   // An error has occurred
}
```

- We can use the *read* command to read a given number of bytes from the current “get” cursor in the file. When a file is opened, the “get” cursor is initially positioned at the beginning of the file.
- Note that the data is stored wherever the first parameter *points to*. In this case, we ***statically*** allocated 100 bytes of memory, but we can also ***dynamically*** allocate it:

```
char *buffer = new char[100];
…
delete [] buffer;
```

Seeking

```
inFile.seekg (100);
```

- The seekg function allows us to move the “get” cursor to another location in the file (called “seeking”). In this example, we move the cursor to the hundredth byte from the *beginning of the file* (***absolute***).
- To move to a position ***relative*** to the current cursor position, use seekg and the function tellg. The following moves the cursor 100 bytes forward from its current position:

```
inFile.seekg (inFile.tellg() + 100);
```

Writing to a File

- Similar commands for writing to a file exist (write, seekp). If you're interested in finding how these work, feel free to search the Internet, or visit the following site:

http://www.angelfire.com/country/aldev0/cpphowto/cpp_BinaryFileIO.html

Closing a File

- Remember to close the file once you’re done using it so the OS knows that your software is finished modifying it:

```
inFile.close();
```

Structures

- Earlier, we had the problem of only being able to return a single variable from a function.
- One way we got around it was to send ***pointers*** to other variables through the parameter list, allowing us to change their values as well.
- A third way to organize data and to modify many values at once is by making a ***structure***.

Structures

- In Pong, we have been faced with keeping track of many variables at once, all relating to a single ball: its x and y positions, its x and y velocities, and perhaps its color.
- What if we had more than one ball? Then we could make all of these variables arrays instead, but in any regard, we'd still have lots of free variables floating around unorganized.
- Wouldn't it be useful if we could group these together and say they're properties of a ball?

Pong Ball Structure

- We can! And this is called a ***structure***, or for short, a struct. Here is how we'd group together all of the ball properties:

```
struct Ball
{
   int x, y;
   int width, height;

   int velX, velY;
};
```

- Notice that we've taken all of the variables that are unique to a single ball and grouped them together in one unit called a ***structure***. By doing this, we're saying that they are all *properties* of a 'Ball'.

Pong Ball Structure

- We also sort of made up our own variable type called ‘Ball’. We can actually count how large it is by summing the bytes required to store its members then rounding up to the nearest 32-bit boundary (compilers often do this automatically for processing speed).
- In the last case, we have six integers, for a total of 6 ints*4 bytes/int = 24 bytes (which is divisible by 4 so it isn’t rounded up).
- The values in a structure are furthermore stored ***contiguously*** in memory! Reordering the variables inside the structure will reorder their location within memory as well. The topmost variable is always stored first in memory.

Accessing Structures

- It’s easy to ***statically allocate*** structures and access their member variables. For instance:

```
Ball pongBall;  // declare a Ball object

pongBall.x = 50;
pongBall.y = 100;
```

- Notice that we access each member using a ‘.’ character. After initially declaring the object, we can treat its members like any other variable.

Dynamically Allocating Structures

- Structures can be ***dynamically allocated*** as well:

```
Ball *pongBall = new Ball;
…
delete pongBall;
```

- Now wait a second. How would we access pongBall’s members?

```
(*pongBall).x = 50;
```

- This could get confusing if we have structs inside of one another:

```
(*pongBall).((*parentBall).x) = 50;
```

-> Notation

- Thus, a new notation was born, the ->:

```
(*pongBall).x = 50;
pongBall->x = 50;

(*pongBall).((*parentBall).x) = 50;
pongBall->parentBall->x = 50;
```

- You’ve already seen examples of using -> after itcc in the graphics framework! For instance:

```
itcc->SetPixel(0, 0, myColor);
```

Classes

- Hmm, but in the graphics framework, I’m calling a … function?
- ***Classes*** are new to C++ -- they can not only hold variables, but functions as well! Thus, a pong ball would be a good example of an ***object*** – an entity that performs actions and has properties. A ball can initialize, it can move, it can change color, and it has lots of the properties discussed before. One way we can turn the ball ***structure*** into a ***class*** is on the next slide.

Pong Ball Class

```
class CBall
{
private:
   int m_x, m_y;
   int m_width, m_height;
   int m_velX, m_velY;

public:
   CBall(int x, int y, int width, int height);  // Constructor
   ~CBall(void);                                // Deconstructor

   int GetX(void);                // Accessor functions
   int GetY(void);
   int SetVelX(int velX);
   int SetVelY(int velY);

   int Move(void);                // Other functions
};
```

m_, C, and g_

- Often, programmers will prefix ***member variables*** with ‘m_’ and ***global variables*** with ‘g_’. Classes are often prefixed with an upper-case ‘C’.
- This is a remnant of Microsoft’s ***Hungarian Notation***. This notation attempted to be a mechanism such that one could tell the type of any variable just by looking at the variable name. It often led to long, complex-looking variable names so it isn’t used as often nowadays.
- However, the good parts of it are still being used as above. (The notation is heavily used in the DirectX and Windows SDK documentation).

public vs. private

- ***Privately*** declared variables and functions can only be accessed from functions within the class itself.
- ***Publicly*** declared variables and functions can be accessed and set at any time.

- In general, all member variables should be declared as ***private*** and only accessed through ***accessor functions***. The general idea behind this is that your class should be treated as a *black box* by other programmers. In other words, no one else except for you should need to know how your class works – all that anyone else needs to know should be specified by your class's *interface* with the outside world (i.e. the public variables and functions).

The Constructor

- Every class can have one or more ***constructors***. This is a function that can be called when an object of the class is instantiated. The constructor always has the same name as the class.

- For instance, to call the CBall constructor, you can do one of the following:

```
CBall pongBall(0,0, 10,10);
CBall *pongBall = new CBall(0,0, 10,10);
```

The Deconstructor

- Every class can have a single ***deconstructor***. This is automatically called whenever an object of the class is about to be destroyed – the programmer **does not** call this function. Often, the constructor and deconstructor are used to ***dynamically*** allocate and deallocate memory.

- The ***deconstructor*** always has the same name as the class, prefixed with a tilde (~).

Member Functions

- ***Member functions*** are always ***defined*** inside the class, but they are usually ***declared*** outside of it (the class definition is often inside a header file and the declaration of its functions is often in a cpp file). To ensure that a function is declared as the member of a class, prefix it with the class name:

```
int CBall::GetX(void)
{
   return (m_x);
}
```

Example: Using a class

```
int main(void)
{
   CBall myBall(0,0, 10,10);

   myBall.SetVelX(1);   myBall.SetVelY(1);
   myBall.Move();
}

int main(void)
{
   CBall *myBall = new CBall(0,0, 10,10);

   myBall->SetVelX(1);  myBall->SetVelY(1);
   myBall->Move();

   delete myBall;
}
```

TODO

- Download ITCC_HW4.zip from the site (will be available soon) http://www.pclx.com/itcc/, and complete the homework exercises, emailing them (FOR THIS WEEK ONLY) to itcc_teachers@pclx.com. Please do not resubmit solutions, even if they are revised. All homework must be submitted by 6:00am PST August 2.
- This is the second-to-last assignment! Lectures will end Monday, August 2!
- If you finish with the homework, experiment!