

Before the lecture begins ...

- Go to <http://pclx.com/itcc>, download Homework 1/1b, and complete it.
- Review the Lecture 2 slides.

Intro to C++

Lecture 2

Variables, Mathematics, while loops, for loops

A Note on Windowed vs. Full Screen

- Games usually are first developed in Windowed mode – only when everything works are they typically tried in full screen.
- Why? Windowed mode has fewer fatal crashes, is easier to debug, and it doesn't have time-consuming monitor switching.
- If full screen debugging is necessary, programmers often use a second monitor or debug from a remote machine.

Variables

- Sometimes we need to **store a value**. We can do this using a ***variable***, memory that is reserved for storing a value.
- All data in a computer is stored the same way – as a number. How a program **interprets** this number gives rise to more complex abstractions such as characters, graphics, and audio.

ASCII

- One example of interpreting numbers as characters is the **ASCII code**. This code is the most popular way to use numbers to encode letters in a computer.
- In **ASCII**, for instance, the number '65' stands for 'A', '97' is 'a', and '32' represents a space.
- There were originally 128 possible numbers in **ASCII**, but to represent international characters, another 128 were added for a total of 256.
- Text files saved with the extension .txt are encoded using **ASCII**.

How Memory Works

- The most fundamental unit of computer memory is the **bit**. A **bit** can only be set to two states – either ‘1’ (on) or ‘0’ (off).
- Memory is comprised of trillions of **bits**, and by manipulating these, we can store data.
- How can we store a number using **bits**?

Base-10

- We are used to a **base-10** (decimal) number system – each digit can have one of ten possible states (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Note that any number in a **base-10** system can be decomposed into powers of ten as follows:

$$13_{10} = 1 \times 10^1 + 3 \times 10^0 = 10 + 3$$

$$8752_{10} = 8 \times 10^3 + 7 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 = 8000 + 700 + 50 + 2$$

Base-2

- Note that **bits** constitute a **base-2** (binary) number system because each digit can only have one of two states (0, 1). Hence,

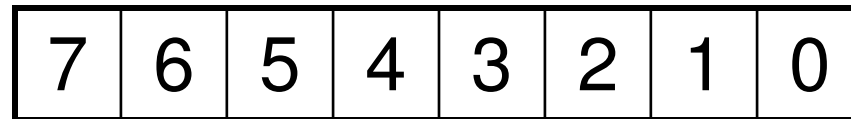
$$10_2 = 1 \times 2^1 + 0 \times 2^0 = 2 + 0 = 2_{10}$$

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

- From this, we have just figured out how we can store numbers by using lots of on/off switches (**bits**)!

The Byte

- Let's assume that we have a **byte**, eight bits, of memory:



- Bit 7 is the **most significant bit** (MSB). When decomposed, it adds $value * 2^7$ to the total sum.
- Bit 0 is the **least significant bit** (LSB). When decomposed, it adds $value * 2^0$ to the total sum.

The Byte

- What is the smallest value we can store in a **byte**? The smallest must be made of all zeros:

$$\mathbf{0000\ 0000}_2 = \mathbf{0} \times 2^7 + \mathbf{0} \times 2^6 + \mathbf{0} \times 2^5 + \mathbf{0} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{0} \times 2^0 = \\ 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 0_{10}$$

- What is the largest value we can store in a **byte**? It must be constructed of all ones:

$$\mathbf{1111\ 1111}_2 = \mathbf{1} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0 = \\ 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255_{10}$$

The Byte

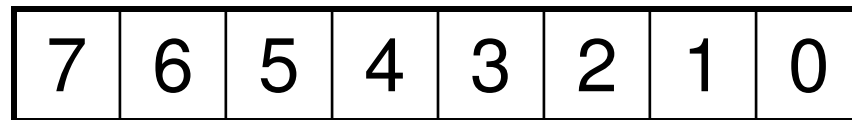
- Smallest number: 0_{10}
Largest number: 255_{10}
- Thus, we can store all numbers between 0 and 255 (inclusive) in a byte, meaning that there are 256 unique numbers that can be stored (1 through 255 in addition to the number 0).

Data Types

- Remember how it takes 256 different numbers to represent one international character using **ASCII**?
- In C/C++, a **byte** is a **char** ('character'). However, remember that a *number* is stored in a **char** – one interpretation of that number is as an **ASCII** character.

Negative Numbers

- So how are negative numbers stored?



- If the **most significant bit** (bit 7) is on, then the number in bits 0–6 is negative. If it is off, then the number is positive.
- Note, however, that now our minimum is -127 and our maximum is +127. Including zero, there are now only 255 possible numbers (127 + 127 + zero)!
- Why? Note that now, zero can be represented two ways – either as 1000 0000₂ or 0000 0000₂ since zero is neither positive nor negative.

32-bit Processors?!

- What does it mean when we say a processor is *32-bit*?
- Different types of memory are available in a computer. In general, memory that is closer to the processor can be accessed faster.
- Thus, RAM installed on the motherboard is faster to access than your hard drive.

32-bit Processors?!

- There is also memory (made up of “registers”) that is stored on the *same chip* as the microprocessor!
- Oftentimes, the microprocessor is optimized to manipulate numbers that are a certain number of bits long, and this optimal number is often reflected by the size of the registers.
- This optimal value is *32-bits* for most modern-day processors, hence their name.
- You can automatically **allocate** this optimal number of **bits** in C/C++ by using the data type **int** (integer).

Data Types

- Variable types in C/C++:
 - `char` (character), 8 bits (256 combinations)
 - `short int` (integer), 16 bits (65,537)
 - `long int` (integer), 32 bits (4,294,967,297)
 - `int` (integer), equivalent to `long int` on 32-bit processors
- `short int` can be abbreviated to `short`, and `long int` can be abbreviated to `long`.

Declaring Variables

- When we **declare variables**, we **allocate**, or set aside, a certain number of **bits** of memory to store a value.

```
char myVar;
```

- This **allocates** 8 bits of memory which can be read and written to by using the **variable** myVar.

```
char myVar = 5;
```

- This **allocates** 8 bits of memory, then it **initializes** those 8 bits to store the value 5 (0000 0101₂).

Variable Scope

- **Variables** can be **local** or **global** in scope:
 - **Global**: Is created at the program's execution and destroyed when the program exits. If a variable is **declared** outside any braces (for example, above the `main()` function), then the variable will exist for the entire program duration.
 - **Local**: Is created and destroyed in the middle of a program's execution. If a variable is **declared** within a pair of braces, then the variable has local scope – it is allocated at the point of declaration and destroyed at the ending brace.

Data Type Prefixes

- We can also add prefixes to the data type of a variable, for instance to determine whether a variable can store negative numbers:
 - `signed` – can be either positive or negative
 - `unsigned` – can only be positive
- By default, newly declared variables are `signed`.

`unsigned char myVar;`

- `myVar` can store a number from 0 to +255.

`signed char myVar;`

- `myVar` can store a number from -127 to +127.

The Assignment Operator

- At any point in our program, we can **assign** a **variable** a value using the **assignment operator =**:

```
myVar = 5;
```

- Whenever we use a **variable name**, we are referring to whatever value is stored in the memory location the **variable** references. Note that the assignment is not made until the entire right hand expression is evaluated.

```
numLives = 5 - startingLevel; // sets numLives based on startingLevel
```

```
numLives = numLives + 1; // adds one to numLives
```

Mathematics Operators

Five basic mathematics *operators*:

- + Addition
- - Subtraction
- * Multiplication
- / Division
- % Modulus (Remainder)

Note, however, that expressions being stored into an integer **variable** are always **truncated** – any fractional portion is ignored:

`long myVar;`

- `myVar = 5 / 2;` (therefore, `myVar = 2`)
- `myVar = 3 * 1.5;` (therefore, `myVar = 4`)

The Modulus Operator (%)

The modulus operator (%) performs a division then takes the *remainder* as the result instead of the whole number:

- $\text{myVar} = 1 \% 2; (1/2 = 0 + \mathbf{1}/2, \text{myVar} = \mathbf{1})$
- $\text{myVar} = 5 \% 3; (5/3 = 1 + \mathbf{2}/3, \text{myVar} = \mathbf{2})$
- $\text{myVar} = 8 \% 8; (8/8 = 1 + \mathbf{0}/8, \text{myVar} = \mathbf{0})$

Prefix/Postfix

- There are shorthands that can be used:

```
numLives += 1; // numLives = numLives + 1
```

```
currHealth -= 50; // currHealth = currHealth - 50
```

- Also note the prefix/postfix operators ++/--:

```
++numLives; // increments numLives by 1 BEFORE the statement is evaluated.
```

```
-- currHealth; // decrements numLives by 1 BEFORE the statement is evaluated.
```

```
numLives++; // increments numLives by 1 AFTER the statement is evaluated.
```

```
currHealth - ; // decrements numLives by 1 AFTER the statement is evaluated.
```

Using Variables in `if` Statements

- The ***comparison operators***:

`<` less than

`>` Greater than

`<=` less than or equal to

`>=` greater than or equal to

`!=` not equal to

`==` equal to // NOTE: DO NOT CONFUSE COMPARISON WITH ASSIGNMENT!!!!

- Comparisons are made within `if` statements. Each comparison results in a TRUE or FALSE response that the `if` statement acts on:

```
if (xPos > 640)
{
    xPos = 640;
}
```

```
if (guess == number) // note the '==', NOT '='
{
    itcc->Text(0,0, "You win!", RGB(255,255,255));
}
```


What can I do now?

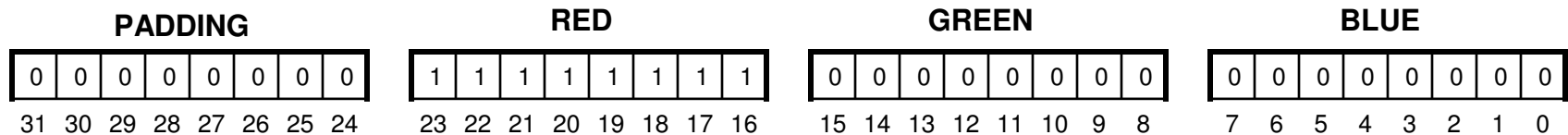
- At this point, you have enough knowledge to make a simple game! By making the x and y values in `itcc->SetPixel` ***variables***, you can change their positions if, say, the user presses an arrow key.
- Try starting with a ball bouncing off the sides of the screen, then if that works, try adding some paddles (this activity will probably be in the coming homework)!

Case Study: Storing a Pixel

- The *bit depth* of a video resolution corresponds to the number of **bits** that are required to display a single pixel.
- For instance, *24-bit* color has an 8-bit channel for red, one for green, and one for blue, for a total of 24 bits.
- *32-bit* color also has three 8-bit RGB channels – the remaining 8 bits are often used for **padding**, unused memory, so we can achieve the optimal number of bits for the 32-bit processor and thus perform operations faster.

Storing a Pixel

- Note that a **long int** in C/C++ is 32-bits long, so we can store one 32-bit color pixel as a **long int** in C/C++.
- Here's how a pure red 32-bit pixel is stored in memory:



- Converting this binary value to decimal, we get:
$$1 \times 2^{23} + 1 \times 2^{22} + 1 \times 2^{21} + 1 \times 2^{20} + 1 \times 2^{19} + 1 \times 2^{18} + 1 \times 2^{17} + 1 \times 2^{16} =$$
$$= 16711680_{10}$$
- Thus, for 32-bit color, **long** redPixel = 16711680;

32-bit Color

- Also note that since there are 8 bits per channel (256 intensities) and three channels (RGB), then it is possible to display $256 \times 256 \times 256 = 1.68$ million colors.
- Can we write a program that will take as input three values – a percentage of red, a percentage of green, and a percentage of blue – then display that color pixel on the screen?

Percentage Conversion

- First, we'll get three values as input:

```
unsigned long red    = 0;
unsigned long green  = 0;
unsigned long blue   = 0;
unsigned long finalColor = 0;
red    = itcc->GetInt("Enter red percentage (0-100%):");
green  = itcc->GetInt("Enter green percentage (0-100%):");
blue   = itcc->GetInt("Enter blue percentage (0-100%):");
```

- Next, we'll scale the values so instead of being between 0 and 100, they are between 0 and 255:

```
red    = (unsigned long)(red    * 2.55);
green  = (unsigned long)(green  * 2.55);
blue   = (unsigned long)(blue   * 2.55);
```

Bit Shifting

- Note in **base-10** that if we shift all of the digits of a number to the left and insert zeros on the right, we *multiply by 10^n* , where n is the number of bits shifted. Shifting the digits to the right is a *division by 10^n* .
- Similarly, in **base-2**, a left shift is a *multiplication by 2^n* and a right shift is a *division by 2^n* (note that this is ONLY when the **variable** is stored as an **unsigned** integer!)

Bit Shifting

- So how are we going to get the *variables* red, green, and blue into the right places in the `long int`?
- One method is to use *bit shifting* to “slide” the bits into place.
- In C/C++, the *operator* `<<` shifts bits to the left, and `>>` shifts bits to the right:

```
unsigned int time = totalTime >> 2; // divides by 4
```

Pixel Packing

- Finally, since each value is now between 0 and 255, we can pack the `ints` into one 32-bit value and display it:

```
finalColor = (red << 16) + (green << 8) + blue;  
itcc->SetPixel(0,0, finalColor);
```

- Looking at the RGB32 macro we were using for `itcc->SetPixel` we find the same method:

```
#define RGB32(r,g,b)      (((r) << 16) + ((g) << 8) + (b))
```


TODO

- Download ITCC_HW2.zip from the site (will not be available until Wednesday, July 7) <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 Monday, July 12.
- Some illustrative examples of the topics in this lecture are given in ITCC_HW2.zip.
- If you still have problems compiling the framework, please make sure to get in contact with us!
- Look over the slides for the third lecture before Thursday, July 8.
- If you finish with the homework, experiment!