Lecture 2

Introduction to Programming:

Data, Arithmetic Statements, Cout, Compiling

Chapters 2, 3

First Program: Console Output

```
#include<iostream>
using namespace std;

int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

Output of the program is:

```
Hello World!
```

- cout (Console Out) allows the programmer to display something to the console

- cout is a predefined class meaning a programmer has already taken the time to write code that interfaces with console.
- Importantly, we do not need to know how cout works, only how to use it.

First Program: Library and Main

```
#include<iostream>
using namespace std;

int main()
{
		cout << "Hello world!" << endl;
		return 0;
}
```

- iostream is the library which contains the code for cout
- Library: a collection of completed defines, functions, and classes to assist in a program.

- int main() : this is a special function which defines the start of the program. **All programs must have one, and only one main function.**
- The first statement of the main is the first statement executed by the program. (With a lot of caveats and exceptions)

First Program: Namespace

```
#include<iostream>
using namespace std;

int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

- cout resides in the namespace of std, to use cout we must first use the namespace it resides in.

- Namespace: group entities, objects, and classes under a name.
- Importance: imagine we include two libraries that both have their own different cout class. When we type cout which one will be used?
  - If we type using namespace std then the cout residing in the std namespace will be used.

First Program: Compilation

```
#include<iostream>
using namespace std;

int main()
{
		cout << "Hello world!" << endl;
		return 0;
}
```

- Unfortunately our CPU does not inherently understand C++; it only understands binary instructions.

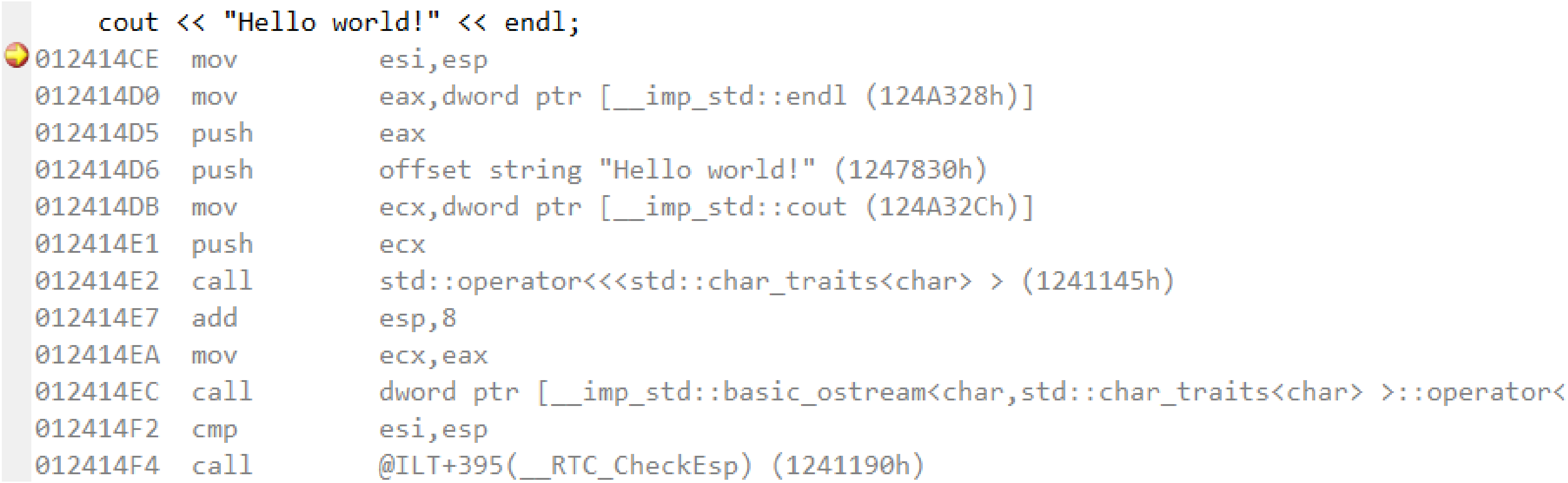
- Therefore we must compile the code to translate from the high level language to a binary executable

Definitions

Assembly:

A low level programming language using the human readable instructions of the CPU.

Looks Like:



```
    cout << "Hello world!" << endl;
012414CE  mov          esi,esp
012414D0  mov          eax,dword ptr [__imp_std::endl (124A328h)]
012414D5  push         eax
012414D6  push         offset string "Hello world!" (1247830h)
012414DB  mov          ecx,dword ptr [__imp_std::cout (124A32Ch)]
012414E1  push         ecx
012414E2  call         std::operator<<<std::char_traits<char> > (1241145h)
012414E7  add          esp,8
012414EA  mov          ecx,eax
012414EC  call         dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<
012414F2  cmp          esi,esp
012414F4  call         @ILT+395(__RTC_CheckEsp) (1241190h)
```

All that for one line of code. Notice in the assembly it performs a call, meaning it will jump to another point in the assembly and start executing instructions.

This one line of code will execute thousands of lines of instructions!!!!

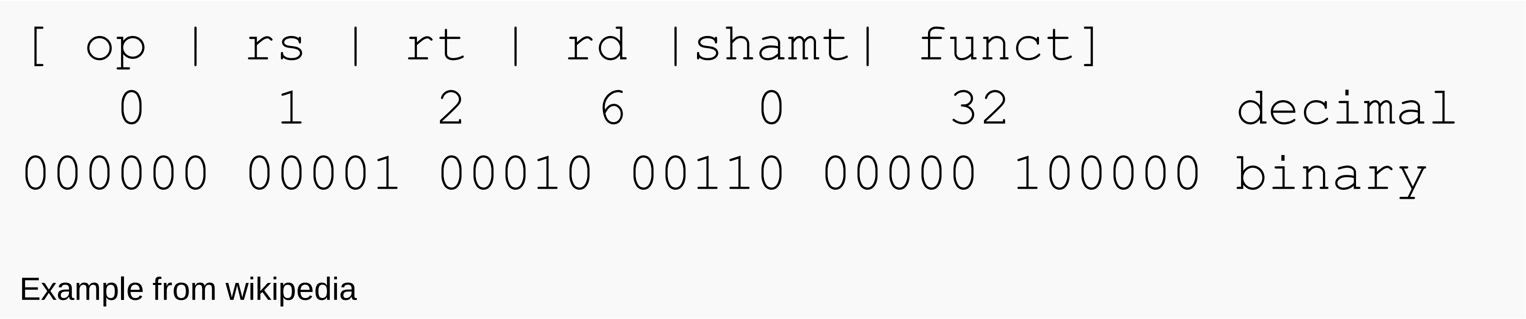
Definitions

- Binary: composed of entirely of:

  1's (represented as a high voltage in computers)

  0's (represented as a low voltage in computers)

- CPU processes only binary instructions (Machine Code)



```
[ op | rs | rt | rd |shamt| funct]
   0    1    2    6    0     32       decimal
000000 00001 00010 00110 00000 100000 binary
```

Example from wikipedia

- Instructions are separated into fields each with its own sets of binary encodings giving the field meaning.
- Example:
  - Opcode: 000000 is Add
  - Opcode: 000001 is Subtract

Translating From Binary to Decimal

- Each position represents a power of 2, starting from the least significant bit (LSB), the right most up to the left most

- Example: Translate binary number 100110 to decimal.
- Step 1: Determine value of each bit

1	**0**	**0**	**1**	**1**	**0**
2^5	2^4	2^3	2^2	2^1	2^0

- Step 2: Multiply the value of the position by the digit and sum them

$1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

$32 + 4 + 2 = 38$

Answer is 38

Translating From Decimal to Binary

- Repeatedly divide by 2, write the remainder to the right
- When the quotient is zero, stop
- The most significant bit is the last remainder wrote, and the LSB is the first
- Example: Translate decimal number 38 to binary.

Op	Quotient	Remainder
38/2	19	0
19/2	9	1
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

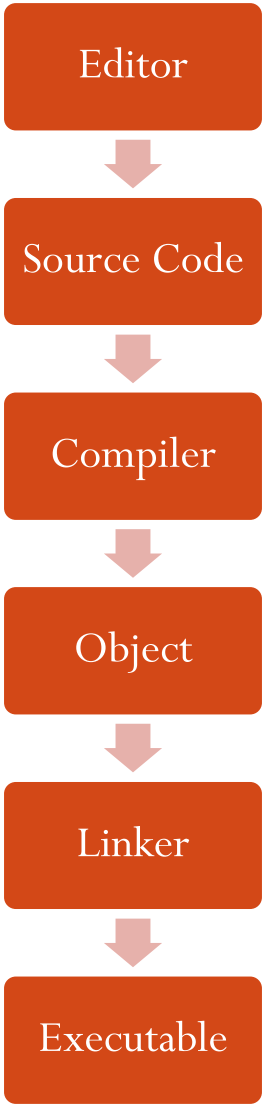
Answer is:
100110

Back to Compiling

- The set of instructions produced targets a specific platform
  - i.e. compilation for a x86 architecture will not work for a freescale imx 35 (different machine code)
- Compiler performs optimizations on source code to increase performance
  - Doesn’t fix inefficient algorithms
  - Trade offs between memory usage and processer time
- The compiler will flag syntax errors but not logical (semantic)
  - i.e. missing semicolons, braces, and failing to declare a variable will be caught, however a mistake in the implementation of the algorithm will not.

Procedure translating source code to working program

- The Editor
  - The interface that allows for the typing of the high level source code
  - Typically performs syntax highlighting
- The object
  - Encapsulation of your written code into binary instructions. (**Still not an executable though**)
- The Linker
  - Links source files together into a single executable.
  - Links or copies libraries into code
- Executable is your final program



Definition: Syntax Error

- An error in the use of the programming language which results in a failure to compile

- Example

```
int main()
{
    int someNum;
    someNum = 4
    return 0;
}
```

- someNum = 4 is a syntax error since a semicolon is missing. This will result in a failure to compile

Definition: Semantic Error

- An error in the program’s logic which causes undesirable behavior at runtime
- Example (Calculating the area of a triangle):

```
int main()
{
    int width = 4;
    int height = 6;
    int area;
    area = width * height * width;
}
```

- An incorrect formula is used to calculate the area. This program will run but return incorrect results

- In general semantic errors are harder to detect and to correct

Exploring cout

```
#include<iostream>
using namespace std;
int main()
{
        // Acceptable
        cout << “Hello world!” << endl;
        cout << “1976 Fiat 124 Spider is the best car ever made.” << endl;
        cout << 1976 << “ Fiat 124 Spider is the best car ever made.” << endl;
        cout << “1976” << “Fiat” << 124 << “Spider is the best car ever made.” << endl;
        return 0;
}
```

endl: creates new line

Rule: When using cout each element must be separated by a << (insertion operator)

Rule: Each statement must end with a semicolon to mark its completion

Definition: Literal

- A fixed value in the source code
  - Is always known at compile time

- Example of String Literal
  - `“here is a string literal”`

- Example of Number Literal
  - 6

Some Bad Examples

```
#include<iostream>
using namespace std;
int main()
{
        // Unacceptable
        // mising << between 1976 and “Fiat
        cout << 1976  “Fiat 124 Spider is the best car ever made.” << endl;

        // Fiat is not in quotes, this makes C++ believe Fiat is a variable
        // All text must be contained in quotes
        cout << 1976 << Fiat << “ 124 Spider is the best car ever made.” << endl;

        // Missing semicolon
        cout << “1976” << “Fiat” << 124 << “Spider is the best car ever made.” << endl
        return 0;
}
```

Pop Quiz

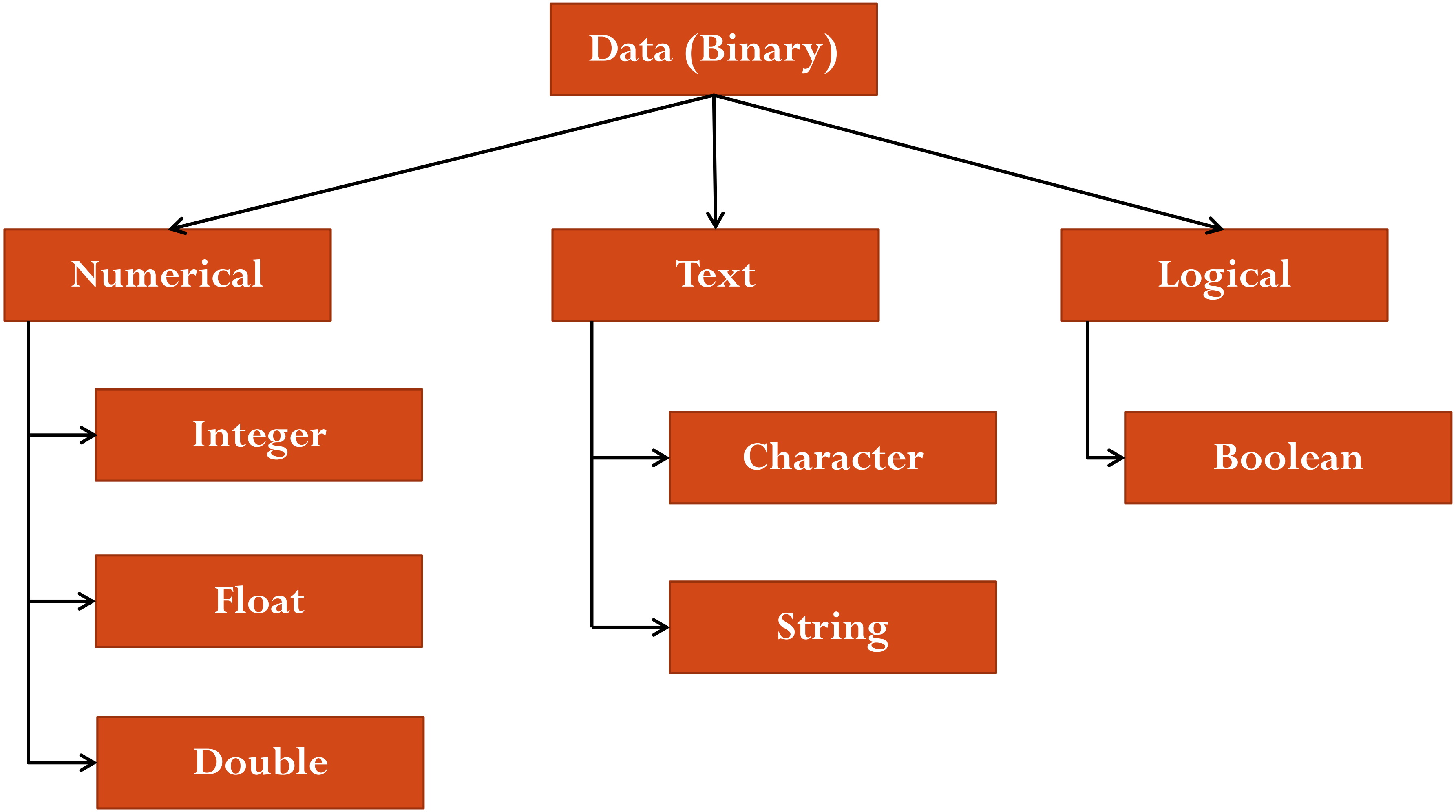
- What does this display?

```
#include<iostream>
using namespace std;
int main()
{
        cout << “My name is”;
        cout << “Luke Pierce”;
}
```

- Answer

```
My name isLuke Pierce
```

Types of Data



Example using data

```
#include<iostream>
using namespace std;
int main()
{
        int year;
        year = 1976;
        cout << year << “ Fiat ” << 124 << “ Spider is the best car ever made.” << endl;
        cout << “year” << “ Fiat ” << 124 << “ Spider is the best car ever made.” << endl;

        return 0;
}
```

- Displays:

```
1976 Fiat 124 Spider is the best car ever made.
year Fiat 124 Spider is the best car ever made.
```

- Notice by not surrounding year in quotes the contents of the variable is printed

Example using data

```
#include<iostream>
using namespace std;
int main()
{
        int year;
        year = 1976;
        cout << year << “ Fiat ” << 124 << “ Spider is the best car ever made.” << endl;
        return 0;
}
```

- Statement: int year;
  - Declares (creates) an integer variable whose name is year
  - The name of the variable only gives context to the programmer as to what the variable contains.
  - The name of the variable never sets the actual contents
    - Naming your variable one will not automatically set it to one

Example using data

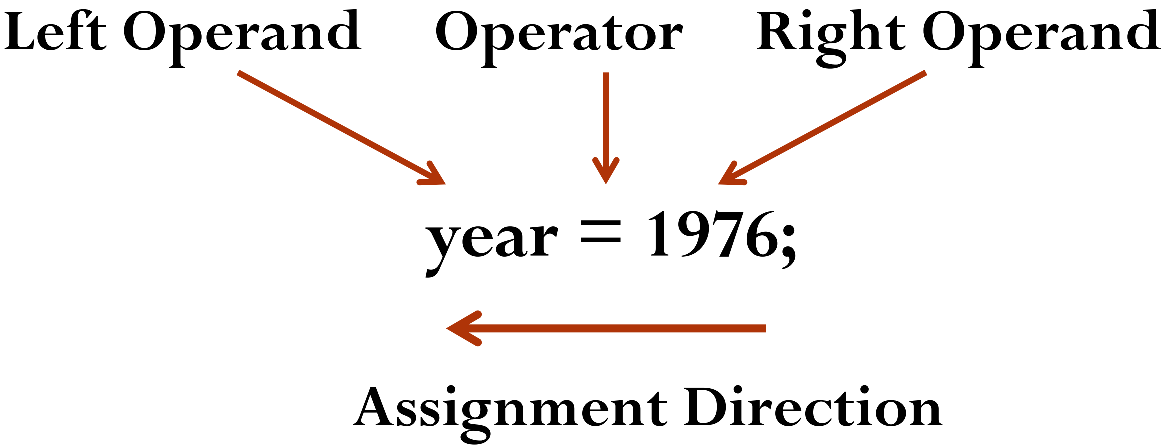
- Statement: int year;
  - All variables must be declared before being used.
  - We will see why in a few slides
  - Good Programming Practice (GPP):
    - Variable names should always give context!!!
  - Variable Naming Rules
    - Can only consist of letters, numbers, underscores
    - May start with a underscore or letter but not a number
  - GPP
    - variable names should start with lowercase letter unless a constant
    - if a name contains multiple words each new word should be capitalized. I.e bondInterest instead of bondinterest

The Assignment Statement

Excluding Program Skeleton:

```
int year;
year = 1976;
cout << year << “ Fiat ” << 124 << “ Spider is the best car ever made.” << endl;
```

Sets variable contents. The left operand is always set to the right operand



Left Operand **Operator** **Right Operand**

year = 1976;

Assignment Direction

Therefore the left operand must always be a variable and never a constant

Arithmetic Operators (Compute Area)

```
int area;
area = 5 * 4;
cout << “Area is: “ << area << endl;
```

- Statement: area = 5 * 4;
  - Arithmetic operators have a high precedence than the assignment and are therefore executed first
    - Step 1: compute 5 * 4 (20)
    - Step 2: assign 20 into area
  - Operators: -, +, / , *, %
  - Order of operations and parentheses work exactly as a calculator would
  - **Implicit multiplication is illegal!!!!**
    - **area = 5(4) will not compile!!!**

Modulus Operator (%)

```
int remainder;
remainder = 15 % 8;
cout << “Remainder of 15 / 8 is: “ << remainder << endl;
```

- Statement: remainder = 15 % 8;
  - % computes the remainder (not percentage)
    - Therefore 7 is stored in remainder
  - Rules:
    - both the left and right operators must be an integer. You cannot take a remainder of a non-whole number

Single Variable Declaration

```
int x;
int y;
```

Multiple Variable Declaration

```
int x, y;
```

Single Variable Declaration with initialization

```
int x = 5;
int y = 2;
```

Multiple Variable Declaration with initialization

```
int x = 2, y = 4;
```

Word of Caution

Consider the following:

```
int number;
cout << “Number is: “ << number << endl;
```

- What is the displayed?

Cannot be determined. This is an example of an uninitialized variable. The contents is whatever has stored previously in that memory position.

Uninitialized variables can result in peculiar outputs.

The Compiler will typically give a warning.

Chaining Operations and Using Variables

```
int width, height, length, volume;
width = 20;
height = 10;
length = 15;
volume = width * height * length;
cout << “Volume of the Box: “ << volume << endl;
```

- Just as a calculator you can perform multiple operations prior to storing the result
- Operations can be applied to constants as well as variable

Integers and a word of caution

```
int triArea, height, width;
width = 11;
height = 1;
triArea= width * height  * .5;
cout << “Area of the Triangle is: “ << triArea<< endl;
```

Displays:

```
Area of the Triangle is: 5
```

- The expected result would be 5.5 however an int (integer) is unable to store a non-whole number
- In the case where a non-whole number is stored into an int, the fraction is **truncated**
- For example: if I try to store the number 3.99 into an int, the number stored shall be 3

Floats

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \overbrace{10^{-4}}^{\text{exponent}}$$

how floats are stored in memory

```
float triArea;
int height, width;
width = 11;
height = 1;
triArea= width * height  * .5;
cout << “Area of the Triangle is: “ << triArea<< endl;
```

Displays:

```
Area of the Triangle is: 5.5
```

- Floats are capable of storing fractions of a number
- Floats have limited precision to typically around 6 decimal places
  - If you store 3.99 into a float it will be stored as 3.99
  - If you store 3.999999999999 into a float it will be rounded to 4
  - This is due to limitations in the instruction size, use double if precision is an issue

If a float can store fractions why use integers at all?

- Most Important Reason – Speed
  - Your CPU has hardware specific for integer operations (IU) and for floating point operations (FPU) in the Arithmetic Logic Unit (ALU)
  - Integer operations require significantly less clock cycles
  - Most architectures include more IUs than FPUs
    - This means more integers can be processed at once vs floats

  - Bottom Line: Anytime a fraction is not required, use an Integer

Chars

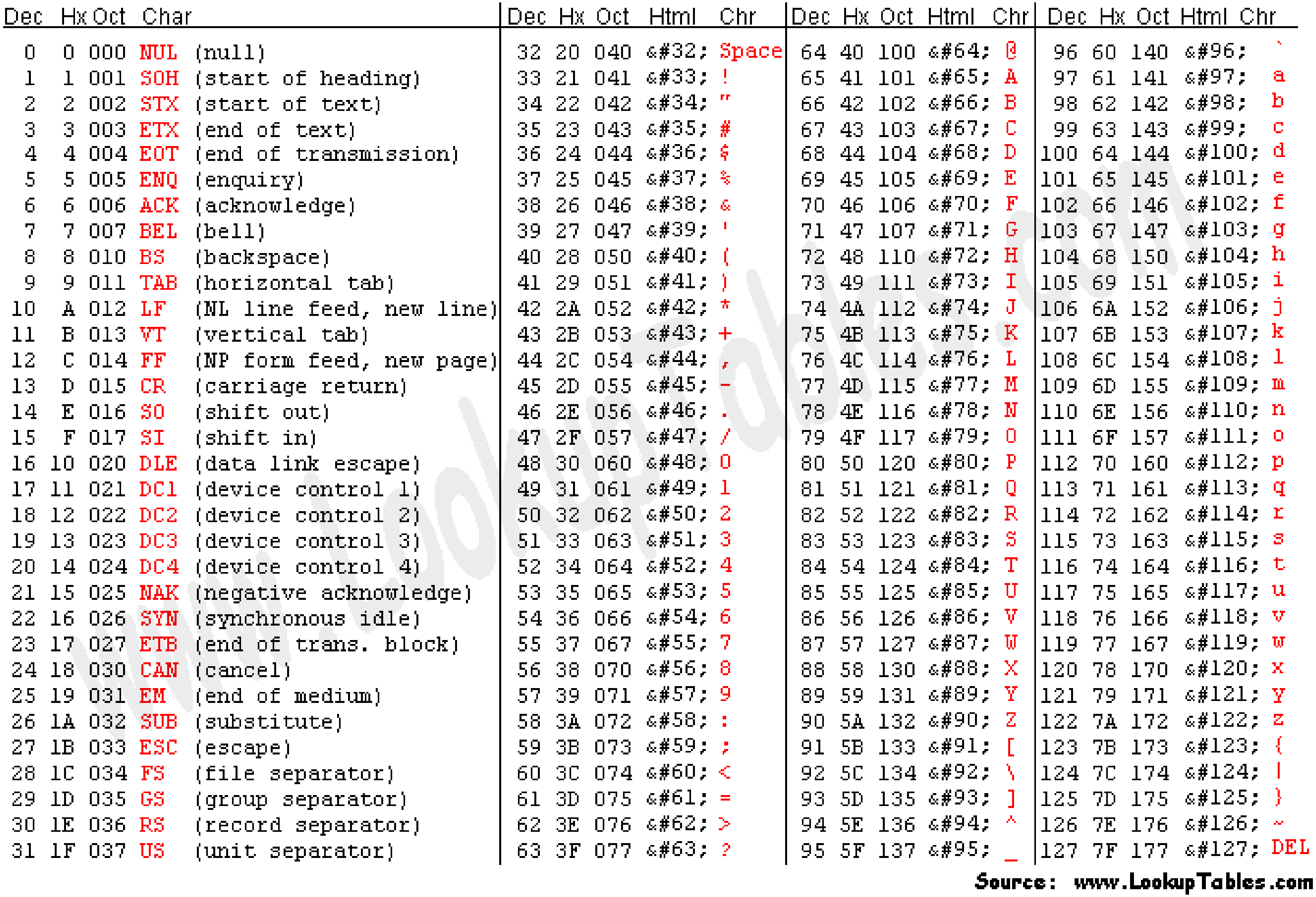
```
char first, second, third, fourth;
first = ‘L’;
second = ‘u’;
third = ‘k’;
fourth = ‘e’;
cout << first << second << third << fourth << endl;
```

Displays:

```
Luke
```

- A char can store a **single** letter
- Keep in mind if you name a char *a* the variable will not automatically contain an *a*
- To refer to a constant character place a character between a set of apostrophes
- Interestingly: a char is nothing more than an integer

Ascii Chart



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Char Pitfalls – MultiChar

```
char name;
name= ‘Luke’;
cout << name << endl;
```

- As tempting as the above maybe, it will not compile
- A char can only take a single character
- Thus why it is called char and not chars

Char Pitfalls – Special Characters

```
char apstrph;
apstrph = '‘’;
cout << apstrph << endl;
```

```
char apstrph;
apstrph = ‘\‘’;
cout << apstrph << endl;
```

- What if I want to print a apostrophe?
  - The left will **not** compile, the compiler will read the first two apostrophes as a closed set, and the third as an open apostrophe generating a compiler error
  - For corner cases we must use the escape character (\) shown on the right
- The escape character can print many special characters listed on page 85

Strings

```
string name;
name= “Mr. Rasputin”;
cout << name << endl;
```

Prints *Mr. Rasputin* to the console

- Allow for a series of characters
- NOT a primitive data type
- String literal must be surrounding by quotations
- Can contain multiple words and lines

Boolean

- Simplest datatype: can only be true (not zero) or false (0)

Examples

```
bool t1, t2, t3, f1, f2;
t1 = 5;  // Will be true
t2 = true; // Will be true
t3 = -1; // Will be true
f1 = 0; // Will be false
f2 = false // Will be false
```

- Booleans are represented in memory as a whole byte because modern computers do not have bit addressable memory, rather byte addressable

Data Types

Type	Range (32-bit System)	Description
int	-2147483648 to 2147483647	All whole numbers both positive and negative
float	+/- 3.4e +/- 38 (~7 digits)	Floating point numbers. Any real number.
double	+/- 1.7e +/- 308 (~15 digits)	Floating point number with twice the precision as float
char	-128 to 127	Each number corresponds to a symbol. The decoding follows an ascii chart.
bool	0 to 1	Can only take the value true or false

C++ Lunacy

```
float answer;
answer = 5 / 2;
cout << answer << endl;
```

What will the answer be?

2.0 Huh?

- The above is an example of quirk in C++ programming called integer division
  - When dividing, if both the left and right operands are integers then the result will always be an integer
  - Essentially the computer divides using the UI
  - answer = 5 / 2.0; would fix the issue

Whitespace

- For the most part C++ is whitespace independent

```
float answer;
answer = 5 / 2;
answer = 5         /       2  ;
answer = 5 /


 2;
answer = 5/2; answer = 5/2;
```

- All of the above is syntax correct, though only the first is acceptable for GPP

Whitespace Cont.

- Newlines in strings cannot be achieved by the following:

```
cout << “This
will not produce a new line”;
```

Displays:

```
Thiswill not produce a new line
```

- Either endl or \n must be used
- Same applies to tabs
- Does not apply to spaces since they are a normal character

Comments

- Comments don’t execute. Use them to document your code. To make it easier for other people to read.

```
// this is a single line comment
// another single line comment
/* this
   is a
   block comment
*/
```

- Block comments are useful for commenting out sections of code for debugging.

When to use Comments

- Personal Preference
- If the functionality of a piece of code is not immediately obvious add a comment
- Generally good things to comment: functions, loops, new stages in algorithm

- For GPP you will be required to have comments on every piece of source code that you submit