CS 107
Lecture 0: Welcome

Monday, September 25, 2017

#include<stdio.h>
#include<stdlib.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}

(gdb) disassemble main
Dump of assembler code for function main:
 0x000000000040052d <+0>:   push %rbp
 0x000000000040052e <+1>:   mov %rsp,%rbp
 0x0000000000400531 <+4>:   mov $0x4005d4,%edi
 0x0000000000400536 <+9>:   callq 0x400410 <puts@plt>
 0x000000000040053b <+14>:  mov $0x0,%eax
 0x0000000000400540 <+19>:  pop %rbp
 0x0000000000400541 <+20>:  retq
End of assembler dump.
(gdb)
Today's Topics

• What is CS107?
• Who We Are
• Course Components and Overview
• The C Language
• Logistics
  • Exams
  • Labs
  • Assignment 0
  • Lab Signup
• Questions / Concerns about CS107?
• Bits and Bytes
What is CS107?

- The CS106 series teaches you how to solve problems as a programmer
- Many times CS106 instructors had to say “just don’t worry about that” or “it probably doesn’t make sense why that happens, but ignore it for now” or “just type this to fix it”
- **CS107 finally takes you behind the scenes**

**How do things really work in there?**
- It’s not quite down to hardware or physics/electromagnetism (those will have to stay even further behind the scenes for now!)
- It’s how things work **inside Java/C++ (we will explore from C)**, and how your programs map onto the components of computer systems
CS107 Learning Goals

- The goals for CS107 are for students to gain **mastery** of
  - writing C programs with complex use of memory and pointers
  - an accurate model of the address space and compile/runtime behavior of C programs
- to achieve **competence** in
  - translating C to/from assembly
  - writing programs that respect the limitations of computer arithmetic
  - identifying bottlenecks and improving runtime performance
  - writing code that correctly ports to other architectures
- working effectively in UNIX development environment
- and have **exposure** to
  - a working understanding of the basics of computer architecture
Who We Are

Julie Zelenski
zelenski@stanford.edu

Chris Gregg
cgregg@stanford.edu
Course Components and Overview

Textbook: Bryant and O’Hallaron, 3rd Edition

You must get the 3rd Edition, as things have significantly changed since the previous editions.

- The suggested C reference is just one suggestion
  - You could do just as well with a different C book
  - Just need somewhere to turn when you have a question about C
## Course Components and Overview

<table>
<thead>
<tr>
<th>Week</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Admin, UNIX environment, bits/bitwise ops</td>
</tr>
<tr>
<td>2</td>
<td>Integer representation, computer arithmetic, C pointers/arrays</td>
</tr>
<tr>
<td>3</td>
<td>C-strings, C stdlib, dynamic allocation</td>
</tr>
<tr>
<td>4</td>
<td>C generics, void *, function pointers</td>
</tr>
<tr>
<td>5</td>
<td>Floating point representation, intro to assembly</td>
</tr>
<tr>
<td>6</td>
<td>x86-64: addressing, ALU ops Midterm: <strong>Fri Nov 3 in-class</strong></td>
</tr>
<tr>
<td>7</td>
<td>x86-64: control, function calls, runtime stack</td>
</tr>
<tr>
<td>8</td>
<td>Address space, dynamic memory management</td>
</tr>
<tr>
<td></td>
<td><em>Thanksgiving recess</em></td>
</tr>
<tr>
<td>9</td>
<td>Performance, case study</td>
</tr>
<tr>
<td>10</td>
<td>Advanced topics, wrap/review</td>
</tr>
<tr>
<td>11</td>
<td><strong>Final:</strong> Wed Dec 13th 3:30-6:30pm</td>
</tr>
</tbody>
</table>
The C Language: History and Background

- **Birthdate around 1970**
- Created to make writing Unix (the OS itself) and tools for Unix easier
- Part of the C/C++/Java family of languages
  - (with C++ and Java coming later)
- **Design principles:**
  - Small, simple abstractions of hardware
  - Minimalist aesthetic
  - C is much more concerned with efficiency and minimalism than safety (Java) or convenient high-level services and abstractions (Java, C++)
Some things will be very familiar:
› Syntax
› Basic data types
› Arithmetic, relational, and logical operators

You may be sad about what’s missing:
› No power features of C++ (overloading operators, default arguments, pass by reference, classes/objects, fancy ADTs)
› Thin standard libraries (no graphics, networking, etc)
› Weak compiler checks, almost no runtime checks

Benefits:
› Small language footprint (not much to learn)

Philosophical difference:
› Procedural (C)
› Procedural + Objects (C++)
› Object-Oriented (Java)
The C Language: Hello, World! Compiling, gdb
Logistics

See the Course Handout for details (link)

Web site: https://cs107.stanford.edu

Class time: 1:30-2:50, M/F, Cubberley Auditorium

Labs: Various Times Tu/We/Th in Gates B08

Exams: In-Class Midterm, Friday, November 3rd
Final Exam: Wednesday, December 13th, 3:30pm-6:30pm
(Note: there are no alternate exam times)
Questions/Concerns About CS107
Assignment 0: Unix!

Assignment page: http://web.stanford.edu/class/cs107/assign0/

Assignment will be released tomorrow (Tuesday morning, due Monday)

Five parts:
1. Read / View Unix Overview Documents / Videos
2. "Clone" Assignment 0 starter code
3. Answer Questions in readme.txt
4. Run make to compile a program, and make minor modifications
5. Submit the assignment
Lab Signup

Online:

https://cs107.stanford.edu/labs

The signup will be available Tuesday, September 26, 10:00am.

Labs will be weekly, starting during week 2.

First-come, first-served for lab signup times, which are held on Wednesdays, Thursdays, Fridays.
Computers are good at detecting "off" or "on"

We have lots of ways to tell the difference between two different states:

- Clockwise / Counterclockwise
- Lightbulb off / on
- Punchcard hole / no hole
Computers are good at detecting "off" or "on"

Electronic computers are built using transistors

A transistor can be set up to either be "off" or "on" -- this gives us our 0 and 1!
One bit doesn't do much for us!

• We call a single on/off representation a 'bit'.
• But having one bit isn't particularly helpful!
• We only have two states we can represent with one bit!

• If we want more states, we simply combine bits together, much like we do with base 10 representation.
• If we want to combine more than ten states with base 10, we add another digit.

• Base 10 has ten digits: 0 1 2 3 4 5 6 7 8 9
• We can represent up to ten numbers with one digit in base 10
• If we want to represent more numbers, we add more digits: 10 11 12 13 14 ...

• Base 2 is the same. We can represent two numbers with one digit: 0 or 1
• To represent more numbers, we add more digits! 10 11 100 101 110 ...
Combinations of bits can represent everything. We can encode anything we want with bits. E.g., the ASCII character set.

### ASCII Code: Character to Binary

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0011 0000</td>
<td>O</td>
<td>0100 1111</td>
<td>m</td>
<td>0110 1101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0011 0001</td>
<td>P</td>
<td>0101 0000</td>
<td>n</td>
<td>0110 1110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0011 0010</td>
<td>Q</td>
<td>0101 0001</td>
<td>o</td>
<td>0110 1111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011 0011</td>
<td>R</td>
<td>0101 0010</td>
<td>p</td>
<td>0111 0000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0011 0100</td>
<td>S</td>
<td>0101 0011</td>
<td>q</td>
<td>0111 0001</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0011 0101</td>
<td>T</td>
<td>0101 0100</td>
<td>r</td>
<td>0111 0010</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0011 0110</td>
<td>U</td>
<td>0101 0101</td>
<td>s</td>
<td>0111 0011</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0011 0111</td>
<td>V</td>
<td>0101 0110</td>
<td>t</td>
<td>0111 0100</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0011 1000</td>
<td>W</td>
<td>0101 0111</td>
<td>u</td>
<td>0111 0101</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0011 1001</td>
<td>X</td>
<td>0101 1000</td>
<td>v</td>
<td>0111 0110</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>0100 0001</td>
<td>Y</td>
<td>0101 1001</td>
<td>w</td>
<td>0111 0111</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0100 0010</td>
<td>Z</td>
<td>0101 1010</td>
<td>x</td>
<td>0111 1000</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0100 0011</td>
<td>a</td>
<td>0110 0001</td>
<td>y</td>
<td>0111 1001</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0100 0100</td>
<td>b</td>
<td>0110 0010</td>
<td>z</td>
<td>0111 1010</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>0100 0101</td>
<td>c</td>
<td>0110 0011</td>
<td>.</td>
<td>0010 1110</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0100 0110</td>
<td>d</td>
<td>0110 0100</td>
<td>,</td>
<td>0010 1111</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0100 0111</td>
<td>e</td>
<td>0110 0101</td>
<td>;</td>
<td>0011 1010</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>0100 1000</td>
<td>f</td>
<td>0110 0110</td>
<td>j</td>
<td>0011 1011</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>0100 1001</td>
<td>g</td>
<td>0110 0111</td>
<td>?</td>
<td>0011 1111</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>0100 1010</td>
<td>h</td>
<td>0110 1000</td>
<td>!</td>
<td>0010 0001</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>0100 1011</td>
<td>i</td>
<td>0110 1001</td>
<td>'</td>
<td>0010 1100</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>0100 1100</td>
<td>j</td>
<td>0110 1010</td>
<td>&quot;</td>
<td>0010 0010</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>0100 1101</td>
<td>k</td>
<td>0110 1011</td>
<td>(</td>
<td>0010 1000</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>0100 1110</td>
<td>l</td>
<td>0110 1100</td>
<td>)</td>
<td>0010 1001</td>
<td></td>
</tr>
</tbody>
</table>

*space: 0010 0000*
**Unsigned Integers**: positive integers and zero only
Ex. 0, 1, 2, ..., 74629, 99999999

**Signed Integers**: negative, positive, and zero integers only
Ex. 0, 1, 2, ..., 74629, 99999999
(represented in "two's complement")

**Floating Point Numbers**: a base-2 representation of scientific notation, for real numbers
Ex. 0.0, 0.1, -12.2, 4.87563 x 10³, -1.00005 x 10⁻¹²
Computers use a limited number of bits for numbers.

Let's write a little program...
Computers use a limited number of bits for numbers.

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

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

$ gcc -g -O0 multTest.c -o multTest
$ ./multTest
-884901888
$
#include<stdio.h>
#include<stdlib.h>

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n", answer);
    return 0;
}

Recall that in base 10, you can represent: 10 numbers with one digit (0 - 9),
100 numbers with two digits (00 - 99),
1000 numbers with three digits (000 - 999)

I.e., with $n$ digits, you can represent up to $10^n$ numbers.

In base 2, you can represent:
2 numbers with one digit (0 - 1)
4 numbers with two digits (00 - 11)
8 numbers with three digits (000 - 111)

I.e., with $n$ digits, you can represent up to $2^n$ numbers

The C int type is a "32-bit" number, meaning it uses 32 digits. That means we can represent up to $2^{32}$ numbers.
Computers use a limited number of bits for numbers

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

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}

$ gcc -g -O0 multTest.c -o multTest
$ ./multTest
884901888
$ 
```

$2^{32}$ $= 4,294,967,296$

$200 \times 300 \times 400 \times 500 = 12,000,000,000$

**problem?**

Turns out it is worse -- ints are signed, meaning that the largest positive number is

$\left(\frac{2^{32}}{2}\right) - 1 = 2^{31} - 1 = 2,147,483,647$
Computers use a limited number of bits for numbers.

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

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n",answer);
    return 0;
}
```

The good news: all of the following produce the same (wrong) answer:

```
(500 * 400) * (300 * 200)
((500 * 400) * 300) * 200
((200 * 500) * 300) * 400
400 * (200 * (300 * 500))
```

```
$ gcc -g -O0 multTest.c -o multTest
$ ./multTest
-884901888
$
```
Let's look at a different program

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

int main() {
    float a = 3.14;
    float b = 1e20;

    printf("(3.14 + 1e20) - 1e20 = %f\n", (a + b) - b);
    printf("3.14 + (1e20 - 1e20) = %f\n", a + (b - b));

    return 0;
}
```

$ gcc -g -Og -std=gnu99 floatMultTest.c -o floatMultTest

$ ./floatMultTest.c

(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000

bigger problem!
Let's look at a different program

$ gcc -g -O0 multTest.c -o multTest
$ ./multTest
-884901888
$

Both C and C++ have specific representations of numbers that allow for these kinds of bugs.

$ gcc -g -Og -std=gnu99 floatMultTest.c -o floatMultTest
$ ./floatMultTest.c
(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000
$

It turns out that Java, which was built from the ground up with security in mind has somewhat different representations.
Information Storage
In C, everything can be thought of as a block of 8 bits
In C, everything can be thought of as a block of 8 bits called a "byte"
We will discuss manipulating bytes on a bit-by-bit level, but we won't be able to consider an individual bit on its own.

In a computer, the memory system is simply a large array of bytes (sound familiar, from CS106B?)

<table>
<thead>
<tr>
<th>values (ints):</th>
<th>7</th>
<th>2</th>
<th>8</th>
<th>3</th>
<th>14</th>
<th>99</th>
<th>-6</th>
<th>3</th>
<th>45</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>address:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Each address (a pointer!) represents the next byte in memory.

E.g., address 0 is a byte, then address 1 is the next full byte, etc.

Again: you can't address a bit. You must address at the byte level.
Because a byte is made up of 8 bits, we can represent the range of a byte as follows:

00000000 to 1111111

This range is 0 to 255 in decimal.

But, neither binary nor decimal is particularly convenient to write out bytes (binary is too long, and decimal isn't numerically friendly for byte representation)

So, we use "hexadecimal," (base 16).
Hexadecimal has 16 digits, so we augment our normal 0-9 digits with six more digits: A, B, C, D, E, and F.

Figure 2.2 in the textbook shows the hex digits and their binary and decimal values:

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Binary value</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Binary value</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>
Hexadecimal

- In C, we write a hexadecimal with a starting `0x`. So, you will see numbers such as `0xfa1d37b`, which means that it is a hex number.
- You should memorize the binary representations for each hex digit. One trick is to memorize A (1010), C (1100), and F (1111), and the others are easy to figure out.
- Let's practice some hex to binary and binary to hex conversions:

Convert: `0x173A4C` to binary.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>A</th>
<th>4</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0001</td>
<td>0111</td>
<td>0011</td>
<td>1010</td>
<td>0100</td>
<td>1100</td>
</tr>
</tbody>
</table>

`0x173A4C` is binary `0b000101110011101001001100`
Convert: 0b1111001010110110110011 to hexadecimal.

<table>
<thead>
<tr>
<th>Binary</th>
<th>11</th>
<th>1100</th>
<th>1010</th>
<th>1101</th>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>

(Start from the right)

0b1111001010110110110011 is hexadecimal 3CADB3
Convert: $0b1111001010110110110011$ to hexadecimal.

<table>
<thead>
<tr>
<th>Binary</th>
<th>11</th>
<th>1100</th>
<th>1010</th>
<th>1101</th>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>

(start from the right)

$0b1111001010110110110011$ is hexadecimal $3CADV3$
Convert: \texttt{0b1111001010110110110011} to hexadecimal.

\begin{center}
\begin{tabular}{cccccc}
\textbf{Binary} & 11 & 1100 & 1010 & 1101 & 1011 & 0011 \\
\textbf{Hexadecimal} & 3 & C & A & D & B & 3 \\
\end{tabular}
\end{center}

\texttt{0b1111001010110110110011} is hexadecimal \texttt{3CADB3}

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\textbf{Hex digit} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\textbf{Decimal value} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\textbf{Binary value} & 0000 & 0001 & 0010 & 0011 & 0100 & 0101 & 0110 & 0111 \\
\hline
\textbf{Hex digit} & 8 & 9 & A & B & C & D & E & F \\
\hline
\textbf{Decimal value} & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\textbf{Binary value} & 1000 & 1001 & 1010 & 1011 & 1100 & 1101 & 1110 & 1111 \\
\hline
\end{tabular}
Convert: \( 0b1111001010110110110011 \) to hexadecimal.

\[
\begin{array}{ccccccc}
\text{Binary} & 11 & 1100 & 1010 & 1101 & 1011 & 0011 \\
\text{Hexadecimal} & 3 & C & A & D & B & 3 \\
\end{array}
\]

(Start from the right)

\( 0b1111001010110110110011 \)
is hexadecimal \( 3CABD3 \)
Hexadecimal

Convert: 0b1111001010110110110011 to hexadecimal.

<table>
<thead>
<tr>
<th>Binary</th>
<th>11</th>
<th>1100</th>
<th>1010</th>
<th>1101</th>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>

(Start from the right)

0b1111001010110110110011 is hexadecimal 3CADB3
Hexadecimal

Convert: \(0b1111001010110110110011\) to hexadecimal.

\[
\begin{array}{cccccccc}
\text{Binary} & 11 & 1100 & 1010 & 1101 & 1011 & 0011 \\
\text{Hexadecimal} & 3 & C & A & D & B & 3
\end{array}
\]

(start from the right)

\(0b1111001010110110110011\) is hexadecimal \(3C\text{CADB}3\)
Convert: \(0b1111001010110110110011\) to hexadecimal.

<table>
<thead>
<tr>
<th>Binary</th>
<th>11</th>
<th>1100</th>
<th>1010</th>
<th>1101</th>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>

(start from the right)

\(0b1111001010110110110011\) is hexadecimal 3CADB3
Decimal to Hexadecimal

To convert from decimal to hexadecimal, you need to repeatedly divide the number in question by 16, and the remainders make up the digits of the hex number:

314156 decimal:

314,156 / 16 = 19,634 with 12 remainder: C
19,634 / 16 = 1,227 with 2 remainder: 2
1,227 / 16 = 76 with 11 remainder: B
76 / 16 = 4 with 12 remainder: C
4 / 16 = 0 with 4 remainder: 4

Reading from bottom up: 0x4CB2C
Hexadecimal to Decimal

To convert from hexadecimal to decimal, multiply each of the hexadecimal digits by the appropriate power of 16:

0x7AF:

$$7 \times 16^2 + 10 \times 16 + 15$$

$$= 7 \times 256 + 160 + 15$$

$$= 1792 + 160 + 15 = 1967$$
Let the computer do it!

Honestly, hex to decimal and vice versa are easy to let the computer handle. You can either use a search engine (Google does this automatically), or you can use a python one-liner:

```bash
$cgregg@myth10:~$ python -c "print(hex(314156))"
0x4cb2c
$cgregg@myth10:~$ python -c "print(0x7af)"
1967
```
Let the computer do it!

You can also use Python to convert to and from binary:

```
cgregg@myth10:~$ python -c "print(bin(0x173A4C))"
0b101110011101001001100

cgregg@myth10:~$ python -c "print(hex(0b1110010110110110011))"
0x3cadb3
cgregg@myth10:~$ 
```

(but you should memorize this as it is easy and you will use it frequently)
References and Advanced Reading

• References:
  • Tiobe Programming Index: https://www.tiobe.com/tiobe-index/
  • The C Language: https://en.wikipedia.org/wiki/C_(programming_language)
  • Kernighan and Ritchie (K&R) C: https://www.youtube.com/watch?v=de2Hsvxf8M
  • C Standard Library: http://www.cplusplus.com/reference/clibrary/

• Advanced Reading:
  • After All These Years, the World is Still Powered by C Programming
  • Is C Still Relevant in the 21st Century?
  • Why Every Programmer Should Learn C