Lexical Rules and Data Types

CSC230: C and Software Tools
N.C. State Department of Computer Science
Contents

• Lexical Scanning
• Comments
• Identifiers and Keywords
• C Variables
• Data Types
• Fundamental C Types
• Constants
Compiling Step #1: *Lexical Scanning*

- Divides the program into *tokens*, which are the smallest meaningful units of a program.
- Tokens in C are:
  - identifiers (e.g., `num_records`, `cust_name`)
  - keywords (e.g., `while`, `if`, `char`)
  - constants/strings (e.g., `3.1415`, “Answer: ”)
  - operators (e.g., `+`, `^`, `=`)
  - explicit separators (e.g., `,`, `;`)
Scanning... (cont’d)

- **White space** (space, tabs/indentation, newlines, comments) are **ignored**, except as explicit separators
Scanning (cont’d)

• Not so easy: what are the tokens in $d=-c++a$;

\[
\begin{align*}
  d &= -c2 + ++a ; \\
  d &= c2++ + a ; \\
  d &= - c2++ + a ; \\
  d &= - c2 + ++a ;
\end{align*}
\]

This is not a precedence issue

- we don’t know or care what the precedence of $=, ==, ++, \text{ and } +$ is at this point
“Max Munch”

- Scan from left to right, always grabbing the largest token possible.

Example (again):

1. \( d = -c2+++a; \) ("d=" not a token)
2. \( d = -c2+++a; \) ("=" not a token)
3. \( d = -c2+++a; \) ("-c" not a token)
4. \( d = -c2 +++a; \) ("c2+" not a token)
5. \( d = -c2 ++ +a; \) ("+++" not a token)
6. \( d = -c2 ++ +a; \) ("+a" not a token)
7. \( d = -c2 ++ +a ; \) ("a;" not a token)
Scanning... (cont’d)

• How many tokens, and what are they?

\[ j = +k^2 + 3; \]
Comments About Comments

• Block Style:

```c
a = c - b;   /* b must be gt 0 */
d = a * 3;
```

Great for commenting out whole sections of code, but **look out** if the code already has comments!

```c
/* Comment out the next two lines
a = c - b;   /* b must be gt 0 */
d = a * 3;
*/
```

% common source of bugs
% attempt to nest comments
Comments (cont’d)

• To-end-of-line comments are allowed in C99:

```c
r = 6 * x;  // compute radius
d = 2 * r;  // now diameter
```
Identifiers (Names, Labels)

- Consist of letters, ‘_’, and digits
  - cannot start with a digit (2_B_or_not_2_B)
- Case sensitive!
  - myVar is not the same as myvar
- Unlimited length (advice: stop at 32)

- gnome_memmgt_insert_into_heap_I_modified_this_because_I_can
Reserved Keywords
• (do not use as identifiers)
• C89:
  - auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while
• C99 adds a few more:
  - __Bool, __Complex, __Imaginary, inline, restrict
C Variables!

• A **variable** = a location in memory + its **interpretation**

• Interpretation of a variable is based on its
  1. *storage class* and
  2. *data type*

• *(We will discuss storage classes later...)*
  - *lifetime* of the variable
  - *how variable is (or can be)* **initialized**
  - *scope (visibility)* of the variable
Data Types

• The data type of a variable defines its interpretation

• Ex: suppose a 32-bit binary value stored in memory is 01000001010000100100001101000100
  – if type float, interpreted to be numerical value 781.03521728515625
  – if type unsigned int, interpreted to be numerical value 1145258561
  – if type char, interpreted to be the ASCII string value ABCD
Static or Dynamic Types

• In C (and Java), variables are statically typed
  – type must be declared when variable is created, and cannot change thereafter

• Languages with dynamic typing (e.g., PHP, Python, Perl, Ruby, Javascript, ...) are more flexible
Fundamental C Types

• (also called built-in, primitive, basic types)
• There are really only 2!
  – integer (includes characters)
  – floating point, or limited precision real number
Derived C Types

• These are composed from the fundamental types
  – arrays
  – functions
  – pointers
  – structs
  – unions
  – these will all be discussed later...

• Enumerated types: we’ll discuss later...

• Complex numbers type: we won’t use this semester
Specializations of Fundamental Types

• Integers can be...
  – signed or unsigned (signed by default)
  – really short (char), short, regular (int by default), long, really long (long long)

• Floating point (always signed) can be...
  – regular precision (float)
  – double precision (double)
  – extended precision (long double)
• The data type of a variable defines its usual meaning, but the programmer may interpret it differently

• Ex.: a `char` can represent...
  – an ASCII-encoded character (most common case)
  – an 8-bit integer
  – eight 1-bit flags
  – …
Min and Max Integer Values

• The lengths (in bits) (and the max and min values) of these types are platform dependent

• Common Platform (/usr/include/limits.h):

<table>
<thead>
<tr>
<th>Type</th>
<th># bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min ‘unsigned anything’</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>Min ‘signed char’</td>
<td>8</td>
<td>-128</td>
</tr>
<tr>
<td>Max ‘signed char’</td>
<td>8</td>
<td>127</td>
</tr>
<tr>
<td>Max ‘unsigned char’</td>
<td>8</td>
<td>255</td>
</tr>
<tr>
<td>Min ‘signed short’</td>
<td>16</td>
<td>-32,768</td>
</tr>
<tr>
<td>Max ‘signed short’</td>
<td>16</td>
<td>32,767</td>
</tr>
<tr>
<td>Max ‘unsigned short’</td>
<td>16</td>
<td>65,535</td>
</tr>
</tbody>
</table>
Integer Values... (cont’d)

<table>
<thead>
<tr>
<th>Type</th>
<th># bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min ‘signed int’</td>
<td>32</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>Max ‘signed int’</td>
<td>32</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>Max ‘unsigned int’</td>
<td>32</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>Min/Max ‘signed long’</td>
<td>64</td>
<td>9,223,372,036,854,775,808 -9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Max ‘unsigned long’</td>
<td>64</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Min ‘signed long long’</td>
<td>64</td>
<td>Same as long</td>
</tr>
<tr>
<td>Max ‘signed long long’</td>
<td>64</td>
<td>Same as long</td>
</tr>
<tr>
<td>Max ‘unsigned long long’</td>
<td>64</td>
<td>Same as long</td>
</tr>
</tbody>
</table>

- Which is big enough to store the **daily** federal deficit?
Floating Point (Real Numbers)

- Warning! Platform dependent! Lots of \texttt{gcc} options!

- Terminology

  \[ +.793 \times 2^{-36} \]

  - sign of the mantissa
  - magnitude of the mantissa
  - base of the exponent
  - exponent

Size of the \texttt{exponent} (# bits) mainly determines the \texttt{range} of numbers that can be represented.

Size of the \texttt{mantissa} (# bits) mainly determines the \texttt{precision} of numbers that can be represented.
Floating Point (Real Numbers)

• **IEEE floating point standard single precision:**
  - 1-bit sign
  - 23-bit (+ 1 implied bit) mantissa
  - 8-bit biased exponent (base 2)
  - 6 decimal digits precision

• **double precision:**
  - 1-bit sign
  - 52+1 bit mantissa
  - 11-bit biased exponent (base 2)
  - 15 decimal digits precision
Floating Point (cont’d)

• **Min** (normalized) positive values (approximate)
  – single precision (**float**): $2^{-126} \approx 10^{-38}$
  – double precision (**double**): $2^{-1022} \approx 10^{-308}$
  – Q: small enough to measure the diameter of an atom, in meters?

• **Max** (normalized) positive values (approximate)
  – single precision (**float**): $2^{127} \approx 10^{38}$
  – double precision (**double**): $2^{1023} \approx 10^{308}$
  – Q: big enough to count the number of atoms in the universe? Distance to the edge of the observable universe, in units of atom diameters?
Floating Point (cont’d)

• `long double = 128 bits`
  – more bits precision than `double`, same range
Reminder: Arithmetic Problems

• Types make a difference in computer arithmetic
  – signed vs. unsigned max and min values (integer)
  – overflow (integer and floating point)
  – underflow and limited precision (floating point)

• More info about floating point:
  see CSC236 or CSC302
What does this do?

```c
int main()
{
    char i;
    for (i=0; i<200; i++) {
        printf("%d\n",i);
    }
}
```
Why?

```c
int main()
{
    char i;
    for (i=0; i<200; i++) {
        printf("%4d %s\n", i, byte_to_binary(i));
    }
}
```
int main()
{
    char i;
    for (i=0; i<200; i++) {
        printf("%4d %s\n", i, byte_to_binary(i));
    }
}
int main()
{
    int i;
    for (i=0; i<200; i++) {
        printf("%4d %s\n", i, byte_to_binary(i));
    }
}

What data type can go here?
Constants with ‘const’

• Don’t want a value to change? Throw a `const` on there.

```cpp
const int BUFFER_SIZE = 1024;
const double PI = 3.141592653589793238;
const char delimiter = ‘,’;
```

• Character constants in single quotes: 'a', 'b'
  - value stored is the numeric value of the character in ASCII
Constants with `#define`

`#define <CONSTANT_NAME> <value>`

- Means “literally replace `CONSTANT_NAME` with `value` every time you see it in my file”.
- Can be very dumb. What does this program do?

```c
#define SLOPE -2
#define Y_INTERCEPT 1

int main()
{
    float x = 1;
    // find the y coordinate of this line
    float y = x * SLOPE + Y_INTERCEPT;
    printf("Coords: (%f,%f)\n",x,y);
}
```

Correct answer: Coords: (1.000000,-1.000000)

Actual output: Coords: (1.000000,0.000000)

Missing `*` operator.
const vs. #define

• The ‘const’ keyword does other stuff we’ll learn later when it comes to arrays/pointers.

• Things can get complicated when it comes to using ‘const’ to declare constants between files; #define doesn’t have these issues.

• Result: Just use #define.
• ASCII is a specific 8-bit encoding of Western characters (punctuation, digits, upper and lower case characters)

• Only the first 128 values (decimal 0-127, octal 000-177) are standardized

• The interpretation of the remaining 128 values (decimal 128-255, octal 200-377) are not standardized, i.e., they are application/platform-specific
# Standardized ASCII (0-127)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Octal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>NUL (null)</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
<td>SOH (start of heading)</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>002</td>
<td>STX (start of text)</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>003</td>
<td>ETX (end of text)</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>004</td>
<td>EOT (end of transmission)</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>005</td>
<td>ENQ (enquiry)</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>006</td>
<td>ACK (acknowledge)</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>007</td>
<td>BEL (bell)</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>010</td>
<td>BS (backspace)</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>011</td>
<td>TAB (horizontal tab)</td>
</tr>
<tr>
<td>A</td>
<td>012</td>
<td>012</td>
<td>LF (NL line feed, new line)</td>
</tr>
<tr>
<td>B</td>
<td>013</td>
<td>013</td>
<td>VT (vertical tab)</td>
</tr>
<tr>
<td>C</td>
<td>014</td>
<td>014</td>
<td>FF (NP form feed, new page)</td>
</tr>
<tr>
<td>D</td>
<td>015</td>
<td>015</td>
<td>CR (carriage return)</td>
</tr>
<tr>
<td>E</td>
<td>016</td>
<td>016</td>
<td>SO (shift out)</td>
</tr>
<tr>
<td>F</td>
<td>017</td>
<td>017</td>
<td>SI (shift in)</td>
</tr>
<tr>
<td>10</td>
<td>020</td>
<td>020</td>
<td>DLE (data link escape)</td>
</tr>
<tr>
<td>11</td>
<td>021</td>
<td>021</td>
<td>DC1 (device control 1)</td>
</tr>
<tr>
<td>12</td>
<td>022</td>
<td>022</td>
<td>DC2 (device control 2)</td>
</tr>
<tr>
<td>13</td>
<td>023</td>
<td>023</td>
<td>DC3 (device control 3)</td>
</tr>
<tr>
<td>14</td>
<td>024</td>
<td>024</td>
<td>DC4 (device control 4)</td>
</tr>
<tr>
<td>15</td>
<td>025</td>
<td>025</td>
<td>NAK (negative acknowledge)</td>
</tr>
<tr>
<td>16</td>
<td>026</td>
<td>026</td>
<td>SYN (synchronous idle)</td>
</tr>
<tr>
<td>17</td>
<td>027</td>
<td>027</td>
<td>ETB (end of trans. block)</td>
</tr>
<tr>
<td>18</td>
<td>030</td>
<td>030</td>
<td>CAN (cancel)</td>
</tr>
<tr>
<td>19</td>
<td>031</td>
<td>031</td>
<td>EM (end of medium)</td>
</tr>
<tr>
<td>20</td>
<td>032</td>
<td>032</td>
<td>SUB (substitute)</td>
</tr>
<tr>
<td>21</td>
<td>033</td>
<td>033</td>
<td>ESC (escape)</td>
</tr>
<tr>
<td>22</td>
<td>034</td>
<td>034</td>
<td>FS (file separator)</td>
</tr>
<tr>
<td>23</td>
<td>035</td>
<td>035</td>
<td>GS (group separator)</td>
</tr>
<tr>
<td>24</td>
<td>036</td>
<td>036</td>
<td>RS (record separator)</td>
</tr>
<tr>
<td>25</td>
<td>037</td>
<td>037</td>
<td>US (unit separator)</td>
</tr>
</tbody>
</table>

Source: www.LookupTables.com
### One Interpretation of 128-255

| 128 | Č | 144 | É | 161 | í | 177 | 193 | 209 | 225 | 241 | ± |
| 129 | ü | 145 | æ | 162 | ó | 178 | 194 | 210 | 226 | 242 | ≥ |
| 130 | é | 146 | AÈ | 163 | ú | 179 | 195 | 211 | 227 | 243 | ≤ |
| 131 | à | 147 | õ | 164 | ñ | 180 | 196 | 212 | 228 | Σ | 244 |
| 132 | ä | 148 | ö | 165 | Ñ | 181 | 197 | 213 | 229 | 245 | × |
| 133 | à | 149 | ô | 166 | ◌ | 182 | 198 | 214 | 230 | 246 | □ |
| 134 | å | 150 | ü | 167 | ° | 183 | 199 | 215 | 231 | 247 | △ |
| 135 | ç | 151 | û | 168 | Ç | 184 | 200 | 216 | 232 | Φ | 248 |
| 136 | ø | 152 | – | 169 | – | 185 | 201 | 217 | 233 | ⊗ | 249 |
| 137 | ë | 153 | Ő | 170 | — | 186 | 202 | 218 | 234 | Ω | 250 |
| 138 | è | 154 | Ü | 171 | ° | 187 | 203 | 219 | 235 | δ | 251 |
| 139 | ı | 156 | £ | 172 | ¼ | 188 | 204 | 220 | 236 | ∞ | 252 |
| 140 | ï | 157 | ¥ | 173 | î | 189 | 205 | 221 | 237 | ϕ | 253 |
| 141 | ï | 158 | – | 174 | « | 190 | 206 | 222 | 238 | € | 254 |
| 142 | Ä | 159 | Ъ | 175 | » | 191 | 207 | 223 | 239 | 255 |
| 143 | Å | 160 | á | 176 | ñ | 192 | 208 | 224 | 240 | ≏ |

Source: [www.LookupTables.com](http://www.LookupTables.com)
(This allowed totally sweet ASCII art in the 90s)

Sources:
Useful Character Constant

Escape Sequences

- \0  Null character
- \`  Single quote
- \“  Double quote
- \\  Backslash
- \n  Newline
- \t  Horizontal tab
- \nnn  Octal value of character (ex: ‘a’ == ‘\141’)
- \xnn  Hexadecimal value of character (== ‘\x61’)

CSC230: C and Software Tools © NC State Computer Science Faculty
Converting ASCII digits to Integers

• You can read ASCII characters and do arithmetic on them, but results **not** what you expect!

• Program: read a number, print it out

```c
int c;
c = getchar();  // read one ascii character
printf("%d\n", c); // interpret c as an integer
  // and print as ASCII
  // (decimal) string
```

Result

- user types: 1
- program prints: 49  Why??

*common source of bugs* difference between ASCII-encoded strings and numbers
Converting ASCII to Numbers

• Converting ASCII-encoded digit to an integer, the right way:

```c
unsigned char c;
c = (unsigned char) getchar();
unsigned int n;
n = c - '0';
printf("%d\n", n);
```

Converting integer to ASCII:

```c
c = (char) (n + '0');
```

How would we convert an ASCII string ("12") to an integer, and vice versa???
(“Wide” Characters)

• For encoding character sets other than ASCII
• Type: `wchar_t`
• Ex. of specifying a wide character constant: `L’å’`
• *We’ll look at support for this later*
String Literals

• Strings are **arrays** of characters
  – terminated (automatically, by the compiler) with **NULL**
  – *we’ll discuss more later…*

• Specifying a string: "abcdefg"
  – cannot contain double quote or span multiple lines (use \" or \n if quote or newline should be in the string)
  – strings of wide characters: L"å∫ç∂f"

• Warning: "a" is not the same as ‘a’!
Multi-line string literals

- Just put quoted string literals one after another; they get glued together automatically.

```c
int main()
{
    printf("Usage:\n" " coolapp [options] <filename>\n" "\n" "Copyright 2014 Tyler Bletsch\n");
}
```
Review: Binary

- Advice: memorize the following (need for 236 anyway...)
  - $2^0 = 1$
  - $2^1 = 2$
  - $2^2 = 4$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - $2^9 = 512$
  - $2^{10} = 1024$
### Review: Decimal to Binary

<table>
<thead>
<tr>
<th>?</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>$457 \div 2 =$</td>
<td>228</td>
<td>1</td>
</tr>
<tr>
<td>$228 \div 2 =$</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>$114 \div 2 =$</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>$57 \div 2 =$</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>$28 \div 2 =$</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>$14 \div 2 =$</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>$7 \div 2 =$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$3 \div 2 =$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$1 \div 2 =$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The binary equivalent of 457 is $111001001_2$. 

- **Decimal to Binary:**
  - Divide the decimal number by 2.
  - Write down the remainder (0 or 1).
  - Repeat the process with the quotient until it becomes 0.
  - The binary representation is the sequence of remainders read in reverse order.
Practice: binary to/from hex

- \(0101101100100011_2 \rightarrow \)
- \(0101\ 1011\ 0010\ 0011_2 \rightarrow \)
- \(5\ B\ 2\ 3_{16}\)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

- \(0001\ 1111\ 0100\ 1011_2 \rightarrow \)
- \(0001111101001011_2 \)
Integer Constants

• Specifying:
  <optional sign> <string of decimal digits>
  – ex: 7940, +7940, −36

• If prefixed by 0, interpreted as base 8 constant
  – only 0–7 allowed as digits

• If prefixed by 0x, interpreted as base 16 constant
  – 0–9, a–f allowed as digits

• Ex.: what’s decimal value of 03, 0x03, 3?
  of 53, 053, and 0x53?
Integer Constants (cont’d)

- If suffixed by `u`, type is `unsigned int`, and value must be positive
  - ex: `123u`
- If suffixed by `L`, type is `long int`
  - ex: `456L`
Floating Point Constants

• Specifying:
  \(<\text{optional sign}>\text{integerpart} \cdot \text{fractionpart}\)
  – either integer part or fractional part can be missing
  – all good: 22.22, +2., −.22
  – warning: 2 is integer constant, 2. is floating point

• Followed (optionally) by exponent (expressed in base 10)
  – specifying:
    \(\text{e}\ <\text{optional sign}>\ <\text{integer constant}>\)
  – ex.: 23.45e−67 means 23.45 * 10^{−67}
Floating Point... (cont’d)

• Default type is **double**
  – suffixed by `f`: force type to be **float**
  – suffixed by `L`: **long double** (extended precision)

• *More about floating point numbers, precision, and range, later...*
A dumb thing that C will let you do, but you shouldn’t do it

• The following is legal C code:
  
  unsigned x;

• What’s the data size?
  – Yeah, I don’t know either
  – Apparently it’s like an int?
  – Let’s just never do this

• Always put the type specifier:
  
  unsigned int x;
### tl;dr

<table>
<thead>
<tr>
<th>Integer Type</th>
<th>Size (on x86!)</th>
<th>Normal use</th>
<th>Signed range (on x86)</th>
<th>Unsigned range (on x86)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bit (1 byte)</td>
<td>ASCII char or small integer</td>
<td>-128..127</td>
<td>0..255</td>
</tr>
<tr>
<td>short</td>
<td>16 bit (2 byte)</td>
<td>Smallish integer</td>
<td>-32768..32767</td>
<td>0..65535</td>
</tr>
<tr>
<td>int</td>
<td>32 bit (4 byte)</td>
<td>Normal integer</td>
<td>-2147483648..2147483647</td>
<td>0..4294967295</td>
</tr>
<tr>
<td>long</td>
<td>64 bit (8 byte)</td>
<td>Big integer</td>
<td>-2^{63}+1 .. 2^{63}-1</td>
<td>0.. 2^{64}-1</td>
</tr>
<tr>
<td>long long</td>
<td>64 bit (8 byte)</td>
<td>Big integer</td>
<td>-9,223,372,036,854,775,808..9,223,372,036,854,775,807</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal Type</th>
<th>Size (on x86!)</th>
<th>Normal use</th>
<th>Decimal digits of precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bit (4 byte)</td>
<td>Lousy decimal</td>
<td>6</td>
</tr>
<tr>
<td>double</td>
<td>64 bit (8 byte)</td>
<td>Good decimal</td>
<td>15</td>
</tr>
</tbody>
</table>
Exercise 03a

ASCII table

• Write a program that prints ASCII characters 32-127.

• Steps to help you along:
  – Write a loop to print integers 32..127.
  – Write a printf statement that prints a single character.
  – Combine them.

We haven’t necessarily covered everything to do this. Ask questions!

Reminder: Go to course web page for link to exercise form. Paste code into ideone.com and submit the link.
Implied Types of Constants

- Default type for integer constants: shortest type compatible with value, starting with `signed int -> unsigned int -> ...`
- Default type for floating point constants: `double`
Base Conversions to/from Binary

...and be **able to do** the following

- \( 2 \times 8^2 + 5 \times 8^1 + 6 \times 8^0 \) == decimal 174 ==

- octal 256 ==

- binary 10 101 110 ==

- \( 2^7 + 2^5 + 2^3 + 2^2 + 2^1 \) ==

- 128 + 32 + 8 + 4 + 2 == decimal 174

...and likewise with hex
Review: binary to/from octal

- $00111000_2 \rightarrow$
- $00\ 111\ 000_2 \rightarrow$
- $0\ 7\ 0_8$

$356_8 \rightarrow$

$11\ 101\ 110_2 \rightarrow$

$11101110_2$

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>