Overview of C

- Basic Data Types
- Constants
- Variables
- Identifiers
- Keywords
- Basic I/O

**NOTE:** There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, form feeds and comments (collectively, “white space”) are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.
Basic Data Types

- **Integer Types**
  - Char – smallest addressable unit; each byte has its own address
  - Short – not used so much
  - Int – default type for an integer constant value
  - Long – do you really need it?

- **Floating point Types** – are “inexact”
  - Float – single precision (about 6 digits of precision)
  - Double – double precision (about 15 digits of precision)
    - constant default unless suffixed with ‘f’

Note that variables of type `char` are guaranteed to always be one byte.

There is no maximum size for a type, but the following relationships must hold:

- `sizeof(short) <= sizeof(int) <= sizeof(long)`
- `sizeof(float) <= sizeof(double) <= sizeof(long double)`
Beside the basic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- **arrays** of objects of a given type;
- **functions** returning objects of a given type;
- **pointers** to objects of a given type;
- **structures** containing a sequence of objects of various types;
- **unions** capable of containing any of one of several objects of various types.

In general these methods of constructing objects can be applied recursively

- An array of pointers
- An array of characters (i.e. a string)
- Structures that contain pointers
Constants

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>'A', 'B'</td>
</tr>
<tr>
<td>int</td>
<td>123, -1, 2147483647, 040 (octal), 0xab (hexadecimal)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>123u, 2107433648, 040U (octal), 0X02 (hexadecimal)</td>
</tr>
<tr>
<td>long</td>
<td>123L, 0x1FFFl (hexadecimal)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>123ul, 0777UL (octal)</td>
</tr>
<tr>
<td>float</td>
<td>1.23F, 3.14e+0f</td>
</tr>
<tr>
<td>double</td>
<td>1.23, 2.718281828</td>
</tr>
<tr>
<td>long double</td>
<td>1.23L, 9.99E-9L</td>
</tr>
</tbody>
</table>

Special characters

- Not convenient to type on a keyboard
- Use single quotes i.e. `\n`
- Looks like two characters but is really only one

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alert (bell) character</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\f</td>
<td>formfeed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>\000</td>
<td>octal number</td>
</tr>
<tr>
<td>\xhh</td>
<td>hexadecimal number</td>
</tr>
</tbody>
</table>


Symbolic constants

- A name that substitutes for a value that cannot be changed
- Can be used to define a:
  - Constant
  - Statement
  - Mathematical expression
- Uses a preprocessor directive
  - `#define <name> <value>`
    - No semi-colon
  - Coding style is to use all capital letters for the name
- Can be used any place you would use the actual value
- All occurrences are replaced when the program is compiled
- Examples (see link below):
  - The use of EXIT_SUCCESS in hello.c code (see stdlib.h)
  - `#define PI 3.141593`
  - `#define TRUE 1`
  - `#define floatingpointnum float`

http://www.cprogrammingexpert.com/C/Tutorial/fundamentals/symbolic_constant_c_programming_language.aspx
Variable Declarations

- **Purpose**: define a variable before it is used.
- **Format**: type identifier [, identifier] ;
- **Initial value**: can be assigned
  - int i, j, k;
  - char a, b, c = 'D';
  - int i = 123;
  - float f = 3.1415926535;
  - double f = 3.1415926535;
  - strings later... array of characters

**Type conversion**: *aka*, type casting
- Coercion, be very cautious.
- (type) identifier;
  - int i = 65; /* what if 258 */
  - char a; /* range -128 to 127 */
  - a = (char) i; /* What is the value of a? */

**INSTANCE VARIABLE**
In object-oriented programming with classes, an *instance variable* is a variable defined in a class (i.e. a member variable), for which each object of the class has a separate copy. They live in memory for the life of the class.
Variable vs Identifier

- An identifier, also called a token or symbol, is a lexical token that “names” an entity
  - An entity can be: variables, types, labels, functions, packages, etc.
  - Naming entities makes it possible to refer to them

- A variable
  - Allows access and information about what is in memory i.e. a storage location
  - A symbolic name (an identifier) that is associated with a value and whose associated value may be changed
  - The usual way to reference a stored value
Identifier Naming Style

- **Rules for identifiers**
  - a-z, A-Z, 0-9, and _
  - Case sensitive
  - The first character must be a letter or _
  - Keywords are reserved words, and may not be used as identifiers

- **Identifier Naming Style**
  - Separate words with ‘_’ or capitalize the first character
  - Use all UPPERCASE for symbolic constant, macro definitions, etc
  - Be consistent
  - Be meaningful

- **Sample Identifiers**
  - i0, j1, abc, stu_score, __st__, data_t, MAXOF, MINOF ...
Keywords

<table>
<thead>
<tr>
<th>auto</th>
<th>double</th>
<th>int</th>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>switch</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
</tbody>
</table>

Purpose: reserves a word or identifier to have a particular meaning

- The meaning of keywords — and, indeed, the meaning of the notion of *keyword* — differs widely from language to language.
- You shouldn't use them for any other purpose in a C program. They are allowed, of course, within double quotation marks.
Arithmetic operators

- **Mathematical Symbols**
  - + - * / %
  - Addition, subtraction, multiplication, division, modulus

- Works for both int and float
  - + - * /
  - / operator performs integer division if both operands are integer
    - i.e. truncates; otherwise, float
  - % operator divides two integer operands with an integer result of the remainder

- Precedence – left to right
  - () always first
  - * / %
  - + -
Binary operators

<< left-shift
>> right-shift
& bitwise AND
| bitwise OR
^ bitwise exclusive-OR
&& logical AND
|| logical OR
< less than
> greater than
<= less than or equal
>= greater than or equal
== equals
!= does not equal

Note: C has no keywords as true or false. They are non-zero or zero values, usually 1 or 0.

`OR`, `AND`, and `NOT`: differences between bitwise and logical operators.

BITWISE OPERATORS:
0 is FALSE
1 is TRUE
Result: 0 is false; everything else is true

LOGICAL OPERATORS:
0 is FALSE
Everything else is TRUE
United operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>positive 1</td>
</tr>
<tr>
<td>-1</td>
<td>negative 1</td>
</tr>
<tr>
<td>~i</td>
<td>one's complement (bitwise complement)</td>
</tr>
<tr>
<td>!i</td>
<td>logical negation (i.e., 1 if i is zero, 0 otherwise)</td>
</tr>
<tr>
<td>++i</td>
<td>adds one to i, and returns the new value of i</td>
</tr>
<tr>
<td>--i</td>
<td>subtracts one from i, and returns the new value of i</td>
</tr>
<tr>
<td>i++</td>
<td>adds one to i, and returns the old value of i</td>
</tr>
<tr>
<td>i--</td>
<td>subtracts one from i, and returns the old value of i</td>
</tr>
</tbody>
</table>

**EXAMPLES**

\[
i = 20 = 0b0000\ldots10100
\]
\[
\sim i = 1111\ldots01011 \quad \text{!}i = 0
\]
\[
++i = 21, \ i = 21 \quad \text{--}i = 19, \ i = 19
\]
\[
i++ = 20, \ i = 21 \quad \text{i--} = 20, \ i = 19
\]
## Assignment operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>assignment</td>
<td>i += 1 → i = i + 1</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>addition assignment</td>
<td>i &amp;= 1 → i = i &amp; 1</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>subtraction assignment</td>
<td>i &lt;&lt;= 1 → i = i &lt;&lt; 1</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>multiplication assignment</td>
<td></td>
</tr>
<tr>
<td><code>/=</code></td>
<td>division assignment</td>
<td></td>
</tr>
<tr>
<td><code>%=</code></td>
<td>remainder/modulus assignment</td>
<td></td>
</tr>
<tr>
<td><code>&amp;=</code></td>
<td>bitwise AND assignment</td>
<td></td>
</tr>
<tr>
<td>`</td>
<td>=`</td>
<td>bitwise OR assignment</td>
</tr>
<tr>
<td><code>^=</code></td>
<td>bitwise exclusive OR assignment</td>
<td></td>
</tr>
<tr>
<td><code>&lt;&lt;=</code></td>
<td>left shift assignment</td>
<td></td>
</tr>
<tr>
<td><code>&gt;&gt;=</code></td>
<td>right shift assignment</td>
<td></td>
</tr>
</tbody>
</table>

Assignment operators take the value on the right and place it into the variable on the left. C provides more derivatives by combining computation and assignment together. Note the difference between assignment and equal operators. **Don't make the mistake of using '==' when you meant '!='!**
Operator Precedence

https://www.cs50.net/resources/cppreference.com/operator_precedence.html

See handout

Do in-class exercises
### Arithmetic type issues

- **Type casting - EXPLICIT**
  - A way to convert a variable from one data type to another data type
  - Uses a cast operator \(\rightarrow\) (type-name)
  - (type-name) expression

- **Type combination and promotion - IMPLICIT**
  - \((‘a’ - 32) = 97 - 32 = 65 = ‘A’\)
  - Smaller type (char) is “promoted” to be the same size as the larger type (int)
  - Determined at compile time - based purely on the *types* of the values in the expressions
  - Does not lose information – convert from type to compatible large type
The **usual arithmetic conversions** are implicitly performed to cast their values in a common type. Compiler first performs *integer promotion*, if operands still have different types then they are converted to the type that appears highest in the following hierarchy.

```
<table>
<thead>
<tr>
<th>long</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>unsigned long long</td>
</tr>
<tr>
<td></td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
</tr>
<tr>
<td></td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>int</td>
</tr>
</tbody>
</table>
```
Usual Arithmetic Conversions ➔ Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions.

If either operand is long double, the other is converted to long double.
If either operand is double, the other is converted to double.
If either operand is float, the other is converted to float.
Otherwise, the integral promotions are performed on both operands;
If either operand is unsigned long int, the other is converted to unsigned long int.
If one operand is long int and the other is unsigned int, the effect depends on whether a long int can represent all values of an unsigned int; if so, the unsigned int operand is converted to long int; if not, both are converted to unsigned long int.
If one operand is long int, the other is converted to long int.
If either operand is unsigned int, the other is converted to unsigned int.
Otherwise, both operands have type int.

NOTE: There are two changes here. First, arithmetic on float operands may be done in single precision, rather than double; the first edition specified that all floating arithmetic was double precision. Second, shorter unsigned types, when combined with a larger signed type, do not propagate the unsigned property to the result type; in the first edition, the unsigned always dominated. The new rules are slightly more complicated, but reduce somewhat the surprises that may occur when an unsigned quantity meets signed. Unexpected results may still occur when an unsigned expression is compared to a signed expression of the same size.
Arithmetic expressions - Truncation

Pitfall -- int vs. float Arithmetic
Here's an example of the sort of code where int vs. float arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```c
{ int score;
  // suppose score gets set in the range 0..20 somehow
  7
  score = (score / 20) * 100; // NO -- score/20 truncates to 0
  ...
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number...

```c
score = ((double)score / 20) * 100; // OK -- floating point division from cast
score = (score / 20.0) * 100; // OK -- floating point division from 20.0
score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating
    // quotient back to 0
```