Lecture 3. More About C

- Programming languages have their lingo

- Programming *language*
  
  Types are ‘categories’ of values \( \text{int, float, char} \)
  
  Constants are values of basic types \( 0, 123.6, "Hello" \)
  
  Variables name locations that hold values \( i, \text{sum} \)
  
  Expressions compute values/change variables \( \text{sum = sum + i} \)
  
  Statements control a program’s *flow of control* \( \text{while, for, if-else} \)
  
  Functions encapsulate statements \( \text{main} \)
  
  Modules collections of related variables & functions a.k.a. ‘compilation units’

- Programming *environment*
  
  Text editor (emacs, vi, sam)
  
  Compiler (lcc, cc, gcc)
  
  Linker/loader (ld); used rarely, because lcc runs it
  
  Debugger (gdb)
Types

• A *type* determines

  a set of *values*, and

  what *operations* can be performed on those values

• *Scalar* types

  char   a ‘character’; typically a ‘byte’ — 8 bits
  int    a signed integer; typically values from –2147483648 to 2147483647
  unsigned an unsigned integer; typically values from 0 to 4294967295
  float  single-precision floating point
  double double-precision floating point

• *Pointer* types: *much* more later…

• *Aggregate* types: values that have *elements* or *fields*, e.g., arrays, structures
# Constants

## Constant values of the scalar types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>‘a’</td>
<td>character constant (use single quotes)</td>
</tr>
<tr>
<td></td>
<td>‘\035’</td>
<td>character code 35 octal, or base 8</td>
</tr>
<tr>
<td></td>
<td>‘\x29’</td>
<td>character code 29 hexadecimal, or base 16</td>
</tr>
<tr>
<td></td>
<td>‘\t’</td>
<td>tab (‘\011’, do ‘man ascii’ for details)</td>
</tr>
<tr>
<td></td>
<td>‘\n’</td>
<td>newline (‘\012’)</td>
</tr>
<tr>
<td></td>
<td>‘\‘</td>
<td>backslash</td>
</tr>
<tr>
<td></td>
<td>‘\’</td>
<td>single quote</td>
</tr>
<tr>
<td></td>
<td>‘\b’</td>
<td>backspace (‘\010’)</td>
</tr>
<tr>
<td></td>
<td>‘\0’</td>
<td>null character; i.e., the character with code 0</td>
</tr>
</tbody>
</table>

| int  | 156   | decimal (base 10) constant |
|      | 0234  | octal (base 8) |
|      | 0x9c  | hexadecimal (base 16) |

| unsigned | 156U | decimal |
|          | 0234U | octal |
|          | 0x9cU | hexadecimal |

| float   | 15.6F | |
|         | 1.56e1F | |

| double  | 15.6 | ‘plain’ floating point constants are doubles |
|         | 1.56E1L | |
Variables

- A variable is the name of a *location in memory* that can hold values

  ```
  int i, sum;
  float average;
  unsigned count;
  ```

  ```
  i = 8;
  sum = -456;
  count = 101U;
  average = 34.5;
  ```

- A variable has a **type**; it can hold only values of that type

- Assignments **change** the values of variables

  ```
  sum = sum + i;
  ```

  changes the value of `sum` to -448

- Variables must be **initialized** before they are used

  ```
  #include <stdio.h>

  int main(void) {
    int x;
    printf("x = %d\n", x);  // output is *undefined*!
    return 0;
  }
  ```
Expressions

- Expressions use the values of variables and constants to compute new values
- Binary arithmetic operators take two operands produce one result
  - addition, subtraction
  - multiplication, division
  - remainder (a.k.a. modulus)
- Type of result depends on type of operands
  ```
  int i; unsigned u; float f;
  ```

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>u</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int</td>
<td>unsigned</td>
<td>float</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>?</td>
<td>unsigned</td>
<td>float</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `i + i` specifies int addition and yields an int result
- int and unsigned division truncate: 7/2 is 3, but 7.0/2 is 3.5
- Unary operators take one operand and produce one result
  - negation, ‘affirmation’ (just returns its operand’s value)
Precedence and Associateivity

• Operator precedence and associativity dictate the order of expression evaluation

• **Precedence** dictates which subexpressions get evaluated first

  - highest unary − +
  - binary * / %

  −2*a + b is evaluated as if written as ((−2)*a) + b

• **Associativity** dictates the evaluation order for expressions with several operators of the same precedence

  - all arithmetic operators have *left-to-right* associativity

  a + b + c is evaluated as if written as ((a + b) + c)

• Use **parentheses** to force a specific order of evaluation

  −2*(a + b) computes −2

  a + b

  the product of these two values
Assignments

• Assignment expressions *store* values in variables

\[ \text{variable} = \text{expression} \]

the type of *expression* must be

the same as the type of *variable*

convertible to the type of *variable*

\[
\begin{array}{c|ccc}
= & i & u & f \\
\hline
i & \text{int} & \text{int} & \text{int} \\
u & \text{unsigned} & \text{unsigned} & \text{unsigned} \\
f & \text{float} & \text{float} & \text{float} \\
\end{array}
\]

• Augmented assignments combine a binary operator with assignment

\[ \text{variable} += \text{expression} \]
\[ \text{variable} -= \text{expression} \]

... 

\[ \text{sum} += \text{i} \quad \text{is the same as} \quad \text{sum} = \text{sum} + \text{i} \]
Increment/Decrement

- Prefix and postfix operators `++` and `--` increment and decrement operand by 1
  - `++n` adds 1 to `n`
  - `--n` subtracts 1 from `n`
- **Prefix** operator increments operand *before* returning the *new* value
  ```
  n = 5;
  x = ++n;
  x is 6, n is 6
  ```
- **Postfix** operator increments operand *after* returning the *old* value
  ```
  n = 5;
  x = n++;
  x is 5, n is 6
  ```
- Operands of `++` and `--` must be *variables*
  ```
  ++1
  2 + 3++
  are illegal
  ```
Idiomatic C

• `sum.c` (in `sum2.c`) rewritten using common idioms involving `+=` and `++`

```c
#include <stdio.h>

int main(void) {
    int i, n, sum = 0;
    printf("Enter n:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        sum += i;
    printf("Sum from 1 to %d = %d\n", n, sum);
    return 0;
}
```

• `scanf` is a form of assignment; it changes `n`
Statements

- Expression statements

  \[ \text{expression}_{\text{opt}} ; \quad \text{sum} += \text{i}; \]
  \[ \text{printf("Sum from 1 to %d = %d\n", n, sum);} \]

- Selection statements

  \[ \text{if (conditional ) statement} \]
  \[ \text{if (conditional ) statement else statement} \]
  \[ \text{if (x > max) max = x;} \]
  \[ \text{if (bit == 0) printf(" "); else printf("*");} \]
  \[ \text{switch (expression ) \{ case constant : statement... default : statement \} } \]

- Iteration statements (loops)

  \[ \text{while (conditional ) statement} \]
  \[ \text{while (i <= n) \{ sum += i; i++; \}} \]
  \[ \text{for (expression}_{\text{opt}} ; conditional_{\text{opt}} ; expression_{\text{opt}} ) \text{ statement} \]
  \[ \text{for (i = 1; i <= n; i++) sum += i;} \]
  \[ \text{for (;;) printf("Help! I’m looping\n"); } \]
  \[ \text{do statement while (expression) ; \}
  \[ \text{do \{ sum += i; ++i; \} while (i <= n);} \]


Statements, cont’d

• Compound statements

\{ \text{declaration}_{\text{opt}} \ldots \text{statement} \ldots \}\n
  \text{for } (j = 0; j < n; j = j + 1) \{ \\
  \text{int } \text{bit} = (\text{rand()} \gg 14) \% 2; \\
  \text{if } (\text{bit} == 0) \\
    \text{printf}(" "); \\
  \text{else} \\
    \text{printf}("*"); \\
  \}\n
• Others

  \text{return } \text{expression}_{\text{opt}}; \quad \text{return;}

  \text{return } 0; \quad \text{return } -2*(a + b);

  \text{break;}

  \text{continue;}

• Keywords (if else while do for switch case ...) \textit{cannot} be used as variables
Conditional Expressions

• A conditional expression is any expression that evaluates to zero or nonzero.

• There is no ‘Boolean’ type; nonzero is true, zero is false.

• Relational operators compare two arithmetic values (or pointers) and yield 0 or 1:
  - `<`    <= less than, less than or equal to
  - `==`  `!=` equal to, not equal to
  - `>`    `>=` greater than, greater than or equal to

• Logical connectives:
  - `conditional_1 && conditional_2` 1 if both conditionals are nonzero; 0 otherwise.
  - `conditional_1 || conditional_2` 1 if either conditional is nonzero; 0 otherwise.

  Conditionals are evaluated left-to-right only as far as is necessary:
  - `&&` stops when the outcome is known to be zero
  - `||` stops when the outcome is known to be nonzero.

• Associativity: left to right; precedence: below the arithmetic operators.

  highest arithmetic operators:
  - `<`    `<=`    `>=`    `>`
  - `==`  `!=`  `&&`

  lowest `||`

  `a + b < max || max == 0 && a == b` is interpreted as if written `(((a + b) < max) || (max == 0 && (a == b)))`.