Lecture 3. More About C

- **Programming languages have their lingo**
- **Programming *language***

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

- **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**

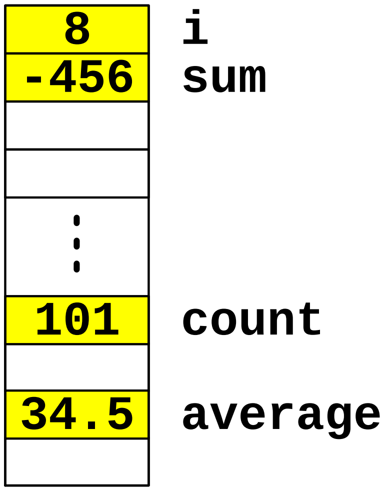
char	’a’	character constant (use single quotes)
	’\035’	character code 35 octal, or base 8
	’\x29’	character code 29 hexadecimal, or base 16
	’\t’	tab (’\011’, do ‘man ascii’ for details)
	’\n’	newline (’\012’)
	’\\’	backslash
	’\’’	single quote
	’\b’	backspace (’\010’)
	’\0’	null character; i.e., the character with code 0
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);
    return 0;
}
```

output is *undefined*!

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;
```

+	i	u	f
i	int	unsigned	float
u	?	unsigned	float
f	?	?	float

`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 Associativity

- Operator precedence and associativity dictate the ***order of expression evaluation***
- ***Precedence*** dictates which subexpressions get evaluated first

highest	unary `-` `+`
	binary `*` `/` `%`
lowest	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

  variable = *expression*

  the type of *expression* must be

  the same as the type of *variable*
  convertible to the type of *variable*

```
int i; unsigned u; float f;
```

=	i	u	f
i	int	int	int
u	unsigned	unsigned	unsigned
f	float	float	float

- Augmented assignments combine a binary operator with assignment

  variable += *expression*
  variable -= *expression*
  ...

  `sum += i` is the same as `sum = sum + i`

Increment/Decrement

- Prefix and postfix operators `++` `--` 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 ++**

```
/*
Compute the sum of the integers
from 1 to n, for a given n.
*/
#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

  expression$_{opt}$ `;`

```
sum += i;
printf("Sum from 1 to %d = %d\n", n, sum);
```

- Selection statements

  `if (` *conditional* `)` *statement*
  `if (` *conditional* `)` *statement* `else` *statement*

```
if (x > max) max = x;
if (bit == 0) printf(" "); else printf("*");
```

  `switch (` *expression* `) {` `case` *constant* `:` *statement*... `default` `:` *statement* `}`

- Iteration statements (loops)

  `while (` *conditional* `)` *statement*

```
while (i <= n) { sum += i; i++; }
```

  `for (` *expression*$_{opt}$ `;` *conditional*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

```
for (i = 1; i <= n; i++) sum += i;
for (;;) printf("Help! I’m looping\n");
```

  `do` *statement* `while (` *expression* `) ;`

```
do { sum += i; ++i; } while (i <= n);
```

Statements, cont'd

- **Compound statements**

  { *declaration*$_{\text{opt}}$… *statement*… }

```
for (j = 0; j < n; j = j + 1) {
    int bit = (rand()>>14)%2;
    if (bit == 0)
        printf(" ");
    else
        printf("*");
}
```

- **Others**

  `return` *expression*$_{\text{opt}}$ `;`

```
return;
return 0;
return -2*(a + b);
```

  `break ;`
  `continue ;`

- **Keywords (`if else while do for switch case` …) *<u>cannot</u>* 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 *conditional*s 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))
```