

CS113: Lecture 3

Topics:

- Variables
- Data types
- Arithmetic and Bitwise Operators
- Order of Evaluation

Variables

Names of variables:

- Composed of letters, digits, and the underscore (" _ ") character. (NO spaces; use underscore instead.)
- First character must be a letter.
- At least the first 31 characters matter – after that, they may not.
- You can't use keywords (like `if`, `else`, etc.) for variable names.

Similar rules for naming functions, etc.

Data types

- C's basic types and typical sizes:
 - `char` - a single byte, capable of holding one integer/character (8/16 bits)
 - `int` - an integer (16/32 bits)
 - `float` - single-precision floating point (32 bits)
 - `double` - double-precision floating point (64 bits)
 - NOTE: there is no basic "string" type.
- Actual size is compiler- and machine-dependent! (You can find out what the size is if you need to.)
- Qualifiers (e.g. `unsigned`, `long`) can be applied.
 - An integer can be `short`, `long`, and even `long long`
 - If it's `unsigned` it means that only positive integers can be represented. (If you try to do any operations with unsigned integers and negative integer constants, weird things might happen.)
 - The type `unsigned long long int` lets you represent very large positive integers!
 - Don't worry too much about all the different types unless you need to.

Variable declarations

- Variables must be declared at the *start* of a function, before use.

```
int lower;  
int upper;  
int step;  
char c;  
char d;
```

- Variables with the same type can be grouped together:

```
int lower, upper, step;  
char c, d;
```

- Variables can also be initialized in the declaration.

```
int lower = 0, upper = 8, step = 1;  
char c = 'f', d = 'z';
```

- What happens if a variable is not initialized and then used?

```
void main() {  
    int a;  
    printf( "The value of a is: %d\n", a );  
}
```

Examples of Constants

- Integer constant: 1234
- long int constant: 12345789 or 123456789L
- Integers can be specified in octal (leading zero) or hexadecimal (leading 0x or 0X): 037, 0x1f.
- Floating-point constant: 123.4

Character constants

- All characters are represented as integers (usually signed), and can be treated as integers.
- Escape codes correspond to characters, for use in single-quotes:
 - Examples: `\n` (newline), `\\` (backslash), `\"` (double quote)
 - Example use: `char a = '\n';`
- Variables of type `char` can be thought of as either a character or an integer.

```
printf( "%c", 'a' ); /* a is printed */
printf( "%d", 'a' ); /* 97 is printed */
printf( "%c", 97 ); /* a is printed */
printf( "%d", 97 ); /* 97 is printed */
```

- Lower-case letters, upper-case letters, digits “consecutive”

```
'a' == 97, 'b' == 98, . . ., 'z' == 122
```

```
'A' == 65, 'B' == 66, . . ., 'Z' == 90
```

```
'0' == 48, '1' == 49, . . ., '9' == 57
```

- Some more examples of the integer values corresponding to character constants:

```
'&' == 38, '*' == 42, '\n' == 10, '\\' == 92, . . .
```

char Example

```
void main() {
    char i;
    printf( "Here's the alphabet, in lower-case:\n" );
    for( i = 97; i <= 122; i++ ) {
        printf( "%c", i );
    }
    printf( "\n\nHere's the alphabet, in upper-case:\n" );
    for( i = 65; i <= 90; i++ ) {
        printf( "%c", i );
    }
}
```

```
void main() {
    char i;
    printf( "Here's the alphabet, in lower-case:\n" );
    for( i = 'a'; i <= 'z'; i++ ) {
        printf( "%c", i );
    }
    printf( "\n\nHere's the alphabet, in upper-case:\n" );
    for( i = 'A'; i <= 'Z'; i++ ) {
        printf( "%c", i );
    }
}
```

String constants

Strictly speaking, there is no string type, so there can't be any string constants. (A string is represented as an array of characters.)

Fortunately, C lets us deal with strings as if they were constants, so that they can be passed to functions that do things with strings. Just put the string in double-quotes:

- Example: `"A string"` is a string.
- The statement `printf("A string");` would print that string.
- You'll never see anything like `String s = "A string";`. There is no string type, and the `char*` type that is used for strings does NOT do what you might expect given this kind of an assignment.

We'll revisit these issues in more detail later in the course.

Type Conversions

C is very flexible with type conversions.

- If an operator has operands of different types, they are converted according to a small number of rules.
- Automatic conversions occur when a “narrower” operand can be converted into a “wider one”. Example: adding a `short` and a `long` will cause the `short` to be converted automatically. (See rules on K&R, p. 44 for details.)
- Keep in mind that a `char` is just a small integer, so you can do arithmetic operations: e.g., `'c' - 'a'` is 2. (No type conversion required!)

Conversions also occur when you try to assign a variable of one type to another. Be careful – the new assigned variable might be different!

- Example: if `x` is `float` and `i` is `int`, then the assignment `i = x` will truncate any fractional part of `x`.

Casting

You can explicitly *cast* a variable of one type to be a variable of another type.

- This is useful if you aren't sure how conversion will work, or you want to force conversion to happen in a specific way.

- Example:

```
int a=15, b=10;  
double x;
```

```
x = a / b;  
/* x is now 1.0 */
```

```
x = (double) a / (double) b;  
/* x is now 1.5 */
```

- Casting ensures floating point division rather than integer division (which truncates the result so that the type is still integer).

Enumeration constants

- An enumeration is a way to specify a list of constant integer values:

```
enum color { red, blue, green };
```

- Unless specified explicitly, the first name in an enum has value 0, the second one 1, etc.
- Example.

```
void main() {  
    enum color { red, blue, green };  
    int fave;  
    printf( "0=red,1=blue,2=green" );  
    printf( "Enter the number of your favorite:" );  
    scanf( "%d", &fave );  
    if( fave == red ) {  
        printf( "Red is also my favorite.\n" );  
    }  
}
```

- When explicit values are provided, unspecified values continue in progression from the most recent specified value.

```
enum month { JAN = 1, FEB, MAR, APR, MAY, JUN,  
            JUL, AUG, SEP, OCT, NOV, DEC };
```

Using printf

- Printing a float

- Simple form:

- ```
printf("%f", 3.141592653);
```

- Fancy form:

- ```
printf( "%6.2f", 3.141592653 );
```

- ...result: two spaces followed by 3.14

- 6 specifies *minimum field width*: at least 6 characters will be printed, with spaces added if necessary

- 2 specifies *maximum* number of digits to be printed after the decimal point

- Printing an int as an octal number

- ```
printf("%o\n", 17);
```

- ...result: 21

- Printing an int as a hexadecimal number

- ```
printf( "%x\n", 31 );
```

- ...result: 1f

- Use %X for upper-case letters

Operators

- Recall the relational operators ($>$, $>=$, $<$, $<=$), equality operators ($==$, $!=$), and the logical operators ($!$, $&&$, $||$).
- C has a number of *arithmetic operators*.
 - Assignment operator: $=$
 - Binary arithmetic operators: $+$, $-$, $*$, $/$, $\%$
 - * Can be applied to `int`, `float`, or `double`, except for $\%$ which can only be applied to `ints`.
 - * $\%$ is the “modulus” or “mod” operator: $a \% b$ is equal to the remainder when a is divided by b . We won’t worry about what happens on non-positive values (implementation dependent). Example: `8 % 3 == 2`.
 - Unary arithmetic operator: $-$. Example:
`x = -y;`
 - Shortcut operators: $+=$, $-=$, $*=$, $/=$
`x += 2; /* equivalent to x = x + 2; */`
`x *= 2; /* equivalent to x = x * 2; */`
 - Increment/decrement operators: $++$, $--$
`x++; /* acts like x = x + 1; */`
`x--; /* acts like x = x - 1; */`

++ and --: tricky expressions

- Both `x++` and `++x` are *expressions*. The expression `x++` acts like `x`, and `++x` acts like the expression `x+1`.
- What's special is the *side effect*: evaluating these expressions causes `x` to be incremented by 1.

```
int a = 10, b, c;  
b = a++; /* a is now 11, b is 10 */  
c = ++b; /* a, b, c are all 11 */
```

- For clarity, try not to mix `++` or `--` into complicated expressions.
- Note that the expression that `++` or `--` is applied to must be an *lvalue*, e.g. a variable.

```
(x + 2)++; /* no good! */
```

- The *left* side of an assignment statement must be an lvalue; hence the “l” in “lvalue”.

```
x + 2 = 8; /* no good! */
```

- The result of applying `++` or `--` to an lvalue is NOT an lvalue.

```
(x++)++; /* no good! */
```

Bitwise operators

- Six operators for bit manipulation which can only be applied to *integral operands* (e.g., variables of type `int` or `char`):
 - Bitwise AND (`&`)
 - Bitwise inclusive OR (`|`)
 - Bitwise exclusive OR (`^`)
 - Left shift (`<<`)
 - Right shift (`>>`)
 - One's complement (`~`)
- All binary except for one's complement.
- Left shifting fills vacated bits with zero.
- Careful! Right shifting a *signed* quantity (e.g. `int` variable) may fill vacated bits with sign bits on some machines.
- See PCP, Chapter 11 for more details. (We probably won't use bit operations again in this class, but it's good to know about them.)

Order of Evaluation

How are expressions with many operators evaluated?

Two considerations:

- Precedence
 - How is $1 + 2 * 3$ evaluated? Is it $(1 + 2) * 3$, or $1 + (2 * 3)$?
 - It's the latter: the $*$ operator has higher precedence than the $+$ operator.
 - Parentheses must be used if we want the addition to be performed first.
- Associativity
 - What about expressions containing operators at the same precedence level? E.g., $(12 / 6 * 2)$ or $(5 - 3 - 1)$?
 - These parse as $((12 / 6) * 2)$ and $((5 - 3) - 1)$: they are left associative. (Most operators are left associative.)

See table on p. 53 of K&R.

True or false?

```
void main() {
    int a = -2, b = -1, c = 0;

    if( a < b < c )
        printf( "True.\n" );
    else
        printf( "False.\n" );

    if (a >= b >= c)
        printf( "True.\n");
    else
        printf( "False.\n");
}
```

- Be careful! Use parentheses!