Lecture Notes for Lecture 2 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

Program Basics: Types, Statements, Operators, and Expressions

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 1 Review

- In this lecture, we learned about what a computer is, and the fact that there are two kinds: special-purpose and general-purpose.
- We looked at the earliest mechanical computers, designed by Charles Babbage and programmed by Ada Lovelace in the 1800s that had many of the elements of a modern computer.
- We also looked at an early electronic computer in the 1940s that was the forerunner of todays electronic computers.
- We saw that the architecture of a computer includes a central processing unit, memory, and input/output. The CPU fetches programs and data from memory and performs I/O functions.

Lecture 1 Review

- Memory on modern computers is binary and organized into 8-bit bytes that each have an address.
- The data in memory can represent characters and types of numbers that occupy one or more bytes. The encoding depends on the type and the representation.
- Instructions executed by the CPU are also stored in memory. Their format includes the operations to perform and what data to use. A program is a sequence of machine instructions.
- This class will use C, a mid-level imperative language that is efficiently compiled into machine instructions. C is easy to learn, and is a foundational language for computer science.

Identifiers and Types

- As we learned in the previous lecture, values are represented by one or more bytes. The number of bytes depends on the type of data.
- There are different machine instructions for performing operations on different types of data. Which instruction to use depends on the types of data.
- The data types supported by the C language reflect those supported by the underlying machine instructions.
- To enable the compiler to generate the correct instructions, a C program must specify the types of the values being used.

Identifiers and Types

- C enables a memory location to be given a symbolic name. The name should reflect its purpose in a program.
- An *identifier* specifies the name of a memory location, as well as type type of data associated with that location.
- This enables the compiler to translate operations in a C program into machine instructions that are appropriate for the types of identifiers.
- Identifiers can be used to designate
  - a *function*: a memory location of a sequence of instructions
  - a *variable*: a memory location whose value can change
  - a *constant*: a memory location whose value remains constant

Identifiers and Types

Function identifiers

- A C program consists of sequences of instructions known as *functions* that perform specific operations. Each function is named by an identifier that is used to *call* the function.
- A C program must have a function named *main* that is called when the program is run.
- The main function can call program-defined or library functions that *return* to the main function that called them.
- The program exits when the main function completes.
- We will learn more about functions in the lecture 3.

Identifiers and Types

- Function *main* calls a library-defined function

```
/**
 * @file hello_world.c
 *
 * This is the first demonstration program for lecture 2.
 */
#include <stdio.h>                      // input/output operations

/**
 * The main function prints the message “Hello World!”.
*/
int main(void) {                        // every C program has function main
        printf("Hello World!\n");       // library function prints message
}
```

Identifiers and Types

- Function *main* calls a program-defined function

```
/** @file hello_world2.c
 */
#include <stdio.h>                      // input/output operations

/** This function prints the message “Hello World!”.
 */
void printHello(void) {                 // function called by main
        printf("Hello World!\n");       // library function prints message
}

/**
 * The main function calls program-defined function to print “Hello World!”.
*/
int main(void) {                        // main function
        printHello();                   // call function to printHello
}
```

Identifiers and Types

Rules for identifier names

- C identifiers start with an upper- or lower-case letter or an underscore (_), followed by any number of upper- or lower-case letters, digits, or underscores.
  - valid identifiers:
    - x
    - averageValue
    - _unknown
    - value_1
  - Invalid identifiers:
    - 0abc (first character not a letter or underscore)
    - a.number (cannot contain other punctuation character)
    - while (cannot be a C language “keyword”)

Identifiers and Types

Identifier types

- Basic types supported by the C language include
  - Boolean: logical type that represents *true* and *false*.
    - Used to represent conditions and relationships
  - Integer type that represents whole values
    - Used in whole-number arithmetic calculations
    - Variants include: *short*, *int*, *long, long long (signed or unsigned)*
  - Real number type that can represent non-whole values
    - Used in scientific, financial, and other operations that require non-whole value calculations
    - Variants include *float*, *double*, and *long double*
  - Character type that represents textual information
    - Used in operations that process textual information
    - Also used as a small integers or binary data (signed or unsigned)

Identifiers and Types

- C does not guarantee the size of any data type. The size depends on the type supported by underlying computer
- C does specify the size relationships among type variants
  - size of *short* ≤ size of *int* ≤ size of *long* ≤ size of *long long*
  - size of *float* ≤ size of *double* ≤ size of *long double*
- A special *sizeof* operator provides the size for a type, variable, or constant. For example, sizeof(char) is 1. It usually returns unsigned long (or unsigned long long on Cygwin).

Identifiers and Types

- Here are typical sizes reported by sizeof() for the basic types.
  - bool: 1 byte (8 bits)
  - char, signed char, unsigned char: 1 byte (8 bits)
  - short (signed short), unsigned short: 2 bytes (16 bits)
  - int (signed int), unsigned int : 4 bytes (32 bits)
  - long (signed long), unsigned long: 8 bytes (64 bits)
  - long long (signed long long), unsigned long long : 8 bytes (64 bits)
  - float: 4 bytes (32 bits)
  - double: 8 bytes (64 bits)
  - long double: 16 bytes (80 bits) (note: 10 bytes stored in 16 byte field)

Identifiers and Types

- Specifying literal values for basic types:
  - bool: true, false
  - char: ‘a’, ‘\x61’, ‘\141’, ‘\n’, ‘\t’, ‘\’’
  - short or int: 0, 123, -45, 0245, 0xa5
  - unsigned short or unsigned int: 0U, 123U, 0245U, 0xa5U
  - long: 0L, 123L, -45L, 0245L, 0xa5L
  - unsigned long: 0UL, 123UL, 0245UL, 0xa5UL
  - long long: 0LL, 123LL, -45LL, 0245LL, 0xa5LL
  - unsigned long long: 0ULL, 123ULL, 0245ULL, 0xa5ULL
  - float: 0F, -45.2F, 5.3e2F (5300.0F), 314159e-5F (3.14159F)
  - double: 0.0, -45.2, 5.3e3 (5300.0), 314159e-5 (3.14159)
  - long double: 0.0L, -45.2L, 5.3e2L (5300.0L), 314159e-5L (3.14159L)

Identifiers and Types

- Specify sequence of characters as a literal *string*:
  - “a”
  - “Hello World”
  - “She said ‘hello’” (single quotes within double quotes)
  - ”She said \”hello\”” (double quotes “escaped” within double quotes
- Internally, a the compiler generates a sequence of characters with an extra byte after the last character whose value is 0. The extra byte *delimits* the string.
  - “Hello World” = ‘H’ ‘e’ ‘l’ ‘l’ ‘o’ ‘ ‘ ‘W’ ‘o’ ‘r’ ‘l’ ‘d’ ‘\0’

Identifiers and Types

Variable identifiers

- A variable is declared by specifying its type and its name. It can also be initialized using the assignment operator (=).
  - bool boolVal = true;
  - char charVal = ‘a’;
  - short shortVal = -3;
  - unsigned short unsignedShortVal = 2;
  - int intVal = -3;
  - unsigned int unsignedIntVal = 2;
  - long longVal = -3L;
  - unsigned long unsignedLongVal = 2UL;
  - float floatVal = 3.0F;
  - double doubleVal = 3.0;
  - long double longDoubleVal = 3.0L;

Identifiers and Types

Constant identifiers

- A constant is declared by specifying its type and name with the qualifier *const*. Constants must be initialized with either a literal value or a constant expression.
  - const int minValue = -100;
  - const int maxVal = 500;
  - const unsigned int range = maxVal – minVal + 1; // const expression
  - const long double pi = 3.14159265358979;
- Once declared and initialized, the value of a const cannot be changed.
  - maxVal = 1000; // C compiler reports an error

Identifiers and Types

- Formatted printing using printf

  printf (*format string*, value1, value2,…, valueN)

  printf(”The answer is %d\n”, 42);

  The answer is 42

  printf(“The answer to problem %d is %f\n”, 1, 42.0);

  The answer to problem 1 is 42.0

  printf(“The answer to problem %d is %s\n”, 2, “cat”);

  The answer to problem 2 is cat

Identifiers and Types

- Format specifiers in printf format string
  - bool: %d, %i
  - char: %c
  - short: %d, %i
  - unsigned short: %u
  - int: %d, %i
  - unsigned int: %u
  - long: %ld, %li
  - unsigned long: %lu
  - long long: %lld, %lli (see note)
  - unsigned long long: %llu (see note)
  - float: %f
  - double: %f
  - long double: %Lf
  - string: %s

  Note: Using MinGW under Microsoft Windows, must define symbol __USE_MINGW_ANSI_STDIO=1

Identifiers and Types

- C provides operations on basic types through pre-defined *operators* that often directly correspond to single machine instructions. The operators take one or more *operands*.
- These operators fall into the following categories:
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Conditional operators
- Operators and operands can be used to create expressions that evaluate to a value that can be printed or assigned.
- These operators are referred to as *polymorphic* because the machine instructions that are compiled for an operator depend on the operand types.

Identifiers and Types

- The order of a C operator and its operand(s) often determines which operation to perform.
- There are three forms:
  - With an *infix* operator, the operands go on either side of the operator. An example is the infix '-' operator that indicates subtraction. For example, *a* – *b*.
  - With a *prefix* operator, the operator comes before the operand(s). An example is the infix '-' operator that indicates negative. For example, -*a* produces the negative of the value a
  - With a *postfix* operator, the operator comes after the operands. There are only a few postfix operators, and we will see them shortly.

Identifiers and Types

- Arithmetic operators produce values based on operand types.
  - Infix binary operators: +, -, *, /, %
    - addition: 5 + 4, 5L + 4L, 5.0F + 4.0F, 5.0 + 4.0e0
    - subtraction: 5 - 4, 5L - 4L, 5.0F - 4.0F, 5.0 - 4.0e0
    - multiplication: 5 * 4, 5L * 4L, 5.0F * 4.0F, 5.0 * 4.0e0
    - division: 5 / 4, 5L / 4L, 5.0F / 4.0F, 5.0 / 4.0e0
    - integer/long remainder: 5 % 4, 5L % 4L, -5 % 4, -5 % -4,
  - Prefix unary operators: ++, --, +, -
    - pre-increment/pre-decrement: ++intVal, --intVal
    - negative: - intVal
  - Postfix unary operators: ++, --
    - post-increment/post-decrement: intVal++, intVal--
  - Arithmetic assignment operators: +=, -=, *=, /=, %=
    - intVal += 2, intVal -= 2, intVal *= 2, intVal /= 2, intVal %= 2

Identifiers and Types

- Examples of arithmetic operators

```
double p1x = 5.0, p1y = 3.5;          // point 1 x and y
double p2x = -2.0, p2y = -7.5;        // point 2 x and y
double deltax = p1x - p2x, deltay = p1y - p2y;
double dist = sqrt(deltax*deltax + deltay*deltay);  // library fn

int count = 0;
int count1 = ++count;                 // count1 is 1, count is 1
int count2 = count++;                 // count2 is 1, count is 2
count *= count;                       // count is 4

int quotient = 27/5;                  // quotient is 5
int remainder = 27%5;                 // remainder is 2
int val = 5*quotient + remainder;  // val is 27
```

Identifiers and Types

- Relational operators produce bool results
  - equals (==):
    - 3 == 3 -> true, 3 == 4 -> false
  - not equal (!=):
    - 3 != 3 -> false, 3 != 4 -> true
  - greater (>):
    - 3 > 3 -> false, 4 > 3 -> true
  - less (<):
    - 3 < 3 -> false, 3 < 4 -> true, 4 < 3 -> false
  - greater or equal (>=):
    - 3 >= 3 -> true, 3 >= 4 -> false
  - less or equal (<=):
    - 3 <= 3 -> true, 3 <= 4 -> true

Identifiers and Types

- Logical operators produce bool results
  - logical AND (&&):
    - false && false -> false
    - false && true -> false
    - true && false -> false
    - true && true -> true
  - logical OR (||)
    - false || false -> false
    - false || true -> true
    - true || false -> true
    - true || true -> true
  - logical NOT (!)
    - ! true -> false
    - ! false -> true

Identifiers and Types

- Conditional operator (?) is a *trinary operator*. It operates on a boolean value and two other values of the same type.
  - expr1 ? expr2 : expr3
- If expr1 evaluates to *false*, the operator evaluates and returns expr3, otherwise the operator evaluates and returns expr2

```
int a, b;
char c;
bool d;
...
(a > b) ? a : b;                          // maximum of a and b
(a < 0) ? -a : a;                         // absolute value of a
(a % b == 0) ? a : 0;                     // a if b is a factor else 0
(c >= 'a' && c <= 'z') ? c - ('a' - 'A') : c;   // upper case c
d ? "true" : "false"                      // "true" if d is true, else "false"
```

Identifiers and Types

- A word about programming style with the conditional operator. The conditional operator uses an expression that evaluates to bool value *true* or *false*.
- In reality, C allows a result of any basic type to be used, and interprets a value of 0 as *false*, and any other value as *true*.
- Never use a non-bool expression with a conditional operator or other conditional operation. Instead of this

```
intValue ? trueValue : falseValue;          // interpret non-0 intVal as true
```

  do this

```
(intValue != 0) ? trueValue : falseValue;   // explicitly test intVal against 0
```

- Dennis Richie made this mistake in the first version of Unix. It made the code unreadable, and he replaced it in the next version of Unix. He considered it one his worst blunders!

Identifiers and Types

Operator precedence

- In an expression, certain operators are evaluated before others. This is known as *operator precedence*. Here are the operators from highest to lowest.
  - postfix () ++, --
  - prefix +, -, !, ++, --, sizeof()
  - multiplicative: *, /, %
  - additive: +, -
  - relational: <, <=, >=, >
  - equality: ==, !=
  - logical AND: &&
  - logical OR: ||
  - conditional ?
  - assignment: =, +=, -=, *=, /=, %=

Identifiers and Types

- Use parentheses to override operator precedence
  - 3 + 5 * 4 -> 23, *not* 32
    - taken as (3 + (5 * 4))
  - (3 + 5)*4 -> 32
    - parentheses override standard operator precedence for +
  - 1 + 9 % 5 -> 5 *not* 0
    - taken as (1 + (9 % 5))
  - (1 + 9) % 5 -> 0
    - parentheses override standard operator precedence for %
  - a > b && 45 <= sum || sum < a + b && d > 90
    - taken as ((a > b) && (45 <= sum)) || ((sum < (a + b)) && (d > 90))
  - a > b && (45 <= sum || sum < a + b) && d > 90
    - parentheses override standard operator precedence for &&