CMPUT 201: Practical Programming Methodology

Guohui Lin
guohui@ualberta.ca
Department of Computing Science
University of Alberta
September 2019

Lecture 9: Basic Types

Agenda:

- Integer types
  - `int`
  - built-in types (no need standard libraries)
  - constants, variables: storage (machine format)
  - type conversion
  - type definitions
  - `sizeof` operator: return #bytes
- Floating types
- Character types
  - ```
    scanf("%c", &ch);
    ch = getchar();
    ch = getc(stdin);
    ```

Reading:

- Textbook: Chapter 7
    ```

Basic built-in types:

- `int`
- `float`
- `bool`: need

  ```
  #include <stdbool.h>
  ```

Integer types:

- Two categories: signed (default) and unsigned
  - `int`: 32 bits / 4 bytes
  - the first (leftmost) bit is the sign bit: 0 for positive
  - largest $2^{31} - 1 = 2,147,483,647$ (unsigned $2^{32} - 1$)

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

  - `short`: 16 bits
  - `long`: 32/64 bits (if on a 64-bit machine)
  - `long long int`: force to have 64 bits

- The corresponding conversion specification
  - `%d` for `int` type
  - `%u` for `unsigned` type
  - `%hd, %hu` for `short`
  - `%ld, %lu` for `long`
  - `%lld, %llu` for `long long`

Integer constants:

- Machine format — binary `0, 1`
- `%[h,l,ll]d, %[h,l,ll]u` — decimal `0, 1, 2, ..., 9`
- Octal constants `%o, %O`: must begin with a zero, e.g. 017, 0377, 07777
- Hexadecimal constants `%x, %X`: must begin with 0x, e.g. 0x17, 0xff, 0X7Fff

```
int i;
unsigned int j;
long long k;
unsigned long long p;
...


i = 500;
j = 015u;
k = 0x3Ffl;
...

p = 0xffff12345UL;
...
```

Floating types:

- Three formats:
  - `float`: single-precision, 32 bits / 4 bytes
  - `double`: double-precision, 64 bits / 8 bytes
  - `long double`: extended-precision, 80 or 128 bits, rarely used

- Not just the number of bits (the standard `%e` format, but in binary):
  - the 1st bit for "sign": `0/1`
  - a floating-point number is written in binary (do you know how to?)
  - the next a few bits represent the exponent, or where the decimal point is
  - i.e., $\mathrm{i}_k\mathrm{i}_{k-1}\ldots\mathrm{i}_1\mathrm{i}_0.\mathrm{d}_1\mathrm{d}_2\ldots\mathrm{d}_\ell$ — i_k is always 1 (unless 0 exponent)
  - for `float`: single-precision, 32 bits

    8 bits for exponent
    23 bits for binary representation (called fraction)
    i_k is not stored, and therefore $23 = k + \ell$

s	e	e	e	e	e	e	e	e	i	i	i	i	i	i	i	·	·	·	·	·	·	·	·	d	d	d	d	d	d	d	d

The precision issue:

- When "`x = 100839.21f;`"
- The storage is:
  - $100{,}839_{10} = 1{,}1000{,}1001{,}1110{,}0111_2$: 17 bits used
  - exponent is $16_{10} + 127 = 1{,}0000_2 + 111{,}1111_2 =$ `1000,1111`
  - leaves with $23 - 16 = 7$ bits for the digits after the decimal point
  - $.21_{10} = .0011010_2$ (.006875) ? $.0011011_2$ (-.0009375)
  - $.21_{10} = .0011011_2 = .2109375$

therefore, in machine memory, `x` is represented as

`0 10001111 1000100111100111 0011011`

- Further notes on type `float`:
  - maximum value is $(2^{24} - 1) \times 2^{127-23} \approx 3.40282 \times 10^{38}$
  - minimum value is $\approx 1.17549 \times 10^{-38}$
- The storage for type `double` is very the same

Floating types:

- The conversion specifications:
  - `%m.pf`, `%e`, and `%g`
  - `p`: number of digits after decimal point or max number of significant digits in `g`
  - `%lf` and `%Lf` for type **`double`** and type **`long double`**, respectively
- Floating constants
  - `x = 100839.21;`
  - `x = .21;`
  - `x = 100839.;`
  - `x = 100839.21f;`
  - `x = 1.008e5;`
  - `x = 10.e-3;`
  - `x = .1008e+39;`
- Recall when "`x = 100839.21f;`", we have

```
ghlin@ug10:~/CMPUT201_19F/Week04>./test
|40|   40|40   |  040|
|100839.211| 1.008e+05|100839    |
```

Precision?

- For example,

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void) {

    int i;
    float x;

    i = 40;
    x = 100839.21f;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|%-10.3g|\n", x, x, x, x);

    return 0;
}
```

- Output:

```
ghlin@ug10:~/CMPUT201_19F/Week04>gcc -Wall tprintf.c -o test
ghlin@ug10:~/CMPUT201_19F/Week04>./test
|40|   40|40   |  040|
|100839.211| 1.008e+05|100839    |1.01e+05  |
```

Precision?

- For example,

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void) {

    int i;
    double x;

    i = 40;
    x = 100839.21; /* previously using 100839.21f */

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|%-10.3g|\n", x, x, x, x);

    return 0;
}
```

- Output:

```
ghlin@ug10:~/CMPUT201_19F/Week04>gcc -Wall tprintf.c -o test
ghlin@ug10:~/CMPUT201_19F/Week04>./test
|40|   40|40   |  040|
|100839.210| 1.008e+05|100839    |1.01e+05  |
```

Character types:

- Values of type `char` are machine dependent
  - different character sets
- ASCII (American Standard Code for Information Interchange):
  - 7-bit code for 128 characters
  - extended to Latin-1 of 256 characters
- Syntax: use of single quotation marks

```
char ch;

ch = 'A'; /* variable ch is assigned a value 'A', upper-case A */
```

Character types:

- Characters are (treated as) small integers

```
char ch;
int i;

ch = 'A'; /* variable ch is assigned a value 'A', upper-case A */
i = ch; /* variable i has a value 'A', 1000001 in binary, or 65 in decimal */
```

- Consequently,
  - all operations on integers can be done with characters, e.g.
    `for (ch = 'A'; ch <= 'Z'; ch++) { ... }`
  - upper-case letters A to Z are represented by `1000001` to `1011010`, consecutively
  - numerical digits 0 to 9 are also consecutive
  - can have `signed` or `unsigned` type

- Conversion specification
  - `%c`

Arithmetic types, summary:

- Integer types
  - `char`
  - signed: `signed char`, `short int`, `int`, `long int`, `long long int`, and extended
  - unsigned: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `bool`, and extended
- Floating types
  - real: `float`, `double`, `long double`
  - complex: `float _Complex`, `double _Complex`, `long double _Complex`

Escape sequences, summary:

- We have known

alert (bell):	`\a`
backspace:	`\b`
new line:	`\n`
horizontal tab:	`\t`
`"`:	`\"`
`\`:	`\\`
(`%`	`%%`)

- Some more

form feed:	`\f`
carriage return:	`\r`
vertical tab:	`\v`
`?`:	`\?`
`'`:	`\'`

- Categories
  - control characters
  - line-feed character
  - characters can be included in strings
- More powerful numeric escapes (any character, e.g. '\33' represents 'ESC')

Character-handling functions:

- Convert case:

```
ch = toupper(ch);
```

- Need the following library:

```
#include <ctype.h>
```

  - `toupper(char)` implements simply

```
if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';
```

Reading and writing characters:

- Conversion specification: `%c`
  - difference between the following — skipping white spaces?

    ```
    scanf("%c", &ch);
    scanf(" %c", &ch);
    ```

- - read a single character by "`ch = getchar();`"
  - write a single character by "`putchar(ch);`"
  - function return values are `int` (not `char`!)

- Skip the rest of line (idiom):

```
do {
    scanf("%c", &ch);
} while (ch != '\n');

do {
    ch = getchar();
} while (ch != '\n');

while ((ch = getchar()) != '\n')
    ;

while (getchar() != '\n')
    ;
```

Example:

- Determine the length of a message (Page 141)
- Appearance:

```
Enter a message: I have a date!
Your message was 14 character(s) long.
```

Type conversion:

- Cases: the size and the stored way must match w/ what defined
  - operands in arithmetic/logical expression not of the same type
  - type of right side of an assignment not matching the type of variable on left side
  - type of argument in a function not matching that of parameter
  - type of function return value not matching the return type
- Expressions may mix basic types
- Compiler does the conversions automatically!
  - implicit conversions
  - "promoting" some type
  - rule of thumb: no losing semantics or precision much (narrowest and safest)

    `bool` → `char` → `short int` → `int` → `unsigned int` → `long int` → `unsigned long int`

    (→) `float` → `double` →`long double`

Type conversion:

- Conversion during assignment
  - type of right side of an assignment not matching the type of variable on left side
  - some are “promotion”, others could be problematic

    e.g., (recall the printing a table of squares?)

    ```
    int i;

    i = 842.97; /* i is now 842 */
    i = -842.97; /* i is now -842 */
    i = 1.0e10; /* meaningless, wrong */
    ```

- Casting
  - form:

    ```
    ( type name ) expression
    ```

  - the value of `expression` is converted to the `type name`, e.g.

    ```
    float f, frac_part;

    fact_part = f - (int) f;
    ```

  - here (`type name`) is regarded as a unary operator

Type definitions:

- `#define BOOL int`
  - macro, every appearance of `BOOL` is replaced by `int`
- Type definition:

  `#typedef int Bool;`
  - the `;`
  - `Bool` is a new data type, and its type is `int`
  - later `Bool` can be used the same as any built-in type to declare variables
  - advantages in code readability, ease of modifying, and portability

    ```
    #typedef int Quantity

    Quantity q;
    ```
  - later may just change to the following to increase the range for `q`:

    ```
    #typedef long Quantity

    Quantity q;
    ```
  - in C library, `#typedef unsigned long int size_t;`

The `sizeof` operator:

- Check how much memory is required to store values of a particular type

```
sizeof ( type name )
```

```
sizeof(int)
sizeof(100.0f)
sizeof(i)
sizeof(i + j)
```

  - unary operator
  - return a `size_t` integer (i.e. `unsigned long int`): # of bytes storing the value
  - for example (conversion specification for `size_t` is typically `%lu` or `%zu`),

```
int i;

sizeof(i); /* is normally 4 */
printf("Size of int: %zu\n", sizeof(int));
```

Graph terminologies:

- $G = (V, E)$
- V is the set of vertices $\{v_1, v_2, \ldots, v_n\}$
- E is the set of edges,
  each edge is a set of two distinct (unordered) vertices, e.g. $\{v_1, v_4\}$, $\{v_4, v_2\}$
  - v_1 and v_4 are adjacent
  - $\{v_1, v_4\}$ is incident at/with v_1
  - $\{v_1, v_4\}$ and $\{v_2, v_4\}$ are adjacent
- A graph can be represented as an adjacency matrix $A_{n \times n}$ (binary)

```
0 1 1 1 0
1 0 0 1 1
1 0 0 0 1
1 1 0 0 1
0 1 1 1 0
```

  - $V = \{v_1, v_2, v_3, v_4, v_5\}$
  - $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

Agenda:

- One-dimensional arrays
  - `int a[length];`
  - aggregate variables storing a collection of data
  - `length` must have an `int`-type value by the time of declaration
  - index starting with `0`
  - trying to access `a[i]` with `i < 0` or `i >= length`? (2 kinds of) violation!
- Multi-dimensional arrays
- Variable-length arrays

Reading:

- Textbook: Chapter 8

Variables:

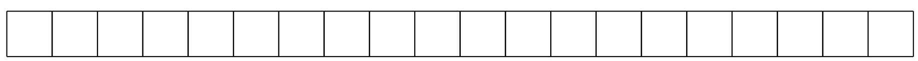
- We have seen scalar
  - holding a single data item
- Aggregate variables
  - store a collection of values
  - arrays
  - structures

One-dimensional arrays:

- A data structure containing a number of data values, of the same type
  - called elements
  - each can be accessed by its position within the array
  - declaration: type of the elements, array name, the number of elements (length)

    `int a[20];`

- Visualization, an array `a`:

  a 

  - length is a **constant** integer expression, such as 10, 1+4, or N (macro definition)
  - conceptually, elements are arranged consecutively in memory
  - index/subscript starting from 0
  - `a[2]` is an lvalue, the 3rd element, treated as a type `int` variable

One-dimensional arrays:

- idioms:

```
#define N 20;
...
int a[N], i;

for (i = 0; i < N; i++)
    a[i] = 0;

for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

sum = 0;
for (i = 0; i < N; i++)
    sum += a[i];
```

- Subscript out of range, undefined behavior (Page 163)

```
    int a[N], i;

    for (i = 1; i <= 2 * N; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i, a[i]);
    }
```

Array initialization:

- When declared,

```
int b[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int c[10] = {1, 2, 3, 4, 5, 6, 7 + 8};

int d[10] = {0};

int e[] = {1, 2, 3, 4, 5, 6, 7};

int f[10] = {[2] = 2, [7] = 9, [9] = 7, [2] = 3};
```

  - by a list of constant expressions
  - shorter? fill with 0's (default values)
  - empty or longer? illegal!
  - no length? use the length of the list
  - designated? the others default

Checking a number for repeated digits (Page 166):

- Appearance:

```
Enter a number: 3456787
Repeated digit

Enter a number: 9758
No repeated digit
```

- The algorithm:
  - check from right (least significant) to left
  - obtain digit by remainder (dividing by 10)
  - first time seen, okay, set a flag (`true`)

    ```
    bool digit_seen[10] = {false}; /* all 10 digits not seen yet */
    ```

  - second time seen, exit for "`Repeated digit`"
  - update the number by quotient, terminate when becomes 0

    ```
    while (n > 0) {
        digit = n % 10;
        if (digit_seen[digit])
            break;
        digit_seen[digit] = true;
        n /= 10;
    }
    ```

`sizeof` operator:

- Recall: `sizeof(int)` returns the number of bytes for storing a type `int` value
- `sizeof(a)` returns size of array `a` in bytes
  - `sizeof(a[0])` returns size of element `a[0]` in bytes
  - thus, length = `sizeof(a) / sizeof(a[0])`
    - type `size_t` vs the type of length
    - useful when the length of the array is unknown!
    - similar to the use of macro definition
- Computing interest (Page 168):
  - saving $100
  - yearly interest rate `rate%`
  - to print out amounts at one-year interval, appearance:

```
Enter interest rate: 1.35
Enter number of years: 3

Years    1.35%    2.35%    3.35%    4.35%    5.35%
  1      101.35   102.35   103.35   104.35   105.35
  2      102.72   104.76   106.81   108.89   110.99
  3      104.10   107.22   110.39   113.63   116.92
```

Multi-dimensional arrays:

- e.g., a two-dimensional array

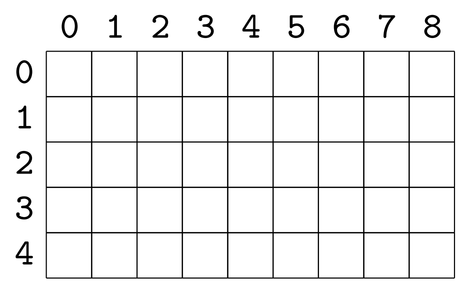
  `int a[5][9];`

  – 5 rows, indexed from 0

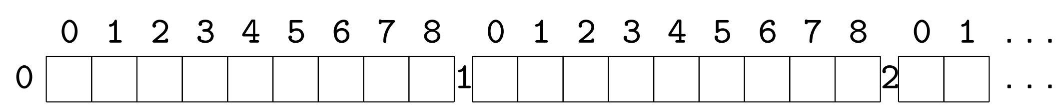
  – 9 columns, indexed from 0

  – each element has type `int`

- Visualization:



- Inside memory:



  – therefore, `a[i][j]` is the same as `a[i * 9 + j]`

  – conceptually, elements are arranged consecutively in memory

Initialization w/ cares:

- Again, non initialized elements set to default value 0:

```
int a[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 1},
               {1, 1, 0, 1, 1, 1},
               [3][2] = 1, [4][8] = 1};
```

- Constant arrays
  - meaning that your program cannot change the value for any element
  - declare using `const`, w/ given values

    ```
    const char DNA_chars[] = {'A', 'C', 'G', 'T'};
    ```
  - used as a "dictionary"

- E.g., generating a hand of random cards (Page 172), appearance

```
Enter number of cards in hand: 6
Your hand: 5s 7d 9h tc 6h kh
```

Representing a Sudoku game:

- The usual file format describing a game (tokens separated by a space or a tab):

```
0 4 0 0 0 0 1 7 9
0 0 2 0 0 8 0 5 4
0 0 6 0 0 5 0 0 8
0 8 0 0 7 0 9 1 0
0 5 0 0 9 0 0 3 0
0 1 9 0 6 0 0 4 0
3 0 0 4 0 0 7 0 0
5 7 0 1 0 0 2 0 0
9 2 8 0 0 0 0 6 0
```

- Denoted as `S[9][9]`:
  - `0` indicates ‘to be filled’
  - check out what the game is!

- Coming up rules for solving (some of) the game? want to implement them?

Variable-length arrays:

- `-std=c99`
- Basically, variable declarations can be any where (define and then use)

```
#include <stdio.h>

int main(void) {

    int i, n;

    printf("Enter the length of array: ");
    scanf("%d", &n);

    int a[n]; /* declare a lenght-n array a */

    for (i = 0; i < n; i++) {
        if (a[i] == 0)
            printf("a[%d] is nicely initialized to %d? :-)\n", i, a[i]);
        else
            printf("a[%d] has a system-leftover value %d\n", i, a[i]);
    }

    return 0;
}
```

Lecture 11: Array applications

Agenda:

- Two sorting algorithms
  - `bubble sort`
  - `insertion sort`

Reading:

- Textbook: Chapter 8

Bubble sort:

- A comparison-based sorting algorithm (You should have known already)
- Goal: sort an array of integers into a non-decreasing order
- The algorithm (mathematically):
  - check every pair of adjacent elements `a[i]` and `a[i+1]`

    ```
    if (a[i] > a[i+1]) {
        ?; // swap these two elements in the array
    }
    ```
  - use `for`-loop
  - two nested `for`-loops
- Question: How many comparisons are made?

Insertion sort:

- A comparison-based sorting algorithm (You should have known already)
- Goal: sort an array of integers into a non-decreasing order
- The algorithm (mathematically):
  - assume `a[0..i-1]` is already sorted
  - how to insert the element `a[i]` to maintain sorted?

    ```
    if (a[i] < a[j]) {
        ?; // insert a[i] right before a[j]
    }
    ```

  - use `for`-loop
  - two nested `for`-loops
- Question: How many comparisons are made?

Code design:

- Program appearance:

```
Enter the length of the array: 10
Enter 10 integers to be sorted: 1 4 11 100 2 7 3 -1 99 6
In sorted non-decreasing order: -1 1 2 3 4 6 7 11 99 100
```

- Perhaps combine two sorting algorithms together, and let user choose

```
>./mysorting
Select sorting algorithm (i for insertsion, b for bubble): i
Enter the length of the array: 10
Enter 10 integers to be sorted: 1 4 11 100 2 7 3 -1 99 6
In sorted non-decreasing order: -1 1 2 3 4 6 7 11 99 100
```

- We will implement later `mysorting` to have options, for example

```
>./mysorting -b|h|i|m|q
```

  - `-b` for `bubble sort`
  - `-h` for `heap sort`
  - `-i` for `insertion sort`
  - `-m` for `merge sort`
  - `-q` for `quick sort`

Agenda:

- Definition
  - recall the following?

```
int main() {}
int main(void) {}
int main(int argc, char *argv[]) {}
```

- Defining and calling functions
- Function declarations
- Arguments: passed-by-value

Reading:

- Textbook: Chapter 9