Lecture 10: Basic Types

By Ahmed E. Khaled
Recap to previous lecture!

- While loop
- Do While loop
- For loop
The **while** Statement

- Using a **while** statement is the easiest way to set up a loop.
- The **while** statement has the form

  ```
  while ( expression )
  {
    statement
  }
  ```

- *expression* is the controlling expression; *statement* is the loop body.
do .. while, loop

• General form of the do statement:

```
do {
    statements
}
} while ( expression );
```

• When a do statement is executed, the loop body is executed first, then the controlling expression is evaluated.

• If the value of the expression is true, the loop body is executed again and then the expression is evaluated once more.
The for Statement

• The for statement is ideal for loops with “counting” variable.
• General form of the for statement:

```plaintext
for ( expr1 ; expr2 ; expr3 )
{
    statements
}
```

expr1, expr2, and expr3 are expressions.
Basic types of variable in C
• So far, we’ve used only two C’s basic (built-in) types: int and float.

• Other built-in types:
  – Full range of Integer type.
  – Double (greater precision than float).
  – Character type.

• How to convert a value of one type to an equivalent value of another.
Integer Types

• C supports two fundamentally different kinds of numeric types: integer types and floating types.
• The integer types (typically stored in either 16 or 32 bits), in turn, are divided into two categories: *signed* and *unsigned*. 
Signed and Unsigned Integers

• The leftmost bit of a *signed* integer (*known as sign bit*) is 0 if the number is positive or zero, 1 if it’s negative.

• The largest 16-bit integer 32,767 ($2^{15} – 1$) has the binary representation of 0111111111111111.

• The largest 32-bit integer 2,147,483,647 ($2^{31} – 1$) has the binary representation of 01111111111111111111111111111111.
Signed and Unsigned Integers

• An integer with no sign bit (the leftmost bit is considered part of the number’s magnitude) is said to be unsigned.

• The largest 16-bit unsigned integer is 65,535 (2^{16} - 1).

• The largest 32-bit unsigned integer is 4,294,967,295 (2^{32} - 1).
• By default, integer variables are *signed*, the leftmost bit is reserved for the sign.

• To tell the compiler that a variable has no sign bit, declare it to be *unsigned*.

```c
unsigned int number = 10;
unsigned number = 10;
```

• Unsigned numbers are primarily useful for *systems programming* and low-level, machine-dependent applications.
Integer Types

• The int type is usually 32 bits, but may be 16 bits on older CPUs.

• **long** integers may have *more bits than ordinary integers*; **short** integers may have *fewer bits*.

• The specifiers **long** and **short**, as well as **signed** and **unsigned**, can be combined with int to form **integer types**.

• short int = short
• unsigned short int = unsigned short
• int = int
• unsigned int = unsigned
• long int = long
• unsigned long int = unsigned long

the word int can be dropped (long int can be abbreviated to just long)
Integer Types

The range of values represented by each of the six integer types varies from one machine to another.

Typical ranges on a 16-bit machine:

<table>
<thead>
<tr>
<th>Type</th>
<th>Smallest Value</th>
<th>Largest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>−32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>−32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>long int</td>
<td>−2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>
Reading and Writing Integers

short s;
scanf("%hd", &s); printf("%hd", s);

long m;
scanf("%ld", &m); printf("%ld", m);

unsigned g;
scanf("%u", &g); printf("%u", g);

unsigned long y;
scanf("%lu", &y); printf("%lu", y);
Also for constants …

To force the compiler to treat a certain constant:

```c
#define CONSTANT_X 377U    //unsigned
#define CONSTANT_Y 377L     //long
#define CONSTANT_Y 377UL    //unsigned long
```
Floating Types

- C provides three floating types, corresponding to different floating-point formats:
  - float  Single-precision floating-point
  - double Double-precision floating-point
  - long double Extended-precision floating-point
double d;
scanf("%lf", &d);  printf("%lf", d);

long double m;
scanf("%Lf", &m);  printf("%Lf", m);
Also for constants …

To force the compiler to treat a certain constant:

```c
#define CONSTANT_X 0.5f //float

#define CONSTANT_Y 0.5L  //long double
```
Character Types

• The values of type `char` can vary from one computer to another, because different machines may have different underlying `character sets`.

• Today’s most popular character set is `ASCII` (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.

• ASCII is often extended to a 256-character code that provides the characters necessary for Western European and many African languages.
Character Sets

• A variable of type `char` can be assigned any single character:

```c
char ch;
ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero */
ch = ' ';    /* space */
```

• Notice that character constants are enclosed in single quotes, not double quotes.
Operations on Characters

• When a character appears in a computation, C uses its integer value.

• Consider the following examples, which assume the ASCII character set:

```c
char ch;
int i;

i = 'a';       /* i is now 97   */
ch = 65;       /* ch is now 'A' */
ch = ch + 1;   /* ch is now 'B' */
ch++;          /* ch is now 'C' */
```
Operations on Characters

• For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:

```c
char ch;
for (ch = 'A'; ch <= 'Z'; ch++) {
    ...
}
```
Operations on Characters

• An if statement that converts a lower-case letter to upper case:

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

• Comparisons such as 'a' <= ch are done using the integer values of the characters involved.
Reading and Writing Characters

• The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```c
char ch;
scanf("%c", &ch);  /* reads one character */
printf("%c", ch);  /* writes one character */
```

• `scanf` doesn’t skip white-space characters, to skip white space before reading a character, put a space in its format string just before `%c`:

```c
scanf(" %c", &ch);
```
• For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.

• `putchar` writes a character: `putchar(ch);`

• Each time `getchar` is called, it reads one character, which it returns: `ch = getchar();`

• Like `scanf`, `getchar` doesn’t skip white-space characters as it reads.
• Consider the `scanf` loop that we used to skip the rest of an input line:

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

• Rewriting this loop using `getchar` gives us the following:

```c
do {
    ch = getchar();
} while (ch != '\n');
```
• Develop a program that displays the length of a message entered by the user:

Enter a message: Brevity is the soul of wit. Your message was 27 character(s) long.

• We could use either scanf or getchar to read characters; most C programmers would choose getchar.