Basic Types

Based on slides from K. N. King
Bryn Mawr College
CS246 Programming Paradigm

Basic Types

C’s basic (built-in) types:
- Integer types, including long integers, short integers, and unsigned integers
- Floating types (float, double, and long double)
- char
- _Bool (C99)

Integer Types

C supports two fundamentally different kinds of numeric types: integer types and floating types.
- Values of an integer type are whole numbers.
- Values of a floating type have a fractional part.
- The integer types, in turn, are divided into two categories: signed (default) and unsigned.
- The leftmost bit of a signed integer (known as the sign bit) is
  - 0 – the number is positive or zero,
  - 1 – negative.

Typical ranges of values for the integer types 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>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</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>

Typical ranges on a 32-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>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</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>

Typical ranges on a 64-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>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</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>

The <limits.h> header defines macros that represent the smallest and largest values of each integer type.
Integers Constants

- **Constants** are numbers that appear in the text of a program.
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

- Octal numbers use only the digits 0 through 7.
- Each position in an octal number represents a power of 8.
  - The octal number 237 represents the decimal number:
    \[ 2 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 128 + 24 + 7 = 159. \]
- A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  - The hex number 1AF has the decimal value:
    \[ 1 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 = 256 + 160 + 15 = 431. \]

Integer Constants

- **Decimal** constants contain digits between 0 and 9, but must not begin with a zero:
  - 15, 255, 32767
- **Octal** constants contain only digits between 0 and 7, and must begin with a zero:
  - 017, 0377, 07777
- **Hexadecimal** constants contain digits between 0 and 9 and letters between a and f, and always begin with 0x:
  - 0xf, 0xff, 0x7fff
  - The letters in a hexadecimal constant may be either upper or lower case:
    - 0xff, 0xFF, 0xFF

Integer Overflow

- When arithmetic operations are performed on integers, it’s possible that the result will be too large to represent.
- If the result can’t be represented as an int (because it requires too many bits), we say that overflow has occurred.
  - When overflow occurs during an operation on signed integers, the program’s behavior is undefined.
  - When overflow occurs during an operation on unsigned integers, the result is defined: we get the correct answer modulo \(2^n\), where \(n\) is the number of bits used to store the result.

Reading and Writing Integers

- When reading or writing an unsigned integer, use the letter u, o, or x instead of d in the conversion specification.
  - unsigned int u;
  - scanf("%u", &u); /* reads u in base 10 */
  - printf("%u", u); /* writes u in base 10 */
  - scanf("%o", &u); /* reads u in base 8 */
  - printf("%o", u); /* writes u in base 8 */
  - scanf("%x", &u); /* reads u in base 16 */
  - printf("%x", u); /* writes u in base 16 */
Reading and Writing Integers

- When reading or writing a short integer, put the letter h in front of d, o, u, or x:
  ```
  short s;
  scanf("%hd", &s);
  printf("%hd", s);
  ```
- When reading or writing a long integer, put the letter l ("ell", not "one") in front of d, o, u, or x.

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 (rarely used)
- Macros that define the characteristics of the floating types can be found in the <float.h> header.

Floating Constants

- By default, floating constants are stored as double-precision numbers.
- To indicate that only single precision is desired, put the letter f (or F) at the end of the constant (for example, 57.0F).
- To indicate that a constant should be stored in long double format, put the letter L (or l) at the end (57.0L).

Reading and Writing Floating-Point Numbers

- %e, %f, and %g: reading and writing single-precision floating-point numbers.
- When reading a value of type double, put the letter l in front of e, f, or g:
  ```
  double d;
  scanf("%lf", &d);  
  ```
- Use l only in a scanf format string, NOT a printf string.
- In a printf format string, the e, f, and g conversions can be used to write either float or double values.
- When reading or writing a value of type long double, put the letter L in front of e, f, or g.

Use of char (character)

- Basic operations
  - Declaration: char c;
  - Assignment: c = 'a';
  - Reference: c = c + 1;
- Constants
  - Single-quoted character (only one)
  - Special characters: \n (tab), \" (double quote), \' (single quote), \ (backslash)

Characters are Integers

- A char type represents an integer value from 0 to 255 (1 byte) or –128 to 127.
- A single quoted character is called a “character constant”.
- C characters use ASCII representation:
  - 'A' = 65 ... 'Z' = 'A' + 25 = 90
  - 'a' = 97 ... 'z' = 'a' + 25 = 122
  - '0' = 48, '9' = '0' + 9
- Never make assumptions of char values
  - Always write 'A' instead of 65
ASCII Table

American Standard Code for Information Interchange
A standard way of representing the alphabet, numbers, and symbols (in computers)

wikipedia on ASCII

Escape Sequences

• A character constant is usually one character enclosed in single quotes.
• Escape sequences provide a way to represent special characters that are invisible (nonprinting) or can’t be entered from the keyboard.
• There are two kinds of escape sequences: character escapes and numeric escapes.

Character Escapes

• A complete list of character escapes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alert (bell)</td>
<td>\a</td>
</tr>
<tr>
<td>Backspace</td>
<td>\b</td>
</tr>
<tr>
<td>Form feed</td>
<td>\t</td>
</tr>
<tr>
<td>New line</td>
<td>\n</td>
</tr>
<tr>
<td>Carriage return</td>
<td>\r</td>
</tr>
<tr>
<td>Horizontal tab</td>
<td>\t</td>
</tr>
<tr>
<td>Vertical tab</td>
<td>\v</td>
</tr>
<tr>
<td>Backslash</td>
<td>\</td>
</tr>
<tr>
<td>Question mark</td>
<td>?</td>
</tr>
<tr>
<td>Single quote</td>
<td>`'</td>
</tr>
<tr>
<td>Double quote</td>
<td>`&quot;</td>
</tr>
</tbody>
</table>

Numeric Escapes

• Character escapes
  o don’t exist for all nonprinting ASCII characters.
  o useless for representing characters beyond the basic 128 ASCII characters.
• Numeric escapes can represent any character.
• A numeric escape for a particular character uses the character’s octal or hexadecimal value.
• For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex.

Escape Sequences

• An octal escape sequence consists of the \ character followed by an octal number with at most three digits, such as \33 or \033.
• A hexadecimal escape sequence consists of \x followed by a hexadecimal number, such as \x1b or \x1B.
• The x must be in lower case, but the hex digits can be upper or lower case.

Escape Sequences

• When used as a character constant, an escape sequence must be enclosed in single quotes.
  o E.g., ‘\33’ (or ‘\x1b’) for decimal value 27.
• It’s often a good idea to use #define to give them names:
  #define ESC ‘\33’
• Escape sequences can also be embedded in strings.
**ctype.h**

- The `ctype` header is used for testing and converting characters.
- To use character-handling functions in `ctype` header, a program need to have
  #include `<ctype.h>`
- For example, `toupper` returns the upper-case version of its argument.
  ```c
  ch = toupper(ch);
  ```

- These functions take an integer (not necessarily a char!) and return 0 or 1.
  ```c
  int isdigit(int c);
  int isalpha, isalnum, isspace, islower, isupper;
  int tolower/toupper (int c);
  ```

**Reading and Writing Characters Using `scanf` and `printf`**

- 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 */
  ```
- To force `scanf` to skip white-space characters.
- To force `scanf` to skip white-space before reading a character, put a space in its format string just before `%c`:
  ```c
  scanf(" %c", &ch);
  ```
- Since `scanf` doesn’t skip white-space before reading a char, it’s easy to detect the end of an input line:
  ```c
  do {
      scanf("%c", &ch);
  } while (ch != \n);
  ```
- When `scanf` is called the next time, it will read the first character on the next input line.

**getchar and putchar**

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
  - To write a character:
    ```c
    putchar(ch);
    ```
  - To read a character:
    ```c
    ch = getchar();
    ```
  - `getchar` returns an int value rather than a char value, so `ch` will often have type int.
  - Like `scanf`, `getchar` doesn’t skip white-space characters as it reads a character.

- 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);
  ```
getchar and putchar

- Moving the call of getchar into the controlling expression allows us to condense the loop:
  ```c
  while ((ch = getchar()) != '\n')
  ;
  ``
- The ch variable isn’t even needed; we can just compare the return value of getchar with the new-line character:
  ```c
  while (getchar() != '\n')
  ;
  ```

- getchar is useful in loops that skip characters as well as loops that search for characters:
  ```c
  while ((ch = getchar()) == ' ')
  ;
  ``
- When the loop terminates, ch will contain the first nonblank character that getchar encountered.

getchar and putchar

- Be careful when mixing getchar and scanf.
- scanf has a tendency to leave behind characters that it has “peeked” at but not read, including the new-line character:
  ```c
  printf("Enter an integer: ");
  scanf("%d", &i);
  printf("Enter a command: ");
  command = getchar();
  scanf will leave behind any characters that weren’t consumed during the reading of i, including (but not limited to) the new-line character.
- getchar will fetch the first leftover character.

scanf Notes

- Beware of combining scanf and getchar().
- Use of multiple specifications can be both convenient and tricky.
  - Experiment!
- Remember to use the return value for error checking.

```c
int main() {
  char c;
  c = getchar();
  printf("Character >%c< has the value %d.\n", c, c);
  return 0;
}
```

The sizeof Operator

- The value of the expression
  ```c
  sizeof ( type-name )
  ```
  is an unsigned integer representing the number of bytes required to store a value belonging to type-name.
- sizeof(char) is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, sizeof(int) is normally 4.

The sizeof Operator

- The sizeof operator can also be applied to constants, variables, and expressions in general.
  - If i and j are int variables, then sizeof(i) is 4 on a 32-bit machine.
  - What about sizeof(i + j)?
## Type Conversions

<table>
<thead>
<tr>
<th>Implicit conversions</th>
<th>Explicit conversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>casting</td>
</tr>
<tr>
<td>assignment</td>
<td><code>int x;</code></td>
</tr>
<tr>
<td>function parameters</td>
<td><code>x = (int) 4.0;</code></td>
</tr>
<tr>
<td>function return type</td>
<td></td>
</tr>
<tr>
<td>promotion if possible</td>
<td></td>
</tr>
</tbody>
</table>