CSCI 2132: Software Development

Basic Types in C

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Basic Types in C

Integer types:

- `int`
- `long` (64 bits)
- `short` (32 bits)
- `signed` (MSB = sign)
- `unsigned` (non-negative)

Most significant bit

Floating point types:

- `float` (32 bits)
- `double` (64 bits)
- `long double`

Boolean types:

- `int`
- (`bool`)

Character types:

- `char`

Examples of Integer Types

Order is not important:

- `long signed int` = `signed long int`

Default is `signed` and `short`:

- `int` = `signed short int`

We can omit `int`:

- `long` = `long int`
- `short` = `short int`

Range of Integers

Typically `int` = one machine word (not on current machines)

"Word":

- Old 16-bit machine: 16 bits = 2 bytes
- Less old 32-bit machine: 32 bits = 4 bytes
- Current 64-bit machine: 64 bits = 8 bytes (but `int` uses only 4 bytes)

16-bit `unsigned int`: `0..65,535` ($= 2^{16} - 1$)

16-bit `signed int`: `-32,768..32,767` ($-2^{15} .. 2^{15} - 1$)

- 2's complement:
  - `0...` = positive number
  - `1...` = negative number
  - Negative number: `1x...` = –(`1x...` `^ 1111 1111 1111 1111 + 1`)

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
                    = - 0000 0000 0000 0101
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
                    = - 0000 0000 0000 0101
                    = - 5
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
                    = - 0000 0000 0000 0101
                    = - 5

1000 0000 0000 0000 = -(0111 1111 1111 1111 + 1)
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
                    = - 0000 0000 0000 0101
                    = - 5

1000 0000 0000 0000 = -(0111 1111 1111 1111 + 1)
                    = - 1000 0000 0000 0000
```

2's Complement Examples

(16 bits)

```
0000 1001 1101 1111 = 2,271

1111 1111 1111 1011 = -(0000 0000 0000 0100 + 1)
                    = - 0000 0000 0000 0101
                    = - 5

1000 0000 0000 0000 = -(0111 1111 1111 1111 + 1)
                    = - 1000 0000 0000 0000
                    = - 32,768
```

More on Integer Sizes

C standard does not specify integer sizes
(implementation-dependent)

More on Integer Sizes

C standard does not specify integer sizes
(implementation-dependent)

Compare: Java standard requires `int` = 32 bits

Java: JVM, **C:** runs directly on hardware, use hardware's `int` type

How to learn the range of integers on your machine?

- `#include <limits.h>`
- `INT_MIN` = minimum integer, `INT_MAX` = maximum integer
- `sizeof(int)` = number of bytes used for integer

Sometimes the compiler needs help to recognize long int constants:

- `15L` = 15 as a `long int` (64 bits)

Floating Point Numbers

`float` = single precision (32 bits)

`double` = double precision (64 bits)

`long double` = extended precision (128 bits)

Representation is implementation-dependent

- Same reason as for `int` types: Use hardware support
- Most hardware/compilers support IEEE 754

Character Type

`char`

- C: 8 bytes (extended ASCII)
- Java: 16 bytes (UTF-16)

`char` can be `signed` or `unsigned` to represent 8-bit integers

Example: What does the following code do?

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

Reading Characters

- `scanf`:
  - Does not ignore space
  - The following are not equivalent:

```
scanf(“%c”, &ch);
scanf(“ %c”, &ch);
```

Reading Characters

- `scanf`:
  - Does not ignore space
  - The following are not equivalent:

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

- `getchar`/`putchar` used for reading/writing characters:

```
int ch = getchar();
putchar(ch);
```

Reading Characters

- `scanf`:
  - Does not ignore space
  - The following are not equivalent:

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

- `getchar`/`putchar` used for reading/writing characters:

```
int ch = getchar();
putchar(ch);
```

  - **Note:** Return type of `getchar` is `int`!
  - **Reason:** Return `EOF` (–1) on end of file

Code Example

```
#include <stdio.h>

int main() {
    int ch;
    while ((ch = getchar()) != EOF && ch != '\n') {
        if ('a' <= ch && ch <= 'z')
            ch = ch - 'a' + 'A';
        putchar(ch);
    }
    putchar('\n');
    return 0;
}
```

Are these parentheses needed?

Type Conversion

```
int x = 3.4;
```

- x =

Type Conversion

```
int x = 3.4;
```

- x = 3

- Works in C, not in Java.

- C uses implicit type conversion for
  - Operands
  - Assignment

Type Conversion of Operands

Operands are converted to the “narrowest” type that accommodates both.

Example:

```
float f;
double d;
int i;
d = d + f;
f = f + i;
```

f is promoted to double here.

i is promoted to float here.

Type Conversion in Assignments

The right side is converted to the type of the left side.

Example: `int i = 8.92;`

Type Casting (Explicit Type Conversion)

Syntax: `(type) expression`

Example: What does this compute?

```
float f, frac_part;
frac_part = f - (int) f;
```

Another Example

```
float quotient;
int dividend = 5;
int divisor = 4;
```

What do these four statements compute?

```
quotient = dividend / divisor;
quotient = (float) dividend / divisor;
quotient = (float) (dividend / divisor);
quotient = 1.0f * dividend / divisor;
```

“Type Definitions”
(Really, Defining Type Aliases)

```
typedef typename alias
```

Example:

```
typedef int Bool;
Bool flag;
```

- This only defines an alias; `int` and `Bool` can be used interchangeably and can be mixed.

Type aliases are useful to make code more readable, especially with complex types:

```
typedef unsigned int banner_number;
typedef int (*binary_operator)(int, int);
```

The sizeof Operator

Many C types have implementation-defined sizes.

`sizeof(type)` = number of bytes needed to represent values of type `type`

`sizeof(var)` = number of bytes occupied by variable `var`

Examples:

```
printf("%d\n", sizeof(char));     /* prints 1 */
int i; printf("%d\n", sizeof(i)); /* prints 4 */
```