Basic Type

Basic Type

Introduction

Integers

- Long, short, unsigned

Floating

- Double, float

Char

Conversion

Typedef

Sizeof

Basic Type

Introduction - Integer

Integers

- The leftmost bit of a signed integer (i.e. sign bit) is 0 if the number is a integer greater than or equal to zero, 1 if it's negative
- The largest value is $2^{(n-1)} - 1$, where n is the number of bits

Unsigned

- The largest value is $2^{(n)} - 1$, where n is the number of bits

By default, integer variables are signed in C

Basic Type

Introduction - Integer

Integer types come in different sizes

- The int type is usually 32 bits, but may be 16 bits on older CPU
- C provides long and short integer type

short int
unsigned short int

int
unsigned int

long int
unsigned long int

Basic Type

Introduction - Integer

On a 16-bit machine

Type	Smallest	Largest
short int	-32,768	32,767
unsigned short int	0	65,535
int	-32,768	32,767
unsigned int	0	65,535
long int	-2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295

Basic Type

Introduction - Integer

On a 32-bit machine

Type	Smallest	Largest
short int	-32,768	32,767
unsigned short int	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
long int	-2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295

Basic Type

Introduction - Integer

On a 64-bit machine

Type	Smallest	Largest
short int	-32,768	32,767
unsigned short int	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
long int	-2^{63}	$2^{63}-1$
unsigned long int	0	$2^{64}-1$

Basic Type

Introduction - Integer constants

Decimal (base 10)

- 15, 255, 16500

Octal (base 8)

- 027, 01364, 07777

Hexadecimal (base 16)

- 0xf, 0x9f, 0x5adf, 0xFF

Basic Type

Introduction - Integer constants

Octal numbers use only the digit 0 through 7

Each position in an octal number represents a power of 8

- The octal number 237 represents the decimal number is

$$2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$$

A hexadecimal (or hex) number is written using the digits 0 through plus the letters A through F, which stand for 10 through 15, respectively

- The hex number 1AF represents the decimal number is

$$1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$$

- The letters in a hexadecimal constant may be either upper or lower case

0xff 0xfF 0xFF 0Xff 0XFf

Basic Type

Introduction - Integer constants

To force the compiler to treat a constant as a long integer, just follow it with the letter L (or l)

15L 0377L 0x7fffL

To indicate that a constant is unsigned, put the letter U (or u) after it

15U 0377U 0x7fffU

L and U can be used in combination

0xffffffUL

Basic Type

Introduction - Overflow

When arithmetic operations are performed on integers, it's possible that the result will be too large to represent

- For example, when an arithmetic operation is performed on two int values, the result must be able to be represented as an int
- If the result can't be represented as an int, we say that ***overflow*** has occurred

The behavior when integer overflow occurs depends on whether the operands were signed or unsigned

- 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

Basic Type

Introduction - Reading and Writing Integers

Reading and writing unsigned, short, and long integers requires new conversion specifiers

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\n", u);      // writes u in base 10
scanf("%o", &u);        // reads  u in base  8
printf("%o\n", u);      // writes u in base  8
scanf("%x", &u);        // reads  u in base 16
printf("%x\n", u);      // writes u in base 16
```

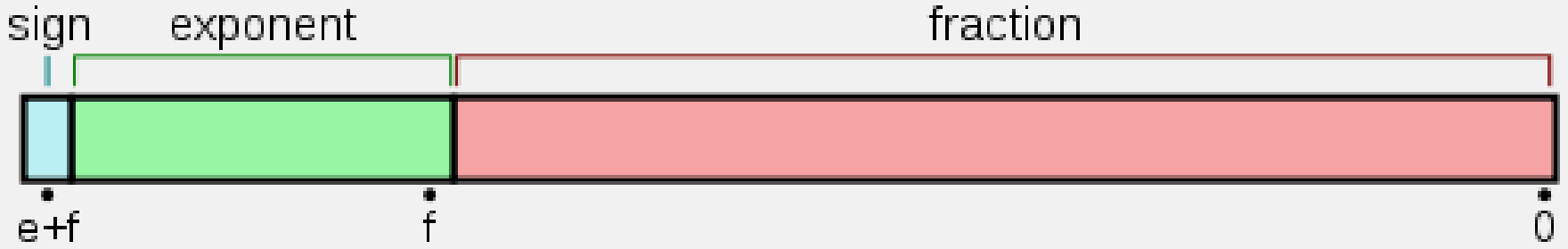
```
unsigned int u;

scanf("%u", &u);        // reads  u in base 10
printf("%u\n", u);      // writes u in base 10
scanf("%o", &u);        // reads  u in base  8
printf("%u\n", u);      // writes u in base  8
scanf("%x", &u);        // reads  u in base 16
printf("%u\n", u);      // writes u in base 16
```

Basic Type

Introduction - Floating

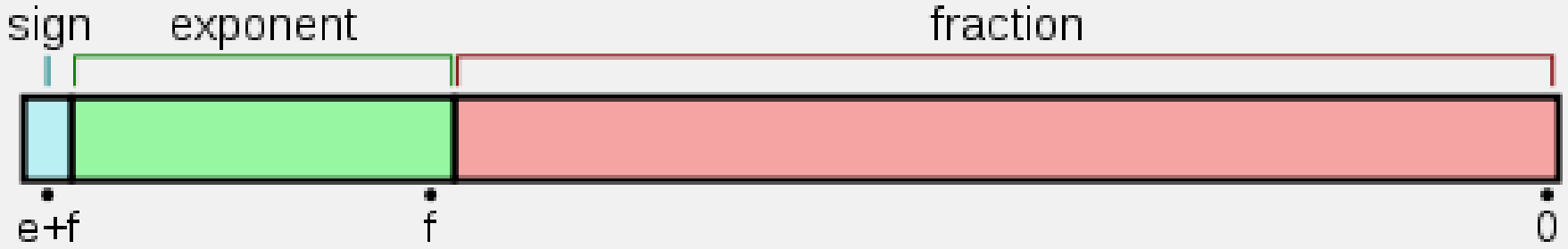
C provides three floating types, corresponding to different floating-point formats

- Float
  - Single-precision floating-point
- Double
  - Double-precision floating-point
- Long double
  - Extended-precision floating-point



Basic Type

Introduction - Floating



If 32 bits, using vector $[x_{31}, x_{30}, \ldots, x_1, x_0]$

x_{31} is the sign bit (s), $[x_{30}, x_{29}, \ldots, x_{23}]$ is the exponent bits (exp), and $[x_{22}, x_{29}, \ldots, x_0]$ is the fraction bits (M)

$n = (-1)^s \times M \times 2^{exp-127}$

Basic Type

Introduction - Floating

$$n = (-1)^s \times M \times 2^{exp-127}$$

ex. -12.625 presented by IEEE-754, single precision (32 bits)

First: convert 12.625 (decimal) to a value (binary)

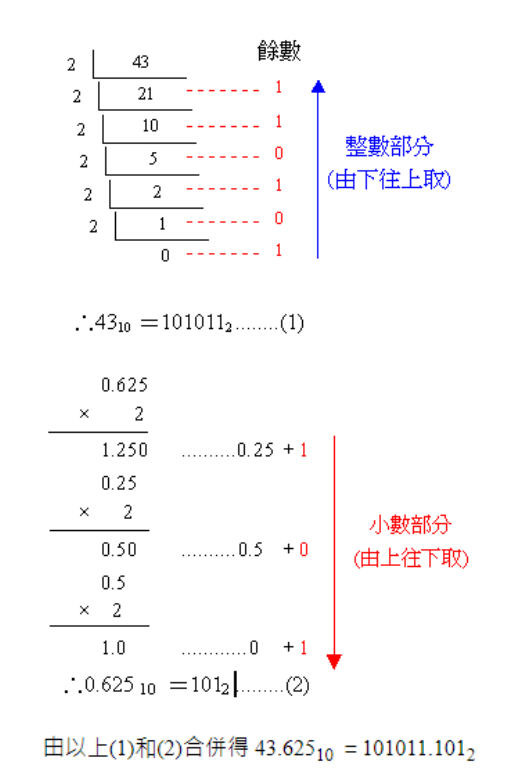
12.625 => 1100.101 = 1.100101 × 23

Second: calculate the exponent

127 + 3 = 130 => 10000010

Third: insert value to the floating format

S	E	M
1	10000010	10010 1000 0000 0000 0000 0



Reference

Basic Type

Introduction - Char

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

A variable of type char can be assigned any single character

```
char ch;

ch = 'A';
ch = 'a';
ch = '0';
ch = ' ';
```

```
char ch;
int i;

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

Basic Type

Introduction - Char

Characters can be compared, just as numbers can

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

It also can be employed by the for statement

```
for (ch = 'a';  ch <= 'z'; ch++)
{
    …
}
```

Basic Type

Introduction - Char

Calling C's toupper library function is a fast and portable way to convert case

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

toupper returns the upper-case version of its argument

Programs that call *toupper* needs the following code

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

The C library provides many other useful character-handling functions

Basic Type

Introduction - Reading and Writing Characters

Using *scanf* and *printf*

- The %c conversion specification

```
char ch;

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

scanf doesn't 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

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

space

Basic Type

Introduction - Reading and Writing Characters

Since scanf doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character

A loop that reads and ignores all remaining characters in the current input line

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

When scanf is called the next time, it will read the first character on the next input line

Basic Type

Introduction - Reading and Writing Characters

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();
```

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

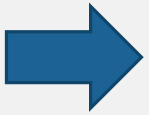
Basic Type

Introduction - Reading and Writing Characters

Using *getchar* and *putchar* (rather than *scanf* and *printf*) saves execution time

- *getchar* and *putchar* are much simpler than *scanf* and *printf*, which are designed to read and write many kinds of data in a variety of formats

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



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

Basic Type

Introduction - Char

Write a program to determine the length of a Message

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

Basic Type

Introduction - Conversion

When a computer perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way

If operands with different types are mixed in expressions, C compiler will create instructions that different types will be adjusted into same types to evaluate the expression

- If adding a *short* type variable and a *int* type variable, the compiler will convert the *short* type variable into *int* type
- If adding a *int* type variable and a *float* type variable, the compiler will convert the *int* type variable into *float* type

Basic Type

Introduction - Conversion

An example of the usual conversions:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;          // c is converted to int
i = i + s;          //s is converted to int
u = u + i;          //i is converted to unsigned int
l = l + u;          //u is converted to long int
ul = ul + l;        //l is converted to unsigned long int
f = f + ul;         //ul is converted to float
d = d + f;          //f is converted to double
ld = ld + d;        //d is converted to long double
```

Basic Type

Introduction - Conversion

Although the implicit conversions are convenient, we also need a greater degree of control over type conversion

How to perform the type conversion?

- Using cast operator (explicit conversions)

(*type-name*) *expression*

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

- If adding a *int* type variable and a *float* type variable, the compiler will convert the *int* type variable into *float* type

Basic Type

Introduction - Type definitions

The #define directive can be used to create a "Boolean type" macro

```
#define Bool int
```

Another way is type definition

```
typedef int Bool;
Bool flag;      //same as int flag;
```

Advantages

- Make a program easier to modify

```
typedef int dollars;          typedef long dollars;
                    dollars money;
                    money = 100000;
```

Basic Type

Introduction - Sizeof Operator

The operator is to represent the number of bytes required to store a value belonging to *type-name*

```
sizeof (type-name)
sizeof(char); // will return 1
sizeof(int);    // will return 4 in a 32-bit machine
```

It can also be applied to constants, variables, and expressions in general

```
int x, y;
sizeof(x);      // will return 4 in a 32-bit machine
sizeof(x+y);  // will return 4 in a 32-bit machine
```

Basic Type

Introduction

Write a program to convert 12-hour time into 24-hour time

```
Enter a 12-hour time: 9:11 PM
Equivalent 24-hour time: 21:11
```