Programming with C++

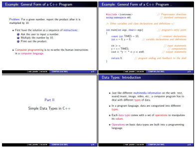
COMP2011: C++ Basics II

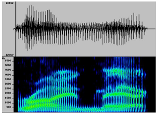
Cecia Chan
Cindy Li
Brian Mak

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Part I

More Basic Data Types in C++





C++ Basic Types

Types	Common Size(#bytes on a 32-bit machine)	Value Range
bool	1	{ true, false }
char	1	[-128, 127]
short	2	[-32768, 32767]
int	4	$[-2^{31}, 2^{31}-1]$
long	4	$[-2^{31}, 2^{31}-1]$
float	4	± [1.17549E-38, 3.40282E+38]
double	8	± [2.22507E-308, 1.79769E+308]

- Not all numbers of a type can be represented by a computer.
- It depends on how many bytes you use to represent it: with more bytes, more numbers can be represented.

Find Out Their Sizes using sizeof

```
#include <iostream>     /* File: value.cpp */
using namespace std;

int main()
{
    cout << "sizeof(bool) = " << sizeof(bool) << endl;
    cout << "sizeof(char) = " << sizeof(char) << endl;
    cout << "sizeof(short) = " << sizeof(short) << endl;
    cout << "sizeof(int) = " << sizeof(int) << endl;
    cout << "sizeof(long) = " << sizeof(long) << endl;
    cout << "sizeof(long long) = " << sizeof(long long) << endl;
    cout << "sizeof(float) = " << sizeof(float) << endl;
    cout << "sizeof(double) = " << sizeof(double) << endl;
    cout << "sizeof(long double) = " << sizeof(long double) << endl;

    return 0;
}
```

Size of Basic Types on 2 Computers

on a 32-bit machine

```
sizeof(bool) = 1
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 4
sizeof(long) = 4
sizeof(long long) = 8
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 12
```

on a 64-bit machine

```
sizeof(bool) = 1
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 4
sizeof(long) = 8
sizeof(long long) = 8
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 16
```

- Note that the figures may be different on your computer.
- A 32(64)-bit machine uses CPUs of which the data bus width and memory address width are 32 (64) bits.

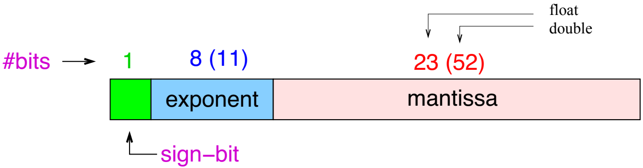
Integers

- Type names: short (int), int, long (int), long long (int)
- Their sizes depend on the CPU and the compiler.
- ANSI C++ requires:
  size of short $\leq$ size of int $\leq$ size of long $\leq$ size of long long
- e.g., What are the numbers that can be represented by a 2-byte short int?
- Each integral data type has 2 versions:
  - signed version: represents both +ve and -ve integers.
    e.g. signed short, signed int, signed long
  - unsigned version: represents only +ve integers.
    e.g. unsigned short, unsigned int, unsigned long
- signed versions are the default.
- Obviously unsigned int can represent 2 times more +ve integers than signed int.

Floating-Point Data Types

- Floating-point numbers are used to represent real numbers and very large integers (which cannot be held in long long).
- Type names:
  - float for single-precision numbers.
  - double for double-precision numbers.
- Precision: For decimal numbers, if you are given more decimal places, you may represent a number to higher precision.
  - for 1 decimal place: 1.1, 1.2, 1.3, ... etc.; can't get 1.03.
  - for 2 decimal places: 1.01, 1.02, 1.03, ... etc.; can't get 1.024.
- In scientific notation, a number has 2 components. e.g., 5.16E-02
  - mantissa: 5.16
  - exponent: -2
- More mantissa bits $\Rightarrow$ higher precision.
- More exponent bits $\Rightarrow$ larger real number.

Floating-Point Data Types ..



- Many programming language uses the IEEE 754 floating-point standard.
- Binary Representation of mantissa: e.g.

$$1.011_2 = 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

- Binary Representation of exponent: signed integer
- All floating-point data types in C++ are signed.
- ANSI C++ requires: size of float $\leq$ size of double

Question: Can every real number be represented by float in C++?

Integer Arithmetic and Floating-Point Arithmetic

- Arithmetic expressions involving *only* integers use integer arithmetic.
- Arithmetic expressions involving *only* floating-point numbers use floating-point arithmetic.
- For $+$, -, $\times$ operations, results should be what you expect.
- However, integer division and floating-point division may give different results. e.g.,
  - $10/2 = 5$ and $10.0/2.0 = 5.0$
  - $9/2 = 4$ and $9.0/2.0 = 4.5$
  - $4/8 = 0$ and $4.0/8.0 = 0.5$

Example: Continuously Halving a float Number

```cpp
#include <iostream>      /* File: halving-float.cpp */
using namespace std;

int main()
{
    int HALF = 2;        // Reduce the number by this factor
    int count = 0;       // Count how many times that x can be halved
    float x;             // Number to halve

    cout << "Enter a positive number: ";
    cin >> x;

    while (x > 0.1)
    {
        cout << "Halving " << count++ << " time(s); "
             << "x = " << x << endl;
        x /= HALF;
    }

    return 0;
}
```

```
Enter a positive number: 7
Halving 0 time(s); x = 7
Halving 1 time(s); x = 3.5
Halving 2 time(s); x = 1.75
Halving 3 time(s); x = 0.875
Halving 4 time(s); x = 0.4375
Halving 5 time(s); x = 0.21875
Halving 6 time(s); x = 0.109375
```

Example: Continuously Halving an int Number

```
#include <iostream>      /* File: halving-int.cpp */
using namespace std;

int main()
{
    int HALF = 2;        // Reduce the number by this factor
    int count = 0;       // Count how many times that x can be halved
    int x;               // Number to halve

    cout << "Enter a positive number: ";
    cin >> x;

    while (x > 0.1)
    {
        cout << "Halving " << count++ << " time(s); "
             << "x = " << x << endl;
        x /= HALF;
    }

    return 0;
}
```

Example: Continuously Halving an int Number ..

```
Enter a positive number: 7
Halving 0 time(s); x = 7
Halving 1 time(s); x = 3
Halving 2 time(s); x = 1
```

Boolean Data Type

- Type name: bool.
- Used to represent the truth value, true or false of logical (boolean) expressions like:

$$a > b \qquad x + y == 0 \qquad \text{true \&\& false}$$

- Since C++ evolves from C, C++ follows C's convention:
  - zero may be interpreted as false.
  - non-zero values may be interpreted as true.
- However, since internally everything is represented by 0's and 1's,
  - false is represented as 0.
  - true is represented as 1.
- Even if you put other values to a bool variable, its internal value always is changed back to either 1 or 0.

Example: Output Boolean Values

```cpp
#include <iostream>     /* File: boolalpha.cpp */
using namespace std;

int main()
{
    bool x = true;
    bool y = false;

    // Default output format of booleans
    cout << x << " && " << y << " = " << (x && y) << endl << endl;

    cout << boolalpha;   // To print booleans in English
    cout << x << " && " << y << " = " << (x && y) << endl << endl;

    cout << noboolalpha; // To print booleans in 1 or 0
    cout << x << " && " << y << " = " << (x && y) << endl;

    return 0;
}
```

Example: Use of bool Variables

```
#include <iostream>      /* File: bool-blood-donation.cpp */
using namespace std;

int main()
{
    char donor_blood_type, recipient_blood_type;
    bool exact_match, match_all;

    cout << "Enter donor's bloodtype: A, B, C (for AB), and O. ";
    cin >> donor_blood_type;
    cout << "Enter recipient's bloodtype: A, B, C (for AB), and O. ";
    cin >> recipient_blood_type;

    exact_match = (donor_blood_type == recipient_blood_type);
    match_all = (donor_blood_type == 'O');

    if (exact_match || match_all)
        cout << "Great! A donor is found!" << endl;
    else
        cout << "Keep searching for the right donor." << endl;

    return 0;
}
```

- Overflow: occurs when a data type is used to represent a number larger than what it can hold. e.g.
  - if you use a short int to store HK's population.
  - when a short int has its max value of 32767, and you want to add 1 to it.
- Underflow: occurs when a data type is used to represent a number smaller than what it can hold. e.g.
  - use an unsigned int to store a -ve number.

Underflow and Overflow in Floating-Point Data Types

- Underflow: when the -ve exponent becomes too large to fit in the exponent field of the floating-point number.
- Overflow: when the +ve exponent becomes too large to fit in the exponent field of the floating-point number.
- To prevent these from happening, use double if memory space allows.
- In fact, all floating literals (e.g., 1.23) is treated as double unless explicitly specified by a suffix (e.g., 1.23f).

Part II

Type Checking and Type Conversion

Type Checking and Coercion

Analogy:

Blood Types	
Receiver	**Donor**
A	A, O
B	B, O
AB	A, B, AB, O
O	O

- For most languages, data types have to be matched during an operation $\Rightarrow$ type checking.
- However, sometimes, a type is made compatible with a different type $\Rightarrow$ coercion.

Operand Coercion

Coercion is the automatic conversion of the data type of operands during an operation.

- Example: $3 + 2.5 \Rightarrow$ int + double.
- The C++ compiler will automatically change it to

  $3.0 + 2.5 \Rightarrow$ double + double
- Thus, the integer 3 is coerced to the double 3.0.

Example: Convert a Small Character to Capital Letter

```
char small_y, big_y;
cin >> small_y;                  // Character in small case
big_y = small_y + 'A' - 'a';  // Character in big case
```

Here big_y, small_y, 'A', and 'a' are "coerced" by "promoting" it to int before addition. The result is converted back (or coerced) to char.

Priority Rules for the Usual Arithmetic Conversions for Binary Operations

- If either operand is of type long double, convert the other operand also to long double.
- If either operand is of type double, convert the other operand also to double.
- If either operand is of type float, convert the other operand also to float.
- Otherwise, the integral promotions shall be performed on both operands.
  - Similar rules are used for integral promotion of the operands.
  - Compute using integer arithmetic.

Question: What is the result of 3/4?

Automatic Type Conversion During Assignment

Examples

```
float x = 3.2;          // Initialize x with 3.2 by assignment
double y = 5.7;         // Initialize y with 5.7 by assignment

short k = x;            // k = ?
int n;
n = y;                  // n = ?
```

- Since float|double can hold numbers bigger than short | int, the assignment of *k* and *n* in the above program will cause the compiler to issue a warning — not an error.

Compiler Warnings

a.cpp:9: warning: converting to 'short int' from 'float'
a.cpp:11: warning: converting to 'int' from 'double'

Automatic Type Conversion During Assignment ..

- A narrowing conversion changes a value to a data type that might not be able to hold some of the possible values.
- A widening conversion changes a value to a data type that can accommodate any possible value of the original data.
- C++ uses truncation rather than rounding in converting a float|double to short | int | long.

Manual Type Conversion (Casting)

```
int k = 5;
int n = 2;
float x = n/k;          // What is the value of x?
```

- In the above example, one can get x = 0.4 by manually converting *n* and/or *k* from int to float|double.

Syntax: static_cast for manual type casting

static_cast<data-type> (value)

- No more warning messages on narrowing conversion.

```
int k = 5, n = 2;
float x = static_cast<double>(n)/k;
float y = n/static_cast<double>(k);
float z = static_cast<double>(n)/static_cast<double>(k);
```

Part III

Constants

Literal Constants

- Constants represent fixed values, or permanent values that cannot be modified (in a program).
- Examples of literal constants:
  - char constants: 'a', '5', '\n'
  - string constants: "hello world", "don't worry, be happy"
  - int constants: 123, 456, -89
  - double constants: 123.456, -2.90E+11

Symbolic Constants

- A symbolic constant is a named constant with an identifier name.
- The rule for identifier names for constants is the same as that for variables. However, by convention, constant identifiers are written in capital letters.
- A symbolic constant must be defined and/or declared before it can be used. (Just like variables or functions.)
- Once defined, symbolic constants cannot be changed!

Syntax: Constant Definition

const <data-type> <identifier> = <value> ;

Example

```
const char BACKSPACE = '\b';
const float US2HK = 7.80;
const float HK2RMB = 0.86;
const float US2RMB = US2HK * HK2RMB;
```

Why Symbolic Constants?

Compared with literal constants, symbolic constants are preferred because they are

- more readable. A literal constant does not carry a meaning. e.g. the number 320 cannot tell you that it is the enrollment quota of COMP2011 in 2015.

```
const int COMP2011_QUOTA = 320;
```

- more maintainable. In case we want to increase the quota to 400, we only need to make the change in one place: the initial value in the definition of the constant COMP2011_QUOTA.

```
const int COMP2011_QUOTA = 400;
```

- type-checked during compilation.

Remark: Unlike variable definitions, memory is not allocated for constant definitions with only few exceptions.

Example: Use of Symbolic Constants

```cpp
#include <iostream>     /* File: symbolic-constant.cpp */
#include <cmath>        // For calling the ceil() function
using namespace std;

int main()
{
    const int COMP2011_QUOTA = 320;
    const float STUDENT_2_PROF_RATIO = 100.0;
    const float STUDENT_2_TA_RATIO = 40.0;
    const float STUDENT_2_ROOM_RATIO = 100.0;

    cout << "COMP2011 requires "
         << ceil(COMP2011_QUOTA/STUDENT_2_PROF_RATIO)
         << " instructors, "
         << ceil(COMP2011_QUOTA/STUDENT_2_TA_RATIO)
         << " TAs, and "
         << ceil(COMP2011_QUOTA/STUDENT_2_ROOM_RATIO)
         << " classrooms" << endl;

    return 0;
}
```