Introductory Robot Programming

ENPM809Y

Reading Material 01 – C++ Data Types

Zeid Kootbally

zeidk@umd.edu

Fall 2020

Overview I

Overview II

C++ Data Types

- The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones.
- Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name.
- What the program needs to be aware of is the kind of data stored in the variable.
- It is not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they do not occupy the same amount of memory.

C++ Data Types | Size of Variables

- The sizeof operator determines the size (in bytes) of a type or variable.
  - sizeof(type): returns size in bytes of the object representation of type.
  - sizeof expression: Returns size in bytes of the object representation of the type that would be returned by expression, if evaluated.

```
float f_var;

std::cout << sizeof(int) << std::endl;    //--4
std::cout << sizeof(double) << std::endl; //--8
std::cout << sizeof(f_var) << std::endl;  //--4
std::cout << sizeof f_var << std::endl;   //--4
```

C++ Data Types | Fundamental Data Types

- Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:
  - Character types: They can represent a single character, such as `'A'` or `'$'`. The most basic type is `char`, which is a one-byte character. Other types are also provided for wider characters.
  - Integral types: They can store a whole number value, such as 7 or 1024. They exist in a variety of sizes, and can either be `signed` or `unsigned`, depending on whether they support negative values or not.
  - Floating-point types: They can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.
  - Boolean type: The Boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

C++ Data Types | Simple Quotation Mark

- Numeric literals of more than a few digits are hard to read.

```
int a{1334872343};
```

- C++14 introduced Simple Quotation Mark as a digit separator, in numbers and user-defined literals.
- This can make it easier for human readers to parse large numbers.

```
int a{1'334'872'343};
```

- The position of the single quotes is irrelevant to the compiler. All the following are equivalent:

```
int a1{1'334'872'343};
int a2{13'34'87'23'43};
int a3{1'33'4872'343};
int a4{1334'872'34'3};
```

Integral Data Types

- The term integral data type in programming languages refers to any data type that is one of the integer types in the language. The word integral, in this case, is an adjective whose dictionary definition is "of or denoted by an integer". It has nothing to do with the integral used in calculus.
- An integer is any number that is not either a decimal or a fraction. However, both 2.000 and 2/2 are integers because they can be simplified into non-decimal and non-fractional numbers, this includes negative numbers.

Integral Data Types | Modifiers

- A signedness modifier is used to specify positive and negative numbers.
- A size modifier is used to specify the range of a data type.
- You can mix signedness and size modifiers but only one of each group can be present in a type name.
    - Signedness modifiers:
        - `signed` indicates that a variable can hold negative and positive values (this is the default one used if omitted).
        - `unsigned` indicates a variable that can hold only positive numbers.
    - Size modifiers:
        - `short` is used for values of at least 16 bits.
        - `long` is used for values of at least 32 bits.
        - `long long` is used for values of at least 64 bits (since C++11).

Integral Data Types | Range of Data Types

- For `signed` types (positive and negative numbers):
  - minimum range = $-2^{\text{bits}-1}$, with bits = size of a data type.
  - maximum range = $2^{\text{bits}-1} - 1$, with bits = size of a data type.
  - Example: A `short` variable takes at least 2 bytes (16 bits) in size. The minimum and maximum limits for a `signed short` are $-2^{16-1} = -32,768$ and $2^{16-1} - 1 = 32,767$, respectively.
- For `unsigned` types (only positive numbers):
  - minimum range = 0
  - maximum range = $2^{\text{bits}} - 1$, with bits = size of a data type.
  - Example: A `short` variable takes at least 2 bytes (16 bits) in size. The minimum and maximum limits for a `unsigned short` are 0 and $2^{16} - 1 = 65,535$, respectively.

Integral Data Types | Range of Data Types

- The range of some data types depends on the compiler and on the operating system. However, there is always a guaranteed size for all data types. That is, regardless of the compiler or the OS, these data types will have at least a known size (in bytes).
- The range of each data type presented in the slides was retrieved using Ubuntu 18.04, 64 bits and you may get different results.
- To get the minimum and maximum limits of a data type you can use specific pre-defined constants (macros) defined in the climits header (`#include<climits>`). Each slide discussing a data type also includes the constants for minimum and maximum limits. Please see here for a complete list of constants which provide the range (min and max) for fundamental data types.

Integral Data Types | short

- `short` was designed to provide integer values of smaller range than an `int` data type.
- `short` is 2 bytes on my machine.
- Guaranteed size: Not smaller than `char`. At least 16 bits (2 bytes).
- Other notations:
  - `short int`
  - `signed short`
  - `signed short int`
- Range: `[SHRT_MIN, SHRT_MAX]`
  - `SHRT_MIN` = `-32768`
  - `SHRT_MAX` = `32767`
- Uses:

```
short a{3'423};
short int a{3'423};
signed short a{3'423};
signed short int a{3'423};
```

- Size:

```
sizeof(a)
sizeof(short)
sizeof(short int)
sizeof(signed short)
sizeof(signed short int)
```

Integral Data Types | unsigned short

- The `unsigned short` data type is used to store only positive `short` integer values.
- `unsigned short` is 2 bytes on my machine.
- Guaranteed size: At least 16 bits (2 bytes).
- Other notations:
  - `unsigned short int`
- Range: `[0, USHRT_MAX]`
  - USHRT_MAX = 65535

- Uses:

```cpp
unsigned short a{3'423};
```

```cpp
unsigned short int a{3'423};
```

- Size:

```cpp
sizeof(a)
sizeof(unsigned short)
sizeof(unsigned short int)
```

Integral Data Types | int

- The int data type is used to store positive and negative integer values.
- Guaranteed size: Not smaller than short and not larger than long. At least 32 bits (4 bytes).
- Other notations:
  - signed
  - signed int
- Range: [INT_MIN, INT_MAX]
  - INT_MIN = -2147483648
  - INT_MAX = 2147483647

- Uses:

```
int a{-3'423};
```

```
signed a{-3'423};
```

```
signed int a{-3'423};
```

- Size:

```
sizeof(a)
sizeof(int)
sizeof(signed)
sizeof(signed int)
```

Integral Data Types | unsigned int

- The `unsigned int` data type stores double more positive integer values than an `int` data type.
- `unsigned int` is 4 bytes on my machine.
- Guaranteed size: At least 32 bits (4 bytes).
- Other notations:
  - `unsigned`
- Range: `[0, UINT_MAX]`
  - `UINT_MAX` = `4294967295`

- Uses:

```
unsigned int a{3'423};
```

```
unsigned a{3'423};
```

- Size:

```
sizeof(a)
sizeof(unsigned int)
sizeof(unsigned)
```

Integral Data Types | long int

- The `long int` data type is used for large integers.
- `long int` is 8 bytes on my machine.
- Guaranteed size: Not smaller than `int` and not larger than `long long int`. At least 32 bits (4 bytes).
- Other notations:
  - `long`
  - `signed long`
  - `signed long int`
- Range: `[LONG_MIN, LONG_MAX]`
  - `LONG_MIN` = `-9223372036854775808`
  - `LONG_MAX` = `9223372036854775807`

- Uses:

```
long a{3'423};
long int a{3'423};
signed long a{3'423};
signed long int a{3'423};
```

- Size:

```
sizeof(a)
sizeof(long)
sizeof(long int)
sizeof(signed long)
sizeof(signed long int)
```

Integral Data Types | unsigned long int

- The `unsigned long int` data type is used for large positive integers.
- `unsigned long int` is 8 bytes on my machine.
- Guaranteed size: Not smaller than `unsigned int` and not larger than `unsigned long long int`. At least 32 bits (4 bytes).
- Other notations:
  - `unsigned long`
- Range: `[0, ULONG_MAX]`
  - `ULONG_MAX` = 18446744073709551615

- Uses:

```
unsigned long a{3'423};
```

```
unsigned long int a{3'423};
```

- Size:

```
sizeof(a)
sizeof(unsigned long)
sizeof(unsigned long int)
```

Integral Data Types | long long int

- The `long long int` data type is also used for large positive and negative integers.
- `long long int` is 8 bytes on my machine.
- Guaranteed size: Not smaller than `long int`. At least 64 bits (8 bytes).
- Other notations:
  - `signed long long int`
  - `signed long long`
  - `long long`
- Range: `[LLONG_MIN, LLONG_MAX]`
  - `LLONG_MIN` = `-9223372036854775808`
  - `LLONG_MAX` = `9223372036854775807`

- Uses:

```
long long int a{};
long long int a{};
signed long long int a{};
long long a{};
```

- Size:

```
sizeof(a)
sizeof(long long int)
sizeof(signed long long int)
sizeof(signed long long)
sizeof(long long)
```

Integral Data Types | unsigned long long int

- The `unsigned long long int` data type is also used for large positive integers.
- `unsigned long long int` is 8 bytes on my machine.
- Guaranteed size: Not smaller than `unsigned long int`. At least 64 bits (8 bytes).
- Other notations:
  - `unsigned long long`
- Range: `[0, ULLONG_MAX]`
  - ULLONG_MAX = 18446744073709551615

- Uses:

```
unsigned long long int a{};
```

```
unsigned long long a{};
```

- Size:

```
sizeof(a)
sizeof(unsigned long long int)
sizeof(unsigned long long)
```

Integral Data Types | Character Types

- Used to represent single characters, e.g., `'A'`, `'b'`, `'@'`, etc
- The most basic type is `char`, which is a 1-byte character.
- Wider types are used to represent wide character sets. Mainly used when single character representation is not enough (e.g., Unicode characters and strings).

Type Name	Size/Precision
`char`	Exactly 1 byte. At least 8 bits.
`char16_t`	At least 16 bits.
`char32_t`	At least 32 bits.
`wchar_t`	Can represent the largest available character set.

Integral Data Types | Character Types

- The char data type is an integral type, meaning the underlying value is stored as an int, and it's guaranteed to be 1-byte in size.
- However, similar to how a bool value is interpreted as true or false, a char value is interpreted as an ASCII character.
- You can initialize char variables using character literals.
- You can initialize char variables with integers as well, but this should be avoided if possible.

```
int main(){
    char var_char1{'a'};
    char var_char2{97};
    std::cout << var_char1 << std::endl; //-- 'a'
    std::cout << var_char2 << std::endl; //-- 'a' based on the ASCII table
    return 0;
}
```

Integral Data Types | Character Types | Conversion to Integer

- To print characters as integers you can use a type cast.
- It creates a value of one type from a value of another type.
- To convert between fundamental data types (for example, from a `char` to an `int`, or vice versa), we use a type cast called `static_cast`.

```
static_cast<new_type>(expression)
```

- Example:

```
#include <iostream>

int main(){
    char var_char{'a'};
    std::cout << var_char << '\n'; //--a
    std::cout << static_cast<int>(var_char) << '\n'; //--97
    std::cout << var_char<< '\n'; //--a
    return 0;
}
```

Integral Data Types | Character Types | Conversion to Integer

- The following program asks the user to input a character, then prints out both the character and its ASCII code:

```cpp
#include <iostream>

int main(){
    std::cout << "Input a keyboard character: ";

    char v_char{};
    std::cin >> v_char;
    std::cout << v_char << " has ASCII code " << static_cast<int>(v_char) << '\n';

    return 0;
}
```

- Note: `std::cin` will let you enter multiple characters.
- However, variable `v_char` can only hold 1 character.
- Consequently, only the first input character is extracted into variable `v_char`.
- The rest of the user input is left in the input buffer that `std::cin` uses.

Integral Data Types | Character Types | Escape Sequences

- There are some characters in C++ that have special meaning.
- These characters are called escape sequences.
- An escape sequence starts with a backslash character, and then a following letter or number.

```
int main(){
    std::cout << "First line\nSecond line\n";
    return 0;
}
```

- More information on escape characters can be found here.

Integral Data Types | Character Types | 16-bit and 32-bit

- The `char16_t` and `char32_t` types represent 16-bit and 32-bit wide characters, respectively.
- Unicode encoded as UTF-16 can be stored in the `char16_t` type, and Unicode encoded as UTF-32 can be stored in the `char32_t` type. Strings of these types and `wchar_t` are all referred to as wide strings, though the term often refers specifically to strings of `wchar_t` type.

Integral Data Types | Character Types | Wide-character Set

- `wchar_t` is similar to `char` data type, except that `wchar_t` takes up twice the space and can take on much larger values as a result.
- `char` can take 256 values which corresponds to entries in the ASCII table.
- On the other hand, `wchar_t` can take on 65,536 values which corresponds to UNICODE values which is a recent international standard which allows for the encoding of characters for virtually all languages and commonly used symbols.

Integral Data Types | Character Types | Wide-character Set

```
#include<iostream>

int main(){
    wchar_t w  = L'A';
    std::cout << "Wide character value:: " << w << std::endl;
    return 0;
}
```

- `L` is the prefix for wide character literals and wide-character string literals which tells the compiler that the character or string is of type `wchar_t`.

Integral Data Types | Best Practice

- You should try your best using the correct modifiers.
    - If you know that some values will never be negative (e.g., size of a file or length of a string) then why use `int` or `signed int`? Use `unsigned int` or `unsigned` instead.
    - By using `unsigned int` or `unsigned` in your program, you will clearly remind yourself (or whoever is reading the code) that this value cannot be negative.
    - If you know for sure that a value will never exceed 32 bits, then explicitly use `short` as the variable type.

Floating-point Types

- Floating-point types are used to represent non-integer numbers.
- They are represented by mantissa and exponent (scientific notation).
- Precision is the number of digits in the mantissa.
- Precision and size are compiler dependent.

Type Name	Size/Typical Precision	Typical Range
`float`	7 decimal digits	1.5×10^{-45} to 3.4×10^{38}
`double`	No less than `float`, 15 decimal digits	2.2×10^{-308} to 1.8×10^{308}
`long double`	No less than `double`, 19 decimal digits	3.3×10^{-4932} to 1.2×10^{4932}

- To-do: Please see here for more information on floating-point numbers.
- To-do: Please see here for the limits on floating-point numbers.

Floating-point Types | float

- `float` is a shortened term for "floating point".
- By definition, it's a fundamental data type built into the compiler that's used to define numeric values with floating decimal points.
- The `float` type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38}, with a precision – the limit of digits – of seven.
- `float` can contain up to seven digits in total, not just following the decimal point so, for example, 321.1234567 cannot be stored in float because it has 10 digits.
- If greater precision – more digits – is necessary, the `double` type is used.

Floating-point Types | double

- `double` is a fundamental data type built into the compiler and used to define numeric variables holding numbers with decimal points.
- A `double` type can represent fractional as well as whole values. It can contain up to 15 digits in total, including those before and after the decimal point.
- The `float` type, which has a smaller range, was used at one time because it was faster than the double when dealing with thousands or millions of floating-point numbers.
- Because calculation speed has increased dramatically with new processors, however, the advantages of `float` over `double` are negligible.
- Many programmers consider the `double` type to be the default when working with numbers that require decimal points.

Boolean Data Type

- The Boolean data type (1 byte) `bool` can only take the values `true` or `false`:

```
bool is_today_sunny = true;
bool is_today_cloudy = false;
std::cout << is_today_sunny << std::endl;  //--1
std::cout << is_today_cloudy << std::endl; //--0
```

- To output `true` or `false` instead of 1 or 0 you can use `std::boolalpha`. `std::noboolalpha` will display 1 or 0.

```
bool is_today_sunny = true;
bool is_today_cloudy = false;
std::cout << std::boolalpha << is_today_sunny << std::endl;//--true
std::cout << std::boolalpha << true << std::endl;//--true
std::cout << std::noboolalpha << true << std::endl;//--1
std::cout << std::boolalpha << is_today_cloudy << std::endl;//--false
std::cout << std::boolalpha << false << std::endl;//--false
std::cout << std::noboolalpha << false << std::endl;//--0
```

Overflow

- Overflow is a phenomenon where operations on 2 numbers exceeds the maximum (or goes below the minimum) value the data type can have.
- Overflow can occur during compilation or at runtime.
- Usually it is thought that integral types are very large and people do not take into account the fact that the sum of two numbers can be larger than the range.

Overflow | Unsigned Overflow

- Imagine we have a made up data type called `var_t` of exactly 1 byte.
- The minimum value that an `unsigned var_t` can store is 0 and the maximum value is 255.

```
unsigned var_t a{200}, b{100};
```

- `a{200}` can be represented as follows using 1-byte memory:
  - 200 = 1100 1000 = $(2^3) + (2^6) + (2^7)$
- `b{100}` can be represented as follows using 1-byte memory:
  - 100 = 0110 0100 = $(2^2) + (2^5) + (2^6)$
- `a + b` = 1100 1000 + 0110 0100 = 0000 0001 0010 1100. The result is more than 1 byte and the higher byte will be rejected. In this case, `a + b` will be read as 0010 1100 = $(2^2) + (2^3) + (2^5) = 44$.
- This is an example of an unsigned overflow, where the value could not be stored in the available number of bytes.

Overflow | Signed Overflow

- A `signed int` is 4 bytes and can store data ranging from -2,147,483,648 to 2,147,483,647 (see slide 14).
- Now consider the code snippet below.

```
int a{2'000'000'000};
int b{2'000'000'000};
int c = a + b;
std::cout << c << std::endl;
```

- The output of this program is -294967296 and not 4000000000.
- Although `a` and `b` hold values within the `int` range, the result of `a + b` is a value outside the range.
- We have another case of overflow.