Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger

LS 11 Algorithm Engineering

Lecture 5 • Winter 2011/12 • Nov 15

technische universität dortmund

department of computer science

File I/O

- C++ provides support for reading from and writing to files
- Reading: input file streams
  - similar as reading from the console
  - **`cin`** is an input stream
- Writing: output file streams
  - similar as writing to the console
  - **`cout`** is an output stream

Example: Read integers from a file

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
     ifstream is("input.txt");
     if(!is)
        cout << "Could not open file!" << endl;

     else {
        int x;
        while(is >> x)
           cout << x << "\n";

        is.close();
     }

     return 0;
}
```

Example: Step-by-Step

```
#include <fstream>
```

- Includes functionality for working with file streams

```
ifstream is("input.txt");
```

- Create a file input stream variable **`is`**
- Try to open the file **`input.txt`**

```
if(!is)
   cout << "Could not open file!" << endl;
```

- Check if file could be opened
  - an input stream can automatically be converted to a bool
- If not, print an error message

Example: Step-by-Step

```
while(is >> x)
   cout << x << "\n";
```

- Read integers as long as possible
- the value of **`is >> x`** is false if no integer could be read

```
is.close();
```

- Finally, close the file
  - You cannot read from the file anymore once it is closed!
  - Files get automatically closed when the scope of the corresponding stream variable ends

Example: Writing to a file

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
     ifstream is("input.txt");
     if(!is) cout << "Could not open file!" << endl;
     else {
        ofstream os("output.txt");
        if(!os) cout << "Could not open output file!" << endl;
        else {
           int x, i = 1;
           while(is >> x)
                os << "line " << i++ << ": " << x << "\n";
        }
     }
     return 0;
}
```

Example: Step-by-Step

```
ofstream os("output.txt");
if(!os) cout << "Could not open output file!" << endl;
```

- Create a file output stream variable **`os`**
- Try to open the file **`output.txt`** and check for errors

```
int x, i = 1;
while(is >> x)
    os << "line " << i++ << ": " << x << "\n";
```

- Write the integers read from input file to the output file
  - We also count the line numbers and print them in front of the integers

Characters

- The data type `char` represents a single character
- Character literals have to be enclosed by single quotation marks: `'c'`
- A limited form of arithmetic is available for char
  - e.g. `'7' - '0' == 7` holds
- You can also think of an `std::string` as a vector of `char`s
- The usual stream I/O is supported, e.g.

```
char x;
cin >> x;
```

- **Caution:** By default, `cin` skips whitespace characters!

Reading Lines and Single Characters

- So far we have problems when we want to read a whole line into a string or a single character into a **`char`**
- The following methods help us (let **`is`** be an input stream):
  - **`is.get(c)`** reads a single character into a **`char c`**
  - **`std::getline(is,str)`** reads a whole line (including any whitespace) into an **`std::string str`**
- Example:

```
ifstream is("input.txt");
char c; string str;

getline(is,str); cout << str << endl; // print first line

while(is.get(c)) // read remaining characters one by one
   cout << c << endl;
```

Maps

- A map (also dictionary or association) stores pairs of keys and values
- Using **`std:map`** requires **`#include <map>`**
- The following example declares a map of (string,int) pairs:

```
std::map<std::string,int> wordcounts;
```

  - the keys are of type std::string
  - the values are of type int
- Keys are always unique within a map

Accessing Values Through Keys

- The map allows us to access the values through the keys:

```
++wordcounts["hi"];
```

  - `wordcounts["hi"]` gives access to the value stored for key `"hi"`
  - We can use it like any other `int` variable (increase it in this case)
- If we access a yet unknown key, a new (key,value)-pair is added, where the value is a default value (e.g. 0 for number types). This can be a problem.

Map Iterators

- Map iterators allow us to iterate over all elements in a map
- This works in the same way as for vectors:
  `begin()`, `end()`, `++` and `*` operators
- A map iterator `it` points to an `std::pair`, which has two components: `first` and `second`
  - key: `(*it).first` or `it->first`
  - value: `(*it).second` or `it->second`

Example: Histogram (compare Ex. 3.3)

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<int,int> count;

    while(true) {
        int x; cin >> x;
        if(x <= 0) break;
        ++count[x];
    }
    map<int,int>::iterator it;
    for(it = count.begin(); it != count.end(); ++it)
        cout << it->first << "\t" << it->second << endl;

    return 0;
}
```

Finding in Maps

- We can use **`m.find(key)`** to check whether a given key exists in a map **`m`**:
  - If the key is in the map, **`find`** returns an iterator pointing to the corresponding (key,value)-pair
  - Otherwise, **`find`** returns the **`end()`** iterator of the map
- Generally, accessing elements in a map (using the []-operator or the find method) is very fast (much faster than iterating over all elements!)

Type Definitions

- The names of data types we use in our programs can become quite long, e.g. **`std::vector<string>::iterator`**
- C++ allows us to give types new names, e.g.

```
typedef std::vector<string>::iterator str_iterator;
```

- The general form of a type definition is:

  typedef *data-type new-name*;

- Type definitions follow the same scope rules as variables
- Choose good names for data types to make your program easy to read and understand!

Constants

- Variables that shall never be changed can be declared as constants:

```
const int months_in_year = 12;
```

- Constants
  - must be initialized when they are declared
  - cannot be changed later:

```
months_in_year = 20; // error
                     // cannot assign to a constant!
```

- You can declare constants of any type
- Prefer constants over literals
  - this makes your program more readable and easier to modify

Types of Integers

- C++ provides different flavors of integers
  - They can be signed or unsigned
  - They can have different sizes, thus allowing a smaller or larger range of numbers that can be represented
- Signed or unsigned:
  - a `signed int` can store positive and negative values (this is the default for `int`, so we can omit the `signed` keyword)
  - an `unsigned int` can only store non-negative values
- Different sizes:
  - an `int` is usually 4 bytes wide
  - a `short int` uses less space, usually 2 bytes
  - a `long int` or a `long long int` can represent even more values
  - **But:** the actual size may vary from system to system!

Types of Integers

- `short`, `long`, `signed`, and `unsigned` are called qualifiers
- They can be combined, e.g.

```
unsigned short int a;
signed long int b;
```

- Integers are signed by default
- Integer representation:
  - signed integers use one bit for the sign
  - a 2 bytes wide signed integer thus supports the following range of values: -32,768 (= -2^{15}) to 32,767 (=$2^{15}-1$)
    (these are 2^{16} distinct values, including the 0)
  - on the other hand, a 2 bytes wide unsigned integer:
    0 to 65,535 (=$2^{16}-1$)

Ranges of Values in Visual C++

Type Name	Bytes	Other Names	Min. Value	Max. Value
short	2	short int, signed short int	–32,768	32,767
unsigned short	2	unsigned short int	0	65,535
int	4	signed	–2,147,483,648	2,147,483,647
unsigned int	4	unsigned	0	4,294,967,295
long	4	long int, signed long int	–2,147,483,648	2,147,483,647
unsigned long	4	unsigned long int	0	4,294,967,295
long long	8		–9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	8		0	18,446,744,073,709,551,615

The sizeof Operator

- The `sizeof` operator returns the size of a data type
- The following programs prints the various sizes:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "short int:      " << sizeof(short int)      << "\n";
    cout << "int:            " << sizeof(int)            << "\n";
    cout << "long int:       " << sizeof(long int)       << "\n";
    cout << "long long int:  " << sizeof(long long int)  << "\n";

    return 0;
}
```

Preparations for next week

- Functions
- References
- The conditional operator (`? :`)
- Switch-Statements (`switch`… `case`…)