C++: Up to Speed

Yngvi Björnsson

Overview

- Different Faces of C++
- Compiler Warnings
- Type Casting
- Command Line Arguments
- I/O Basics
- Header Files
- Class vs. Struct
- Parameter Passing
- Overloading

Different Faces of C++ EC[#1]

- Four “sub-languages”:
  - C
    - Designed around efficiency
    - Only very minor (but sometimes subtle) difference
  - Object-Oriented C++
    - Powerful OO language
    - (Albeit, maybe not the purest OO)
  - Template C++
    - Very powerful generic programming capabilities
  - STL
    - Extensive library of generic containers and algorithms

My First Program

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!\n";
    return 0;
}
```

Pay Attention to Compiler Warnings EC[#53]

- Must fix errors for the program to compile.
- Also pay attention to *all* compiler **warnings**
  - Get in the habit of eliminating them, even when we know them to be "harmless"
  - Otherwise we run the risk of overlooking potentially important warnings
    - "Úlfur, úlfur, …"
- Compile with high-warning settings, e.g.,
  - -Wall -Wconversion

Compiler Warnings: Example 1

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

int main( )
{
    vector<int> data;
    for ( int i=0; i < data.size(); ++i ) {
        // ... do something with data.
    }
}
```

```
Building file: ../main.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"main.d" -MT"main.d" -
o "main.o" "../main.cpp"
../main.cpp: In function 'int main()':
../main.cpp:8: warning: comparison between signed and unsigned integer
expressions
Finished building: ../main.cpp
```

Compiler Warnings: Example

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

int main( )
{
    int i = 1, l;
    i = 2 * i;
    cout << i << endl;
}
```

```
>g++ main.cpp
>
>g++ -Wall main.cpp
main.cpp:6:16: warning: unused variable 'l' [-Wunused-
variable]
    int i = 1, l;
               ^
1 warning generated.
```

Compiler Warnings: Example

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

int main( ){
    vector<int> data;
    // ... add to data.
    for ( int i=0; i < data.size(); ++i ) {
        if ( data[i] = 5 ){
            // ... do something.
        }
    }
}
```

```
main.cpp:10:22: warning: using the result of an
assignment as a condition
      without parentheses [-Wparentheses]
        if ( data[i] = 5 ) {
```

Compiler Warnings: Example

```
#include <iostream>
using namespace std;
int lookup(int A[], int len, int to_find) {
    for (int i = 0; i < len; ++i) {
        if (A[i] == to_find) {
            return i;
        }
    }
}

int main( ) {
    int data[10] = {0,3,4,5,2,5,1,4,3,8};
    cout << lookup(data, 10, 1) << endl;
    cout << lookup(data, 10, 6) << endl;
}
```

```
main.cpp:10:1: warning: control may reach end of non-void
function
      [-Wreturn-type]
}
```

Compiler Warnings: Example

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

int main( ) {
   int i = 100;
   i = i / 2.5;                  // Converts expression to float and then back to int. Intentional?
   i = static_cast<int>( i / 2.5 );      // Better to be explicit about this!
   cout << i << endl;
}
```

```
>c++ -Wall main.cpp
>
>c++ -Wall -Wconversion main.cpp
main.cpp:8:11: warning: implicit conversion turns floating-point
number into
      integer: 'double' to 'int' [-Wfloat-conversion]
    i = i / 2.5;
      ~ ~~^~~~~
1 warning generated.
```

Type Casting (EC[#27] for details)

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

int main( ) {
    vector<int> data;
    // ... add to data.
    for (int i=0; i < static_cast<int>(data.size()); ++i) {
        // ... do something with data
    }
}
```

- Casting (compile time)
  - `(type) var // C-style, to be discouraged.`
  - `static_cast<type>( expression ) // Preferred.`
- Casting (run time):
  - `dynamic_cast<type>( expression ) // See later!`

Command Line Arguments

```
#include <iostream>
using namespace std;
//
// This program prints out its
// command-line arguments.
//
int main( int argc, char *argv[] )
{
    cout << "Arguments:\n";
    for ( int i = 0; i < argc; ++i ) {
        cout << ' ' << argv[i] << '\n';
    }
    return 0;
}
```

```
./prg arg1 arg2 arg3
Arguments:
 ./prg
 arg1
 arg2
 arg3
```

Note that argv[0] is the name of the program.

Reading from standard input

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

//
// This program reads in text from
// standard input and prints out the
// inputted 'words', one per line.
//
int main( )
{
    string word;
    while ( cin >> word ) {
        cout << "  '" << word << "'\n";
    }
    return 0;
}
```

```
./prg
hello, how are you?
  'hello,'
  'how'
  'are'
  'you?'
```

'words' are separated by white-spaces (space, tab, newline,...)

Reading word-by-word (from cin)

```
// This program reads in text from standard
// input word-by-word and counts them.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    int count = 0;
    string word;
    while ( cin >> word ) {
        ++count;
    }
    cout << count << '\n';

    return 0;
}
```

File test.txt

This sentence has only 6 words

```
./prg < test.txt
6
```

Note how file content is redirected to standard input.

Reading word-by-word (from file)

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

int main( int argc, char *argv[] )
{
    if ( argc > 1 ) {
        ifstream ifs( argv[1] );
        if ( ifs.is_open() ) {
            int count = 0;
            string word;
            while ( ifs >> word ) {
                ++count;
            }
            cout << count << '\n';
            ifs.close();
        }
    }
}
```

File test.txt

This sentence has only 6 words

```
./prg test.txt
6
```

Now the name of the file is provided as an argument.

Reading word-by-word (from istream)

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

int countWords( istream& in ) {
    int count = 0;
    string word;
    while ( in >> word ) {
        ++count;
    }
    return count;
}

int main( int argc, char *argv[] ) {
    int count = 0;
    if ( argc > 1 ) {
        ifstream ifs( argv[1] );
        if ( ifs.is_open() ) {
            count = countWords( ifs );
            ifs.close();
        }
    }
    else { count = countWords( cin ); }
    cout << count << '\n';
}
```

File test.txt

This sentence has only 6 words

```
./prg test.txt
6
```

I/O Stream Hierarchy

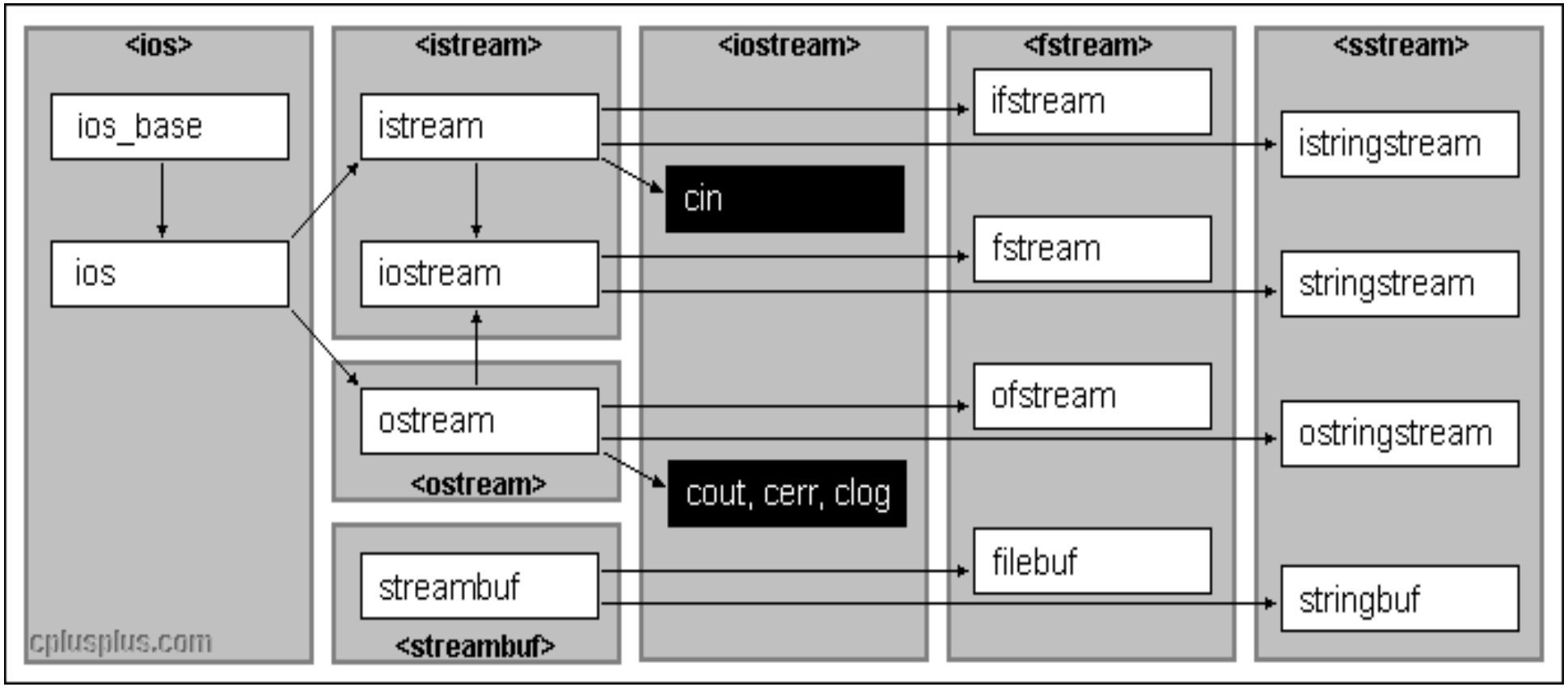

Image from http//cplusplus.com

Reading line-by-line (from istream)

```
//
// This functions counts the number of
// lines in the stream passed down.
//
int countLines( istream& in )
{
    int count = 0;
    string line;
    while ( getline( in, line ) ) {
        ++count;
    }
    return count;
}
```

Prefer this function over method:
char line[256];
in.getline(line, 256);

Reading char-by-char (from istream)

```
//
// This functions counts the number of
// characters in the stream passed down,
// including white-spaces.
//
int countChars( istream& in )
{
    int count = 0;
    char ch;
    while ( in.get( ch ) ) {
        ++count;
    }
    return count;
}
```

Can “read” from strings (from istream)

```
//
// Counts words in a string (using already
// existing function countWords(istream&) ).
//
. . .
#include <sstream>

void foo( ) {
    string line( "This is a line." );
    stringstream ss( line );
    cout << countWords( ss );
}
```

Example: Counting

```
void countLinesWordsChars( istream& is, int& countLines,
                           int& countWords, int& countChars )
{
    countLines = countWords = countChars = 0;
    bool processingWhiteSpaces = true;
    char ch;
    while ( is.get( ch ) ) {
        ++countChars;
        if ( iswspace( ch ) ) {
            processingWhiteSpaces = true;
        }
        else if ( processingWhiteSpaces ) {
            processingWhiteSpaces = false;
            ++countWords;
        }
        if ( ch == '\n' ) { ++countLines; }
    }
}
```

```
./prg < test.txt
2 6 32
```

Classes

```
class Date {
public:
   // Accessible to all.
protected:
   // Accessible to derived classes.
private:
   // Accessible only to class members.
   // Make data-members private (if possible).
   int day_;
   int month_;
   int year_;
};
```

Note '_' postfix naming convention of member variables (different conventions)

Classes

- Separate **declaration and definition** of class.
  - Declaration in a header file (.h)
    - Methods
    - Data members
  - Definition in a C++ file (.cpp)
    - Implementation of methods
    - Initialization (e.g. of static members)
- Exception for small classes/methods
  - Declaration and definition all in header file.
  - Pros and cons?

Header file

- **Include guards**
  - Prevents header file to be included multiple times in the same compilation unit (could cause errors).

```
#ifndef DATE_H
#define DATE_H

class Date {
    …
};

#endif DATE_H
```

C++ File

```
#include “date.h”

Date::Date(…)
{
};

.
.
.
```

Difference between
#include <filename> or
#include “filename” ?

Example

```
// calender.h
#include “date.h”
...
```

```
// main.cpp
#include “date.h”
#include “calender.h”
...
```

Headerfile Pitfalls

- In a header file avoid “using namespace”:

```
using namespace std;
int countWords( istream& in );
```

instead, use quantifiers:

```
int countWords( std::istream& in );
```

- Do not include things in the header, unless needed in header-file itself

```
#include <algorithm> //Needed in .h or only .cpp?
```

Class vs. Struct

```
class Date {
   int day;
   int month;
   int year;
};
```

```
struct Date {
   int day;
   int month;
   int year;
};
```

- Only difference is that
  - struct members have public access by default
  - class members have private access by default
- Convention to use structures only for object types that
  - do little more than storing data, with little or none logic (maybe, at most, use “light-wrappers”, e.g. to initialize).

Example

```
struct Date {
   Date( int d, int m, int y )
      : day(d), month(m), year(y) {}
   int day;
   int month;
   int year;
};

struct Move {
    Move(int f, int t) : from(f), to(t) {}
    int from;
    int to;
};
```

Parameter Passing

- Pass-by-value:
  - a copy of the variable is passed down

```
void foo( int x ) {
   x = x + 2;
}

int main( ) {
   int x = 0;
   foo( x );
   cout << x << '\n';
}
```

Parameter Passing (cont.)

- Pass-by-reference:
  - address of the variable is passed down (under the hood)

```
void foo( int& x ) {
   x = x + 2;
}

int main( ) {
   int x = 0;
   foo( x );
   cout << x << '\n';
}
```

int& x
vs.
int &x

Parameter Passing

- Using pointers:
  - Address of variable passed down as a pointer (it is passed-by-value)

```
void foo( int* x ) {
   *x = *x + 2;
}

int main( ) {
   int x = 0;
   foo( &x );
   cout << x << '\n';
}
```

Parameter Passing (cont.)

- Often preferable to pass-by-reference, even though we do not want to modify value in function.
  - Why?
- Pass-by-reference but make const.

```
void foo( const Data& data ) {
    // We can work with the original copy
    // of data, but we cannot modify it.
    . . .
}

int main( ) {
    Data d;
    foo( d );
    cout << data << '\n';
}
```

Default Parameters

- May specify default values for omitted arguments.
  - But only for the right-most parameters. Why?
- Note:
  - If separate declaration/definition (e.g., a class method), then specify default value in declaration only.

```
int sum( int v1, int v2 = 0, int v3 = 0 ){
    return v1 + v2 + v3;
}

int main( )
{
    cout << sum( 1, 2, 3 ) << '\n';
    cout << sum( 4, 5 ) << '\n';
    cout << sum( 6 ) << '\n';
}
```

Overloading

- Can have many “versions” of a function
  - Differ by parameter signature

```
int sum( int v1, int v2, int v3 ) {
    return v1 + v2 + v3;
}

int sum( int v1, int v2 ) {
    return v1 + v2;
}

int sum( int v1 ) { return v1; }

int main( ) {
    cout << sum( 1, 2, 3 ) << '\n';
    cout << sum( 4, 5 ) << '\n';
    cout << sum( 6 ) << '\n';
}
```

Operator Overloading

```
class Rational {
public:
    Rational( int n, int d = 1 ) : n_( n ), d_( d ) {
       // ... need to simplify quotient;
    }
    bool operator==( const Rational& rhs ) {
       return (n_ == rhs.n_) && (d_ == rhs.d_);
    }
private:
    int n_;
    int d_;
};

int main() {
    Rational r1(1, 1), r2(1);
    cout << (r1 == r2) << '\n';
    cout << (r1 == 1) << '\n'; // Works!
    cout << (1 == r1) << '\n'; // Compile error!
}
```

If we do not want this to work, we must make constructor **explicit**

Operator Overloading (cont.)

```
class Rational {
public:
    Rational( int n, int d = 1 ): n_( n ), d_( d ) {
       // ... need to simplify quotient;
    }
    friend bool operator==(const Rational& lhs,
                           const Rational& rhs ) {
       return (lhs.n_ == rhs.n_) && (lhs.d_ == rhs.d_);
    }
private:
    int n_;
    int d_;
};

int main() {
   // ...
    cout << (1 == r1) << '\n'; // Works now!
}
```

Not a method (member-function), even though declared in class!

Operator Overloading (cont.) (EC[#24] for details)

```
class Rational {
public:
    Rational( int n, int d = 1 ) : n_( n ), d_( d ) {
       // ... need to simplify quotient;
    }
     int getN() const; { return n_; }
     int getD() const; { return d_; }
private:
    int n_;
    int d_;
};
inline bool operator==( const Rational& lhs,
                        const Rational& rhs ) {
    return ( lhs.getN() == rhs.getN() ) && ( ... );
}
int main() {
    // ...
   cout << (1 == r1) << '\n'; // Works now!
}
```

Alternative to using friends (sometimes possible)

Pros and cons?

Questions?

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```