# Basics

## 1 Introduction

C++ can be seen as an extension to C, although there are some details that may prevent a C program from directly compiling under C++ compiler, for example because C++ has extended set of keywords that could have been used as variable names in a C program (e.g., **new** and **delete**). The basic syntax, structures and data types from the C language are still available in C++.

C++ is standardised by the International Standard Organization (ISO). There are multiple versions of the standard, and this material assumes **C++11** (ISO/IEC 14882:2011), the 2011 version of the standard, that brought a few notable new features to the language that did not exist earlier.

g++ -std=c++0x -g -Wall -Wextra -pthread -Wno-missing-field-initializers

"-std=c++0x" tells that we assume the C++11 standard, "-g" brings in the program symbols to support debugging (as with C), "-Wall" and "-Wextra" increase the number of warnings the compiler produces "-pthread" is required by our exercise checking environment.

The C++ source files are named with "**.cpp**" file name suffix, to separate them from other languages, particularly C. The C++ header files are named with "**.hpp**" suffix. These are just naming conventions, and the compiler would allow also other styles of naming. In different environments different naming styles may be used.

Typically the files are in 'src' directory of the package, and there is a makefile, that compiles and runs the program, when you type 'make run-main'.

## Basic diff between C and C++

C++ is backwards compatible with C: A program written in C should also work in C++ compiler, and for example as part of a larger C++ software.

### C++ standard library and namespaces

Unlike in C, where function and type names are all in same global namespace, C++ introduces namespaces that allow organising names hierarchically. This is useful in large software, when there may be collisions with common type of names, such as "size", "count", etc.

a commonly used namespace, **std**, that is used by the C++ standard library. The identifiers in this namespace have "**std::** as prefix, such as in std::cout. It refers to the "cout" object (for standard output stream), that is defined under the "std" namespace.

## 2.2 Boolean data type

C++ has a dedicated data type for boolean true/false type variables, called bool. A bool variable can take values true and false, that are new reserved words that can be used in assignments and comparisons.

#include <iostream>

int main(void) {

bool flag;

flag = true;

if (flag) {

std::cout << "Flag is true" << std::endl;

}

flag = 8 > 9;

std::cout << "Flag: " << flag << std::endl;

}

The first line tells that we will use the **iostream** header for I/O streams. Worth noting is, that the headers in C++ standard library do not use ".h" suffix, or ".hpp" suffix. Even without the suffix they are normal C++ header files, containing definitions for data types and other names.

We can see that C++ statements have similar properties as in C: they end in semicolon, and program blocks are still represented by curly braces. (In fact, with C's enum and typedef definitions we could simulate quite similar behavior).

In conditional expressions bool works similarly as int in C programs, and there usually is no need to compare it against true or false. An int-type value can be assigned to bool type variable, in which case it will be converted either to true (non-zero integer) or false (zero integer), similarly to the behavior of integers in conditional expressions.

std::cout is a C++ object that allows **outputting data to standard output** stream, to replace C's use of **printf()**.

std::endl means that a new line is started from that point (std::endl is similar to **'\n'**).

we do not need to care about type-specific formatting specifiers anymore.

**From this point on, in C++ we will avoid using the old printf() function.**

## 2.3 Reference variables

C++ introduces a new kind of variable, a **reference**, that points to another variable. Reference variable has some similarity to traditional C pointer, but it is more strictly considered as a compile-time alias: reference does not represent any particular address like pointer, and cannot have a NULL value. Therefore, with references we cannot have invalid pointers, making them safer to use in programs while avoiding a full copy of the variable (which may be expensive with larger data structures).

Reference variable is identified with & following the variable type when variable is defined.

#include <iostream>

int main(void) {

int someA = 10;

int& someB = someA;

someB = 20;

std::cout << "someA: " << someA << " someB: " << someB << std::endl;

}

Line 5 defines reference variable someB that is a reference to someA. Therefore, when we assign value to **someB**, the value of **someA** changes at the same time. The properties of reference variable imply, that it must always be initialized when it is defined.

Also function arguments can be passed by reference following a similar notation. this causes the function to access the original content of the argument, instead of making a copy for function's local use. This is similar to using pointers as function arguments, except that the function can assume that the reference is always valid: there are no NULL pointers.

#include <iostream>

void multiply(int& value, int mul)

{

value \*= mul;

}

int main(void) {

int someA = 10;

multiply(someA, 3);

std::cout << "someA: " << someA << std::endl;

}

In the above example, the **multiply** function does not have return value. Instead, it directly modifies the **value**reference target in the calling context. Therefore the value of **someA** changes as a result of the function call, and the program outputs value 30 at the end. Note that the **mul** argument is passed by value in the normal way, and is copied for the function. If it was modified, the changed value would not be visible in the **main** function.

Note that where a reference argument is used, fixes constant numbers cannot be used as a parameter: multiply(6, 3) would fail at compilation. The caller must use an **lvalue** as parameter, i.e., a variable or some other expression that can be used in the left side of an assignment operation.

Even though reference variables are a convenient and safer way to pass data by reference, the traditional pointers can be used with C++ in the similar way as in C.

## 2.4 auto type

Starting from the C++11 standard, a variable can be defined with auto type specifier. This is not an actual type, but rather tells the compiler to determine what the actual type of the variable is. In most cases compiler can deduce the type based on the initialization. Therefore, the definition with **auto** type must come with an initialization, so that the compiler can do this trick.

## 2.5 Overloading

In C++ a **function can be overloaded**, i.e., multiple definitions of function with same name, but different argument lists can be given. This was not possible in C.

#include <iostream>

void multiply(int& value, int mul)

{

value \*= mul;

}

void multiply(double& value, int mul)

{

value \*= mul;

}

int main(void) {

int someA = 10;

double someB = 2.54;

multiply(someA, 3);

multiply(someB, 4);

std::cout << "someA: " << someA << " someB: " << someB << std::endl;

}

The above extends the previous example such that our **multiply()** function now supports both **int** and **double** types. It happens here that the implementation for both functions looks the same, but they would not have to be similar. However, a sane design is to have the overloaded functions to behave semantically in the same way.

Another aspect in function overloading is that some of the arguments can be assigned a default value. When an argument is given a default value, the calling code can omit the argument, in which case the function implementation will assume the default value instead.

In addition, also an **operator can be overloaded**. In the previous examples we see how the **cout** objects have overloaded the << operators. In this case they do not have nothing to do with bit manipulation, as we have been accustomed to in C.The I/O classes have overloaded these operators to mean something else related to input and output. Operator overloading makes it possible to write very confusing code if used inappropriately, so one should try to apply common sense, especially when overloading widely used operators, such as **+** or **-**.

## 3. I/O stream

 In C++ there is a different -- and easier -- way to read and write data to a stream, that does not require the use of the inconvenient format specifiers, but thanks to function overloading and operator overloading (explained later), C++ allows easy way of handling any type of data. also in C++ there are a few default streams available for all programs: standard input, standard output, and standard error streams.

### 3.1 Using standard input and output

The C++ standard library comes with various classes and operations related with input and output, but for now we discuss **istream** objects that process user input, and **ostream** objects that process user output. In addition, the library defines a few default objects, particularly, **cin** that handles the standard input (similar to the stdin variable in C), and **cout** that handles the standard output (similar to stdout variable in C).

C++ accesses the streams using the << and >> operators, depending on whether we are handling output to stream (the former), or input from stream (the latter).

Particularly, the code does not need to care about format specifiers (%d, %f, %s, and such), but when objects are passed to the iostream operations, their output format is determined "automatically" through the the I/O operators.

int num = 5;

std::string str = "Hello!";

std::cout << "num: " << num << " string: " << str << std::endl;

The output expression can have a mix of string literals (such as "num: ") and variables, that will be concatenated in output. In addition, it can have **manipulators**, such as **endl**, that will cause the output to move to the next line (instead of "\n"). This becomes especially handy when we need to output more complex types, such as vectors or structured data types.

int num;

std::cin >> num;

**cout** is an **ostream** type object, and **cin** is an **istream** type object that refer to system standard output and input streams. They are available by default, but in addition other streams can be opened for files, communication, or other uses, as needed.

# Vectors and Strings

The C++ Standard Library comes with helpful tools for managing information. For now, we focus on two basic types: the **string** class allows handling strings in easier and safer way than the traditional C did. The **vector** class that allows operating with arrays of data (of arbitrary type).

Term Standard Template Library (STL) is also sometimes used, referring to an earlier library which the Standard Library is based on. The name indicates that the classes are based on templates that allow generic handling of any type of data, such as in **vector** class.

## String

The **string** type contains a variable-length sequence of characters. For C++, however, **string** is an abstract data type that takes care of space allocation, safe initialization and management of the string, on behalf of the programmer. Because string is part of the the C++ standard library, it belongs to the std namespace. Strings are defined in a separate "string" header.

string s1; // an empty string

string s2 = "Hello";

string s3 = s2;

**s1** is an empty string. Different from C, it **is initialized**, and one can assume the string to be always empty after the variable declaration. An initial value can be assigned together with variable declaration, as done for **s2**. The syntax of this looks rather similar than what a C programmer would do with char \* type, but here string object **s2** is modified based on the assignment operation, and **s2** is not a pointer. **s3** is assigned based on the content of **s2**, but because it is not a pointer, the content is copied, and there are two copies of string "Hello".

With string objects the assignment operator changes the content of the string. This is another example of **operator overloading** that was mentioned above.

#include<iostream>

#include<string>

int main(void)

{

std::string s1 = "Hello";

std::string s2;

std::string s3;

s2 = "world";

s3 = s1 + " " + s2;

std::cout << s3 << std::endl;

}

The example shows that for strings, the + operator is also overloaded to concatenate two strings. As seen before, I/O streams, on the other hand, overload the << operator for output.

When a string needs to be read from the user, the **cin** stream can be used as shown above:

std::string s1;

std::cin >> s1;

std::cout << "String was: " << s1 << std::endl;

unsigned int length = s1.size();

std::cout << "Length of the string is: " << length << std::endl;

The above reads a string from user into **s1**, and then outputs it with some additional text, and a newline mark. For input streams, the characters until the first whitespace are read (space, tab, newline, etc.).

In addition, the **size()** function is used to get the length of the string that was given by the user. The notation means that size of string **s1** is returned by the function, and also that can be output to the screen. Strings can also be passed as function arguments and return values, as any other data type.

#include <string>

#include <iostream>

int longerLength(const std::string& a, const std::string& b)

{

if (a.size() > b.size())

return a.size();

else

return b.size();

}

int main(void)

{

std::string s1 = "Hei";

std::string s2 = "Hello";

std::cout << longerLength(s1, s2);

}

The strings are passed as references, i.e., the string object is not copied with the function call, but the function operates on the original strings in the main function. The **const** declarations in front of the arguments also tell, that the longerLength function is not going to modify the strings.

### Vector

Vector is a list of objects of a certain type that can be dynamically resized. It is a bit like a dynamic array in C, but C++ provides an abstract data type for handling a collection of objects, that is easier and safer to use than C arrays. The Vector implementation takes care of memory management and needed data structures, so our implementation does not need to take care of those (unlike in C).

The vector type definition requires an additional specification about what type of objects are stored in the vector. The stored object type is indicated inside "<" and ">".(These are called templates in C++)

Below there are three vectors, for storing 1) integer numbers; 2) strings; and 3) Car - type objects (we assume that in our include header "Car.hpp" we have defined the "Car" type).

#include <vector>

#include <string>

#include <Car.hpp>

std::vector<int> numbers = { 1, 2, 3 };

std::vector<std::string> words;

std::vector<Car> automobiles;

Vectors "words" and "automobiles" are empty by default, but we set the initial content of vector "numbers" to contain integers 1, 2 and 3.

New members can be added to the vector using the **push\_back()** function.

#include <vector>

#include <iostream>

int main(void)

{

std::vector<int> numbers;

numbers.push\_back(5);

numbers.push\_back(7);

std::cout << "Size: " << numbers.size() << std::endl;

}

Vectors can be used as function arguments or return values like any other type or class.

#include <vector>

#include <iostream>

int largest\_number(const std::vector<int>& v)

{

int largest = -1000; // hmm... -1000 is the smallest number

for (unsigned int i = 0; i < v.size(); i++) {

if (v[i] > largest)

largest = v[i];

}

return largest;

}

int main(void)

{

std::vector<int> numbers = { 1, 2, 3 };

numbers.push\_back(5);

numbers.push\_back(7);

std::cout << "Size: " << numbers.size() << std::endl

<< "Largest: " << largest\_number(numbers) << std::endl;

}

The **largest\_number()** function takes one argument, a reference to int-type vector. The argument is const, i.e., the function cannot modify the vector content. Similarly to C arrays, a subscript operator ([ ]) can be used to access a particular element in vector. Use of subscript is not safe against accessing out of bounds elements, and like in C, using invalid index causes a run-time failure. Therefore one needs to be aware of vector's current length before using it.

## Classes and Objects

One of the most important differences between C++ and C is that C++ supports **object oriented programming model**. In C++, data is modeled in a **class** that defines a structured data type (like "struct" in C), and a set of class-specific functions that are used to operate on the instances of the class.

Whereas in traditional C struct only defines data that is operated using globally named functions, a C++ class contains also the functions to operate on the data inside the class.

Some parts of the class definition are private, not intended to be accessed from outside the class. Other parts (typically the functions operating the class) are public, and can be used from other classes and software modules.

The public functions are the interface to the class, and the only means of the users of the class to interact with it. The implementation of the class logic and private data members can be changed as long as the interface semantics are not changed.

Classes are a good tool for implementing abstract data types: a type is accessed only through given function interfaces, and details of how data is stored are hidden in private part of the class.

Such model leads towards more robust software design. This separation also makes maintenance of larger software easier, because changes to the private parts of the class are not visible to the other software modules.

A class can have multiple instances (called objects) that differ by their data content (i.e., by their state), each using their required slot in computer memory.

The class interface is defined in the C++ header file. There are different naming conventions for header file names, but in this course we use ".hpp" for header. The functionality of the class is defined in files with ".cpp" suffix.

### 5.1 Simple class and its use

#include <string>

class Car

{

public:

Car(const std::string& reg); // construct a brand new car

void drive(double distance);

const std::string& getRegNro() const;

double getKM() const;

private:

std::string regnro;

double km; // distance the car has travelled

};

Class definition looks a bit like struct in C, but in addition to data fields, there are function definitions, and the definitions are split into public and private part. Even though the notation is a bit different from C, otherwise functions have similar parts as in C: return value (with defined type), name, and a list of arguments.

The data fields (regnro and km) of "Car" cannot be directly accessed because they are **private**, but there are functions **getRegNro()** for asking the register number of the car, and **getKM()** for asking the distance the car has travelled so far. These are defined with public visibility, and are therefore accesible from outside the class.

**const** at the end of the function definition tells that calling the function is not going to change the state of the instance, i.e. none of the object data is changed as a result of the function call. The **drive()** function increases the distance driven by the car, therefore it does not have the "const" specifier in the end.

On the other hand, **getRegNro()** returns a reference to string that cannot be modified, therefore there is also **const**declaration before the return value type. **const** in the parameter list means that the referenced variable cannot be modified.

In addition, there is a function that is named after the class name. This is the constructor of the class: it is the function that is executed when a new instance of the class is created.

#include "Car.hpp"

Car::Car(const std::string& reg)

{

regnro = reg;

km = 0;

}

void Car::drive(double distance)

{

km += distance;

}

const std::string& Car::getRegNro() const

{

return regnro;

}

double Car::getKM() const

{

return km;

}

each function definition is prefixed with the class name, because the function belongs to the class. As with namespaces in general, two colons are used in the names to separate the namespace. The class defines its own namespace, and other classes can have functions with same names.

The function implementations of a class assume the namespace of the class, and therefore can refer to the class members without explicit namespace definition. Therefore **regnro** and **km** can be used as in the example. Because they are private members, it would not be possible to refer to them from outside the class.

int main()

{

Car toyota("ABC-313");

Car mazda("XXX-666");

toyota.drive(10.5); // drive a bit

toyota.drive(4.1); // drive some more

std::cout << toyota.getRegNro() << " has driven "

<< toyota.getKM() << " km" << std::endl;

return 0;

}

The notation works similarly as when accessing struct members, except that we now have a function: dot (.) is used, unless the data type is a pointer that refers to class. If **toyota** had been a pointer to a class, we would use -> to refer its members, similar to C.

## 5.2 Interaction between classes

#include <vector>

#include "Car.hpp"

class Garage

{

public:

Garage() { }

void driveIn(Car& car);

void printCars();

private:

std::vector<Car> cars;

};

There is only one private member in the Garage class, a vector that stores Car-type objects. The constructor of the Garage class (line 7) does not have to do anything, so an empty implementation is sufficient. The vector is initialized automatically as an empty vector, so the constructor does not need to do anything about it.

when a function implementation is simple, it **can be included in the header file**inside the class definition. Therefore the constructor defintion is followed by brackets, and not semicolon. When the implementation block is given, the **semicolon is not needed**. The same trick can also be applied to other functions, but with common sense: only do this if your implementation has one or two simple statements. Otherwise it is better to just define the function interface in header, and have the implementation in the .cpp file as shown in earlier examples.

#include <iostream>

#include "Garage.hpp"

void Garage::driveIn(Car& car)

{

cars.push\_back(car);

}

void Garage::printCars()

{

std::vector<Car>::size\_type i;

for (i = 0; i < cars.size(); i++) {

std::cout << cars[i].getRegNro() << " (" << cars[i].getKM()

<< " km)" << std::endl;

}

}

The constructor for Garage is an empty implementation. This is sufficient, because the only member, 'cars' vector is automatically initialized to be empty. std::vector<Car>::size\_type. This is a type definition associated with the class that is used to refer to the size objects. It is similar to size\_t type defintion in C, and could in practice refer to unsigned int, for example. Garage class can refer to Car class, but only through its public interface. The private members of the Cars class are not accessible to Garage implementation.

### 5.3 Reference, Pointers and Classes

class Garage {

public:

Garage(Car c) { vehicle = c; }

private:

Car vehicle;

}

class Garage {

public:

Garage(Car& c) : vehicle(c) { }

private:

Car& vehicle;

};

class Garage {

public:

Garage(Car\* c) { vehicle = c; }

private:

Car\* vehicle;

};

In **variant (a)** the Car class content is stored as part of the Garage: whenever a new Garage object is created, the Car object is allocated at the same time. The content of the Car class is copied from the constructor argument c.

In **variant (b)** the Car class is stored elsewhere in the memory. However, the compiler tries to ensure that the reference variable always points to a valid Car class, and does its best to avoid memory reference errors. Thefore the vehicle member must be initialized as **explicit initialization** at the same time a new Garage object is created. No copying of Car class content is not involved in this case, which is more efficient that in variant (a), and usually desirable.

In **variant (c)** the Car class is stored elsewhere in the memory, but we are referencing to it using the good old pointer variable that contains a memory address of the object. In this case the compiler does not ensure that the pointer is valid, and because of programming mistake we may be using invalid pointer and cause a segmentation fault when using it. On the other hand, we can now also use a NULL pointer to indicate that the Car object is not set, which is not possible with reference variable. Although use of traditional pointers is still sometimes needed, it is good to try to avoid them in C++ when possible.