C++ Crash Course

Nicolas Rougier

C++ crash course for C programmers

Author: Nicolas P. Rougier
Sources: crash-course.rst

1 Foreword

This is an introduction to C++ for C programmers:

- If you can't understand the code below, you'd better start with a C tutorial.

```
#include <stdio.h>

void main (int argc, char **argv)
{
    printf("Hello World!\n");
}
```

- If you don't know what are the stack and the heap, you'd better have a look at some architecture & system introduction.
- If you know java, that might help a bit.
- If you think python is cool, you're right, but still, this is not the place.
- If you never heard about Bjarne Stroustrup, you might be at the right place.
- Here is a list of C++ specific keywords:

```
asm         dynamic_cast  namespace  reinterpret_cast  try
bool        explicit      new        static_cast       typeid
catch       false         operator   template          typename
class       friend        private    this              using
const_cast  inline        public     throw             virtual
delete      mutable       protected  true              wchar_t
```

2 From C to C++

Even if C++ is slanted toward object-oriented programming (OOP), you can nevertheless use any c++ compiler to compile c code and benefits from some c++ goodies.

2.1 Input/Output

Prefer the use of `<iostream>` for input/output operations (see stream section for explanation).

```
#include <iostream>

int main (int argc, char **argv)
{
    int i;
    std::cout << "Please enter an integer value: ";
    std::cin >> i;
    std::cout << "The value you entered is " << i  << std::endl;
    return 0;
}
```

2.2 New/Delete

The `new` and `delete` keywords are used to allocate and free memory. They are "object-aware" so you'd better use them instead of `malloc` and `free`. In any case, never cross the streams (new/free or malloc/delete).

```
int *a = new int;
delete a;

int *b = new int[5];
delete [] b;
```

`delete` does two things: it calls the destructor and it deallocates the memory.

2.3 References

A reference allows to declare an alias to another variable. As long as the aliased variable lives, you can use indifferently the variable or the alias.

```
int x;
int& foo = x;

foo = 42;
std::cout << x << std::endl;
```

References are extremely useful when used with function arguments since it saves the cost of copying parameters into the stack when calling the function.

2.4 Default parameters

You can specify default values for function parameters. When the function is called with fewer parameters, default values are used.

```
float foo( float a=0, float b=1, float c=2 )
{return a+b+c;}

cout << foo(1) << endl
     << foo(1,2) << endl
     << foo(1,2,3) << endl;
```

You should obtain values 4, 5 and 6.

2.5 Namespaces

Namespace allows to group classes, functions and variable under a common scope name that can be referenced elsewhere.

```
namespace first  { int var = 5; }
namespace second { int var = 3; }
cout << first::var << endl << second::var << endl;
```

You should obtain values 3 and 5. There exists some standard namespace in the standard template library such as std.

2.6 Overloading

Function overloading refers to the possibility of creating multiple functions with the same name as long as they have different parameters (type and/or number).

```
float add( float a, float b )
{return a+b;}

int add( int a, int b )
{return a+b;}
```

It is not legal to overload a function based on the return type (but you can do it anyway)

2.7 Const & inline

Defines and macros are bad if not used properly as illustrated below

```
#define SQUARE(x) x*x

int result = SQUARE(3+3);
```

For constants, prefer the const notation:

```
const int two = 2;
```

For macros, prefer the inline notation:

```
int inline square(int x)
{
    return x*x;
}
```

2.8 Mixing C and C++

```
#ifdef __cplusplus
extern "C" {
#endif

#include "some-c-code.h"

#ifdef __cplusplus
}
#endif
```

2.9 Exercises

1. Write a basic makefile for compiling sources

   solution: Makefile

2. How would you declare:

   - A pointer to a char
   - A constant pointer to a char
   - A pointer to a constant char

- A constant pointer to a constant char
- A reference to a char
- A reference to a constant char

solution: crash-course-2.1.cc

3. Create a two-dimensional array of integers (size is n x n), fill it with corresponding indices (a[i][j] = i*n+j), test it and finally, delete it.

   solution: crash-course-2.2.cc

4. Write a function that swap two integers, then two pointers.

   solution: crash-course-2.3.cc

5. Is this legal ?

   ```
   int add( int a, int b ) { return a+b; }

   int add( int a, int b, int c=0 ) { return a+b+c; }
   ```

   solution: crash-course-2.4.cc

6. Write a `const correct` division function.

   solution: crash-course-2.5.cc

7. What's the difference between `int const* p`, `int* const p` and `int const* const p` ?

   solution: crash-course-2.6.cc

3 Classes

A class migh be considered as an extended concept of a data structure: instead of holding only data, it can hold both data and functions. An object is an instantiation of a class. By default, all attributes and functions of a class are private (see below Access control). If you want a public default behavior, you can use keyword `struct` instead of keyword `class` in the declaration.

```
class Foo {
    int attribute;
    int function( void ) { };
};

struct Bar {
    int attribute;
    int function( void ) { };
};

Foo foo;
foo.attribute = 1; // WRONG

Bar bar;
bar.attribute = 1;  // OK
```

3.1 Constructors

It is possible to specify zero, one or more constructors for the class.

```
#include <iostream>

class Foo {
public:
    Foo( void )
    { std::cout << "Foo constructor 1 called" << std::endl; }
    Foo( int value )
    { std::cout << "Foo constructor 2 called" << std::endl; }
};

int main( int argc, char **argv )
{
    Foo foo_1, foo_2(2);
    return 0;
}
```

3.2 Destructor

There can be only one destructor per class. It takes no argument and returns nothing.

```
#include <iostream>

class Foo {
public:
    ~Foo( void )
    { std::cout << "Foo destructor called" << std::endl; }
}
int main( int argc, char **argv )
{
    Foo foo();
    return 0;
}
```

Note that you generally never need to explicitly call a destructor.

3.3 Access control

You can have fine control over who is granted access to a class function or attribute by specifying an explicit access policy:

- **public**: Anyone is granted access
- **protected**: Only derived classes are granted access
- **private**: No one but friends are granted access

3.4 Initialization list

Object's member should be initialized using initialization lists

```
class Foo
{
    int _value;
public:
    Foo(int value=0) : _value(value) { };
};
```

It's cheaper, better and faster.

3.5 Operator overloading

```
class Foo {
private:
    int _value;

public:
    Foo( int value ) : _value(value) { };

    Foo operator+ ( const Foo & other )
    {
        return Foo( _value+ other._value );
    }

    Foo operator* ( const Foo & other );
    {
        return Foo( _value * other._value );
    }
}
```

3.6 Friends

Friends are either functions or other classes that are granted privileged access to a class.

```
#include <iostream>

class Foo {
public:
    friend std::ostream& operator<< ( std::ostream& output,
                                      Foo const & that )
    {
        return output << that._value;
    }
private:
    double _value;
};

int main( int argc, char **argv )
{
  Foo foo;
  std::cout << "Foo object: " << foo << std::endl;
  return 0
}
```

3.7 Exercices

1. Why the following code doesn't compile ?

```
class Foo { Foo () { }; };

int main( int argc, char **argv )
{
    Foo foo;
}
```

solution: crash-course-3.1.cc

2. Write a `Foo` class with default and copy constructors and add also an assignment operator. Write some code to highlight the use of each of them.

   solution: crash-course-3.2.cc

3. Write a `Point` class that can be constructed using cartesian or polar coordinates.

   solution: crash-course-3.3.cc

4. Write a `Foo` class and provide it with an input method.

   solution: crash-course-3.4.cc

5. Is is possible to write something like `foo.method1().method2()` ?

   solution: crash-course-3.5.cc

4 Inheritance

4.1 Basics

Inheritance is done at the class definition level by specifying the base class and the type of inheritance.

```
class Foo                           { /* ... */ };
class Bar_public : public Foo       { /* ... */ };
class Bar_private : private Foo     { /* ... */ };
class Bar_protected : protected Foo { /* ... */ };
```

`Bar_public`, `Bar_private` and `Bar_protected` are derived from `Foo`. `Foo` is the base class of `Bar_public`, `Bar_private` and `Bar_protected`.

- In `Bar_public`, public parts of `Foo` are public, protected parts of `Foo` are protected
- In `Bar_private`, public and protected parts of `Foo` are private
- In `Bar_protected`, public and protected parts of `Foo` are protected

4.2 Virtual methods

A `virtual` function allows derived classes to replace the implementation provided by the base class (yes, it is not automatic...). Non virtual methods are resolved statically (at compile time) while virtual methods are resolved dynamically (at run time).

```
class Foo {
public:
    Foo( void );
    void method1( void );
    virtual void method2( void );
};

class Bar : public Foo {
public:
    Bar( void );
    void method1( void );
    void method2( void );
};

Foo *bar = new Bar();
bar->method1();
bar->method2();
```

Make sure your destructor is virtual when you have derived class.

4.3 Abstract classes

You can define pure virtual method that prohibits the base object to be instantiated. Derived classes need then to implement the virtual method.

```
class Foo {
public:
    Foo( void );
    virtual void method( void ) = 0;
};

class Bar: public Foo {
public:
    Foo( void );
    void method( void ) { };
};
```

4.4 Multiple inheritance

A class may inherit from multiple base classes but you have to be careful:

```
class Foo               { protected: int data; };
class Bar1 : public Foo { /* ... */ };
class Bar2 : public Foo { /* ... */ };
class Bar3 : public Bar1, public Bar2 {
    void method( void )
    {
       data = 1; // !!! BAD
    }
};
```

In class Bar3, the `data` reference is ambiguous since it could refer to Bar1::data or Bar2::data. This problem is referred as the **diamond problem**. You can eliminete the problem by explicitely specifying the data origin (e.g. Bar1::data) or by using virtual inheritance in Bar1 and Bar2.

4.5 Exercices

1. Write a `Bar` class that inherits from a `Foo` class and makes constructor and destructor methods to print something when called.

   solution: crash-course-4.1.cc

2. Write a `foo` function and make it called from a class that has a `foo` method.

   solution: crash-course-4.2.cc

3. Write a `Real` base class and a derived `Integer` class with all common operators (+,-,*,/)

   solution: crash-course-4.3.cc

4. Write a `Singleton` class such that only one object of this class can be created.

   solution: crash-course-4.4.cc

5. Write a functor class

   solution: crash-course-4.5.cc

5 Exceptions

5.1 The Zen of Python

(by Tim Peters)

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> **Errors should never pass silently.**
> **Unless explicitly silenced.**
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.

Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

5.2 Catch me if you can

You can catch any exception using the following structure:

```
try
{
    float *array = new float[-1];
}
catch( std::bad_alloc e )
{
    std::cerr << e.what() << std::endl;
}
```

If the raised exception is different from the ones you're catching, program will stop.

5.3 Creating your own exception

Creating a new exception is quite easy:

```
#include <stdexcept>

class Exception : public std::runtime_error
{
public:
    Exception() : std::runtime_error("Exception") { };
};
```

5.4 Standard exceptions

There exist some standard exceptions that can be raised in some circumstances:

```
#include <stdexcept>
```

- bad_alloc
- bad_cast
- bad_exception
- bad_typeid
- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - range_error
  - overflow_error
  - underflow_error

5.5 Exercices

1. How to handle a constructor that fails ?

   solution: crash-course-5.1.cc

2. Write a program that raise 3 of the standard exceptions.

   solution: crash-course-5.2.cc

3. Write a correct division function.

   solution: crash-course-5.3.cc

4. Write a `Integer` (positive) class with proper exception handling (`Overflow`, `Underflow`, `DivideByZero`, etc.)

   solution: crash-course-5.4.cc

6 Streams

C++ provides input/output capability throught the iostream classes that provide the stream concept (iXXXstream for input and oXXXstream for output).

6.1 iostream and ios

Screen outputs and keyboard inputs may be handled using the iostream header file:

```
#include <iostream>

int main( int argc, char **argv )
{

    unsigned char age = 65;
    std::cout << static_cast<unsigned>(age)     << std::endl;
    std::cout << static_cast<void const*>(&age) << std::endl;

    double f = 3.14159;
    cout.unsetf(ios::floatfield);
    cout.precision(5);
    cout << f << endl;
    cout.precision(10);
    cout << f << endl;
    cout.setf(ios::fixed,ios::floatfield);
    cout << f << endl;

    std::cout << "Enter a number, or -1 to quit: ";
    int i = 0;
    while( std::cin >> i )
    {
        if (i == -1) break;
        std::cout << "You entered " << i << '\n';
    }
    return 0;
}
```

6.2 Class input/output

You can implement a class input and output using friends functions:

```
#include <iostream>

class Foo {
public:
    friend std::ostream& operator<< ( std::ostream & output, Foo const & that )
    { return output << that._value; }
    friend std::istream& operator>> ( std::istream & input, Foo& foo )
    { return input >> fred._value; }

private:
    double _value;
};
```

6.3 Working with files

```
#include <fstream>

int main( int argc, char **argv )
{
    std::ifstream input( filename );
    // std::ifstream input( filename, std::ios::in | std::ios::binary);

    std::ofstream output( filename );
    // std::ofstream output( filename, std::ios::out | std::ios::binary);

    return 0;
}
```

6.4 Working with strings

```
#include <sstream>

int main( int argc, char **argv )
{
    const char *svalue = "42.0";
    int ivalue;
    std::istringstream istream;
    std::ostringstream ostream;

    istream.str(svalue);
    istream >> ivalue;
    std::cout << svalue << " = " << ivalue << std::endl;

    ostream.clear();
    ostream << ivalue;
    std::cout << ivalue << " = " << ostream.str() << std::endl;

    return 0;
}
```

6.5 Exercices

1. Write an `itoa` and an `atoi` function
2. Write a foo class with some attributes and write functions for writing to file and reading from file.

7 Templates

Templates are special operators that specify that a class or a function is written for one or several generic types that are not yet known. The format for declaring function templates is:

- template <typename identifier> function_declaration;
- template <typename identifier> class_declaration;

You can have several templates and to actually use a class or function template, you have to specify all unknown types:

```
template<typename T1>
T1 foo1( void ) { /* ... */ };

template<typename T1, typename T2>
T1 foo2( void ) { /* ... */ };

template<typename T1>
class Foo3 { /* ... */ };


int a = foo1<int>();
float b = foo2<int,float>();
Foo<int> c;
```

7.1 Template parameters

There are three possible template types:

- **Type**

```
template<typename T>  T foo( void ) { /* ... */ };
```

- **Non-type**

```
template<int N>  foo( void ) { /* ... */ };
```

- **Template**

```
template< template <typename T> > foo( void ) { /* ... */ };
```

7.2 Template function

```
template <class T>
T max( T a, T b)
{
    return( a > b ? a : b );
}

#include <sstream>

int main( int argc, char **argv )
{
    std::cout << max<int>( 2.2, 2.5 ) << std::endl;
    std::cout << max<float>( 2.2, 2.5 ) << std::endl;
}
```

7.3 Template class

```
template <class T>
class Foo {
    T _value;

public:
    Foo( T value ) : _value(value) { };
}

int main( int argc, char **argv )
{
    Foo<int> foo_int;
    Foo<float> foo_float;
}
```

7.4 Template specialization

```
#include <iostream>

template <class T>
class Foo {
    T _value;
public:
    Foo( T value ) : _value(value)
    {
        std::cout << "Generic constructor called" << std::endl;
    };
}

template <>
class Foo<float> {
    float _value;
public:
    Foo( float value ) : _value(value)
    {
        std::cout << "Specialized constructor called" << std::endl;
    };
}

int main( int argc, char **argv )
{
    Foo<int> foo_int;
    Foo<float> foo_float;
}
```

7.5 Exercices

1. Write a generic swap function
2. Write a generic point structure
3. Write templated factorial, power and exponential functions (exp(x) = sum_n x^n/n!, exp(-x) = 1/exp(x))
4. Write a smart pointer class

8 Standard Template Library

8.1 Containers

STL containers are template classes that implement various ways of storing elements and accessing them.

Sequence containers:

- vector
- deque
- list

Container adaptors:

- stack
- queue
- priority_queue

Associative containers:

- set
- multiset
- map
- multimap
- bitset

See http://www.cplusplus.com/reference/stl/ for more information.

```
#include <vector>
#include <map>
#include <string>

int main( int argc, char **argv )
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::map<std::string,int> m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    return 0;
}
```

8.2 Iterators

Iterators are a convebient tool to iterate over a container:

```
#include <map>
#include <string>
#include <iostream>

int main( int argc, char **argv )
{
    std::map<std::string,int> m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    std::map<std::string,int>::iterator iter;
    for( iter=m.begin(); iter != m.end(); ++iter )
    {
        std::cout << "map[" << iter->first << "] = "
                  << iter->second << std::endl;
    }
    return 0;
}
```

8.3 Algorithms

Algorithms from the STL offer fast, robust, tested and maintained code for a lot of standard operations on ranged elements. Don't reinvent the wheel !

Have a look at http://r0d.developpez.com/articles/algos-stl-fr/ (French) and http://www.cplusplus.com/reference/algorithm/ for an overview.

```
#include <vector>
#include <algorithm>

bool compare( const int & first, const int  & second )
{
    return (first < second);
}

int main( int argc, char **argv )
{
    std::vector<int> v(10);
    std::sort(v.begin(), v.end(), &compare);

    return 0;
}
```

8.4 Exercices

1. Write a template stack class using the STL vector class
2. Write a generic vector class with iterators and benchmark it againt the STL vector class

9 External links

- C++ FAQ — Frequently Asked Questions
  http://www.parashift.com/c++-faq-lite/
- Boost free peer-reviewed portable C++ source libraries
  http://www.boost.org/
- Bjarne Stroustrup homepage
  http://www2.research.att.com/~bs/
- Complete reference on C++ Standard Library
  http://en.cppreference.com/w/cpp

- C++11 main features
  http://en.wikipedia.org/wiki/C%2B%2B11
- The definitive C++ book guide
  http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- comp.lang.c++
  http://groups.google.com/group/comp.lang.c++/topics
- GNU make
  http://www.gnu.org/s/make/manual/make.html
- Les meilleurs cours et tutoriaux (in **French** as you may have already guessed...)
  http://cpp.developpez.com/cours/