Online C++ FAQ/Tutorial and Advanced Questions

Alexis Angelidis

January 11, 2005

1 Quick notes to C programmers

- instead of macros use
  - `const` or `enum` to define constants
  - `inline` to prevent function call overload
  - `template` to declare families of type and families of functions
- use `new`/`delete` instead of `free`/`malloc` (use `delete[]` for arrays)
- don't use `void*` and pointer arithmetic
- an explicit type conversion reveals an error of conception.
- avoid to use C style tables, use vectors instead.
- don't recode what is already available in the C++ standard library.
- variables can be declared anywhere: initialization can be done when variable is required.
- whenever a pointer cannot be zero, use a reference.
- when using derived class, destructors should be virtual.

2 What is C++

C++ is C with classes. It was designed for one person to manage large amounts of code, and so that a line of C++ express more thing than a line of C. The main functionalities C++ adds to C are:

- control of the accessibility to code within the source files (`namespace`, `struct` and `class`).
- mechanisms that make a line of code more expressive (constructors, destructors, operators, ...).
- object programming (class derivation).
- generic programming (templates).

3 How do I use namespaces?

Namespace allow to code different entities without bothering about unicity of names. A namespace is a logical entity of code. A namespace can be included into another namespace (and so on). If a namespace is anonymous, its code is only accessible from the compilation unit (.cpp file), thus equivalent to using `static` and `extern` in C.

```
// in file1.cpp

namespace name1
{
  void function1() { /* ... */ }

  namespace name2
  {
    void function2() { /* ... */ }
  }
}

namespace // anonymous namespace
{
  void  function() { /* ... */ }
}

void function3() { function(); }          // ok

// in file2.cpp

void function4a() { function1(); }              // error
void function4c() { name1::function2(); }       // error
void function4c() { function(); }               // error
void function4b() { name1::function1(); }       // ok
void function4d() { name1::name2::function2(); } // ok
using namespace name1;           //makes accessible the entire namespace name1
void function4e() { function1(); }              // ok
void function4f() { name2::function1(); }       // ok
using name1::name2::function2;   //makes accessible function2 only
void function3g() { function2(); }              // ok
```

4 How do I use references?

A reference is merely an initialized pointer. This reduces significantly zero pointer and un-initialized pointers errors. Prefer references to pointers. Although the two following codes look equivalent, the second implementation prevents invalid compilation.

```
// in C                     // in C++

int     i;                  int    i;
int     *pi = &i;           int    &ti = i;
int     *pj;                int    &rj; // g++ refuses to compile that
```

```
*pi = 0;                    ri = 0;
*pj = 0; // segfault        rj = 0;
```

5 Functions

5.1 How do I declare, assign and call a pointer to a function?

```
int f(int, char)
{
  // ...
}

// ...

int (*pf)(int, char); // pointer to a function

pf = f;  // assignment
pf = &f; // alternative

int r = (*pf)(42, 'a');
```

5.2 How do I declare a type of a pointer to a function?

```
typedef int (*pf)(int, char);
```

5.3 How do I declare an array of pointers to a function?

```
typedef int (*pf)(int, char);

pf  pfarray[10]; // array of 10 pointers to a function
```

6 class and struct

6.1 How do I use class and struct?

A C++ `struct` (or `class`) is almost like a struct in C (i.e. a set of attributes), but has two additional kinds of members: *methods* and *data types*. A class' method has to be used with an instance of that class. The important is that methods have always access to their instance's attributes and data types.

```
struct A
{
  typedef char  t_byte; // type
  unsigned char i;      // attribute
  void          m()     // method
  {
    t_byte      b = 1;  // m can access type t_byte
```

```
    i = b;              // m can access attribute i
  }
};

void function()
{
  A     a;

  a.m();               // an instance is required to call a method.
}
```

6.2 When do I use a class or a struct?

In `struct`, the members are `public` by default. In `class`, the members are `private` by default (Question 6.3 explains `private` and `public`). Since in C++ privacy is a vertue, prefer `class` to `struct`.

6.3 Who has access to class members?

There are three levels of access to members

- `private` : access is granted only to the class' methods and to `friend` functions and classes (Question 6.17 explains `friend`).
- `protected` : access is granted only to the methods and to derived classes' methods.
- `public` : access is granted to everyone.

Restricting access to members is usefull for detecting illicit use of the members of a `class` when compiling, as shown in the following code.

```
class A
{
private:
  int   a0;
  void  f0() { /* ... */ }
protected:
  int   a1;
  void  f1() { f0(); }      // ok
public:
  int   a2;
  void  f2() { /* ... */ }
};

void    function()
{
  A     a;

  a.a0 = 0;                 // error
  a.f0();                   // error
  a.a1 = 0;                 // error
```

```
  a.f1();                  // error
  a.a2 = 0;                // ok
  a.f2();                  // ok
}
```

6.4 How do I use private, protected or public?

Restricting access to member is important to prevent illicit use. Use them in this order of increasing preference: `public`, `protected`, `private`.

6.5 How do I create an instance of a class?

The methods called when a class is created are called *contructors*. There are four possible ways of specifying constructors; the fifth method is worth mentioning for clarifying reasons:

- default constructor
- copy constructor
- value constructor
- conversion constructor
- copy assignment (not a constructor)

```
struct A
{
  A() { /* ... */ }                          // default constructor
  A(const A &a) { /* ... */ }                // copy constructor
  A(int i, int j) { /* ... */ }              // value constructor
  A     &operator=(const A &a) { /* ... */ } // copy assignment
};

struct B
{
  B() { /* ... */ }           // default constructor
  B(const A &a) { /* ... */ } // conversion constructor
};

void function()
{
  A a0(0, 0); // shortcut, value constructor
  A a1(a0);   // shortcut, copy constructor
  B b1(a1);   // shortcut, conversion constructor
  B b;        // shortcut, default constructor

  b1 = a0;        // conversion contructor
  a0 = a1;        // copy assignment
}
```

6.6 How do I initialize members of a class?

There are two ways.

```
struct  A
{
  int   a;
  int   b;
  A(): a(0) { b = 0; } // attribute a and b are initialized to 0
};
```

6.7 How do I initialize a const member?

Since the value of a const member cannot be assigned with operator =, its value must be initialized as follows:

```
struct A
{
  const int   id;
  A(int i): id(i) {} // attribute id is initialized to the value of parameter i
};
```

6.8 How do I call a parent constructor?

```
struct A
{
  A() { /* ... */ }
};

struct B
{
  B(): A() { /* ... */ } // call to parent's constructor.
};
```

6.9 What is a destructor?

See Question 6.10.

6.10 How do I free the memory allocated to a class?

The method called when the memory occupied by a class is freed is called the *destructor*. With derived classes, destructor should always be virtual. If a class is destroyed through a base class pointer whose destructor isn't virtual, the result is undefined (only part of the destructors will be called).

```
struct A
{
  A() {}
  virtual ~A() {}
};
```

```
struct B: public A
{
  B() {}
  ~B() {}
};

void function()
{
  B *b = new B();
  A *a = b;

  delete a; // calls ~A and ~B. If ~A wasn't virtual, only ~A would be called.
}
```

6.11 How do I put the code of methods outside classes?

It is possible to put in a class only the prototype of the methods, and to put all the algorithms outside. This is recommended, because it allows the programmer to read a short prototype of the class, and makes re-usability easier:

```
// in a header file (.h file)

struct  A
{
  int   a;
  A();
  virtual ~A();
  void    f();
};

// in a compilation unit (.cpp file)

A::A()
{
  /* ... */
};

A::~A()
{
  /* ... */
};

void    A::f()
{
  /* ... */
};
```

6.12 How do I put the code of methods outside classes in a header file?

The code of a method specified in a header file must be declared `inline`. It means that when compiling, the calls to the method will all be replaced by the code of that method.

```
struct  A
{
  int   a;
  A();
};

inline A::A()
{
  /* ... */
};
```

6.13 How do I declare, assign and call a pointer to a method?

Note that a pointer to a method does not hold a second pointer to an instance of a class. To use a pointer to a method, you need an instance of the class onto which this method can be called (possibly a second pointer).

```
struct A
{
  void m(int) {}
};

void    function()
{
  void  (A::*pm)(int) = &A::m; // pointer on method
  A     a;                     // instance of a class

  (a.*m)(1);                   // calling the method with parameter value 1.
}
```

6.14 How can I handle easily a pointer to a method?

If you use templates, you won't have to write the annoying type of a pointer to a method (at the 'cost' of using a template). You can also use `typedef` to declare the type.

```
struct A
{
  void m() {}
};

template <class PMETHOD>
void    f(A       &a
```

```
          PMETHOD pm)
{
  (a.*pm)()
}

void    function()
{
  A     a;

  f(a, &A::m);
}
```

6.15 How do I declare a method that returns a pointer to a function?

The following method takes parameter a char and returns a pointer to a function. To avoid this heavy syntax, you may use a typedef.

```
struct A
{
  void (*m(char))(int) { /* ... */ }
};

// ...

A       a;
void   (*pf)(int);

pf = a.m('a');
```

6.16 How do I specify a pointer to a static method?

It works exactly like pointers to functions in C.

```
struct A
{
  static void sm(int) {}
};

void    function()
{
  void  (*psm)(int) = A::sm;  // assignment
  void  (*psm)(int) = &A::sm; // alternative

  (*psm)(1);                  // calling the method with parameter value 1.
}
```

6.17 How can I access the private part of a class from another class or function?

A class can define friends: functions or classes. The friends will have access to the private part of that class. If a function is friend, it has to be prototyped or defined **before** specifying its friendship.

```
class A
{
};

void f();       // f prototyped before;

class B;
{
  friend A;        // A can access private part of B
  friend void f(); // f can access private part of B
};

void f()
{
  /* ... */
}
```

6.18 How do I prevent the compiler from doing implicit type conversions?

Use the keyword `explicit` on the constructor. Forcing explicit conversions is usefull to make the programmers aware of the conversion. This is especially interesting for time consuming conversions.

```
struct A
{
  A() {}
};

struct B
{
  B() {}
  B(const A &a) {}
};

struct C
{
  C() {}
  explicit C(const A &a) {}
};

void fb(B b) { /* ... */ }
```

```
void fc(C c) { /* ... */ }

void function()
{
  A a;
  B b;
  C c;

  fb(a);    // ok, conversion is implicit
  fc(a);    // error
  fc(C(a)); // ok, conversion is explicit
}
```

6.19 When should I use const methods?

When a method isn't going to modify a class, it should be `const`. This prevents from modifying attributes unwantedly, and reduces significantly errors:

```
struct A
{
  int   a;
  bool  f(int i) const
  {
    if (a = i)     // error. f shouldn't  modify a.
      return true;
    return false;
  }
};
```

6.20 How do I modify attributes in a const method?

When you have no other choice (which happens), use the keyword `mutable`:

```
struct A
{
  int           a;
  mutable int   b;
  void f() const
  {
    a = 0; // error
    b = 0; // ok
  }
};
```

6.21 What are static members?

Static members are members that exist independently from an instantiation of a class. They are shared by all instances of that class, and can be used without requiring an

instance. Only methods and attributes can be static members[1]. A static attribute must be defined in a .cpp file.

```
struct A
{
  static int id;
  static int genId() { return id++; }
};

int      A::id = 0; // defined in a .cpp file
```

6.22 When should I use a static method or a function?

A static method has full access to the members of a class. If this isn't required, the method should be a function.

```
class   A
{
private:
  int   i;
public:
  static void f(A &a)
  {
    a.i = 0;
  }
};
```

6.23 When should I use a static method or a friend function?

A static method has full access to the members of a single class. A function can be the friend of more than one class, and therefore can have access to the members of one of more classes.

6.24 When should I use a global variable or a static attribute?

If possible, avoid global variables.

6.25 How do I call a static member outside a class?

```
struct A
{
  static int id;
  static int genId() { return id++; }
};
```

[1]This adds a third meaning to the keyword static: the first is a local definition of a function or variable inside a compilation unit, the second is a variable instantiated only once inside a function or method.

```
int     function()
{
  A::id = 0;          // call to a static attribute
  return A::gendId(); // call to a static method
}
```

6.26 How do I derive classes?

The purpose of deriving classes is to factor code: if two classes derive from a *parent* class, the members defined in the *parent* will be accessible by both, and have to be coded only once. The level of access to a parent's member is specified with `public`, `private` and `protected`. The following examples show the three types of derivation and their effect.

```
class A
{
  /* ... */
};

//access to members of A are transmitted to B.
class B: public A
{
  /* ... */
};

//public members of A become protected members of C
class C: protected A
{
  /* ... */
};

//public and protected members of A become private members of D
class D: private A
{
  /* ... */
};
```

6.27 How do I avoid ambiguities with multiple class derivation?

Consider the two following valid examples: the left one if non-ambiguous, the right one is.

```
struct A
{
  void a() {}
};

struct B: public virtual A
{
```

```
struct A
{
  void a() {}
};

struct B: public A
{
```

```
};

struct C: public virtual A
{
};

struct D: public B, public C
{
};

void        function()
{
  D        d;

  d.a();
}
```

```
};

struct C: public A
{
};

struct D: public B, public C // D has two a
{
};

void function()
{
  D        d;

  d.a();    // error, ambiguous
  d.B::a(); // ok
  d.C::a(); // ok
}
```

6.28 What is a virtual method?

A virtual method in a parent allows children to have a different implementation for it. A pure virtual method in a parent forces children to have an implementation for it (interface in Java). A class with a pure virtual method is called virtual.

```
struct A
{
  virtual void f1() = 0;
  virtual void f2() { /* ... */ }
};

struct B: public A
{
  void f1() { /* ... */ }
};

struct C: public A
{
  void f1() { /* ... */ }
  void f2() { /* ... */ }
};
```

6.29 What is a pure virtual method?

See Question 6.28.

6.30 What are templates?

Template allow the programmers to implement algorithms once for various data types. Contrarily to macros, the compiler checks the syntax. Functions, methods and classes can be templated. Template parameters are of two kinds: data types or integers.

6.30.1 How do I specify a function template?

In a header (.h file) :

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

6.30.2 How do I specify a class template?

In a header (.h file). The template parameter can be used for defining any member of the class.

```
template <class T, int N>
class   Vector
{
  T array[N];
  void method(T t, int i) { array[i] = T; }
};
```

6.30.3 How do I specify a template method?

In a header (.h file) :

```
class   Vector
{
  int     array[3];

  template <class TVECTOR2>
  void  eqAdd(TVECTOR2 v2);
};

template <class TVECTOR2>
void Vector::eqAdd(TVECTOR2 a2)
{
  for (int i(0); i < 3; ++i) array[i] += a2[i];
}
```

6.30.4 How do I put the code of template methods outside classes?

```
template <class T, int N>
class   Vector
{
  T array[N];
  void reset();
};

template <class T, int N>
void Vector<T, N>::reset()
```

```
{
  for (int i(0); i < N; ++i) array[i] = 0;
}
```

6.30.5 How do I write a template method of a template class?

The syntax is a bit heavy. There is no point of using it unless you realy need to.

```
template <class T, int N>
class   Vector
{
  T array[N];
  template <class F>
  void apply(F f);
};

template <class T, int N>
template <class F>
void Vector<T, N>::apply(F function)
{
  for (int i(0); i < N; ++i) array[i] = f(array[i]);
}
```

6.30.6 How do I specify a friend template class?

Like that:

```
class A
{
  template<class> friend class B;
  friend class B <class T>;
};
```

6.30.7 How do I write different code for different template parameters?

Use specializations:

```
template <class T, int N>
class Vector
{
  T a[N];
public:
  Vector(const T v) { for (unsigned i(0); i < N; ++i) a[i] = v; }
};

template <>
class Vector <double, 3>
{
  double x, y, z;
public:
```

```
  Vector(const double v): x(v), y(v), z(v) {}
};
```

6.31 How do I write to the standard output in a C++ way?

Include iostream and use the operator <<.

```
std::cout << "Hello!" << std::endl;
```

6.32 How do I read from the standard input in a C++ way?

Include iostream and use the operator >>. For a string of undefined length, use `getline`.

```
float f;
char str[255];

std::cin >> f;
std::cin.getline(str, 255);
```

6.33 How do I specify a copy constructor for a class whose code is not accessible?

Use the keyword `operator`.

```
// suppose class A's code is not accessible
A::A(const int i) { /* ... */ }

// This does not prevent you to make a B to A convertor

struct B
{
  int b;
  B(const int i): b(i) {}
  operator A() { return A(b); }
};
```

6.34 How do I redefined arithmetic operators?

There are two ways of redefining an operator: with a method or with a function (usually friend function, for most operators need to access the private members of a class). Some operators can only be redefined with a method.

6.34.1 Method/function operators

The following operators can be redefined with a method or a function: binary +, unary +, binary −, unary −, ∗, /, %, ==, ! =, &&, ||, !, <, >, <=, >=, + =, − =, ∗ =, / =, % =, &, |, ∧, ∼, <<, >>, & =, | =, ++, −−. The advantage of a function

operator over a method operator is the possibility to define it independently from the class. For example, to use a method:

```
struct   Number
{
  Number  operator+(const Number &n) const { /* ... */ }
  Number  operator-() const { /* ... */ }
  bool    operator==(const Number &n) const { /* ... */ }
  Number  &operator+=(const Number &n) { /* ... */ }
  Number  &operator++() { /* ... */ }   // postfixed
  Number  operator++(int) { /* ... */ } // prefixed
  double  &operator*() { /* ... */ }    // prefixed
  double  &operator->() { /* ... */ }   // prefixed
};

struct   Stream
{
  Stream   &operator<<(const Number &n) { /* ... */ }
};
```

They can also be redefined with a function (possibly friend). For example:

```
Number  operator+(const Number &n0, const Number &n1) { /* ... */ }
Number  operator-(const Number &n) { /* ... */ }
bool    operator==(const Number &n0, const Number &n1) { /* ... */ }
Stream  &operator<<(Stream &is, const Number &n) { /* ... */ }
Number  &operator+=(Number &n0, const Number &n1) { /* ... */ }
Number  &operator++(const Number &n) { /* ... */ }     // postfixed
Number  operator++(const Number &n, int) { /* ... */ } // prefixed
double  &operator*(const Number &n) { /* ... */ }      // prefixed
double  &operator->(const Number &n) { /* ... */ }     // prefixed
```

6.34.2 Method-only operators

The following operators can be redefined with a non-static method only: (), []. For example:

```
struct    Matrix
{
  double &operator()(int i, int j) { /* ... */ }
};

struct    Vertex
{
  double &operator()(int i) { /* ... */ }
  double &operator[](int i) { /* ... */ }
};
```

6.35 When should I use operator [] or ()?

Use `()` instead of `[]`: calling `[]` on a pointer to the array (instead of the array) will compile too, but will have a different effect.

6.36 When should I use operator i++ or ++i?

Since `i++` returns a copy of `i`, it is preferable to use `++i`.

6.37 How do I cast types?

Remember that an explicit type conversion often reveals an error of conception. To cast a variable `a` into another type `T`, use one of the following:

```
static_cast<T>(a)      // explicit and standard conversion
dynamic_cast<T>(a)     // validity of object checked at run-time
reinterpret_cast<T>(a) // binary copy
const_cast<T>(a)       // changes the constness
```