Introduction to C++11 and its use inside Qt

Olivier Goffart

February 2013

About Me

woboq

http://woboq.com

http://code.woboq.org

History

- 1979: C With classes
- 1983: C++
- 1991: Qt development begins (Qt 1.0 in 1995)
- 1998: C++98 - First standardized C++
- 2003: C++03 - Only few corrections
- 2005: Qt 4.0
- 2011: C++11
- 2012: Qt 5.0

Goals of C++11

From the C++11 FAQ:

- Make C++ a better language for systems programming and library building

  that is, to build directly on C++'s contributions to programming, rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development).

- Make C++ easier to teach and learn

  through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts).

Rules of Thumbs

From the C++11 FAQ:

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

Example

```
1  QString processString(const QString &str)
2  {
3      enum State {Ok, Cancel};
4      QVector<QPair<State, QString>> stack;
5      stack.append({State::Ok, str});
6      //...
7  }
```

What is specific to C++11 in this code?

Example

```
1  QString processString(const QString &str)
2  {
3      enum State {Ok, Cancel};
4      QVector<QPair<State, QString>> stack;
5      stack.append({State::Ok, str});
6      //...
7  }
```

What is specific to C++11 in this code?

- use of bracket initilizer

Example

```
1  QString processString(const QString &str)
2  {
3      enum State {Ok, Cancel};
4      QVector<QPair<State, QString>> stack;
5      stack.append({State::Ok, str});
6      //...
7  }
```

What is specific to C++11 in this code?

- use of bracket initilizer
- use of >> without a space

Example

```
1  QString processString(const QString &str)
2  {
3      enum State {Ok, Cancel};
4      QVector<QPair<State, QString>> stack;
5      stack.append({State::Ok, str});
6      //...
7  }
```

What is specific to C++11 in this code?

- use of bracket initilizer
- use of >> without a space
- Scoped `State::Ok`

Example

```
1  QString processString(const QString &str)
2  {
3      enum State {Ok, Cancel};
4      QVector<QPair<State, QString>> stack;
5      stack.append({State::Ok, str});
6      //...
7  }
```

What is specific to C++11 in this code?

- use of bracket initilizer
- use of >> without a space
- Scoped `State::Ok`
- use of local type `State` as template parameter

New Features

auto

```
1  QString Obj::foo(int blah)
2  {
3    QHash<int, QString> dict = bar();
4    auto it = dict.find(blah);
5    //...
6  }
```

Member Initializers

```
1  class X {
2    //  ...
3  private:
4    bool m_enabled = false;
5    int m_state = 0;
6    QByteArray m_name = "None";
7  }
```

Range Based for

```
1  bool Obj::foo(const QStringList &list)
2  {
3      for(const QString &it : list) {
4          //...
5      }
6  }
```

Uniform Initialisation

In C++98

```
1  int b(1);  // variable definition
2  QVariant c(); // function declaration
3  QVariant d(QString()); // oops
4
5  QPoint p(1,2);
6  int i = {2}; //yes, you can do that
7  QString a[] = { "foo", "bar" }; // ok
8  //QVector<QString> v = { "foo", "bar" }; // error
```

Uniform Initialisation

In C++11

```
1  int b{1};  // variable definition
2  QVariant c{}; // default constructor
3  QVariant d{QString()}; // Works as expected
4
5  QPoint p{1,2};
6  int i = {2}; // = is optional
7  QString a[] { "foo", "bar" };
8  QVector<QString> v = { "foo", "bar" }; // works
```

Uniform Initialisation

In C++11

```
1  struct X { int a, b; };
2  X x1 = X{1,2};
3  X x2 = {1,2}; // the = is optional
4  X* p = new X{1,2};
5
6  struct D : X {
7      D(int x, int y, int z) : X{x,y}, c{z}
8      { /* ... */ };
9      int c;
10 };
```

Uniform Initialisation

In C++11

```
1  X foo(const X &a) {
2      X b{a};
3      return {2,3};
4  }
5  foo({4,5});
6
7  X x5{4.2, 4.5}  //error: narrowing
8
9  //using initializer_list constructor (from Qt 4.8)
10 QVector<X> { {1,2}, {2,4}, {4,5} };
```

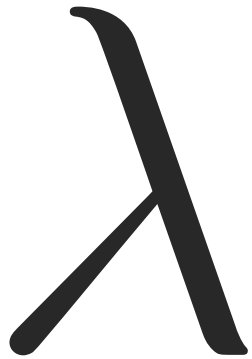
initializer_list

```
#ifdef Q_COMPILER_INITIALIZER_LISTS
template <typename T>
QVector<T>::QVector(std::initializer_list<T> args)
{
 d = Data::allocate(args.size());
 copyConstruct(args.begin(), args.end(), d->begin());
 d->size = args.size();
}
#endif
```

C K&R syntax

```
1  int plus(x, y)
2      int x;
3      int y;
4  {
5       return x + y;
6  }
```

Lambda



Lambda

```
[foo] (int a) -> int { return a + foo; }
```

- Capture: Variables that you capture
- Parametter list: The perametters of the function
- Return type (optional)
- Function body

Lambda

```
[foo] (int a) -> int { return a + foo; }
```

```
struct {
    double foo;
    int operator()(int a)
    { return a + foo; }
}
```

Lambda Example

```
1  QList<int> f(const QList<int> &list, double foo)
2  {
3      auto functor = [foo](int a) { return a + foo; };
4      return QtConcurrent::mapped(list, functor);
5  }
```

Lambda capture

```
1  int a{1}, b{2}, c{3};
2
3  // 'a' by value, 'b' by reference
4  auto f1 = [a, &b]() { b = a; };
5
6  // everything by reference
7  auto f2 = [&]() { b = a; };
8
9  // everything by value
10 auto f3 = [=]() { return a + c; };
11
12 // everything by value, 'b' by reference
13 auto f4 = [=,&b]() { b = a + c; };
```

Lambda Example (Qt5)

```
1  void Doc::saveDocument() {
2    QFileDialog *dlg = new QFileDialog();
3    dlg->open();
4    QObject::connect(dlg, &QDialog::finished,
5                     [dlg, this](int result) {
6      if (result) {
7        QFile file(dlg->selectedFiles().first());
8        // ...
9      }
10     dlg->deleteLater();
11   });
12 }
```

Function Declaration

```
1  struct Foo {
2      enum State { /*...*/ };
3      /*...*/
4      State func(State);
5  };
6
7  //Error
8  State Foo::func(State arg) { /*...*/ }
```

Function Declaration

```
1  struct Foo {
2      enum State { /*...*/ };
3      /*...*/
4      State func(State);
5  };
6
7  //Error
8  State Foo::func(State arg) { /*...*/ }
```

Alternative function syntax

```
1  auto Foo::func(State arg) -> State
2  { /* ... */ }
```

decltype

```
1  template <typename A, typename B>
2  auto dotProduct(A vect_a, B vect_b)
3    -> decltype(vect_a[0]*vect_b[0])
4  {  /*...*/  }
5
6
7
8  QList<int> a{/*...*/};
9  QVector<float> b{/*...*/};
10 auto p = dotProduct(a, b);
```

Default and Deleted function

```
1  struct MyObject {
2      virtual ~MyObject() = default;
3      MyObject(int);
4      MyObject() = default;
5      int someFunction(QObject *);
6      int someFunction(QWidget *) = delete;
7  };
```

```
1  #define Q_DISABLE_COPY(Class) \
2      Class(const Class &) = delete; \
3      Class &operator=(const Class &) = delete;
```

Q_DECL_DELETE

```
1  struct MyObject {
2      void myFunction(int);
3      // ...
4    private:
5      void myFunction(bool) Q_DECL_DELETE;
6  };
```

Rvalue References

```
1  struct string {
2    char *data;
3    string(const char *str) : data(strdup(str)) {}
4    string(const string &o) : data(strdup(o.data)) {}
5    ~string() { free(data); }
6    string &operator=(const string &o) {
7      free(data); data = strdup(o.data)
8      return *this;
9    }
10 };
11
12 template<class T> swap(T &a, T &b) {
13   T tmp(a); a = b; b = tmp;
14 }
```

Rvalue References

```
1  struct string {
2    /* ... */
3    string(const string &o); // copy constructor
4    string(string &&o) // move constructor
5          : data(o.data) { o.data = 0; }
6    string &operator=(const string &o); // copy assignme
7    string &operator=(string &&o) // move assignment
8    { swap(data, o.data); }
9  };
10
11 template<class T> swap(T &a, T &b) {
12   T tmp = std::move(a);
13   a = std::move(b);
14   b = std::move(tmp);
15 }
```

Rvalue References

```
1  X a;
2  X f();
3
4  X &r1 = a;     // bind r1 to a (an lvalue)
5  X &r2 = f();   // error: f() is an rvalue; can't bind
6
7  X &&rr1 = f(); // ok: bind rr1 to temporary
8  X &&rr2 = a    // error: bind a is an lvalue
9  X &&rr3 = std::move(a) // ok
```

Move Sementic

```
1  #ifdef Q_COMPILER_RVALUE_REFS
2  inline QString::QString(QString &&other)
3      : d(other.d)
4  {
5    other.d = Data::sharedNull();
6  }
7
8  inline QString::QString &operator=(QString &&other)
9  { qSwap(d, other.d); return *this; }
10 #endif
```

Perfect Forwarding

```
1  template <class T> class QSharedPointer {
2      /* ... */
3  #if defined(Q_COMPILER_RVALUE_REFS) \
4      && defined(Q_COMPILER_VARIADIC_TEMPLATES)
5   template <typename... Args>
6   static QSharedPointer<T> create(Args && ... args)
7   {
8    QSharedPointer<T> result(Qt::Uninitialized);
9    result.d = Private::create(&result.value);
10
11   // now initialize the data
12   new (result.data()) T(std::forward<Args>(args)...);
13   result.d->setQObjectShared(result.value, true);
14   return result;
15  }
16 #endif
```

constexpr

```
1  switch (btn) {
2      case Qt::LeftButton: /*...*/ break;
3      case Qt::RightButton: /*...*/ break;
4
5      // error?  the operator| convert to QFlags
6      case Qt::LeftButton | Qt::RightButton:
7      /*...*/ break;
8      default: /*...*/
9  }
```

constexpr

```
1  template<typename Enum> class QFlags {
2      int i;
3  public:
4      constexpr QFlags(Enum e) i{e} {}
5      constexpr operator int() { return i; }
6      //...
7  };
8
9  #define Q_DECLARE_OPERATORS_FOR_FLAGS(ENUM) \
10 constexpr inline QFlags<ENUM> operator|(ENUM a,
11                                         ENUM b) \
12 { return QFlags(ENUM(a | b));  }
```

constexpr

```
1  template<typename Enum> class QFlags {
2      int i;
3  public:
4    Q_DECL_CONSTEXPR QFlags(Enum e) i{e} {}
5    Q_DECL_CONSTEXPR operator int() { return i; }
6      //...
7  };
8
9  #define Q_DECLARE_OPERATORS_FOR_FLAGS(ENUM) \
10 Q_DECL_CONSTEXPR inline QFlags<ENUM> operator| \
11                                 (ENUM a,ENUM b) \
12 { return QFlags(ENUM(a | b)); }
```

Explicit Override

```
1  struct A {
2      virtual void foo(int);
3      virtual void bar(int) const;
4  };
```

```
1  struct B : public A {
2      void foo(float); // oops
3      void bar(int); // oops
4  };
```

Explicit Override

```
1  struct A {
2      virtual void foo(int);
3      virtual void bar(int) const;
4  };
```

```
1  struct B : public A {
2      void foo() override; // compilation error
3      void bar(int) override; // compilation error
4      void baz(int) override; // compilation error
5      void foo(int) override; // ok
6  };
7
8  struct C : public A {
9      void foo(int) final;
10 };
```

Explicit Override

```
1  struct A {
2      virtual void foo(int);
3      virtual void bar(int) const;
4  };
```

```
1  struct B : public A {
2      void foo() Q_DECL_OVERRIDE; //compilation error
3      void bar(int) Q_DECL_OVERRIDE; //compilation erro
4      void baz(int) Q_DECL_OVERRIDE; //compilation erro
5      void foo(int) Q_DECL_OVERRIDE; //ok
6  };
7
8  struct C : public A {
9      void foo(int) Q_DECL_FINAL;
10 };
```

String Literals

Unicode strings

```
1 //utf-8
2 const char *str1 = u8"v\u00e6r s\u00e5 god";
3
4 //utf-16
5 const char16_t *str2 = u"v\u00e6r s\u00e5 god";
6
7 //utf-32
8 const char32_t *str3 = U"v\u00e6r s\u00e5 god";
```

```
1   QStringLiteral("Hello");
```

(http://woboq.com/blog/qstringliteral.html)

Raw Literals

C++98

```
1 ret.replace(QRegExp("(\\\\*)\""),
2               "\"\\1\\1\\^\"\"")
```

Raw litteral

```
1 ret.replace(QRegExp(R"-((\\*)")-"),
2               R"("\1\1\^"")");
```

enum class

```
1  enum Color { Blue, Red, Yellow };
2  enum class State { Ok, Cancel };
3  enum class SomeEnum; //forward declare (sizeof(int))
4
5   enum OtherEnum : char {A, B}; // specify the size
6
7  Color color1 = Blue, color2 = Color::Red;
8  State state1 = Cancel; //error
9  State state2 = State::Cancel;
```

...

- `nullptr`
- Generalized POD
- Generalized unions
- user defined litteral
- explicit conversion operators
- `noexcept` (`Q_DECL_NOTHROW` , `Q_DECL_NOEXCEPT`, `Q_DECL_NOEXCEPT_EXPR`)
- delegating constructors
- Inline namespace

Templates

- Variatic template
- Default template arguments for function templates
- SFINAE for expressions
- Extern templates
- Local and unnamed types as template arguments
- Template aliases
- `static_assert` (`Q_STATIC_ASSERT` , `Q_STATIC_ASSERT_X`)

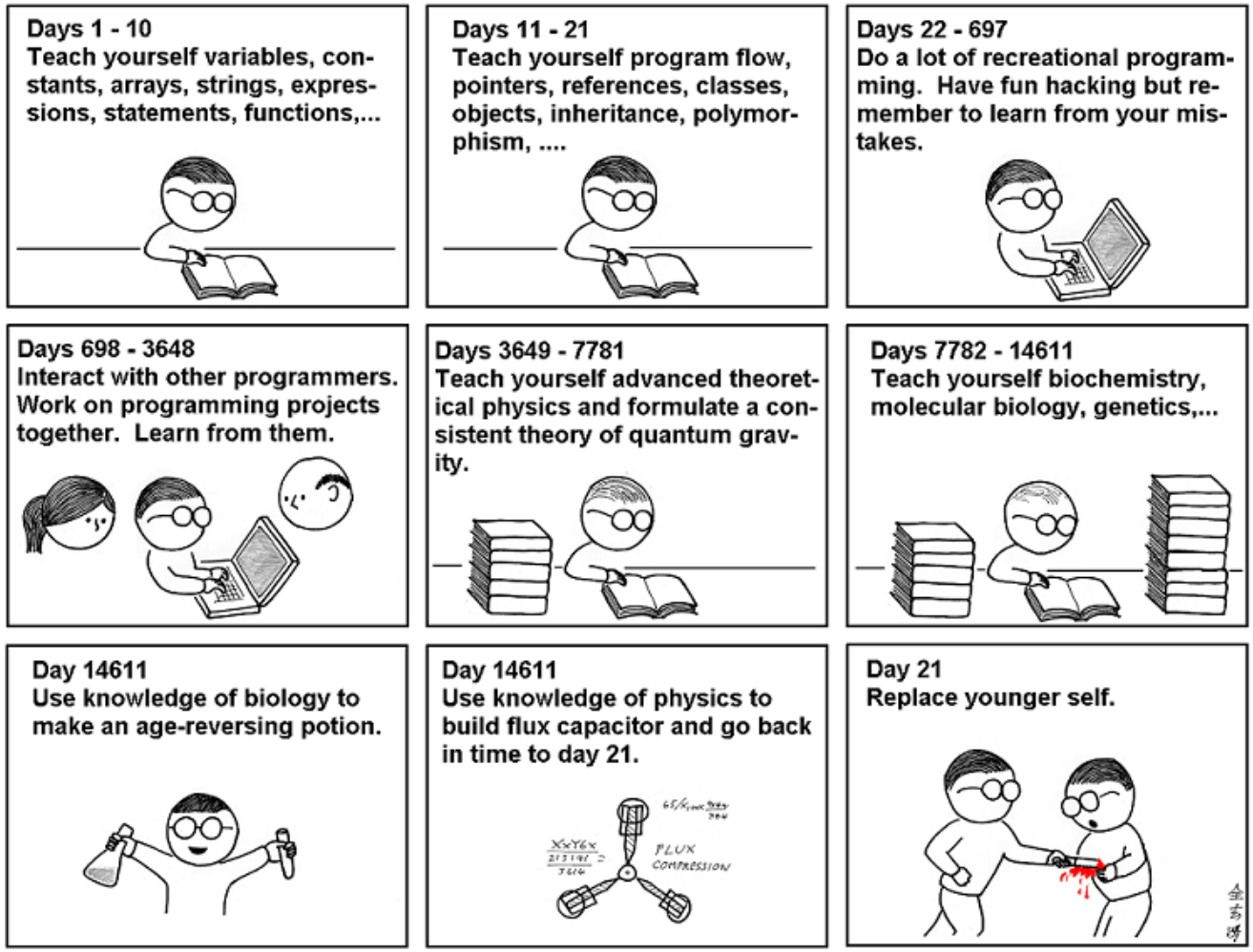
Concurrency

- Memory model
- Thread Local Storage (`thread_local`)
- Atomic operations
- futures, mutexes, ...

More STL Features

- `unordered_map` (QHash)
- Smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`)
- `bind`, `function`
- more algorithms (`min`, `max`, `minmax`, `all_of`, `any_of`, `is_sorted`, `copy_if`, `copy_n`, `partition_copy`, `iota` ...)
- Regular expressions
- Random number generation
- ...

http://abstrusegoose.com/249



The End

```
CONFIG += c++11
```

Questions

olivier@woboq.com

visit http://woboq.com