What is the level of my current skills in C++?

(C) Richèl Bilderbeek

June 6, 2015

Chapter 1

Overview

1. Introduction
2. Domain #1

3. Domain #2
4. Domain #3
5. Domain #4

1.1 Introduction

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it [1]

- What is the level of my current skills in C++?
- Visit the subdomains of C++ to assess this
- Question: Can you name some programming ideals?
- Question: Can you name the subdomains of C++?

[1] Alan Perlis

1.2 Programming Ideals[2]

- Correctness
- Reliability
- Affordable
- Maintainable

[2] Stroustrup. Programming. §1.6

1.3 C++

View C++ as a federation of languages [3]:

- C
- Object-Oriented C++
- Template C++
- The STL

Effective programming requires that you change strategy when you switch from one sublanguage to another [4]

[3]Scott Meyers. Effective C++ (3rd edition). Item 1: 'View C++ as a federation of languages' (also list from this reference)

[4]Scott Meyers. Effective C++ (3rd edition). Item 1: 'Effective programming requires that you change strategy when you switch from one sublanguage to another'

Chapter 2

Domain #1: C

2.1 Domain #1: C

- Working with built-in types
- Function design
- The best C is not always the best C++

2.2 C: working with built-in types

- Type choice
- Choice of modifers: const, static, volatile
- Consequences of this choice

2.3 C example: working with built-in types

What can be concluded from the following code?

```
unsigned int n_countries = 27;
```

2.4 C example: working with built-in types

What can be concluded from the following code?

```
unsigned int n_countries = 27;
```

Conclusions

- n_countries is probably a number of countries
- n_countries is always positive
- n_countries will not be used in arithmetic [1] or will be checked for its range when doing arithmetic with it [2]

[1] Bjarne Stroustrup. The C++ Programming Language(3rd edition). Chapter 4.10 'Advice', item 18: 'Avoid unsigned arithmetic'

[2] C++ FAQ Lite [29.12] '[...] at least if you are careful to check your ranges'

- n_countries must be checked for implicit conversions [3]
- n_countries will have its value changed at least once

[3]Bjarne Stroustrup. The C++ Programming Language (3rd edition). Chapter 4.10 'Advice', item 19: 'View signed to unsigned and unsigned to signed conversions with suspicion'

2.5 C example: working with built-in types

What can be concluded from the following code?

```
const int n_countries = 27;
```

2.6 C example: working with built-in types

What can be concluded from the following code?

```
const int n_countries = 27;
```

Conclusions

- n_countries is probably a number of countries
- n_countries is always positive
- n_countries can be used inituitively in arithmetic
- n_countries will never have its value changed

2.7 C: function design

- Return type choice
- Argument type choice
- Name choice
- Error handling policy

2.8 C example: function design

Comment on the following. Assume all code resides in C-only code.

```
int Sum(int * a, int * b);

int DisplayValue(const int value); /* returns an error code

void Swap(T * const lhs_array, T * const rhs_array);

const double * Divide(const double numerator, const double n
```

2.9 C example: function design

Comment on the following. Assume all code is found in C++ code.

```
const int GetRows(const Database d);

void Set(std::vector<std::vector<double> >& v, const int& y,
  const double& value);

const int Swap(int& a, int& b);

int DisplayValue(const int value); //returns an error code

const double MeanAndStdDev(const std::vector<double>& v, dou

void CoutWidget(const Widget& w);

void SetSquare(const Square& s, const Color& c, RubiksCube&
```

2.10 C example: exercise

Write a safe and correct Divide function, that divides two doubles.

For example, if this function would be called with 3.0 as a numerator and 4.0 as denominator, it should somehow produce 0.75.

2.11 C example: exercise answer #1

```
const double Divide(
  const double numerator,
  const double denominator)
{
  assert(denominator != 0.0);
  if (denominator == 0.0)
  {
    throw std::logic_error("Cannot divide by 0.0");
  }
  return numerator / denominator;
}
```

2.12 C example: exercise answer #2

```
const std::vector<double> Divide(
  const double numerator,
  const double denominator)
{
  std::vector<double> v;
  if (denominator != 0.0)
  {
    v.push_back(numerator / denominator);
  }
  return v;
}
```

2.13 C example: exercise answer #3

Comment on this correct solution

```
const double * Divide(
  const double numerator,
  const double denominator)
{
  return denominator == 0.0
    ? 0 //C++11: nullptr
    : new double(numerator / denominator);
}
```

2.14 C example: exercise answer #4

Comment on this correct solution

```
const boost::scoped_ptr<double> Divide(
  const double numerator,
  const double denominator)
{
  boost::scoped_ptr<double> p;
  if (denominator != 0.0)
  {
    p.reset(new double(numerator / denominator));
  }
  return p;
}
```

2.15 C example: exercise answer #5

Comment on this correct solution

```
void Divide(
  const double numerator,
  const double denominator,
  std::vector<double>& v)
{
  assert(v.empty());
  assert(v.capacity() >= 1);
  if (denominator != 0.0)
  {
    v.push_back(numerator / denominator);
  }
  return v;
}
```

2.16 C example: bigger exercise

Write a safe and correct quadratic equation function

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Remember, the number of solutions is dependent on the discriminant, $\sqrt{b^2 - 4ac}$: if it is bigger than zero, there are two solutions, if it is equal to zero there is one solution, if it is less than zero there are no solutions.

Chapter 3

Domain #2: Object-Oriented C++

3.1 Domain #2: Object-Oriented C++

- Member function design
- Class design
- Design Patterns

3.2 OO C++: Member function design

- Function design
- Choise of modifiers: const, static

3.3 OO C++ example: Member function design

```
struct Person
{
  const bool IsFemale();

  //...
};
```

3.4 OO C++ example: Member function design

```
struct Database
{
  Data * GetData() const;

  //...

  private:
  Data * m_data;

  //...
};
```

3.5 OO C++ example: Member function design

```
struct Line
{
  ///Calculate the length of a line using the Pythagorian eq
  ///dx: horizontal length of line
  ///dy: vertical length of line
  const double Length(const double dx, const double dy) cons

  //...
};
```

3.6 OO C++: Class design

- Member function design
- Member variable type choice
- Choice of member variable modifiers: const, mutable, static, volatile
- Interface design
- The Big Four
- Class hierarchy
- Design Patterns

3.7 OO C++ example: Class design

```
struct Person
{
  //...

  private:
  bool m_is_female;

  //...
};
```

3.8 OO C++ example: Class design

```
struct Database
{
  void Init();

  //...
};
```

3.9 OO C++ example: Class design

```
struct Data {
  ///Expensive calculation
  const int Sum() {
    ++m_cnt;
    //...
  }
  //...
  private:
  ///Monitor how often Sum is called
  int m_cnt;
  //...
};
```

3.10 OO C++ example: Class design

```
///PrimeNumber can only contain numbers that are prime
struct PrimeNumber
{
  PrimeNumber();
  void SetValue(const int prime_number);

  //...
};
```

3.11 OO C++ example: Class design

```
template <class T>
struct SmartPointer
{
  SmartPointer() : m_p(new T) {}
  ~SmartPointer() { delete m_p; }
  T * Get() { return m_p; }
  private:
  T * m_p;
};
```

3.12 OO C++ example: Class design

```
struct Parameters
{
  int m_x;

  //...
};

struct Simulation : public Parameters
{
  const int GetX() const { return m_x; }

  //...
}
```

3.13 OO C++ example: Design Patterns

A Design Pattern is 'a description of communicating objects and classes that are customized to solve a general design problem in a particular context', examples: [1]

- Command: encapsulates a request as an object
- Decorator: attach additional responsibilities to an object dynamically
- Iterator: provide a way to access the elements of an aggregate object sequentially
- Observer: when one object changes state, all its dependents are notified and updated

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. 1995. ISBN: 0201633612.

- State: allow an object to alter its behavior
- Strategy: defines a family of algorithms, encapsulates each one, and makes them interchangeable

3.14 OO C++ example: Design Patterns

```
struct Duck
{
  virtual void Fly() = 0;
  virtual void Quack() = 0;
  //...
};

struct FlyWithWingsNormalQuackDuck : public Duck {};
struct FlyRocketPoweredNormalQuackDuck : public Duck {};
struct FlyWithWingsSqeakDuck : public Duck {};
struct FlyRocketPoweredSqueakDuck : public Duck {};
```

3.15 OO C++ example: Strategy Design Pattern 1/3

```
struct Duck
{
  void Fly() { m_fly_behavior->Fly(); }
  void Quack() { m_quack_behavior->Quack(); }
  void SetFlyBehavior(
    const std::shared_ptr<const FlyBehavior> fb);
  void SetQuackBehavior(
    const std::shared_ptr<const QuackBehavior> qb);

  private:
  std::shared_ptr<const FlyBehavior> m_fly_behavior;
  std::shared_ptr<const QuackBehavior> m_quack_behavior;
};
```

3.16 OO C++ example: Strategy Design Pattern 2/3

```
struct FlyBehavior
{
  virtual ~FlyBehavior() {}
  virtual void Fly() = 0;
};

struct FlyWithWings : public FlyBehavior {
  void Fly() { /* */ }
};

struct FlyRocketPowered : public FlyBehavior {
  void Fly(){ /* */ }
};
```

3.17 OO C++ example: Strategy Design Pattern 3/3

```
struct MallardDuck : public Duck {
  MallardDuck()
  {
    //Set default behaviors
  }
};

struct SuperDuck : public Duck {
  SuperDuck()
  {
    //Set default behaviors
  }
};
```

Chapter 4

Domain #3: Template C++

4.1 Domain #3: Template C++

- What is it?
- Host class design
- Policy design

4.2 Template C++: what is it?

- More than containers of T
- Calculations and checks that run at compile-time
- No cost in run-time speed!
- Examples
  - Compile-time assert
  - Compile-time calculations
  - Compile-time polymorphism
  - Unit checking
  - Lookup tables

4.3 Template C++: Compile-time assert

```
template<bool> struct CompileTimeAssert;
template<> struct CompileTimeAssert<true> {};

int main()
{
  CompileTimeAssert< 1+1 == 2 >();
  CompileTimeAssert< 1+1 == 3 >(); //Will not compile
}
```

4.4 Template C++: Compile-time calculation

```
template <unsigned int N> struct factorial {
  static unsigned const value
  = N * factorial<N-1>::value;
};

template <> struct factorial<0> {
  static unsigned const value = 1;
};

int main() {
  CompileTimeAssert<(factorial<5>::value==120)>();
}
```

4.5 Template C++: Compile-time polymorphism

```
///A compile-time Strategy Design Pattern
enum Policy { A, B };

template <Policy> struct Strategy
{
  static void DoIt();
};

template<> void Strategy<A>::DoIt()
{
  //Do it the A way
}
```

```
template<> void Strategy<B>::DoIt()
{
  //Do it the B way
}

int main()
{
  const Strategy<A> x; x.DoIt();
  const Strategy<B> y; y.DoIt();
}
```

4.6 Template C++: Unit checking

```
int main()
{
  //Create a length
  const boost::units::quantity<boost::units::si::length> m(
    1.0 * boost::units::si::meter);
  //Create another length
  const boost::units::quantity<boost::units::si::length> n(
    1.0 * boost::units::si::milli * boost::units::si::meter)
  //Create a force
  const boost::units::quantity<boost::units::si::force> f(
    1.0 * boost::units::si::newton);
  //Add the two lengths
  std::cout << (m + n); //OKAY: can add meters to meters
  //Try to add force to a length
  std::cout << (m + f); //FAILS: cannot add newtons to meter
```

}

Chapter 5

Domain #4: The STL

5.1 Domain #4: The STL

- Container choice
- Iterators
- Algorithm choice
- Smart pointer choice

5.2 The STL: Containers

- How many do you know?
- When to use which one?

5.3 The STL: Containers

- std::string
- std::vector
- std::set
- std::map
- std::list

5.4 The STL: Containers

- std::string: text
- std::vector: dynamic-sized array
- std::set: sorts items, stores each instance once
- std::map: lookup table

5.5 The STL: Containers

- Use std::string and std::vector by default
- Both can communicate with C API's

5.6 The STL: Iterators

- Allows a uniform way to work with STL containers
- Some iterators might be implemented as plain pointers
- Examples:
  - std::vector<int>::iterator
  - std::set<std::string>::iterator
  - std::back_inserter
  - std::ostream_iterator

5.7 The STL: Iterators

```
//v is a container of type T, e.g. std::vector<int> or std::
const T::const_iterator j = v.end();
for (T::const_iterator i = v.begin(); i!=j; ++i)
{
  std::cout << (*i) << '\n';
}
```

5.8 Algorithms question

- What are algorithms?
- Why use algorithms?

5.9 Algorithms answers

- What are algorithms?
  - named operations on multiple elements
- Why use algorithms?
  - verbosity/readability
  - increase run-time speed: naive for-loops might result in higher Big-O

5.10 Algorithm example

```
template<typename In, typename Out, typename Pred>
Out Copy_if(In first, In last, Out res, Pred Pr)
{
  while (first != last)
  {
    if (Pr(*first)) *res++ = *first;
    ++first;
  }
  return res;
}
```

5.11 The STL: Algorithms

- How many do you know?
- When to use which one?
- How to extend these?

5.12 The STL: Some algorithm names

- (amongst) others: std::sort, std::random_shuffle, std::for_each, std::accumulate, std::transform
- Some use predicates: std::copy_if (accidentally ommitted in C++98 standard), std::count_if
- Some expected sorted ranges: std::binary_search, std::merge

5.13 The STL: Algorithm example

```
//Write all elements to std::cout
std::copy(
  v.begin(),
  v.end(),
  std::ostream_iterator<std::string>(
    std::cout,"\n"
  )
);
```

5.14 The STL: Extending algorithms

- Algorithms can be extended by functors
- Functor: class that has a defined function call operator

5.15 The STL: Extending algorith example

```
struct GetStringLengther {
 int operator()(const std::string& s) const { return static_
};

int main()
{
  std::vector<std::string> v = /* */;
  //Obtain the std::string lengths present
  std::set<int> w;
  std::transform(v.begin(),v.end(),
    std::inserter(w,w.begin()),GetStringLengther());
}
```

5.16 The STL: Smart pointers

- RAII idiom