From NAG to C++

Richel Bilderbeek 

June 6, 2015

Chapter 1

Introduction

1.1 About NAG

- NAG is a commercial library
- NAG is developed by the Numerical Algorithms Group
- For C++ the NAG C library can be used
- The NAG C library supports multithreading

1.2 Goal

- How to work with C++ The Right Way?

1.3 Overview

- C example from NAG: 'Hello NAG'
- Initial conversion from C to C++
- Analyse this initial solution
- Improve solution
- Discuss trade-offs in solution

Chapter 2

From NAG...

2.1 Hello NAG

- The example following is adapted from 'Essential Introduction to NAG'[1]
- Question: what is the benefit of using NAG in the example?

[1] http://www.nag.com/numeric/CL/nagdoc_cl23/html/GENINT/essint.html

2.2 Example: Hello NAG for C

```
/* For C compiler */
#include <nag.h>
#include <nag_stdlib.h>

int main(void)
{
  char * s = 0;
  s = NAG_ALLOC(12,char);
  if (!s) return 1;
  strcpy(s, "Hello NAG");
  puts(s);
  NAG_FREE(s);
  return 0;
}
```

2.3 Example: Hello NAG without NAG

```
/* For C compiler */
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char * s = 0;
  s = malloc(22 * sizeof(char)); /* NAG_ALLOC(22,char); */
  if (!s) return 1;
  strcpy(s, "Hello NAG without NAG");
  puts(s);
  free(s); /* NAG_FREE(s); */
  return 0;
}
```

2.4 Exercise: Hello NAG in C++

- The best C style is not always the best C++ style
- Convert the code to the best C++ possible
- Only care about the operations in the code

2.5 Example: Hello NAG in C++

```
/* For C++ compiler */
int main(void)
{
  char * s = 0;
  s = NAG_ALLOC(32, char)
  if (!s)
  {
    /* ... */
    return 1;
  }
  /* ... */
  NAG_FREE(s);
  return 0;

}
```

2.6 (Initial) correct solution

- C++ specification that a function takes no arguments
- C++ comment
- Const-correct
- Implicit return

2.7 (Initial) correct solution 1/4

- In C: a function that has no arguments has '(void)'

```
int main(void) {}
```

- In C++: a function that has no arguments has '()'

```
int main() {}
```

- Benefits
  - less typing
  - using preferred syntax

2.8 (Initial) correct solution 2/4

- In C: comments only '/* */'

```
/*  ...  */
```

- In C++: comments '/* */' for multi-line and '//' for single-line at end of line

```
//...
```

- Benefits
  - Connect forget a closing '*/'
  - Single line comment can be enclosed in a multi-line comment

2.9 (Initial) correct solution 3/4

- In C: all declarations at beginning of function

```
char * s = 0;
s = NAG_ALLOC(32, char);
```

- In C++: declarations can be postponed until definition

```
char * const s = NAG_ALLOC(32, char);
```

- Benefits
  - No uninitialized variables
  - RAII ('Resource Aquisition Is Initialization') idiom
  - Const-correctness

2.10 (Initial) correct solution 4/4

- In C, main must return an int

```
int main(void)
{
  /* ... */
  return 0;
}
```

- In C++, main implicitly returns zero

```
int main() { /* */ }
```

- Benefits
  - No needless typing

2.11 (Initial) correct solution

- Next slide: result of putting these in
- Question: can the resulting code be improved?
  - Exception safety
  - Scalability

2.12 (Initial) correct solution code

```
#include <nag.h>
#include <nag_stdlib.h>

int main()
{
  char * const s = NAG_ALLOC(32,char);
  if (!s)
  {
    //...
    return 1;
  }
  //...

  NAG_FREE(s);
}
```

2.13 (Initial) correct solution code critique

- 's' is not initialized with an initial value (only allocated memory for these)
- 's' cannot hold const chars, because the chars must be set after allocation their memory
- 's' has size 32, this might be concluded from an initial value
- 's' will only be freed if the code reaches the NAG_FREE in the end
- Thrown exceptions must be caught to free the resources
- Scales badly for more allocations
- Question: How to solve all this critique?

Chapter 3

... to C++

3.1 Resolving the (initial) correct solution code critique

- Use a class
- Exercise
  - Write a class that
    - has a proper name
    - allocates its resources automatically
    - always has an initial value
    - frees its resources automatically
  - Use this class in main
  - Initialize it with a text
- Do not care about copying behavior (yet)

3.2 Answer outline

```
struct NagString
{
  NagString(const char * const s) { /* ... */ }
  ~NagString() { /* ... */ }
  const char * const m_s;
};

int main()
{
  const NagString s("Hello NAG");
}
```

3.3 NagString in detail 1/2

```
struct NagString
{
  explicit NagString(const char * const s)
    : m_s(DeepCopy(s)) { }

  ~NagString() { NAG_FREE(m_s); }

  static const char * DeepCopy(const char * const s)
  {
    //...
  }
  const char * const m_s;
};
```

3.4 NagString in detail 2/2

```
struct NagString
{
  static const char * DeepCopy(const char * const s)
  {
    if (!s) return 0;
    //Care about trailing \0
    char * const t
      = NAG_ALLOC(std::strlen(s) + 1,char);
    if (!t) throw std::bad_alloc();
    std::strcpy(t,s);
    return t;
  }
};
```

3.5 NagString exercise

- Strong points?
- Weak points?
- Points that are unknown to be strong or weak?

3.6 NagString exercise answer 1/2

- Strong
  - RAII idiom
  - Simple interface
  - m_s can be read without making a copy
- Weak
  - Copy constructor is not disabled nor implemented correctly
  - operator= is not disabled nor implemented correctly
  - m_s can still be deleted

3.7 NagString exercise answer 2/2

- Unknown
  - Does allowing empty strings benefit the client?
  - Strong or no guarantee, depending on NAG
    - Strong if NAG_FREE has nothrow guarantee
    - None if NAG_FREE can throw
- Bonus: why not use std::string?

3.8 Exercise

- Write a (one of many) correct NagString class implementation
- Next: two possible solutions
  - Which tradoffs are chosen for?

3.9 Answer 1

```
struct NagString {
  explicit NagString(const char * const s) { /* */ }
  ~NagString() { /* */ }
  static const char * Create(const char * const s) {
    assert(s);
    //...
  }

  const char * const m_s;

  private:
  NagString(const NagString& rhs); // = delete
  NagString& operator=(const NagString& rhs); // = delete
};
```

3.10 Answer 2

```
struct NagString {
  explicit NagString(const char * const s) { /* */ }
  ~NagString() { /* */ }
  static const char * Create(const char * const s) {
    if (!s) throw std::invalid_argument("s null");
    //...
  }

  const std::string Get() const { return std::string(m_s); }

  private:
  NagString(const NagString& rhs); // = delete
  NagString& operator=(const NagString& rhs); // = delete
  const char * const m_s;
};
```

Chapter 4

Conclusion

4.1 Conclusion

- The best C style is not the best C++ style
- C++ allows for scalability, exception safety, automatic memory management
- C++ allows for tradeoffs