A C++ Class Designer's Checklist

Steven J. Zeil

August 28, 2013

Outline

Outline I

1. The Checklist
2. Discussion and Explanation
   - Is the interface complete?
   - Are there redundant functions or functions that can be generalized?
   - Have you used names that are meaningful in the application domain?
   - Preconditions and Assertions
   - Are the data members private?
   - Does every constructor initialize every data member?
   - Have you appropriately treated the default constructor?
   - Have you appropriately treated the "Big 3"?
   - Does your assignment operator handle self-assignment?
   - Does your class provide == and < operators?

Outline II

Video Recording

A recording of this presentation is available.

The Checklist I

1. Is the interface complete?
2. Are there redundant functions in the interface that could be removed? Are there functions that could be generalized?
3. Have you used names that are meaningful in the application area?
4. Are pre-conditions and assumptions well documented? Have you used `assert` to "guard" the inputs?
5. Are the data members private?
6. Does every constructor initialize every data member?
7. Have you appropriately treated the default constructor?
8. Have you appropriately treated the "big 3" (copy constructor, assignment operator, and destructor)?
9. Does your assignment operator handle self-assignment?

The Checklist II

10. Does your class provide == and < operators?
11. Does your class provide an output routine?
12. Is your class const-correct?

Purpose

This is a *checklist*

- Use it whenever you have the responsibility of designing an ADT interface
- These are not absolute rules, but are things that you need to think about
  - Violate them if necessary, but only after careful consideration

Outline I

1. The Checklist

2. Discussion and Explanation
   - Is the interface complete?
   - Are there redundant functions or functions that can be generalized?
   - Have you used names that are meaningful in the application domain?
   - Preconditions and Assertions
   - Are the data members private?
   - Does every constructor initialize every data member?
   - Have you appropriately treated the default constructor?
   - Have you appropriately treated the "Big 3"?
   - Does your assignment operator handle self-assignment?
   - Does your class provide == and < operators?

Outline II

- Does your class provide an output routine?
- Is your class const-correct?

3 Summary

Is the interface complete?

An ADT interface is *complete* if it contains all the operations required to implement the application at hand (and/or reasonably probable applications in the near future).

The best way to determine this is to look to the requirements of the application.

Is Day complete?

day.h

For example, if we were to look through proposed applications of the `Day` class and find designs with pseudocode like:

```
if (d is last day in its month)
   payday = d;
```

we would be happy with the ability of our `Day` interface to support this.

Is Day complete? (2)

On the other hand, if we encountered a design like this:

```
while (payday falls on the weekend)
   move payday back one day
end while
```

we might want to consider adding a function to the Day class to get the day of the week.

Are there redundant functions or functions that can be generalized?

- Avoid unneded redundancies that make your class larger and provide additional code to maintain.
- Generalize when possible to provide more options to the application.

Example: Day output

```
class Day
{
public:
  ⋮
  void print() const;
private:
  ⋮
};
```

Future applications may need to send their output to different places. So, it makes sense to make the print destination a parameter:

```
void print (std::ostream& out);
```

Day output op

Of course, most C++ programmers are used to doing output this way:

```
cout << variable;
```

rather than

```
variable.print (cout);
```

So we would do better to add the operator. . .

Day output op

```
class Day
{
public:
  ⋮
  void print(std::ostream out) const;
private:
  ⋮
};

inline
std::ostream& operator<< (std::ostream& out, Day day)
{
  dat.print (out);
  return out;
}
```

Have you used names that are meaningful in the application domain?

An important part of this question is the "*in the application domain*".

- Good variable names should make sense to anyone who understands the application domain,
- even if they don't understand (yet) how your program works.

Example: Book identifiers

- `getTitle`, `putTitle`, or `getNumberOfAuthors` are all fine.
- Not everyone who has worked with books would understand `getIdentifier`
- Better would be to recognize that suitable unique identifiers already exist

```
class Book {
public:
  ⋮
    std::string getISBN();
  ⋮
};
```

  - the ISBN appears on the copyright page of every published book.

Are pre-conditions and assumptions well documented?

A *pre-condition* is a condition that the person calling a function must be sure is true, before the call, if he/she expects the function to do anything reasonable.

- Pre-conditions must be documented because they are an obligation upon the caller
  - And callers can't fulfill that obligation if they don't know about it

Example: Day Constructor

day.h What pre-condition would you impose upon the *Day* constructor?

- A fairly obvious requirement is for the month and day to be valid.

```
Day(int aYear, int aMonth, int aDate);
//pre: (aMonth > 0 && aMonth <= 12)
//   && (aDate > 0 && aDate <= daysInMonth(aMonth, a
```

  - All pre-conditions are, by definition, boolean expressions
  - As we will see shortly, there's a significant advantage to writing them as proper C++ expressions

Example: Day Constructor (cont.)

This comment

```
class Day
{
   /**
      Represents a day with a given year, month, and day
      of the Gregorian calendar. The Gregorian calendar
      replaced the Julian calendar beginning on
      October 15, 1582
   */
   ⋮
```

suggests a more rigorous pre-condition

```
  Day(int aYear, int aMonth, int aDate);
  //pre: (aMonth > 0 && aMonth <= 12)
  //  && (aDate > 0 && aDate <= daysInMonth(aMonth,aYear))
  //  && (aYear > 1582 || (aYear == 1582 && aMonth > 10)
  //        || (aYear == 1582 && aMonth == 10 && aDate >= 15)
```

Example: MailingList getContact

mailinglist.h What pre-condition, if any, would you write for the `getContact` function?

Example: MailingList getContact

mailinglist.h What pre-condition, if any, would you write for the `getContact` function?

```
// Find and retrieve contacts
bool contains (const Name& name) const;

Contact& getContact (const Name& name) const;
//pre: contains(name)
```

Have you used `assert` to "guard" the inputs?

An `assert` statement takes a single argument,

- a boolean condition that we believe should be true unless someone somewhere has made a programming mistake.
- It aborts program execution if that condition evaluates as false.
    - Can be "turned off" in release versions by a simple compiler switch
        - Though that might not be a good thing...

Example: Guarding the Day Constructor

```
#include "day.h"
#include <cassert>

using namespace std;

Day::Day(int aYear, int aMonth, int aDate)
//pre: (aMonth > 0 && aMonth <= 12)
//   && (aDate > 0 && aDate <= daysInMonth(aMonth,aYear))
//   && (aYear > 1582 || (aYear == 1582 && aMonth > 10)
//         || (aYear == 1582 && aMonth == 10 && aDate >= 15)
{
  assert (aMonth > 0 && aMonth <= 12);
  assert (aDate > 0 && aDate <= 31);
  assert (aYear > 1582 || (aYear == 1582 && aMonth > 10)
          || (aYear == 1582 && aMonth == 10 && aDate >= 15
  daysSinceStart = ...
}
```

Example: guarding getContact I

```
#include "mailinglist.h"
#include <cassert>

using namespace std;


Contact& MailingList::getContact (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
```

Example: guarding getContact II

```
      current = current->next;
    }
  assert (current != NULL
      && name == current->contact.getName());
  return current->contact;
}
   ⋮
```

Do Assertions Reduce Robustness?

Why do

```
assert (current != NULL
        && name == current->contact.getName());
```

instead of making the code take corrective action? E.g.,

```
if (current != NULL
   && name == current->contact.getName())
   return current->contact;
else
   return Contact();
```

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.
  - The ability to recover from errors is called *robustness*.

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.
  - The ability to recover from errors is called *robustness*.
  - Robust handling of human interface errors is a good thing

Do Assertions Reduce Robustness?

- Many people argue in favor of making code “tolerate” errors.
    - The ability to recover from errors is called *robustness*.
    - Robust handling of human interface errors is a good thing
- However, tolerance of pre-conditions is reckless.

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.
  - The ability to recover from errors is called *robustness*.
  - Robust handling of human interface errors is a good thing
- However, tolerance of pre-conditions is reckless.
  - A pre-condition violation is evidence of a bug in the *application*.

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.
  - The ability to recover from errors is called *robustness*.
  - Robust handling of human interface errors is a good thing
- However, tolerance of pre-conditions is reckless.
  - A pre-condition violation is evidence of a bug in the *application*.
  - If the application is under test, it's self-defeating to hide mistakes.

Do Assertions Reduce Robustness?

- Many people argue in favor of making code "tolerate" errors.
  - The ability to recover from errors is called *robustness*.
  - Robust handling of human interface errors is a good thing
- However, tolerance of pre-conditions is reckless.
  - A pre-condition violation is evidence of a bug in the *application*.
  - If the application is under test, it's self-defeating to hide mistakes.
  - If application has been released, hiding bugs can lead to plusible but incorrect output, corrupted files & databases, etc.

Are the data members private?

As discussed earlier, we strongly favor the use of encapsulated `private` data to provide information hiding in ADT implementations.

- E.g., permitting alternate implementations of the *Day* class

Providing Access to Attributes

Two common styles in C++:

```
class Contact {
public:
  Contact (Name nm, Address addr);

  const Name& name() const {return theName;}
  Name&       name()       {return theName;}

  const Address& getAddress() const {return theAddress;}
  Address&       getAddress()       {return theAddress;}
     ⋮
```

- `getAttr`, `setAttr` as used for the Address attribute
- Attribute reference functions

Attribute Reference Functions

```
class Contact {
     ⋮
  const Address& getAddress() const {return theAddress;}
  Address&       getAddress()       {return theAddress;}
```

Attribute reference functions can be used to both access and assign to attributes

```
void foo (Contact& c1, const Contact& c2)
{
  c1.name() = c2.name();
}
```

but may offer less flexibility to the ADT implementor.

- Consequently, not used as often as get/set

Does every constructor initialize every data member?

Simple enough to check, but can prevent some very difficult-to-catch errors.

Example: MailingList

Check this...

```
class MailingList
{
   ⋮
private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  int theSize;
  ML_Node* first;
  ML_Node* last;
};
```

against this:
mailinglist.cpp

Have you appropriately treated the default constructor?

Remember that the default constructor is a constructor that can be called with no arguments.

Your options are:

1. The compiler-generated version is acceptable.
2. Write your own
3. No default constructor is appropriate
4. (very rare) If you don't want to allow other code to construct objects of your ADT type at all, declare a constructor and make it `private`.

Have you appropriately treated the "Big 3"?

- Recall that the Big 3 in C++ class design are the copy constructor, the assignment operator, the destructor.
    - For each of these, if you do not provide them, the compiler generates a version for you.
- The *Rule of the Big 3* states that,

> *if you provide your own version of any one of the big 3, you should provide your own version of all 3.*

Handling the Big 3

So your choices as a class designer come down to:

1. The compiler-generated version is acceptable for all three.
2. You have provided your own version of all three.
3. You don't want to allow copying of this ADT's objects
   - Provide private versions of the copy constructor and assignment operator so the compiler won't provide public ones, but no one can use them.

The Compiler-Generated Versions are wrong when...

Copy constructor Shallow-copy is inappropriate for your ADT

Assignment operator Shallow-copy is inappropriate for your ADT

Destructor Your ADT holds resources that need to be released when no longer needed

The Compiler-Generated Versions are wrong when... (2)

Generally this occurs when

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

The Rule of the Big 3

The *Rule of the Big 3* states that,

> *if you provide your own version of any one of the big 3, you should provide your own version of all 3.*

Why? Because we don't trust the compiler-generated...

- ...copy constructor if our data members include pointers to data we don't share
- ...assignment operator if our data members include pointers to data we don't share
- ...destructor if our data members include pointers to data we don't share

So if we don't trust one, we don't trust any of them.

Does your assignment operator handle self-assignment?

If we assign something to itself:

```
x = x;
```

we normally expect that nothing really happens.

But when we are writing our own assignment operators, that's not always the case. Sometimes assignment of an object to itself is a nasty special case that breaks thing badly.

Does your class provide == and < operators?

- *The compiler never generates these implicitly*, so if we want them, we have to supply them.
- The == and < are often required if you want to put your objects inside other data structures.
  - That's enough reason to provide them whenever practical.
- Also heavily used in test drivers

Does your class provide an output routine?

- Even if not required by the application, useful (essential?) in testing and debugging.
  - Again, that's enough reason to always provide one if at all practical to do so

Is your class const-correct?

In C++, we use the keyword `const` to declare constants. But it also has two other important uses:

1. indicating what formal parameters a function will look at, but promises not to change
2. indicating which member functions don't change the object they are applied to

These last two uses are important for a number of reasons

- This information often helps make it easier for programmers to understand the expected behavior of a function.
- The compiler may be able to use this information to generate more efficient code.
- This information allows the compiler to detect many potential programming mistakes.

Const Correctness

A class is *const-correct* if

1. Any formal function parameter that will not be changed by the function is passed by copy or as a const reference (`const &`).
2. Every member function that does not alter the object it's applied to is declared as a const member.

Example: Contact

Passed by copy, passed by const ref, & const member functions

contactcc.h

Const Correctness Prevents Errors I

- This code will not compile:

```
void foo (Contact& conOut, const Contact& conIn)
{
  conIn = conOut;
}
```

- nor will this:

```
void bar (Contact& conOut, const Contact& conIn)
{
  conIn.setName (conOut.getName());;
}
```

- The error messages aren't always easy to understand ("discards qualifier", "no match for...")
  - Resist the temptation to strip away the "const"s or use type-casting to bypass the errors.

Example: MailingList

Passed by copy, passed by const ref, & const member functions

mailinglistcc.h

Outline I

Outline II

Summary

This is a *checklist*

- Use it whenever you have the responsibility of designing an ADT interface
- These are not absolute rules, but are things that you need to think about
  - Violate them if necessary, but only after careful consideration