Common Coding Problems

Jim Fawcett
CSE687 – Object Oriented Design
Spring 2014

References

- Some of this material is based on the text:
    - C++ Gotchas in Coding and Design, Stephen Dewhurst, Addison-Wesley, 2003

Comments

- For each package provide Manual Page and Maintenance Page
- For each prototype file provide a brief prologue with title and author
- Don’t comment the obvious
- Prefer names with meaning over comments
  - Don’t   void aFunction(string, string) // some inadequate comment
  - Do: void CopyAnalysisResults(string destination, string source)
- Comments aren’t bad, but they have to be maintained.
- Better to write code so that a lot of comments are not necessary

Magic Numbers

- Prefer named, initialized constants over literal numbers:
- Don't: `char buffer[100];  …  int x = 100; …`
- Do:
```
const unsigned int BufSize = 100;
char buffer[BufSize];
```

If you need to change the buffer size should you also change the value of x? Would some maintainer know? Will you know in two weeks?

- Don't: `readflag = 0; writeflag = 1; openflag = 2; …`
- Do: `enum flags { read, write, open, … }`

Global Variables

- Globals are almost never needed.
  - std::ostream::cout is a predefined global
  - sout from threads module is a predefined global
  - These are about the only acceptable uses, except when forced by a framework, e.g., MFC or COM.
- You can make shared data easily available like this:

```
class share
{
public:
   vector<std::string> getNames() { return _names; }
   void addName(std::string) { // validation here … }
private:
   static vector<std::string> _names;
};
```

- This way we can validate or log changes, or change the internal representation, if needed.

Globals

- If you think you must use globals, here is the correct way to do that:
- In header file: extern std::vector<std::string> names;
- In implem. file: vector<std::string> names;
- The problems with globals are:
  - No way to track access – who modified? Who used?
  - No way to validate changes
  - Changing representation breaks every user. Who are the users?
  - Destroys reusability of all users
  - Creates mutual dependency among all users
  - Require either compile-time or load-time initialization
    - May not have the knowledge to do that until later

References

- A reference is an alias for its intializer:
  - ```
    int a = 0;
    int &ra = a;      // alias for a
    int *ptr = &ra;  // refers to a
    ra = 1;            // a now has value 1
    a = 2;             // ra now has value 2
    ```
- C++ (before C++11) does not allow:
  - References to references
  - Pointers to references
  - Arrays of references
  - Null references
  - References to void
  - Resetting references after declaration
- References can’t be const or volatile
    ```

References

- A reference can refer to any lvalue (something with a memory location), e.g.:
  - int &elem = array[i][j]; // refers to the array element currently
    // indexed by i and j
  - std::string& name = p->info[n].name;
- You can assign to references returned by functions.

  double& vec::operator[](int n); ➔ vec v; … v[3] = 3.14159;
- References support multiple return values through side-effects:

  std::string lookup(const std::string& id, std::string& status);

  returns value of lookup, const id, and mutable status.

std::ref

- If a function accepts its argument(s) by value you can force a pass by reference by using std::ref, like this:

```
someFunction(std::ref(anInstance))
```

std::move

- If you choose to create some data structure type in a function to be used elsewhere you MUST return by value. If the type provides a move constructor that will be called, else its copy constructor is called.
- If you pass an instance of a type to a function by value and the caller will not use it subsequently you should use std::move, like this:

```
someFunction(std::move(theInstance))
```

  That will cause a move constructor to be called if defined, else a copy constructor is called. If moved, theInstance is no longer valid.

Const and Null Pointers

- const int* pInt = &j;   // can’t change *pInt, can change pInt
- int* const pInt = &j;   // can’t change pInt, can change *pInt
- const int* const pInt = &j; // can’t change *pInt or pInt

- To define a null pointer:

  pInt = 0; or pInt = nullptr; (C++11)

- Don’t use predefined NULL, Null, null:
  - #define NULL ((char*)0)   ??
  - #define NULL ((void*)0)   ??
  - #define NULL 0   ??

□unless documentation tells you to do so (Win32 API)

Copy Construction and Assignment

- ***EVERY*** class design should explicitly decide to provide copy construction, assignment, and a destructor, let the compiler do so, or disallow them.
  - Provide them if class members do not have the copy and assignment semantics you need. Then use the standard declarations:
    - X::X(const X&)
    - X& X::operator=(const X&);
  - If class members have correct copy and assigment semantics let the compiler implement them and the destructor by not declaring them.
    - That results in member wise copy, assignment, and destruction
  - If class semantics don't require copy and assignment, then disallow them by declaring them private and not implementing them.

Assertions

- Assertions can be useful debugging tools:

  ```
  #include <cassert> … assert(arg);
  ```

  This statement:
  - prints line number and file name and aborts whenever arg is false.
  - The assert macro can be turned off by #define NDEBUG ***before*** the include of <cassert>.
  - If you do not make this definition, Visual Studio will enable asserts in Debug builds and disable them in Release builds.

- ***Never*** make assignments or evaluate functions in an assert. The corresponding side-affects will be removed when you make a Release build. Even if you are positive a function has no side effect, a maintenance programmer may add one later.

Stroustrup’s Assert

- Bjarne Stroustrup suggests (pg 751) the use of the following template instead of asserts:

```
template <typename T, typename A>
inline void Assert(A assertion)
{
  if(!assertion) throw T();
}
```

So, this now will work:

```
Assert<range_error>(0<=n || n<MAX);
```

To check only when debugging:

```
Assert<range_error>(NDEBUG || 0<=n || n<MAX)
```

Conversion Constructors

- Conversion constructors (I often call them promotion constructors, but I'm trying to convert) can be called at times when you don't mean them to be called, perhaps because of a logic error.

  You can prevent that by making them explicit:

```
class Y { … };
class X { public: explicit X(Y& y); … };

X x = Y();      // fails to compile
X x = X(Y());  // succeeds
```

Casts

- C++ now defines four casts:
  - X x = Static_cast<X>(y);
    - calls a ctor to convert y to an X instance if one exists or the conversion is supported by the language.
    - Otherwise statement fails to compile.
  - D* pD = dynamic_cast<D*>(pB);
    - Succeeds if pB is a base pointer to an instance of D and D inherits publically from B. Then it returns the address contained in pB.
    - Fails if pB does not point to a D instance. In this case it returns 0.
    - RTTI must be enabled in Visual Studio or the cast will throw an exception.
  - X *pX = const_cast<X*>(&x);
    - Returns a pointer to non-const even if x is const.
  - X *pX = reinterpret_cast<X*>(&y);
    - Return address of y, but interpret it's type as pointer to X.
      Usually bad thing to do.

Casts

- Prefer the new casts over the old-style cast:

  ```
  X* pX = (X*)&y;
  ```

  The intended semantics of this could be any of the four distinct operations, discussed on the previous slide. Which is it???

  If you believe you have a cast error - quite common - how do you find it? Search for ‘(‘ ???

Temporaries

- Be careful with temporaries. They live only for the lifetime of the statement in which they are embedded.

```
Std::string s1 = “now is the hour “;
std::string s2 = “for all good men to come to the aid “;
std::string s3 = “of their country”;
const char* pChars = (s1 + s2 + s3).c_str();  // *pChars valid
std::cout << pChars;   // now invalid
```

Returning References

- Return a reference or pointer from a function <u>only if the object referred to existed before the function call</u>. Otherwise one of the following must hold:
  - The reference is bound to a temporary created in the function, e.g., disaster.
  - The reference is bound to an object the function created on the heap with new. Now ownership of memory is shared, usually a bad idea.
  - The reference is bound to a global or static to which the function assigned a new value. The client's reference may change the next time the function is called.

End of Presentation