Modern C++ Programming for Macintosh

Darin Adler
Bent Spoon Software

MacHack, June 1998

How I'll tell this story

- I recently wrote a simple program called CD Researcher using PowerPlant and the C++ library.
- This is some of what I learned by doing that project.
- I'll use examples from CD Researcher.

What I won't cover

- I had intended to discuss PowerPlant Appearance Manager classes.
- I had intended to discuss PowerPlant Internet and Networking classes.
- They are great!
- Sadly, not enough time for them.

Do you program for Macintosh using C++?

- If so, you may find these tips useful.
- You can go right back to work and use these right away.
- If you are a C++ veteran (like me, I started on the Finder in C++ in 1988), some of these features will seem new.

A few principles

- No compromise on the interface and features that the user sees.
- “Leave no place for the bugs to hide.” — Bill Atkinson
- Include error handling from the start.
- Write as little code as possible.

No compromises?

- Design of the program comes first.
- Try not to let implementation issues affect the design too much.
- Start implementing so you can iterate the human interface.
- Stop designing at some point so you can finish the program.

How to write less code

- Learn the libraries.
- Use the libraries.
- Find high-quality free software.
- Make thin classes.
- Don’t copy and paste code — make and use helper functions and classes.

Learn the libraries

- Decide early what environment you are programming for.
- Current versions of Metrowerks tools support many C++ features and library functions that were not available a few years back.
- Read books.

C++ books

- *The C++ Programming Language, Third Edition*
  Bjarne Stroustrup, 1997
  ISBN 0-201-88954-4
- Way beyond the first two editions.
- Indispensable.
- Get as late a printing as possible.

C++ books (2)

- *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*
  Scott Meyers, 1998
  ISBN 0-201-92488-9
- Important advice on every page.

C++ books (3)

- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*
  Scott Meyers, 1996
  ISBN 0-201-63371-X
- As useful as the first, but covers newer language and library features.

C++ books (4)

- ***Design Patterns: Elements of Reusable Object-Oriented Software***
  Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995
  ISBN 0-201-63361-2
- **Learn it here or learn it the hard way.**

C++ books (5)

- *Ruminations on C++: A Decade of Programming Insight and Experience*
  Andrew Koenig, Barbara Moo, 1997
  ISBN 0-201-42339-1
- Tons of useful stuff.

C++ books (6)

- *The Design and Evolution of C++*
  Bjarne Stroustrup, 1994
  ISBN 0-201-54330-3
- Fascinating insight into language design.
- If read carefully, explains much about the language as it stands today.

Exceptions

- Include exception code from the start.
- Two kinds of exception code.
  - Exception-safe coding: routines that won't leave anything in a bad state if terminated by exception.
  - Exception handling: catching any expected exceptions and reporting the corresponding errors to the user.

Exception-safe coding

- Use auto_ptr instead of delete.
- For any “open/close” idiom, use a constructor/destructor cover class.
  - PowerPlant provides many of these.
  - Make your own.

No pointers

- Most crashes in C++ programs come from problems with pointers.
- Three kinds of problems:
  - Nil pointers.
  - Stale pointers to deleted objects.
  - Incorrect pointer arithmetic.

Nil pointer problems

- Use exceptions when creating objects with new.
- Use references when possible instead.
- Check for nil after every dynamic_cast.

Stale pointer problems

- Use library collection classes instead of C built-in arrays created with new.
- Use auto_ptr instead of delete.
  - Note that auto_ptr does not work properly with arrays created by new [ ].
- Don’t retain pointers or iterators into collections when modifying them.

Pointer arithmetic problems

- Keep direct manipulation of collections to a few routines.
- Most code should use high level operations instead.
- Keep type casts to a minimum.

Minimizing type casts

- Use the new type cast syntax.
- Study each cast carefully to know why you are doing a type cast.
- Think of each type cast as a place for a possible bug.
- Use implicit type conversion instead.

Old style casts

- Two syntaxes, both dangerous.
- (X)y is traditional C cast syntax.
- X(y) is function call style syntax.

implicit_cast

- Not part of the language.
- A way to express automatic type conversions without having to introduce a local variable of the desired type.
- Use a local variable instead.

Code snippet

```
template <class X, class Y>
X implicit_cast(const Y& x)
{
  return x;
}
```

const_cast

- Safest cast; still avoid when possible.
- Can only change “const” and “volatile” characteristics
- Use mutable keyword instead.
- Needed when providers of a programming interface neglect const.

static_cast

- Used for “upcasting” to a class higher in the class hierarchy.
- Rarely needed, but fairly safe.

dynamic_cast

- Used for “downcasting” in a class hierarchy with virtual functions.
- If used on a pointer, check for nil.
- If used on a reference, note exception.
- Required by idiom in almost all PowerPlant pane programming.

reinterpret_cast

- Most dangerous cast, still needed sometimes.
- Only way to convert pointers between two unrelated types.
- For example, needed to change char* to unsigned char*.
- Also converts pointers to integers.

Casts in CD Researcher

- No uses of implicit_cast.
- 3 uses of const_cast.
  - LDragTask constructor
  - LTCPEndpoint::SendData
  - ICLaunchURL
- No uses of static_cast.

Code snippet

```
std::string
AsString(const LString& in)
{
  ConstStr255Param str255(in);
  return AsString(str255);
}
```

Code snippet

```
void
CInternetConfig::LaunchURL(
  const std::string& location,
  const std::string& defaultScheme)
{
  long start(0);
  long end(location.length());
  ICLaunchURL(mInternetConfig.mInstance,
    AsStr255(defaultScheme),
    const_cast<Ptr>(location.data()),
    location.length(), &start, &end);
}
```

Casts in CD Researcher (2)

- 8 uses of dynamic_cast.
  - 4 pointer casts.
  - 2 reference casts.
  - 2 versions of FindPane.
- 22 uses of FindPane.
- Traversing PowerPlant's hierarchies.

Code snippet

```
template <class T> T
FindPane(LPane* pane, PaneIDT paneID, T& result)
{
  result = dynamic_cast<T>
    (pane->FindPaneByID(paneID));
  ThrowIfNil_(result);
  return result;
}
```

Casts in CD Researcher (3)

- 25 uses of reintepret_cast.
  - Getting at data in Macintosh handles.
  - Converting between Str255 unsigned char and C and C++ strings with char.
  - Converting void* pointers used by PowerPlant Lbroadcaster interface.

Code snippet

```
std::string
CCDResearcher::Version()
{
  // Read the 'vers' 1 resource.
  StCurResFile application(LMGetCurApRefNum());
  StResource resource('vers', 1);
  VersRecHndl handle(reinterpret_cast
    <VersRecHndl>(versionResource.mResourceH));
  return AsString((**handle).shortVersion);
}
```

std::auto_ptr

- Just say no to delete.
- *Effective C++* shows that you can't do delete in a destructor correctly for all exception cases without auto_ptr.
- Do not use auto_ptr on array pointers.
- Implementations differ, but the concept is stable.

Code snippet

```
class CCDDBThread { // most of class omitted
  auto_ptr<CCDDBConnection> mConnection;
};

CCDDBThread::CCDDBThread(CCDCatalog* catalog)
  : LThread(false)
  , mCatalog(catalog)
  , mConnection(CreateCDDBConnection())
{
  catalog->ThreadBirth(this);
  Resume();
}
```

Collections

- Use them each when appropriate.
- Learn their theory, implementation, how to use standard algorithms.
- Avoid obsessing on implementation.
- Example: stack and queue are built from vector and deque; ignore that.

Collections (2)

- std::vector is a dynamic replacement for built-in C arrays.
- std::stack, stacks, std::queue, queues.
- std::deque when you need to delete from or insert at both ends.
- std::list when you need add and delete in the middle, don’t need to index.

Collections (3)

- std::map and std::multimap make a dictionary out of any two types.
- std::set and std::multiset keep items in sorted order.
- Maps slightly more useful then sets.
- Use [ ] syntax for getting at maps.

Code snippet

```
mutable map<ResIDT, IconSuiteRef> mIconCache;

IconSuiteRef CCDCatalogOutlineDisc::Icon() const
{
  IconSuiteRef icon(mIconCache[mCurrentIconID]);
  if (icon == nil) {
    GetIconSuite(&icon, mCurrentIconID,
      kSelectorAllSmallData);
    mIconCache[mCurrentIconID] = icon;
  }
  return icon;
}
```

Collections (4)

- Avoid collections of pointers.
- Must use __MSL_FIX_ITERATORS__ when using certain collections.

The iterator trap

- Iterator interfaces work on all collections, are used by generic algorithms.
- It's easy to forget that there are clearer interfaces to many collections.
- For example, you can index into a vector with [ ] and an integer.

A pearl from Perl

- Perl programmers love hashes, also known as associative arrays.
- Make a Perl-style hash in C++.
  - std::map<std::string, std::string>
  - Use [ ] to get or set elements.
  - Use iterators to check if element exists.
    - Or write a function or template.

std::string

- Easy to use: like C strings, but with automatic storage management.
- May not be suitable for large amounts of text that need to be modified.
  - Consider SGI's rope class instead.
- Used everywhere in CD Researcher.

Code snippet

```
class CCDDBConnection {
  public:
    virtual ~CCDDBConnection();
    virtual std::string SendCommand(
      const std::string&) = 0;
};
```

std::ostringstream

- Replacement for sprintf().
- Easy to use. Safe.
- Best reference is Stroustrup, 3rd ed.
- Used 20 places in CD Researcher.

Code snippet

```
std::string
CCompactDisc::TimeAsString(unsigned long time)
{
  std::ostringstream result;
  unsigned seconds((time + kFramesPerSecond / 2)
    / kFramesPerSecond);
  unsigned minutes(seconds / 60);
  seconds %= 60;
  result << minutes << ':'
    << seconds / 10 << seconds % 10;
  return result.str();
}
```

std::istringstream

- Replacement for sscanf().
- Easy to use. Safe.
- Streams go into an error state when they see bad input data.
- Best reference is Stroustrup, 3rd ed.
- Used 12 places in CD Researcher.

Code snippet

```
unsigned
StringAsTime(const std::string& timeString)
{
  std::istringstream in(timeString);
  unsigned time(0); in >> time;
  char c;
  if (in.get(c) && c == ':') {
    float seconds(0); in >> seconds;
    time *= kFramesPerMinute;
    time += (seconds * kFramesPerSecond + .5);
  }
  return time;
}
```

String conversions

- Use c_str() to make a C string from a std::string.
- Use data() to get a pointer to raw data in a std::string.
- Use str() to make a std::string from a std::ostringstream.

Make std::string work with Pascal string (Str255)

- Need conversion from Str255 to std::string.
- Need conversion from std::string to suitable Pascal-string parameter.
- C++ wrapper with conversion operator is an excellent solution for the string parameter case.

Code snippet

```
class Str255Converter {
  public:
    Str255Converter(const std::string&);
    operator const unsigned char*() const;
  private:
    Str255 mStr255;
};

std::string AsString(ConstStr255Param);
Str255Converter AsStr255(const std::string&);
void CopyToStr255(const std::string&, Str255);
```

Make std::string work with Macintosh Handle

- Need a routine that converts a Handle into a std::string.
- Also useful to have a version that disposes the Handle afterwards.

Code snippet

```
std::string
AsString(Handle handle)
{
  MoveHHi(handle);
  StHandleLocker lock(handle);
  return std::string(*handle,
    GetHandleSize(handle));
}
```

Learn C++ “physics”

- *Effective C++* books cover this well.
- Special members:
  - Constructors
  - Destructors
  - Copy constructors
  - Assignment operators
  - Type conversion operators

Learn C++ “physics” (2)

- Consider implementing or disallowing copying of each class you define.
- Make destructors virtual when using inheritance for polymorphism.
- Use operator overloading when it makes sense, but avoid “clever” uses.

Module building

- You are building a program, not a set of general purpose libraries.
- Class designs should be “industrial strength,” but implementations need not be thorough.
- Missing implementation is better than partial or incorrect implementation.

Module building (2)

- Feel free to have more interface than implementation.
  - You can implement when you need it.
- Learn to use the linker to tell you what code you still need to write.

Thin classes

- Sure sign of design problems in C++ is a “fat” interface.
- Strive for thinnest possible interface.
  - Easy to understand, maintain.
- PowerPlant classes tend a bit to fatness but are thinner than those of its ancestor, MacApp.

Thin classes (2)

- Thin interfaces are more important than thin implementations.
- Avoid temptation to make all operations member functions.
- If it can be done with the public interface, consider separate function.

Code snippet

```
class CCDDrive {
  public:
    class Iterator {
      public:
        Iterator();
        bool More() const;
        CCDDrive Next();
    };
    CCompactDisc ReadTrackList() const;
    bool AudioCD() const;
    bool Empty() const;
    void Eject() const;
};
```

3 interfaces per class

- Caller interface.
  - Public members, including any inherited public members.
  - Copy constructor and default assignment operator are part of the public interface unless explicitly declared.
  - Default constructor is part of the public interface unless constructor is declared.

3 interfaces per class (2)

- Subclass interface.
  - Public and protected members, including inherited public and protected members.
  - Virtual functions and calls to virtual functions.
- If all constructors are private, then there is no subclass interface.

3 interfaces per class (3)

- Calling interface.
  - The most subtle of the three interfaces.
  - Calls made by the class, including calls to inherited member functions and to outside functions and member functions.

Kinds of classes

- To implement polymorphism, make base classes with virtual functions.
  - Must use pointers and references.
  - Use auto_ptr to hold these.
  - Don’t forget to either implement or disallow copying and assignment.

Kinds of classes (2)

- To implement simple interfaces, make small concrete classes that can be used on stack.
  - No virtual functions.
  - Sometimes use pointers and polymorphism within the implementation.
  - Great for parameters, return values.

CLocalCDDatabase

```
class CLocalCDDatabase {
  public:
    class DatabaseFull { };
    CLocalCDDatabase();
    ~CLocalCDDatabase();
    bool Get(CCompactDisc&);
    void Set(const CCompactDisc&)
      throw(ExceptionCode, DatabaseFull);
};
```

Kinds of classes (3)

- Smooth interfaces with tiny classes.
  - Iterator classes, not to be confused with library iterators.
  - Type conversion classes like the one used to convert std::string to Str255.
  - Aggregates for simple concepts like “longitude/latitude pair”.

Code snippet

```
class CMapLocation {
  public:
    CMapLocation();
    static CMapLocation Degrees(
      double latitude, double longitude);
    static CMapLocation Radians(
      double latitude, double longitude);
    double LatitudeInDegrees() const;
    double LongitudeInDegrees() const;
    double LatitudeInRadians() const;
    double LongitudeInRadians() const;
    bool operator ==(const CMapLocation&) const;
};
```

Things that don’t work

- Collection class implementations have excessive overhead if you have lots of small collections; Metrowerks can fix.
- Member function templates do not work yet, which can be awkward.
- Multiple inheritance — virtual base classes with data members.

Things that don't work (2)

- Exception specifications.
- Too much is imported when you use a header like <stdlib.h>, because the current Metrowerks headers do "using namespace std", which imports the entire namespace, instead of importing each item in the header.

Features I wish for

- Warning on use of old-style casts.
- A workable model for checking exception specifications.
- Debugging that works well with template classes and the standard collection classes.

CD Researcher

- Did it for fun.
- Pieces of the source code available on request (send me email).
- Available from Spinfree as Audiofile Internet Companion.

Go do it!

- I learned lots of new techniques while working on CD Researcher.
- The new C++ language and library features make it easier to write intricate but bug-free programs.
- Consider using these tools when working on your next project.