Effective C++

- Based on the Book "Effective C++" (third edition) By Scott Meyers
- Presentation By: Brian Braatz

ITEM#1

"View C++ as a federation of languages"

- C
- Object Orientated C
- C++Template Meta-Programming
- STL
- Rules for Effective C++ vary depending on which language you are using

ITEM#2

"Prefer const, enum and inlines to #defines"

- (might also be called prefer the compiler to the preprocessor)

Example

- Macro:

  #define ASPECT_RATIO 1.6533

- How will most debuggers treat usage of a macro?
- Poorly! Error msg might refer to 1.6533
- How can we solve the debugger problem?
- Solution: Replace with a constant

  const double AspectRatio = 1.6533

  Does this work with "strings"?....

Const With Pointers

- const char * const CompanyName = "Acme";
- We have to write const twice
  - (WHY twice will be covered in Item 3)
- How could we improve above?
- String objects are generally preferable:
- const std::string CompanyName = "Acme";

Working with Const Pointers

- To limit the scope
  - Must make it a member
- To ensure there is only one copy
  - Must make it static

Compile Time Constants

```
class info
{
const double AspectRatio = 1.6533;
};
```

- Declaration or definition?
- This is a declaration - not a definition
  - usually C++ requires a definition
  - except with class specific integral constants
- Is this really legal C++?
- Yes. As long as you don't take an address, you can use and declare without providing a definition

Compile Time Const

- What if we need to take the address of a constant?
- We then must provide a definition
- const double info::AspectRatio;
- This is only allowed for constant integral types
- Are there any other methods of creating a compile time value?

Enum

```
class info
{
enum { AspectRatio = 1.6533 };
};
```

- Enum is more portable to older compilers
- Enum provides similar functionality to const values
- Is enum more limiting than const?
- We cannot take the address of AspectRatio.
  - This might be your intention

Enum Notes

- The enum as a constant is a good technique to be aware of
- Many boost libs uses this for setting results of compile time expressions (BB)

```
struct AlwaysTrue
{
    enum {VALUE = mpl::true_ };
};
```

Things to remember:

- For simple constants- prefer const objects or enums to #defines
- For function like macros prefer inline functions to defines

ITEM #3

- "Use const whenever possible"

- const communicates to both programmers and compilers the usage model of an object

Const is Versatile

- Const can be used:
- Outside of classes
  - constants can be at global or namespace scope (see item2)
- For objects declared static at file, function, or block scope
- inside classes use it for both static and non-static data members

Const with Pointers

- With pointers const can
  - Specify pointer is const
  - Specify data pointed to is const
  - both
  - neither

Const Pointer Example

```
char name[]="Billy";
char * p = name;            // non-const ptr / non-const data
const char * p = name;    // non-const ptr / const data
char * const p = name;    // const ptr / non-const data
const char * const p = name; // const ptr / const data
```

How to think about const:

```
char name[]="Billy";
char * p = name;             // non-const ptr / non-const data
const char * p = name;      // non-const ptr / const data
char * const p = name;      // const ptr / non-const data
const char * const p = name; // const ptr / const data
```

- const on the left of asterisk
  - What is pointed to is const
- const on the right
  - pointer is const

Const Left or Right of Type

- With the type, const may be on the left of or right
  - but always same meaning
- both of these are equivalent

```
void f( const Foo * fp);

void f( Foo const * fp);
```

STL iterators

- Modeled after pointers
- Iterators acts similar to a pointer with const

```
const std::vector<int>::iterator it =v.begin(); // like T* const
                                               //  const ptr / non-const data
*it = 77; // ok changes data pointed to
++it;   // Error iter is const!

std::vector<int>::const_iterator cit = // like a const T *
v.begin();                          // non-const ptr / const data

*cit = 88;        // Error! object pointed to is const!
++cit;            // ok we can incrememnt the iterator
```

const with functions

- const can be applied to :
  - return value
  - individual arguments
  - to the function as a whole
    - (for member functions)

Returning const:

- Having a function return a constant can reduce client errors

```
struct Rational {};
const Rational operator*(const Rational& lrhs, const Rational& rrhs);
```

- Why make the return const?
- to stop this:

```
Rational a,b,c;

(a*b) = c; // invoking operator= on the result of (a*b) !!!!
```

- Would anyone intentionally do this?
- probably not... but typos happen
- Consider:

```
Rational a,b,c;
// ...
if ( (a*b) = c) // oops! forgot to use "=="
```

Const member functions

- What are they for?
- To identify which members maybe invoked on const objects
- 2 Important Reasons to use const member functions
  - Make the interface of a class easier to understand
  - Make it possible to work with const objects

Const Performance

- Const is a critical aspect of efficient code
- Item 20 explains that one of the best ways to aid efficiency is to pass by ref to const

Constness affects overloading

```
class my_string
{
public:
   // ....

   // for const objects
   const char & operator[](std::size_t pos) const
   {  return sText[pos];
   }
   // for non-const objects
   char & operator[](std::size_t pos)
   {
      return sText[pos];
   }
private:
   string sText;
};


/* my_string can be used like this: */

my_string s1("Billy");
cout << s1[0]; // calls non-const operator

const my_string s2("Bob");
cout << s2[0]; // calls const operator
```

Two forms of const- bitwise and logical

- Bitwise
  - if a member is const, and it doesn't modify members
  - i.e. none of the "bits" inside the object
- Logical
  - bitwise is easy for compiler to detect
  - While logical is more of a technique

Bitwise const can be counterintuitive

```
class my_string
{
public:
   // ....

   // BAD- returning char & from const function
   char & operator[](std::size_t pos) const
   {
      return sText[pos];
   }
private:
   char * sText;
};
```

- Passes the bitwise test. but the member can be used to modify the object
- Is this legal C++?...

Bitwise const can be counterintuitive

- This is LEGAL due to bitwise const rules in C++!

```
class my_string
{
public:
   // ....

   // BAD- returning char & from const function
   char & operator[](std::size_t pos) const
   {
      return sText[pos];
   }
private:
   char * sText;
};

const my_string s("ho"); // CONSTANT object
char * n ;
char * p = &s[0];       // note p is NOT const
*p ='Y';

cout << s; // prints "Yo"
```

Logical Constness

- Philosophy
  - A const member function might modify some of the bits in the object on which it's invoked
  - But only in ways clients cannot detect

Logical Constness

- Lets say we wish to track how many times the operator[] was called with an internal variable;

```
class my_string
{
public:
// ....

   const char & operator[](std::size_t pos) const
   {
      num_calls++; // increment member variable
      return sText[pos];
   }
private:
   mutable int num_calls; // mutable member
};
```

- Mutable lets the function be "const", but still modify specific member variables

- Without the mutable keyword- the above will not compile as it fails bitwise constness

Avoiding Duplication

- We now have better methods for expressing constness
- HOWEVER, if we need additional code in our operators, our techniques leave us with code bloat.
- Do we really need to have a const and non-const version of our operators have duplicate code?
- NO!
- Cast away const!
- ?
- Generally casting is a bad thing
- In this case it is quite useful...

Casting Away Const

```
class my_string
{
public:
// ....

   const char & operator[](std::size_t pos) const
    {
      // .. large body of code here ....
      // ... logging \ tracing \ calculate pi etc ..
      return sText[pos];
    }
   char & operator[](std::size_t pos)
    {
      return const_cast<char&> // cast away const
       (
         // call our const operator[]
         static_cast<const my_string&>(*this)[pos]
       );
    }
};
```

- Notice what this does:
  - static_cast "adds const" to this
  - const_cast "removes const" to this
- (without the const in the static_cast<> we have infinite recursion)

Going the other way

- What about having the const version call the non-const version?
  - Not a good idea
  - Not as safe as it is more likely the code will modify the underlying object in ways not intended

Things to remember:

- Declaring something const helps compilers detect usage errors
- const can be applied to
  - Objects at any scope
  - Function parameters
  - Return types
  - Member functions as a whole
- Compilers enforce bitwise constness
- You should program using conceptual constness
- When const and non-const members have identical implementations
- Code duplication can be avoided by having the non-const version call the const version

ITEM#4

- "Make sure the objects are initialized before they're used."

Example

- int x;
- Initialized or not?
- Sometimes yes, sometimes no.
- Depends on what dialect of C++ you are using
- C - initialization not guaranteed to take place
- Non-C parts of C++ - things sometimes change
- char s[100]; - not initialized
- vector<char>; IS initialized
- Rules for when this happens is complicated

What to do?

- Unless you are hyper sensitive about performance in a critical piece of code, always initialize
- Make sure ctors always initialize everything in the object
- Use member initialization lists instead of code in the ctor

Example

```
struct Address
{
  string fname;
  string lname;

  Address( const string & _fname, const string & _lname)
  : fname(_fname), lname(_lname)   // initialize HERE
  {
  // NOT here
  }
};
```

Initialization vs assignment

- Using the initialization list means fname and lname will be initialized with their values;
- If the code was placed in the body of the ctor,
  - fname and lname would be initialized, and THEN have values assigned to them
  - If there are many ctors, this might be unwieldy, but in general it is a good practice

Order of initialization of of non-local static objects

- "The relative order of initialization of non-local static objects defined in different translation units is undefined"
- if a module level static in one cpp references a module level static in another cpp,
  - The target is NOT guaranteed to be initialized
- Huh?...

Example:

```
///////////
// one.cpp
statc int X = 22;

-------------------------

///////////
// two.cpp
static int  Y = X; // what is the value of X? Undefined!
```

Solution:

- Move static access into static functions

```
-------------------------
// one.cpp

static int & get_x()
{
  statc int X = 22;
}
-------------------------
// two.cpp

static int  Y = get_X(); //  OK!
```

- C++ Guarantees that a local static will be called on initial use, problem solved.

Things to remember:

- Manually initialize objects of built in types-
- C++ only "sometimes' initializes them by itself
- In a ctor, prefer use of member initialization lists to assignment in the body
- list data members in the same order as defined
- Avoid initialization order problems across translation units by replacing non-local static

ITEM#5

- Know what functions C++ silently writes and calls

Example

```cpp
// THIS:
class Empty {};

// is the same as:
class Empty
{
public:
   Empty()                            {...}
   Empty(const Empty& x)              {...}
   ~Empty()                           {...}
   Empty & operator=(const Empty &x) {...}
};
```

- C++ will generate these extra functions when you use them.

What is in the generated members?

```
Empty e1;   // default ctor
Empty e2(e1);  // copy ctor
e1 = e2; //    // copy assignment operator
```

- Default ctor & dtor
- Initialization & destruction of non-static member variables
- Base class invocation of destruction and construction
  - Note: generated dtor is non-virtual- unless the base class declares a virtual dtor
- Copy ctor & copy assignment operator
  - Copy each non-static data member of the source over to the target object

Example

```
template <typename T>
class NamedObject
{
public:
NamedObject(const char * name, const T& value);
NamedObject(const string& name, const T& value);

//...
private:
string nameValue;
T ObjectValue;
};
```

- Points of Note:
  - Since a ctor was defined, compilers wont generate a default ctor
  - No copy ctor or assignment ctor
    - compiler will generate if needed
  - if T == int (integral type), compiler will generate a bitwise copy for ObjectValue
  - if T == string, Compiler will generate a call to the copy or assignment operator in string

Compiler Generated Functions-References

```
template <typename T>
class NamedObject
{
public:
NamedObject(const char * name, const T& value);
NamedObject(const string& name, const T& value);

//...
private:
string & nameValue;
const T ObjectValue;
};

string dog1("percy");
string dog2("skip");

NamedObject p(dog1, 1);
NamedObject s(dog2, 37);

p = s; // what happens to data members in p?
```

- Will this compile?
- Problem: C++ doesn't have a way to make a reference refer to a different object.
  - The "generated" assignment code is invalid.
  - Example will not compile
- Problem: The same goes for const T ObjectValue;
  - Can't modify const members

Solution

- Solution:
  - You must define the assignment operators yourself
- Additionally:
  - Compilers reject implicit copy assignment operators in derived classes that inherit from base classes declaring the copy assignment private

Things to Remember:

- Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator and destructor

ITEM#6

- "Explicitly disallow the use of compiler-generated functions you don't want."

Example

```
class UnCopyable
{
protected:
   UnCopyable();  // allow construction & destruction
   ~UnCopyable();    // of derived objects

private:
   UnCopyable(const UnCopyable &);  // prevent copying
   UnCopyable & operator=(const UnCopyable &);
};
```

- Uncopyable has the following qualities
  - Cannot be created directly
  - Cannot be destroyed directly
    - (must be derived from)
  - Cannot be copied (even by derived)
- There still is a hole however.... What is it?
- Friend classes can break these rules.

Solution:

- Declare the functions but do not provide implementation
  - if the rules are broken- the users code wont link
- Disadvantage:
  - Error is put off until link time

Move Error to Compile Time

- How do we get the error moved to compile time?
  - Put the private \ protected members into a base class

```
class my_uncopyable : private UnCopyable
{
// ...
};
```

- This works nicely because compiler will try to generate a copy ctor and copy assignment operator.
- If anyone tries to copy my_uncopyable, it will fail at compile time.
- Note:
- This is the functionality behind boost::noncopyable

Move Error to Compile Time

- Also:
- Compilers will sometimes generate warning messages about private\protected operators.
- These should be disabled with a #pragma. In this case, we specifically intended to do what the compiler is warning us about.

Things to remember:

- To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class like uncopyable is one way to do this.
- Boost provides such a class

ITEM#7

- “Declare destructors virtual in polymorphic base classes.”

Example

```
struct base
{
    base();
    ~base();
};

struct derived : public base
{
    // ...
    char derived_data[1024];
};

base * b = new derived;

// ....

delete b; // memory leak!!
          //generated code does not know about derived_data member
```

- This leads to a partially destroyed object.
- C++ prefers performance over safety, hence there is no check runtime check to make sure we have the "correct" object to delete.

Example w/ virtual dtor

```
struct base
{
   base();
   virtual ~base() {};
};

struct derived : public base
{
   // ...
   char derived_data[1024];
};

base * b = new derived;

// ....

delete b; //NO memory leak - calls virtual dtor in base
```

- While this might seem like a silver bullet to solve the problem it is not.
- This technique should only be used when a class is intended to be a base class. Why?
- Additional pointer in memory over head
- Indirection (vptr) incurred in destruction

STL as bases classes?

- Where this item can really crop up is the fairly common (bad) technique of deriving from std:: classes.

```
class SpecialString : public std::string
{
   // ...
};

// SpecialString **MIGHT** look ok, but consider:

SpecialString * pss = new SpecialString;
std::string *ps;
...
ps = pss;
...
delete ps; // SpecialString's resources will be leaked!
```

- This problem applies to any class lacking a virtual destructor.

Things to Remember:

- Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphicly should not declare virtual destructors.

ITEM#8

- “Prevent exceptions from leaving destructors.”

- C++ does not prohibit destructors from emitting exceptions, but it certainly discourages the practice.

Problem Code:

```
struct Widget
{
public:
// ...
   ~widget(){...}
};
void dosomething()
{
   std::vector<Widget> v;
   ...
}
```

- Suppose the vector has 10 Widgets in it.
- During the deletion of the first one, an exception is thrown
- The other nine Widgets will have to be destroyed, so v should invoke their destructors
- Suppose during those calls a second Widget dtor throws an exception.
- Now there are two simultaneous active exceptions
- Program execution either terminates or is undefined!
- C++ does NOT like destructors that emit exceptions

What to do?

- What if you have a class of database connections?
- The dtor, SHOULD close the db handle if it is open, right?
- If the close call throws, we have problems....
- Two primary ways to handle this:
  - Catch the exception in the dtor and terminate the program
  - Swallow the exception- maybe make a log entry
- Neither of these are especially appealing.

Suggested approach:

```
struct DbConnection
{
   // ...
   void close()
   {
       db.close();
       closed = true;
   }
   ~dbconn()
   {
       if (!closed)
       {
           try
           {
               db.close();
           }
           catch(...)
           {
           // LOG entry!
           }
       }
   }
};
```

Things to Remember:

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (non destructor) function that performs the operation.

ITEM#9

- Never call virtual functions during construction or destruction.

Example

```
struct base
{
    virtual void log() =0;
     base()
     {
         log(); // Call virtual function
     };
};

struct derived : base
{
    derived() : base() {};

    virtual void log()
     {
    // send information to the log file
     }
};

derived d;
```

- When d is constructed, the base class is initialized before the derived class
- During this initialization, the base class attempts to call a virtual function
- However, "derived" hasn't been initialized yet
    - The call exhibits undefined behavior!!!!!!!
    - If log() were not pure virtual, it would call base::log()
- Destruction works in the opposite manner
- Derived classes are deallocated before base classes.

Solution:

- Init
  - have an init() member which is virtual
  - track whether the object has been initialized
  - error on methods where object initialization is a precondition

Things to Remember:

- Don't' call virtual functions during construction or destruction, because such calls will never go to a more derived class than of the currently executing constructor or destructor.

ITEM#10

- “Have assignment operators return a reference to *this.”

Example

```
struct myclass
{
  // ...
  myclass & operator=(const Widget &)
  {
     //...
     return *this; // return ref to this
  }
};

// Allows chaining
myclass a,b,c;
a = b = c;

// This is standard practice in STL
//and for built in types.
```

Things to Remember:

- Have assignment operators return a reference to *this.
- Allows chaining

End Part 1

- Thank you!

ITEM#11

- Handle assignment to self in operator=.

Example

```
struct myclass
{
   // ...

   string * pstr;
   myclass & operator=(const myclass & mc)
   {
      delete pstr;
      pstr = new string(*(mc.pstr) );
      return *this;
   }
};
```

- Looks good, except:

```
myclass mclass;
myclass & mclass2 = mclass;
// ...
mclass = mclass2; // Self assginment!
```

- pstr in the mclass object is now holding a pointer to a deleted object.
- This problem can be averted by checking for self assignment...

Self Assignment

```
struct myclass
{
// ...

   string * pstr;
   myclass & operator=(const myclass & mc)
    {
      if (this == &ms) return *this; // check identity

      delete pstr;
      pstr = new string(*(mc.pstr) );
      return *this;
    }
};
```

- Done?
- No! ..... Why?
- The code is not exception safe.
- If the constructor in string throws, pstr is left pointing to a deleted object
- Can we fix this?????

Better

```
struct myclass
{
   // ...
   string * pstr;
   myclass & operator=(const myclass & mc)
   {
      if (this == &ms) return *this;

      // Save original
      string * orig_pstr = pstr;

      // Make new
      pstr = new string(*(mc.pstr) );

      // Delete original
      delete orig_pstr;

      return *this;
   }
};
```

- We now have a check for self assignment
  - and we are exception safe!
- How are we exception safe?
- If "new string" throws, pstr is still pointing to a valid object.
- Item #29 explores this topic in further detail

Things to Remember

- Make sure operator= is well behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy and swap.
- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

ITEM#12

- Copy all parts of an object.

More

- If you write a copy ctor or an operator=, remember to take full responsibility for copying all parts of the object.
- Remember to copy your data members and refer to your base class(es) appropriate copying members.
- Both for copy and assignment.
- It is a good practice to have a private copy function and have both copy ctor and operator=() call that one, in all but the most trivial cases.

Things to Remember:

- Copying functions should be sure to copy all of an objects class members and all of it's base class parts.
- Don't try to implement one of the copying functions in terms of another. Instead put common functionality in a third function that both call.

ITEM#13

- Use objects to manage resources.

Managing Resources

- Instead of:

```
string * p = new string("data");
// ...
delete p;
```

- Use an object to manage the resources.
- options:
  - std::auto_ptr
  - tr1::shared_ptr or any of the boost smart pointers
- ANY RAII object is better than doing it by hand for most situations

Things to Remember:

- To prevent resource leaks, use RAII objects that acquire resources in their ctors and release them in their dtors.
- Two commonly useful RAII classes are tr1::shared_ptr and auto_ptr, tr1::shared_ptr is usually the better choice because it's behavior when copied is intuitive. Copying an auto_ptr sets it to null

ITEM#14

- Think carefully about copying behavior in resource managing classes.

Resource Managing Classes - Copying

- Designers of RAII classes have many tough decisions on copying:
- Do I Prohibit it?
- Do I Nullify the original? (transfer ownership)
- Do I reference count and share the handle?
- Do I copy the underlying resource?
- Decisions made by the designer(s) of a RAII style class should be understood before using any such class.

Auto_ptr Copying

- std::auto_ptr
- Is it reference counted?
- NO!
  - When copied, original is set to null
- What use is it?
- Only useful in small scopes to make sure you "don't forget" to free it
- That being said, std::auto_ptr is faster than boost \ tr1::shared_ptr

Boost \ Tr1 Shared_ptr

- Holds onto ONE object pointer
- Reference counts on copy
- When last copy is destroyed, frees the dynamic object

Things to Remember:

- Copying an RAII object entails copying the resources it manages, so the copying behavior of the resource determines the copying behavior of the RAII object.
- Common RAII classes copying behaviors are disallowing copying and performing reference counting, but other behaviors are possible.

ITEM#15

- "Provide access to raw resources in resource managing classes."

“Provide access to raw resources in resource managing classes.”

- RAII wrappers are great at hiding resource management mechanisms
- But, should we expose underlying managed object?
- Yes, occasionally one needs access to the underlying object to call a API or C function, or for just plain debugging.
- This can either be done explicitly or implicitly.

Examples:

- operator char *(); // allows object to be passed as a char *
- const char * c_str(); // returns the char* inside this object

Things to Remember:

- APIs often require access to raw resources, so each RAII class should offer a way to get at the resource it manages.
- Access may be explicit conversion or implicit conversion. In general, explicit conversion is safer, but implicit is more convenient for the users.

ITEM#16

- Use the same form in corresponding uses of new and delete.

"Use the same form in corresponding uses of new and delete."

```
string * s = new string[100];
// ...
delete s; // OOPS! - only deletes one object
```

- Better:

```
string * s = new string[100];
// ...
delete [] s; // now deletes the array
```

Things to Remember:

- If you use [] in a new expression you must use [] in the corresponding delete expression. If you don't use [] in a new expression, you mustn't use [] in the corresponding delete expression.

ITEM#17

- Store newed objects in smart pointers in stand alone statements.

“Store newed objects in smart pointers in stand alone statements.”

- What is wrong with this code?

```
int get_priority();
void process ( boost::shared_ptr<string> sp, int iPriority);

void foo()
{
process( boost::shared_ptr<string>( new string("data") ), get_priority());
}
```

- Order of argument expression validation between arguments in C++ is undefined.

Arg Expression Evaluation Order

```
int get_priority();
void process ( boost::shared_ptr<string> sp, int iPriority);

void foo()
{
process( boost::shared_ptr<string>( new string("data") ), get_priority());
}
```

- Order of code MIGHT be:
  - new string("data")
  - shared_ptr constructor
  - call get_priority()
- OR it might be
  - new string("data")
  - call get_priority()
  - shared_ptr constructor
- IF get_priority() throws an exception, we have leaked memory!

What is the Solution?

```
int get_priority();
void process ( boost::shared_ptr<string> sp, int iPriority);

void foo()
{
boost::shared_ptr<string> sp( new string("data") );
process( sp, get_priority());
}
```

Things to Remember:

- Store newed objects in smart pointers in standalone statements. Failure to do this can lead to subtle resource leaks when exceptions are thrown.

ITEM#18

- Make interfaces easy to use correctly and hard to use incorrectly.

Consider a Date class:

```
struct Date
{
Date(int month, int day, int year);
};
```

- Do you see any problems with design?
- It is easy to use incorrectly.
- Think for a moment on how YOU would fix it

Better Date Class

```
struct Month
{
   explicit Month(int m)
   : val(m) {}

private:
   int val;
};


struct Date
{
Date(const Month & m, const Day & d, const Year & y);
};
// ...  do same for day & year
```

- Smart use of the type system can make usage less error prone
- Could it be even better?

Even Better Date Class?

- Hmm..
- Only 12 possible values for month
  - Make month an enum?
  - This would work, but enums are not very type safe
- enums can be accepted like ints, so initial problem persists

Yet Another Date Class

```
struct Month
{
explicit Month(int m)
: val(m) {}

static Month Jan() { return Month(1); }
static Month Feb() { return Month(2); }

static Month Dec() { return Month(12); }

private:
int val;
};

Date d(Month::Mar(), Day(30), Year(1995) );
```

- Date is now strongly typed
- Interface is
  - Safer
  - Consistent
- What makes this version of Date less likely to be used incorrectly?

Interface Consistency

- The more consistency achieved in an interface the better.
- Things are easier to use if you have consistent concepts in interfaces.
- (brian) Heavy HARP point :)
- This concept applies to UIs, processes, code etc..

Consistency

- STL is not perfect
  - but it is largely consistent
  - Every STL container has a size() member

InConsistency

- Java
  - Arrays
    - Length property
  - Lists
    - Size() method
  - Strings
    - Length() method
- C#
  - Arrays
    - Length property
  - ArrayLists
    - Count property
  - Strings
    - Length property
- Inconsistency imposes mental friction
- The more an interface imposes something the user has to remember the more it is prone to misuse

(Brian Additions)

- Place yourself in the MIND of the user
- Get a feel for how "it reads".
- Think of the general rules you would use in English
- "read" the usage of the library
- Have meaningful English in mind when thinking about it

(Brian Additions)

- This is bad

```
if ( !NotDisabled() ) ....

return ( !NotDisabled() ? !bState : !(bState | bSecond) );
```

- Also Consider:

```
Investment * createInvestment();
```

- What is the problem?
- User has to remember to delete it
- Clients COULD use a smart pointer
- So what's the big deal?

The Big Deal

- Question: "We shouldnt have to make better interfaces. Shouldn't people just use it correctly in the first place?"
- Answer: We "shouldnt" but we have to....
- We all make mistakes as users of code. (and systems)
- As code (or a system) grows in complexity, the amount of things we have to "remember to do correctly" goes up exponentially

The Big Deal

- The more a mechanism is easy to use correctly and difficult to misuse, the more the user of the mechanism can focus on their specific problem.
- The better job we can do of disallowing common mistakes in our interfaces, the more the users of the interfaces can concentrate on their specific problem.
- It it very important to always strive for a strong interface, which prevents misuse.
- Failing to do this, eventually our own sloppiness will catch up with us.

Consider:

```
Investment * createInvestment();
```

- This can be improved by giving the user a smart pointer back

```
boost::shared_ptr<Investment> createInvestment();
```

Consider

```
boost::shared_ptr<Investment> createInvestment();
```

- Advantages:
- No leaked memory
- shared_ptr also can have a custom deleteor
- Internal knowledge about deleting a Investment REMAIN inside the createInvestment() function & class
- Cross DLL problem solved
  - Deleting memory from a different HEAP causes leaks
  - shared_ptr<> handles this automatically

Things to Remember:

- Good interfaces are easy to use correctly and hard to use incorrectly. You should strive for these characteristics in all your interfaces
- Ways to facilitate correct use include consistency in interfaces and behavioral compatibility with built-in types.
- Ways to prevent errors include creating new types, restricting operations on types, constraining object values, and eliminating client resource management responsibilities.
- boost\tr1:: shared_ptr<> supports custom deletors. This prevents the cross-dll problem, it can also be used to unlock mutexs or other types of RAII style problems.

ITEM#19

- Treat class design as type design

Questions to ask during design:

- How should objects of your new type be created and destroyed?
  - Influences ctor and dtor design
- How should object initialization differ from object assignment?
  - Determines behavior of assignment operators
- What does it mean for objects of your new type to be passed by value?
  - Influences the copy ctor

Questions to ask during design:

- Which restrictions for legal values for your new type?
  - Effects your handling of invalid values
  - Class design
  - Error handling mechanism
- Does your new type fit into an inheritance graph?
  - If you inherit- effects what you can do
  - If you intend inheritance for use- affects which functions you provide

Questions to ask during design:

- What kind of type conversions are allowed for your new type?
- Do you allow implicit or explicit conversions?
  - Both to and from your object
- What operators and functions make sense for your new type?

Questions to ask during design:

- What standard functions should be disallowed?
  - I.e. copy, assignment etc
- Who should have access to the members of your new type?
  - public private or protected?
- What is the "undeclared interface" of your new type?

Questions to ask during design:

- What guarantees do you provide in
  - performance?
  - exception safety?
  - resource usage?
- These guarantees will impose constraints in implementation.

Questions to ask during design:

- How general is your new type?
  - Consider using templates instead of additional types.
- Is a new type really what you need?
  - Consider adding functionality to an existing class.

Things to Remember:

- Class design is also type design. Before defining a new type, be sure to consider all the issues discussed in this item.

ITEM#20

- Prefer pass-by-reference-to-const to pass-by-value.

Prefer pass-by-reference-to-const to pass-by-value.

```
struct Person
{
   string name;
   string address;
};

struct Student : Person
{
   string schoolname;
   string schooladdress;
};

bool validateStudent( Student s);

///////////////// usage
Student plato;
bool isok = Validate(s);
```

- What happens when validateStudent() is called?
  - six constructors
  - four copies of strings
  - six destructors
- Can we do better???

Better:

```
struct Person
{
   string name;
   string address;
};

struct Student : Person
{
   string schoolname;
   string schooladdress;
};

bool validateStudent( const Student  & s);
```

- Effects:
- a pointer copy
- much more efficient!
- Avoids the slicing problem!
    - (Slicing Problem???)

Slicing Problem:

```
struct Person
{
    string name;
    string address;
    virtual void savetodisk();
};

struct Student : Person
{
    string schoolname;
    string schooldaddress;
    virtual void savetodisk();
};

void SaveObject( Person p)
{
    p.savetodisk();
}

// Usage:

Student s;
/// .. fill in variables in s

SaveObject(s);  // save object to disk
```

- Student object is copy-constructed into a temp Person object
- The derived class is effectively "sliced" off
- The virtual now calls Person::savetodisk()
- Not Student::savetodisk()
- (Slicing problems can commonly crop up in exception handlers)

Slicing Solution

```
struct Person
{
   string name;
   string address;
   virtual void savetodisk() const;
};

struct Student : Person
{
   string schoolname;
   string schooldaddress;
   virtual void savetodisk() const;
};

void SaveObject( const Person & p)
{
   p.savetodisk();
}
```

```
// Usage:

Student s;

/// .. fill in variables in s

SaveObject(s);   // save object to disk
```

- Person is now passed as const &
- Further derivations of Student now work as expected!

Things to Remember:

- Prefer pass-by-reference-to-const over pass-by-value.
    - Typically more efficient and it avoids the slicing problem.
- For built-in types, STL iterators and function objects (functors), Pass-by-value is usually appropriate.
    - (Brian) These objects usually don't have data members- or their members are small.

End Part 2

- Thank You

ITEM#21

- Don't try to return a reference when you must return an object.

Example

```
Rational & operator*( const Rational &lhs, const Rational &rhs)
{
    return Rational(lhs.value * rhs.value):
}
```

- Is this code ok?
- Problem: Returns a reference to a temporary.
- A better approach would be to return an object:

```
const Rational  operator*( const Rational &lhs, const Rational &rhs)
{
    return Rational(lhs.value * rhs.value):
}
```

- Can we do better?.......

Improvements?

```
const Rational  operator*( const Rational &lhs, const Rational &rhs)
{
    return Rational(lhs.value * rhs.value):
}
```

- We **could** use a static to reduce the copy
- This **would** increase performance
- Does a using a static have any negatives?
- Negative: Thread safety!
- Conclusion: It's not worth it
- (Note the use of a const return type. Remember item 3!)

Things to Remember:

- Never return a pointer or a reference to a local or stack object, a reference to a heap-allocated object, or a pointer or reference to a local static object, if there is a chance that more than one such object will be needed.
- (Item 4 provides an example of a design where returning a reference to a local static is reasonable- at least for single threaded only code.)

ITEM#22

- Declare member variables private.

(Brian)

- In Effective C++ Scott Meyers makes some strong arguments for always using get and set methods. Even for derived classes.
- His key reasons for doing this are:
  - One can change the access or model of the storage variable later
  - Easier to debug
  - Easier to track down misuse & invariant values

(Brian)

- The get \ set idea is an interesting notion
  - Though I would only apply it when and where I was LOOKING for the effects of this Effective C++ item
- Also:
  - One thing that is not mentioned is which dialect of C++ one is using.
  - If one is in "C", then it is typical to make Plain Old Data structures (POD) where the members are public

Things To remember:

- Declare data members as private. It gives clients syntactically uniform access to data, affords fine-grained access control, allows invariants to be enforced, and offers class authors implementation flexiablity.

ITEM#23

- Prefer non-member non-friend functions to member functions.

"Prefer non-member non-friend functions to member functions."

- This item is about making functions instead of member functions.

```
struct myclass
{
    const string data_stream();
};

void save(myclass & mc)
{
    ofile("saved") f;
    f << mc.data_stream();
}
```

- Using this idiom has some interesting effects....

Non-Member Non-Friend Function Effects

```
struct myclass
{
    const string data_stream();
};

void save(myclass & mc)
{
    ofile("saved") f;
    f << mc.data_stream();
}
```

- Smaller classes
- Classes are more "to the point"
- Saves compile & re-compile time
- Allows #including different packages of methods seperately
- Similiar to STL
  - Algorithms like for_each are seperate
- functions operate on objects
- With or without templates, functions can be more generic

Counter-point (Brian)

- I like this item, however I need to point out that "even STL" doesn't always follow it.
- There are times when it seems natural\ intutive to provide methods:

```
std::vector<int> v;
....
cout << v.size();
```

Is more clear than:

```
std::vector<int> v;
....
cout << size(v);
```

- Or is it?.....
- (something to further think about) :)

Namespaces

- Scott Meyers also suggests placing such functions in namespaces to reduce clutter.

Things to Remember:

- Prefer non-member non-friend functions to member functions. Doing so increases encapsulation, packaging flexiablity, and functional extensibility.

ITEM#24

- Declare non-member functions when type conversions should apply to all parameters.
- This item pertains to the situation where one is making a class which can interoperate with built in types.

Implicit type conversion

```
struct Rational
{
    // purposely NOT explict
    Rational(int numerator =0, int denom =1);
    int numerator() const;
    int denom() const;
    // ...

    const Rational operator*(const Rational& rhs) const;
};
```

Is "Rational" interoperable?...

```
Rational oneEighth(1,8);
Rational oneHalf(1,2);

Rational result = oneHalf *
oneEighth; // OK
result = result * oneEighth; // OK
```

YES! Except for...

```
result = oneHalf * 2; // OK
result = 2 * oneHalf; // Error!
```

Why doesnt this work?

Implicit type conversion

think of it as the actual function calls:

```
result = oneHalf.operator*(2); // OK
result = 2.operator*(oneHalf); // Error!

Implicit type conversion

result = oneHalf * 2; // OK
```

- This works because the ctor is not "explicit"
- Allows the compiler to take the "2" and promote & convert it to a Rational type
- Parameters are only eligable for implict type conversion ONLY if they are listed in the parameter list.

Think of it this way:

```
struct Rational
{
    Rational(int numerator =0, int denom =1); // purposely NOT explict
    int numerator() const;
    int denom() const;
    ...

    const Rational operator*(const Rational& rhs) const;
};
```

- That operator member function "says":
- "This is the member for applying the "*" operator to me from another type"
- The constructor "says":
- "I can be implictly converted from an int"

"explicit"

- If the constructor was "explicit", the parameter type would have to exactly match.
- This allows an integral constant, like "12", to be passed into the constructor expecting an "int"
- This is because a integral constant "12" is convertable to "int"
- Without "explicit", one would have to only pass "int" types

How to support mixed mode operators properly:

- The operator must be defined as a non-member function:

```
struct Rational
{
...

};

const Rational operator*(const Rational& lhs, const Rational& rhs)
{
return Rational(lhs.numerator() * rhs.numberator(),
    lhs.denom() * rhs.denom() );
}
```

- Now conversions work in mixed mode:

```
Rational oneFourth(1,4);
Rational result;

result = oneFourth * 2; // OK
result = 2 * oneFourth; // NOW it works!
```

Things to remember:

- If you need type converstions on all parameters to a function (including the one pointed to by the this pointer), the function must be a non-member.

Item#25

- Consider support for a non-throwing swap.

Swap

- swap() was originally introduced as a part of STL.
- Is used by STL for swapping values in containers
- i.e. std::sort<>
- Has become a key piece of exception safe programming.

Typical stl swap:

```
namespace std
{
    template <typename T>
    void swap( T& a, T& b)
    {
        T temp(a);
        a = b;
        b = temp;
    }

};
```

- As long as the types support copying, default swap works.

What about with:

```
struct impl;

class Pimpl
{
    impl * pImplementation;
    public:
    void operator=(const Pimpl & rpimpl)
    {
        // DEEP copy of pimpl
    }
};
```

- The STL Swap then becomes a very inefficient mechanism.
- It would be much more efficient to swap the pImplementation pointers.

Swap Your Own Types

```
struct impl;

class Pimpl
{
    impl * pImplementation;
    public:
    void operator=(const Pimpl & rpimpl)
    {
        // DEEP copy of pimpl
    }
    void swap( Pimpl& rPimpl) // member- due to need
    {                         // to access private data
        impl * pImplTmp( pImplementation);
        pImplementation = rPimpl.pImplementation;
        rPimpl.pImplementation = pImplTmp;
    }
};

// swap function for STL to find
void swap( Pimpl & lPimpl, Pimpl & rPimpl)
{
    lPimpl.swap(rPimpl);
}
```

- IF your swap implementation requires private access to member variables make it a member function.
- If not, make it a non-member function.
- In either case, provide a swap function in your namespace.
- In the case of Pimpl, we must do both

How does STL find my swap?

- Koenig \ Argument Dependant Lookup!
  - (ADL)
- When compilers see the call to swap, they search for the proper one.
- C++'s name lookup rules ensure that whatever namespace is used for the type Pimpl, will be the first place it looks to find the associated swap function.

Things to Remember:

- Provide a swap member function when std::swap would be inefficient.
- Make sure your swap is exception safe.
- If you offer a member swap, also offer a non-member swap that calls the member.
- Never call std::swap on a type, employ a using namespace std, then call swap in it's bare form.
- (allows ADL to kick in)
- It is fine to totally specialize std templates for user-defined types, but never try to add something completely new to std.

Item#26

- Postpone variable definitions as long as possible.

Brian's Summary

```
void func()
{
    int x;
    if (x == 10)
    {
        // do something with x
        return;
    }
    int y(x); // x is unknown - but NOT 10

// do something with y
}
```

- In a nutshell, limit the scope of the variables to where it is needed
- Heap related objects should be close to the usage of them.
- Their scope should be contained to where you need them
- Note: y is not heap allocated unless it is ACTUALY needed

Also:

- Scott suggests that variables inside loops are better for readability.
- Though not as efficient
- He suggests you consider the readability argument strongly against the performance argument

Things To Remember:

- Postpone variable definitions as long as possible. It increases program clarity and improves efficiency.

Item #27

- Minimize Casting

Casting in STL:

- const_cast<T>(expression)
  - Used to cast away constness. Only C++ style cast that can do this.
- dynamic_cast<T>(expression)
  - Uses RTTI to safely downcast a type
- reinterpret_cast<T>(expression)
  - low level casts that yeild implementation-dependant (unportable) results
  - i.e. casting a pointer to an int
  - rarely used outside of low level code

Casting in STL

- static_cast<T>(expression)
  - explicit conversions
  - non-const to const object (Item #3)
  - int to double etc..
  - Also used to to perform reverse
  - void * to typed pointer
  - pointer to base
  - pointer to derived
  - cannot cast from const to non-const objects
- Old style casts continue to be legal.
- New forms are preferred.
- New forms have better compile time error checking support

Things to Remember:

- Avoid casts whenever possible, especially dynamic_casts in performance-sensitive code. If a design requires casting, try to develop a cast free alternative.
- When casting is neccessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their code.
- Prefer C++ style casts to the old C style casts. Easier to see and are more specific about what they do.

Item #28

- Avoid returning "handles" to object internals.

Avoid returning "handles" to object internals.

- While it maybe faster to return a pointer to internal private data at times. Prefer not to do this.
- Sometimes you have to
- a smart pointer class usually has to return the raw pointer with an explict member
- a window class might need to return a handle for an API call
- Only do this if you "have to".

Things to Remember:

- Avoid returning handles (references, pointers, or iterators) to object internals. It increases encapsulation, helps const member functions act const and minimizes the creation of dangling handles.

Item#29

- Strive for exception safe code.

Consider:

```
struct PrettyMenu
{
    void changeBkgrnd(istream & imgSrc); // change image background
    // ...

private:
    Mutex mutex;        // mutex
    Image * bgImage;    // current background
    int ImageChanges;   // number of times changed
};

void PrettyMenu::changeBkgrnd(istream & imgSrc)
{
    lock(&mutex); // acqure mutex

    delete bgImage;  // get rid of old background
    ++ ImageChanges; // update count
    bgImage = new Image(imgSrc); // install new background

    unlock(&mutex); // release mutex
}
```

- How exception safe is this code?
- From an exception safety perspective this is as bad as it gets.
- There are 2 requirments for exception safety and this code satisfies neither.

Exception Safety Requirements

- When an Exception is thrown, exception safe functions:
  - Leak no resources.
  - Do not allow data structures to become corrupted.
- Addressing the resource leak is easy.
- Item #13- Use objects to manage resources.
- Item #14- (in the Book) introduces the lock class

Improved code:

```
void PrettyMenu::changeBkgrnd(istream & imgSrc)
{
    Lock    ml(&mutex); // acqure mutex in an object

    delete bgImage;  // get rid of old background

    ++ ImageChanges; // update count
    bgImage = new Image(imgSrc); // install new background

}
```

- What do you think about above?
- Resource leaking is now gone, structure corruption is still there.

"Abrahams guarantees"

- The Abrahams Guarantees are a set of contractual guidelines that class library implementors and clients use when reasoning about exception safety in C++ programs.
- The **BASIC** guarantee: that the invariants of the component are preserved, and no resources are leaked.
- The **STRONG** guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The **NO-THROW** guarantee: that the operation will not throw an exception.

“Abrahams Gaurantees”

- The guarantees are named for David Abrahams, the member of the C++ Standard committee who formalized the guidelines

Basic Gaurantee

- No resources leaked
- All objects are internally consistent
- However the exact state of the program may not be predictable

- In the example code, if a exception were thrown, which background image do we have?
- (unpredicable)

```
void PrettyMenu::changeBkgrnd(istream & imgSrc)
{
    Lock    ml(&mutex); // acqure mutex in an object

    delete bgImage;  // get rid of old background

    ++ ImageChanges; // update count
    bgImage = new Image(imgSrc); // install new background

}
```

Strong Guarantee

- Promises that if an exception is thrown, the state of the program is unchanged.
- calls to such functions are considered "atomic"
- They either completely succeed or completely fail

No-throw Guarantee

- These functions promise to never throw
- All operations on built in types (int, pointers, char) are no-throw
- Exception safe code must offer one of the three guarantees above.
- if it doesn't it isnt exception safe
- The choice is to determine which gaurantee to offer for the functions you write.

Better code:

```
struct PrettyMenu
{
    // ...
    boost::shared_ptr<Image> bgImage;
};

void PrettyMenu::changeBkgrnd(istream & imgSrc)
{
    Lock ml(&mutex);

    bgImage.reset( new Image(imgSrc)); // replace internal ptr
    // with result of new
    ++ imageChanges;
}
```

If image ctor has strong gaurantee & reset uses "swap" this code is then "Strong"

- Does this satisfy the Strong Guarantee?
- What problems do we still have?
- Image constructor
- if it throws, it is possible that the read marker for imgSrc has been moved!
- (lets set that aside and assume the istream copy ctor CAN offer a strong gaurantee)

Point:

- Consider the functions you call and what their guarantee(s) are.
- A function can usually offer a guarantee no stronger than the weakest guarantee of the function(s) it calls.
- There is a general design strategy that typically leads to a strong guarantee.

"copy and swap" Design strategy

- make a copy of the object you wish to modify
- make all needed changes to the object
- if any operations throw, original is unchanged
- Finally, swap the modified object with the original in a non-throwing opertation
- usually implemented as a PIMPL

Example

```
struct Pimpl
{
    boost::shared_ptr<Image> bgImage;
    int imageChanges;
};

class PrettyMenu
{
    // ...
    Mutex mutex;
    boost::shared_ptr<Pimpl> pImpl;
};

void PrettyMenu::changeBackground(istream & imgSrc)
{
    using std::swap; // see item #25
    Lock ml(&mutex);
    boost::shared_ptr<Pimpl> pNew( new Pimpl(*pImpl) );

    // modify the copy
    pNew->bgImage.reset(new Image(imgSrx) );
    ++ pNew->imageChanges;

    // swap the new data in place
    swap(pImpl, pNew);
}
```

- Positives vs Negatives of example?
- Offers strong guarantee
- Is more difficult to code
- Is less efficient
- Such is the tradeoff between basic vs strong guarantee.

Things to Remember:

- Exception Safe functions leak no resources and allow no data structures to become corrupted, even when exceptions are thrown. Such functions offer the basic, strong, or nothrow guarantees.
- The strong guarantee can often be implemented via copy-and-swap, but the strong guarantee is not practical for all functions.
- A function can usually offer a guarantee no stronger than the weakest guarantee of the function(s) it calls.

Item#30

- Understand the ins and outs of inlining.

Points:

- "inline" is a request to a compiler. It may or may not inline it in reality.
- Inlining creates bigger executables.
- Inlining saves jmp instructions, allowing tight code to run faster.
- Library headers which, from verstion to version, have functions which go from inline to non-inline (and vice versa) can create problems for users.
- Depending on the compiler, templates may or may not be inlined.
- Template instantiation and inlining are *NOT* the same thing

Things to Remember:

- Limit most in-lining to small, frequently called functions. This facilitates debugging and binary upgradeability, minimizes potential code bloat and maximizes the changes of greater program speed.
- Don't declare function templates inline just because they appear in header files.

End Part 3

- Thank You

Item#31

- Minimize compilation dependencies between files.

In a nutshell:

- If you have a class in a header which relies upon lower level types, put those in a separate header.
- Put interface classes & declarations in a separate file form the implementation.

Example

Instead of This

```
class Date;
class Person
{
    Person( Date & birth);
};

class Date
{
  //...
}
```

Do This

```
#include <date.h> // header file DECLARING not defining date
                  // Impl would then be in ANOTHER header
class Person
{
    Person( Date & birth);
};
```

- Date is now not in the same header as person
- date.h only has the declaration , not the definition.
- A Pimpl class (or Handle class) can also reduce compile dependencies
- An interface class can serve the same purpose
- Both Pimpl and Interface classes incur some runtime overhead due to virtual function dereferencing

Things to Remember:

- The general idea behind minimizing compilation dependencies is to depend on declarations instead of definitions. Two approaches based on this idea are Handles classes and Interface classes.
- Library header files should exist in full and declaration only forms. This applies whether or not templates are involved.

Item#32

- “Make sure public inheritance models "is-a".”

Make sure public inheritance models "is-a"

- If you only remember one thing from this book, remember the most important rule in object orientated programming in C++
- "Public inheritance means "is-a" "
- Commit this to memory.

Point:

- Everything that applies to base classes must also apply to derived classes, because every derived class object IS A base class object.
- I.e. if you have an class called "animal", and you place a "fly()" method in it , you are violating this principle.

Things to Remember:

- Public inheritance means "is-a". Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object.

Item#33

- Avoid hiding inherited names.

Consider:

```
struct Base
{
    virtual void mf1() = 0;
    virtual void mf1(int);

    virtual void mf2();

    void mf3();
    void mf3(double d);
};

struct Derived : Base
{
    virtual void mf1();
    void mf3();
    void mf4();
};
```

```
Derived d;
int x;

d.mf1(); // OK calls Derived::mf1()
d.mf1(x); // error! Derived::mf1 hides Base::mf1

d.mf2(); // OK calls Base::mf2()
d.mf3(x); // error! Derived::mf3 hides Base::mf3
```

Why does C++ work this way?

- Prevents you from accidentally inheriting overloads from distant base classes when you create a new derived class
- Unfortunately, one typically WANTS to inherit the overloads
- If you are using public inheritance and do not inherit the overloads, you're violating the is-a relationship between base and derived.
- "using declarations" can be used to speak to this problem:

"Using" Declarations

```
struct Base
{
    virtual void mf1() = 0;
    virtual void mf1(int);

    virtual void mf2();

    void mf3();
    void mf3(double d);
};

struct Derived : Base
{
    using Base::mf1(); // make all mf1 & mf3
                       // things in base
    using Base::mf3(); // visible (& public)
                       // in deriver's scope

    virtual void mf1();
    void mf3();
    void mf4();
};
```

```
Derived d.
int x;

d.mf1();     // Derived::mf1
d.mf1(x);    // Base::mf1
d.mf2();     // Base::mf2
d.mf3();     // Derived::mf3
d.mf3(x);    // Base::mf3
```

- The using declaration brings in everything with that "name".
- If you only wish to inherit the Base::mf3(double) function you must:
  - delete the "using Base::mf3()"
  - provide a forwarding function

Things to Remember:

- Names in derived classes hide names in base classes. Under public inheritance, this is never desirable.
- To make hidden names visible again, employ using declarations or forwarding functions

Item#34

- Differentiate between inheritance of interface and inheritance of implementation.

Notes:

- Alot of discussion in the book on this point.
- Many pages are spent examining virtual vs pure virtual functions and the conceptual design implications of each
- The "things to remember" seems to summarize this well.

Things to Remember:

- Inheritance of interface is different from inheritance of implementation. Under public inheritance, derived classes always inherit base class interfaces.
- Pure virtual functions specify inheritance of interface only.
- Simple (impure) virtual functions specify inheritance of interface plus inheritance of a default implementation.
- Non-Virtual functions specify inheritance of interface plus inheritance of mandatory implementation.

Item#35

- Consider alternatives to virtual functions

Example

```
struct GameCharacter
{
// return character's health value
// derived classes may redefine
virtual int healthValue() const;
};
```

- User code, derives from GameCharacter and either uses supplied healthValue method or supplies it's own
- healthValue is not pure virtual
  - Suggests there is a default algorithm
- Pretty common model of design
  - that is also a weakness
- design is obvious- may not give proper consideration to alternatives
- There are other ways of solving the same problem

Template Method Pattern via Non-Virtual Interface Idiom

- This school of thinking argues
- virtual functions should almost always be private
- a better design would have healthValue as a public member
- make it non-virtual
- have it call private virtual function to do the real work

Example

```
struct GameCharacter
{
  // return character's health value
  // derived classes DO NOT redefine
  int healthValue() const
  {
  //... "before" stuff
  int retVal = doHealthValue(); // real work
  //... "after" stuff
}

private:
  // derived classes may redefine this
  virtual int dohealthValue() const
  {
   // default algorithm
  }
};
```

- Basic design:
- Have clients call private virtual functions indirectly through public non-virtual member functions
- known as "Non-Virtual Interface" idiom (NVI Idiom)
- Advantages:
- The "before" and "after" code is a key advantage
- Intelligent resource handling is possible
- Better control over internal state of object

Example

```
struct GameCharacter
{
  // return character's health value
  // derived classes DO NOT redefine
  int healthValue() const
  {
  //... "before" stuff
  int retVal = doHealthValue(); // real work
  //... "after" stuff
}

private:
  // derived classes may redefine this
  virtual int dohealthValue() const
  {
   // default algorithm
  }
};
```

- Weirdness
  - NVI involves derived classes redefining private virtual functions
  - Functions they can't call!!!
  - "I meant to do that!" - Pee Wee Herman
- The base class controls when the replaceable function gets called
- Positive:
  - Allows for strong isolation

Another way: Strategy Pattern via Function Pointers

- This is a common implementation of the Strategy design pattern.

```
// function for default health calc
int defaultHealthCalc(const GameCharacter & gc);

struct GameCharacter
{
  typedef int (*HealthCalcFunc)(const GameCharacter&);
  explicit GameCharacter(HealthCalcFunc hcf= defaultHealthCalc)
   : healthFunc(hcf) {}

  int healthValue() const
  { return healthFunc(*this); }

  // ...
private:
  HealthCalcFunc healthFunc;
};
```

Interesting flexibility

- Different instances of the same character type can have different health
- EvilCharacter might derive from GameCharacter
- Multiple EvilCharacters can be instantiated
- All with different algorithms for calculating health
- This also allows for the algorithm to CHANGE at runtime
- Could there be negatives???......

On the other hand:

- health calculation is no longer a member function
- no special access to internals of GameCharacter
- syntax is not pretty
- health calculation MUST be a function
- cannot be a functor or something that looks like a function
- health calculation function must return an int
- not something convertible to an int
- This leaves us wondering if there is a better way?......

Strategy Pattern via boost::function

```
short calcHealth(const GameCharacter & gc);

struct GameCharacter
{
  typedef boost::function<int (const
GameCharacter&)> HealthCalcFunc

  explicit
   GameCharacter(HealthCalcFunc hcf=
   defaultHealthCalc)
  : healthFunc(hcf)  {}

  int healthValue() const
  { return healthFunc(*this); }

  // ...
  private:
  HealthCalcFunc healthFunc;
};
```

```
// NOTE: boost::function is a
// generalized function pointer.
struct HealthCalculator {
  int operator()(const GameCharacter &)
const
{ ...}
};

struct GameLevel {
  float health(const GameCharacter &)
const
{ ...}
};

struct EvilBadGuy : GameCharacter
{ .. };

struct EyeCandyCharacter : GameCharacter
{ .. };
```

```
// Usage:
EvilBadGuy ebg1(calcHealth); // using a function

EyeCandyCharacter ecc1(HealthCalculator()); // function object

GameLevel currLevel;

EvilBadGuy ebg2( boost::bind(&GameLevel::Health, currentLevel, _1) );
```

Boost::function

- The constraints with function pointers disappear if we use boost::function
- boost::function is a tr1 library which is essentially a "better" function pointer

Boost::Function Notes:

- it is convertible to the function pointer type
- can receive the results of a bind expression
- can also take a function object (functor)
- Now the syntax is much better
- More flexibility in how we pass in a Health Calculation

Things To Remember:

- Alternatives to virtual functions include the NVI idiom and various forms of the Strategy design pattern.
- A disadvantage of moving functionality from a member function to a function outside the class is that the non-member function lacks access to the class's non-public members
- boost::function objects act like a generalized function pointers. Such objects support all callable entities compatible with a given target signature.

Item#36

- Never redefine an inherited non-virtual function.

Consider:

```
struct B
{
    void mf();
};

struct D : B {...};

D x;

B * pB = &x;
pB->mf();

D* pD = &x;
pD->mf();
```

- Nothing unexpected here...

Now consider:

```
struct B
{
    void mf();
};

struct D : B
{
    void mf(); // hides B::mf()
};

D x;

B * pB = &x;
pB->mf();    // calls B::mf()

D* pD = &x;
pD->mf();    // calls D::mf()
```

- non-virtual functions are statically bound to the pointer or reference type
- virtual functions (on the other hand) are dynamically bound
- This can lead to many confusing situations when trying to read code.
- "Don't do it." - Clint Eastwood

Things to Remember

- Never redefine a inherited non-virtual function

Item#37

- Never redefine a function's inherited default parameter value.

Consider:

```
struct Shape
{
    enum ShapeColor {Red, Green Blue};

    virtual void Draw(ShapeColor = Red) const =0;
};

struct Rectangle : Shape
{
    virtual void Draw(ShapeColor = Green) const;
};

Rectangle R;
Rectangle * pR = &R;
Shape * pS = &R;

pR->Draw(); // Green is used as the default
pS->Draw(); // Red is used as the default
```

- Virtual functions are dynamically bound.
- The default arguments to the dynamically bound call are STATICALLY bound.
- Leaves us wondering WHY C++ does this...
- (well doesn't it?)
- Would you like the next slide now?
- Sure?
- Ok...

Why does C++ do this?

- Performance
- If the arguments were dynamically bound, this would mean a runtime check

Things to Remember

- Never redefine an inherited default parameter value, because default parameter values are statically bound, while virtual functions- the only functions you should be overriding- are dynamically bound.

Item #38

- Model "has-a" or "is-implemented-in-terms-of" through composition.

Example

```
struct Address { ... };
struct PhoneNumber { ... };
struct Person
{
    //  ...
    std::string name;
    Address address;
    PhoneNumber voiceNumber;
    PhoneNumber faxNumber;
};
```

- Person demonstrates "has-a"
- Composition means either "has-a" or "is-implemented-in-terms-of"
- Which definition for composition depends on which domain used....

Application Domain

- Person is using embedded objects to model a real world scenario
- Person, with it's objects, is defined more as a "model" of the domain
- Person implements "has-a" composition

Implementation Domain

- One might have other member variables
- Buffers, counters, mutex
- generally these are implementation details used as member variables
- A class like this would implement "is-implemented-in-terms-of" composition

Basic Point of this item

- Do not use inheritance with "is-implemented-in-terms-of" composition
- this confuses the notion of inheritance "is-a" with "is-implemented-in-terms-of" composition
- It maybe seductive to simply derive from some base class which already has the buffer, counters and mutxes and use those variables directly in the derived class.
- Scott Meyers considers this a bad practice and should be avoided.

Instead..

- Instead, to implement "is-implemented-in-terms-of" composition one should
  - Use a member variable for the "in-terms-of" object
  - Implement forwarding functions to the member variable object
- Gets around hidden gotchas with accidentally inheriting things you did not intend
- Idea that something is "implemented-in-terms-of" is an implementation detail
- Code should not use inheritance, because that works against the "hidden" aspect
- Leaves open accidental or unexpected functionality

Things to Remember

- Composition has meanings completely different from that of public inheritance.
- In the application domain, composition means "has-a". In the implementation domain, it means "is-implemented-in-terms-of".

Item#39

- Use private inheritance judiciously.

Consider:

```
struct Person { ... };

struct Student : private Person
{ ...};
```

- Clearly private inheritance doesn't mean "is-a"
- What does it mean?

2 Rules of private inheritance

- Compilers will generally not convert a derived class object (like student) into a base class object (Person) if the inheritance is private.
- Members inherited from a private class become private members of the derived class, even if they were protected or public in the base class.

What private inheritance means:

- "is-implemented-in-terms-of"
  - Private inheritance is purely an implementation technique
  - means nothing during software design, only during software implementation
- Item 38 points out that composition can be used to implement "is-implemented-in-terms-of"
- so can private inheritance

How does one choose between the two?

- Use composition whenever you can
- Use private inheritance when you must

When must you use Private inheritance?

- when protected members and\or virtual functions enter the picture
- space concerns (Empty Base Optimization (EBO))

Example:

```
struct Timer
{
    explicit Timer(int tick);
    virtual void OnTick() const;
};
```

- Lets say we want to have a class that tracks how many times a member in Widget is called
- However, this means that Widget must derive from Timer.
- Public inheritance is inappropriate in this case
- It is not true that Widget "is-a" Timer
- Widget clients should NOT know about Timer
- Not a part of the conceptual interface
- This also has the artifact of allowing Widget clients to call functions in Timer directly
- NOT GOOD!

So we inherit privately:

```
struct Timer
{
  explicit Timer(int tick);
  virtual void OnTick() const;
};

struct Widget : private Timer
{
  private:
    virtual void OnTick() const;
};
```

- Due to private inheritance, Timer's public OnTick function becomes private in Widget
- This is nice but not necessary...

If we used composition instead:

```
struct Widget
{
private:
    struct WidgetTimer : public Timer
    {
    virtual void onTick() const;
    // ..
    };
    WidgetTimer timer;
public:
    // ...
};
```

- Design is more complicated
- However, derived classes are not permitted to override OnTick()
- which maybe crucial to your design
- Allows for similar functionality to Java's "final" functionality
  - i.e. disallow derived classes from redefining methods

Empty Base Optimization (EBO)

- Classes may qualify for EBO if they are without
  - Data
  - Non-static data members
  - Virtual functions
  - Virtual base classes
- Conceptually, these type of classes should use no space

EBO

```
struct Empty {};

struct HoldsAnInt
{
private:
    int x;
    Empty e;
};
```

- On many compilers:
  - sizeof(HoldsAnInt) > sizeof(int)
- With most compilers sizeof(Empty) is 1
- many compilers will silently add a "char" into storage space of empty so that empty meets the C++ standard requirements
- Alignment requirements may cause compilers to add padding to classes like HoldsAnInt
- it is likely that HoldsAnInt would enlarge enough to hold a char + an int
- (Scott Meyers said he tested many compilers and found this to be the case)

EBO

- C++ standard dictates that "freestanding objects" mustn't have zero size
- Constraint doesn't apply to base class parts of derived class objects
- because they are not freestanding

EBO

```
struct Empty {};
struct HoldsAnInt : private Empty
{
private:
    int x;
};
```

- If you inherit from empty instead of containing an object of that type you are likely to find (compiler dependent) that
  - sizeof(HoldsAnInt) == sizeof(int)
- This is known as the Empty Base Optimization (EBO)
- Scott Tested many compilers and found they all supported EBO
- he does not list however, the compilers

(Brian)

- In meta-programming MOST classes are empty
- This is a key element of why meta-programs are faster at runtime
- You might pull together 30 objects to generate some code, but they are all "empty"
- hence what is left is the code that was generated
- (now back to Effective C++....)

STL & EBO

- STL has many "empty" classes, though in practice most classes are not empty
- it is then rarely a justification for private inheritance
- most inheritance corresponds to "is-a"
- that's a job for public inheritance, not private

Private Inheritance

- Private inheritance is most likely to be a legitimate design strategy when you're dealing with two classes not related by "is-a" where one either needs access to the protected members of another or needs to redefine one or more of it's virtual functions.
- Even in this case, a mixture of public inheritance and containment can often yield the behavior you want
- albeit with greater design complexity

Private Inheritance

- using Private inheritance judiciously means employing it when, having considered all the alternatives, its the best way to express the relationship between two classes in your software

Things to Remember:

- Private inheritance means is-implemented-in-terms-of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions.
- Unlike composition, private inheritance can enable the empty base optimization. This can be important for library writers who strive to minimize object sizes.

Item#40

- Use multiple Inheritance judiciously.

“Use multiple Inheritance judiciously”

- In the community multiple Inheritance (MI) usage breaks into two camps:
- Believe if Single Inheritance (SI) is good, MI must be better
- Single Inheritance is good, MI is not worth the trouble

Consider:

```
struct BorrowableItem
{
  void checkOut();
  // ...
};

struct ElectronicGadget
{
private:
  void checkOut();
// ...
};

struct MP3Player :
public BorrowableItem,
public ElectronicGadget
{
  // ...
};

MP3Player mp;

// ambiguous! Which checkOut()???
mp.checkOut();
```

- checkOut() is ambiguous even though only one of the two functions is accessible
- C++ rules for resolving calls to overloaded functions:
  - before seeing whether a function is accessible, C++ first identifies the function that's the best match
  - Only then does C++ check for accessibility
- To resolve this you must:
- mp.BorrowableItem::checkOut();

Multiple Inheritance

- Multiple Inheritance just means inheriting from more than one base class
- It is not uncommon for MI to be found in hierarchies that have higher level base classes too

Consider:

```
class File { ... };
class InputFile : public File {...};
class OutputFile : public File {...};
class IOFile :
public InputFile, public OutputFile
{ ... };
```

- Data Members?
- where does std::filename go???
- Put it in File, but what does this mean for IOFile?
- it now has "two" filenames

Virtual Inheritance

```
class File { ... };
class InputFile : virtual public File {...};
class OutputFile : virtual public File {...};
class IOFile :
public InputFile, public OutputFile
{ ... };
```

- Virtual Inheritance can solve this problem:
- Note this example is almost directly taken from std streams.

```
class basic_ios { ... };
class basic_istream : virtual public basic_ios
{...};
class basic_ostream : virtual public basic_ios
{...};
class basic_iostream :
public basic_istream, public basic_ostream
{ ... };
```

Virtual Inheritance Negatives

- Larger objects
- access to variables in base classes can be slower
- (both of these are compiler dependant)
- Other costs
- rules governing initialization of virtual base classes are more complicated
- responsibility for initializing a virtual base is borne by the most derived class

Implications of costs:

- classes derived from virtual bases that require initialization must be aware of their virtual bases, no matter how distant
- when a new derived class is added to the hierarchy, it must assume initialization responsibilities for it's virtual bases

Virtual inheritance Advice

- Don't use it unless you need to
- by default use non-virtual inheritance
- If you must, then try to avoid putting data in the classes
- removes the weirdness about initialization
- the same weirdness also comes into play with assignment

Interesting Note

- It is interesting to note that Interfaces in Java and .Net which are comparable to virtual inheritance, are not allowed to contain data.

Is there value in MI?

- Brian Notes:
- I suggest you read Item#40 from the book.
- I do not like his example for how MI is valuable.
- I believe the concept is well founded, but the example is difficult to see his point.

MI

- Basically, his example boils down to this:
- When you have a class that needs to both implement an interface AND "is-implemented-in-terms-of" at the same time, MI is very handy.

Brian Example

```
// [ Implementation Framework Code]
struct Timer { ...}; // For recording how many
times member called
struct Debugable { ... }; // for allowing extra
debugging
struct Loggable { ... }; // for logging to a file

struct TrackableObject : private Timer, private
Debugable, private Loggable
{   ....  };

struct DatabaseCon : private TrackableObject
{...};
struct Grid : private TrackableObject {...};

// [App code]

struct Person : private TrackableObject {...};
struct Place : private TrackableObject {...};
```

- I believe this shows a value in MI for large code-bases
- This is semi-contractdictory to "Effective C++" thinking

Brian's point?

- In the Implementation Framework layer, one may have several dimensions of "is-implemented-in-terms-of"
- Being able to compose\combine "implementation" from other implementations is valuable
- Both inside the Framework layer itself
- And inside the app code
- Having a "core" place to go to affect functionality across the system is key to being able to control large code-bases
- sans this design ability, as time goes on, the code becomes more cluttered and unwieldy
- None of this is easily possible without MI
- The only alternative is LOTS of forwarding functions
- The MI issues due crop up in this scenario, however typically this is done with unrelated "is-implemented-in-terms-of" concepts

Things to Remember:

- Multiple inheritance is more complex than single inheritance. It can lead to new ambiguity issues and to the need for virtual inheritance.
- Virtual inheritance imposes costs in size, speed, and complexity of initialization and assignment. It's most practical when virtual base classes have no data.
- Multiple Inheritance does have legitimate uses. One scenario involves combining public inheritance from an interface class with private inheritance from a class that helps with implementation.

END PART 4

- THANK YOU