Index

Section 7.3. Runtime Type Identification

Section 7.4. Efficient, but Inflexible?

Ch. 1 Object Lessons

- C separates data and operations
- C++ combines data and operations into independent ADT
- allows for class hierarchies and template based parameterization on types (and integral values)

Layout Costs for Adding Encapsulation

- no additional space or runtime costs to support abstraction in C++
- each inline function has 0 o 1definition of itself generated within each module in which it is used
- primary layout and access-time overheads in C++ are associated with virtuals
  - virtual function mechanism used for run-time binding
  - virtual base class in support of single shared instance of base class occurring multiple times in inheritance hierarchy
- overhead in multiple inheritance in conversion of derived class in support of second base class
- no inherent reason why C++ program should be slower than C equivalent

The C++ Object Model

- class data member types
  - static
  - nonstatic
- member function types
  - static
  - nonstatic
  - virtual

A Simple Object Model

- oversimplified to build to closer view of object model
- data members and member functions are stored in slots without either actually being stored in structure
- not used but useful for understanding pointers to members

A Table-driven Object Model

- alternate option to maintain uniform repr across classes
- split data members into one structure and pointers to member
- class holds pointer to data member structure and pointer to member function table

The C++ Object Model

- original and prevailing (as of 2003) object model
- nonstatic data members allocated directly within each class object
- static data members are stored outside of individual class objects
- static and nonstatic member functions are hoisted outside of class object
- virtual functions are supported by
  - generating a table of pointers to virtual member functions for each class (virtual table or vtable)
  - single pointer to vtable is stored within each class object (vptr)
    - setting, resetting and not setting of vptr handled through code generated in ctors, dtor, and assignment operators
    - type_info objects associated with classes when RTTI is used is also stored within the virtual table
- good for space and runtime efficiency
- bad due to need to recompile code when a modification to a class' nonstatic data members is made
  - two table model would avoid at cost of runtime efficiency

Adding Inheritance

- supports
  - single inheritance
  - multiple inheritance
  - virtual inheritance
- virtual inheritance -> only single occurrence of base class is maintained no matter how many times it occurs in the inheritance hierarchy (occurrence of base class is shared)
- original C++ object model removed all indirection for storage of base classes and stored base class members and derived class members within derived class instance
- virtual bases caused modification; originally used a pointer to the base class object and other implementations add a virtual base class table or add pointers to each base class to existing virtual table

How the Object Model Effects Programs

- object model support modifies existing code and adds additional code

A Keyword Distinction

- C++ is considerably more complex due to efforts to maintain C compatibility
- adds meta-language rule to resolve most vexing parse (inability to distinguish between declaration and expression) by assuming declaration
- support for both class and struct
- details unnecessary philosophical issues concerning class and struct

An Object Distinction

- C++ supports
  - C-style procedural model
  - abstract data type model
  - object-oriented model
  - (generic model)
- OO and ADT models differ in the use of polymorphism (ADT does not use it)
- C++ allows polymorphism only through manipulation of pointers and references in relation to public class hierarchies
- primary use is to effect type encapsulation through a shared interface
  - allows interface to be used across many varieties of a type
- memory requirements to represent a class object
  - accumulated size of nonstatic data members
  - additional padding to alignment constraints (or efficiency requirements)
  - any additional overhead to support virtuals
- pointers are fixed memory size

```
class ZooAnimal {
public:
    ZooAnimal();
    virtual ~ZooAnimal();
// ...
    virtual void rotate();
protected:
    int loc;
    string name;
};
ZooAnimal za( "Zoey" );
ZooAnimal *pza = &za;
```

```
// layout

            ________________________
0x1000:     |_int_loc_______________|
            | string name           |
            |           int length|
            |           char * str|
            | ...                   |
            |_______________________|
            | __vptr__ZooAnimal     |
            |_______________________|
```

The Type of a Pointer

- type of a pointer is used to instruct the compiler on how to interpret the memory at a location and what size that memory is
- void * can only hold address and not operate on the memory it points to

Adding Polymorphism

```
class Bear : public ZooAnimal {
public:
    Bear();
    ~Bear();
    // ...
    void rotate();
    virtual void dance();
    // ...
protected:
    enum Dances { ... };
    Dances dances_known;
    int cell_block;
};

// layout
0x1000:     ________________________
            | ZooAnimal           |
            | subobject           |
            | ...                 |
            |_____________________|
            | Bear members        |
            | dances_known        |
            | cell_block          |
            |_____________________|
```

- the type of a pointer to a base class the is used to point to a derived class has the following determined at compile time
  - the fixed available interface, i.e. only the public interface of the base class is available to the pointer
  - the access level of the interface

- the actual function invoked is determined by the vptr table for the actual instance of the object pointed to by a polymorphic base pointer
- direct assignment of a variable of derived class type to base class variable results in slicing and any method calls will be the base class methods
    - polymorphism is not physically possible in directly accessed objects
    - direct object manipulation is not supported in object-oriented paradigm (here direct object manipulation means object manipulation at a type level, not an instance level)
- object-orientation is defined by
    - encapsulation of related data behind an abstract public interface
    - adds an additional level of indirection (both memory acquisition and type resolution)
    - result is polymorphism
- polymorphism is an essential component of object-orientation
- object-based, ADT style programming of non-polymorphic data is also supported
    - technically not OO
    - often faster and more compact than OO
    - less flexible than OO and polymorphism
- (the distinctions specified here have largely disappeared in current OO descriptions and considerations)

Ch. 2 The Semantics of Constructors

- compilers perform magic in C++ that can be the source of frustration and confusion
- example is conversion operators
    - implicit conversions of types to boolean or integer values
    - see http://en.cppreference.com/w/cpp/language/cast_operator
    - can be achieved by defining operators of types for a class (both explicit and implicit)
- explicit keyword added to suppress application of single argument constructor as a conversion operator
- most compiler meddling comes in the form of default member generation, memberwise initialization, and name return value optimization (NRV)

Default Constructor Conversion

- default constructors generated by compiler when needed
    - global objects are guaranteed to have their memory zeroed out by default on program start
    - local and heap objects are not guaranteed to have their memory zeroed out
    - default ctor for simple classes does not include code to zero members
    - when no user ctor for a class, default ctor is implicitly declared
    - trivial ctor is an implicitly declared default ctor

    - nontrivial default ctor is one that is required by the implementation and synthesized by the compiler when necessary

Member Class Object with Default Constructor

- if class contains member with a default ctor, then the class is nontrivial and compiler must generated default ctor
- default ctor in this case is only generated if the ctor actually needs to be invoked
- prevents multiple generation from separate translation units by having the synthesized functions inlined which causes them to have static linkage
- if function is too complex to be inlined, a non-inline static instance is synthesized
- member ctors for class members are invoked within generated ctor
- multiple class members have their default ctors invoked in order of declaration

Base Class with Default Constructor

- if class without default ctors is derived from base class with ctors, derived class is nontrivial and needs synthesized ctors etc

Class with a Virtual Function

- default ctor also needed when:
    - class declares or inherits virtual functions
    - class is derived from inheritance chain in which one or more base classes are virtual
- in each case, a default ctor will be generated when necessary
- classes are also augmented in following ways
    - a virtual function table is generated and populated with the addresses of the active virtual functions for that class
    - the additional virtual table pointer member (vptr) is synthesized to hold the address of the associated class vtbl
- additionally virtual method invocations are rewritten to point to and call virtual function entry within the vtable

```
this->flip() // becomes
(this->vptr[1])(this);
// or
(*class_instance.vptr[1])(&class_instance)
```

Class with a Virtual Base Class

- derived classes need to have access to virtual base classes at runtime
- method for this is different across compiler implementations
- compilers insert code within constructors to manage this access via a pointer to base class

Summary

- 4 characteristics of a class under which a compiler needs to generate a default ctor
  - implicit & nontrivial default ctors
- common incorrect beliefs
  - a default ctor is generated for every class that does not define one
  - compiler generated default ctors provide explicit default initializers for each data member of a class

Copy Constructor Construction

- 3 methods for invocation of copy ctor
  - copy assignment -> X x_instance = x;
  - pass by value to function taking class as argument
  - return (by value) of a class from a function
  - (explicit copy construction)

Default Memberwise Initialization

- when no copy ctor is defined, default memberwise initialization is used
  - copies value of each built-in (i.e. primitive) or derived data member (pointer or array) from one class to the other
  - member class objects a recursively memberwise initialized
- most compilers can generate bitwise copies for most class objects (is_trivially_copyable)
- as with default ctors, implicitly declared and implicitly defined copy constructor are generated as needed if a class does not declare one
- trivially copyable means a class exhibits bitwise copy semantics

Bitwise Copy Semantics

```
// from cppreference for TriviallyCopyable
Requirements:
Every copy constructor is trivial or deleted
Every move constructor is trivial or deleted
Every copy assignment operator is trivial or deleted
Every move assignment operator is trivial or deleted
at least one copy constructor, move constructor, copy assignment operator, or
  move assignment operator is non-deleted
Trivial non-deleted destructor

This implies that the class has no virtual functions or virtual base classes.
Scalar types and arrays of TriviallyCopyable objects are TriviallyCopyable as well
  as well as the const-qualified (but not volatile-qualified) versions of such typ

// from cppreference for TrivialType
```

```
Specifies that a type is trivial type.
Note, that the standard doesn't define a named requirement or concept with this na
  This is a type category defined by the core language. It is included here as
  concept only for consistency.
Requirements:
TriviallyCopyable
Has one or more default constructors, all of which are either trivial or deleted,
  and at least one of which is not deleted.
```

Bitwise Copy Semantics-Not!

- four cases for bitwise copy semantics
  - when the class contains a member object of a class for which a copy constructor exists
  - when the class is derived from a base class for which a copy constructor exists
  - when the class declares one or more virtual functions
  - when the class is derived from an inheritance chain in which one or more base classes are virtual

Resetting the Virtual Table Pointer

- when vtable is created and vptr is added to class a class no longer exhibits bitwise copy semantics

Handling the Virtual Base Class Subobject

- inheritance from a virtual base class invalidates bitwise copy semantics
- compiler must insert and maintain integrity of a reference to a virtual base class within a derived class instance

Program Transformation Semantics

- common assumptions about return by value
  i. every invocation of a function that appears to return by value actually returns by value
  ii. any return by value invokes a copy ctor if a copy ctor is defined/exists
- 1 depends on definition of class
- 2 depends on definition of class and optimizations performed by compiler
- a high-quality C++ compiler might result in both assumptions always being false

Explicit Initialization

- all of the following explicitly initialize x with x0

```
X x0;

X x(x0);
X x{x0};
```

```
X x = x0;
X x = X(x0);
```

- program is required to be transformed as follows
  - each definition is rewritten to remove the initialization
  - the initialization is replace by an invocation of the class copy ctor

Argument Initialization

TODO: more here

Ch. 3 The Semantics of Data

- sizeof with inheritance yields increasing sizes
- empty class is never empty (needed to provide an address of an object)

```
class X {};
sizeof(x) == 1; // true
```

- compiler inserts a 1 byte char member to ensure objects are given addresses
- this combined with inheritance results in
  - language supported overhead (both memory and runtime resolution of virtual methods and base classes)
  - compiler optimization of recognized special cases
    - adds 1 byte char
    - compilers now support empty virtual bases
  - alignment constraints
- !!!! empty base class manual optimization is similar/equivalent to the curiously recurring template pattern (CRTP) which is used for inheritance of interface without additional virtual lookup overhead
- additional size confusion issues
  - virtual base class is included only once in size of overall class inheritance hierarchy
- size of class is determined by the following
  - size of single shared instance of virtual base
  - size of any intermediate inherited classes minus the size of any shared instance of a virtual base class
  - size of derived class -> 0 in inheritance hierarchy like this
  - any additional alignment requirements
  - (with empty base class optimization the padding and size of the base class is removed)

- C++ optimizes for space and access time for nonstatic data members by storing them directly within class instances
- static data members are maintained within the global data segment of the program and do not affect the size of class instances
- static data members of a class template are slightly different
- class instance is the size required to contain nonstatic data members of class along with special considerations previously discussed
    - additional data members added by the compilation system to support some language functionality (i.e. virtuals)
    - alignment requirements on data members and data structures as a whole

The Binding of a Data Member

- old implementations would sometimes resolve a global name before a name within a class, which led to 2 styles of defensive programming
    - placing all data members first in a class declaration to ensure correct binding
    - placing all inline functions, regardless of size, outside of class declaration

Data Member Layout

- static data members are stored in program's data segment independent of individual class instances
- nonstatic data members are stored directly within memory bounds of the class instance
- adjacent data members may not be stored contiguously due to alignment constraints and related padding
- additionally, an object instance may contain one or more synthesized data members to reference vtable and virtual base classes
- order of data members may be implementation dependent when data members are split across multiple access sections (i.e. separate segments within the class definition separated by (potentially multiple uses of) public, private, protected, etc)
- generally, however, multiple access sections are concatenated

Access of a Data Member

- many combinations for data access rules
    - static vs. nonstatic data members
    - single concrete class vs. derivation from base class
    - single inheritance vs. multiple inheritance vs. virtual inheritance

Static Data Members

- static data members are lifted/hoisted outside of class definition
  - member's access permissions and type associations are maintained during compilation without incurring space or runtime overhead
  - single instance per class is stored within data segment of the program and accesses of that member are translated to accesses of instance in global data segment
  - accesses via . operator and -> operator result in equivalent instructions
  - address of static data member yields a normal pointer to the type of the static member rather than pointer to member
  - resolves naming conflicts via name mangling and scope resolution

Nonstatic Data Members

- stored directly within class instance
- member accesses within class code is implicitly resolved through the this pointer
- direct access requires beginning address of class object and offset of location of data member within class

```
&class_instance._member // equivalent to
&class_instance + (&Class::_member - 1)
```

- offsets of pointer-to-data-member always increased by 1
- offsets of nonstatic data members are known at compile time
- nonstatic data member access is equivalent to access of C struct member
- virtual inheritance adds an additional level of indirection for base class subobject member access

Inheritance and the Data Member

- a derived class object is represented as the concatenation of its members with those of its base classes
- ordering is left unspecified by standard
- review
  - single inheritance, no virtual functions
  - single inheritance, virtual functions
  - multiple inheritance
  - virtual inheritance

Inheritance without Polymorphism

- in general, concrete inheritance adds no space or access time overhead
- in this relationship, need to carefully choose inlined functions and ensure they are actually inlined by compiler
- even flat inheritance is affected by alignment constraints (in C as well as C++)
- these rules remain true because the integrity of any base class subobject instance must be preserved by a derived object instance
    - failure to do so could lead to situations that would result in data corruption

Adding Polymorphism

- support for flexibility added by adding a virtual base interface (e.g. polymorphism) adds space and access time overheads
    - vtable
        - slots for each virtual function
        - potentially 1 or 2 slots for RTTI
    - vtable pointer space
    - additional runtime code to support management of vtable and vtable pointer
        - ctor
        - dtor
- additional runtime code can be minimized by an aggressively optimizing compiler
- compiler debate at time of publish was where to place vptr within a class instance layout
    - placing at end is linked to usage of C structs and a legacy compatibility with C
    - placing at start is more efficient for some access patterns
    - tradeoff is loss of C language compatibility, but no evidence to support that this was a real world requirement

Multiple Inheritance

- multiple inheritance is not as well behaved or as easily modeled as single inheritance
- standard does not require ordering for placement of base class subobjects, often based on order of declaration
- ordering may be switched based on compiler optimizations
- no additional costs for accesses to additional base class subobjects because subobject ordering is set at compile time and access addresses are essentially hard-coded in

Virtual Inheritance

- iostreams are commonly cited example

- semantics of virtual inheritance are complex and implementation within compiler is even more complex
  - a class containing one or more virtual base class subobjects, is divided into two regions
    - an invariant region
    - a shared region
  - data within the invariant region remains at a fixed offset from the start of the object regardless of subsequent derivations
  - members within the invariant region can be accessed directly
  - shared region represents the virtual base class subobjects
  - location of data within the shared region fluctuates with each derivation which mean members within shared region need to be accessed indirectly
- weaknesses of model
  - object of class carries additional pointer for each virtual base class
  - indirection chains lengthen as virtual inheritence chain lengthens
- some compilers promote (copy) all virtual base class subobjects into derived class instance
- other solutions exist for first problem including virtual base class table

Object Member Efficiency

- review test performed here and recreate on modern hardware

Pointer to Data Members

- mainly useful for probing underlying data layout of a class

```
class Point3d {
public:
   virtual ~Point3d();
   // ...
protected:
   static Point3d origin;
   float x, y, z;
};

float Point3d::*p1 = 0;
float Point3d::*p2 = &Point3d::x;

if (p1 == p2) {
   cout << " p1 & p2 contain the same value – ";
   cout << " they must address the same member!" << endl;
}

// also consider
& 3d_point::z;
// and
& origin.z
```

Efficiency of Pointers to Members

- review benchmarks here and adjust for modern compilers/machines

Ch. 4 The Semantics of Function

- what happens for member function calls through both pointers and direct access
- depends on type of member function
  - static
  - nonstatic
  - virtual

Varieties of Member Invocation

- originally only supported only nonstatic members
- virtual functions were added in the mid-80's
  - originally viewed as a form of inefficiency
- static members added last

Nonstatic Member Functions

- requires that nonstatic member function is at least as efficient as analogous nonmember function

```
func(const Class* this)
Class::func()
```

- achieved by internally transforming member to nonmember equivalent
  i. rewrite the signature to insert an additional argument to the member function that provides access to the invoking class object
     - this is called the implicit this pointer
     - add const for a const member function
  ii. rewrite each direct access of a nonstatic data member of the class to access the member through the this pointer
  iii. rewrite the member function into an external function, mangling its name so that it is lexically unique within the program
- in some cases a return value that might require a default ctor to be invoked will be transformed to an in/out reference parameter that is assigned to

Name Mangling

- adds information about class to internal function name by prepending class name and sometimes size, etc to ensure unique member function names
- different across compilers

Virtual Member Functions

```
instance_ptr->member()
// becomes
(*instance_ptr->vptr[1])(instance_ptr)
```

- assumes
  - vptr represents pointer to internally generated virtual function table (name is mangled in practice)
  - 1 is index within virtual function table referencing the member function
  - argument passed of instance pointer represents the implicit this pointer
- inlined virtual member functions are more efficient because the compiler rewrites each invocation to directly invoke the function as described for nonstatic member functions

Static Member Functions

- all static member function invocations are translated to normal function invocations
- primary characteristic is that it is not called with the implicit this argument
  - cannot directly access nonstatic members of an object instance
  - cannot be declared const, volatile, or virtual
  - does not need to be invoked through an object instance of the class, but be invoked this way for convenience
- this mechanism can be used to successfully interface with C based APIs

Virtual Member Functions

- to support the virtual function mechanism, some form of runtime type resolution applied to polymorphic objects must be supported
- many options presented here
- C++ facilitates polymorphism through the ability to assign a derived instance pointer to a base pointer (also through reference mechanism)
- polymorphism is passive when the base pointer object is just being passed to and from functions without being called
- polymorphism is active when member functions are actually called, thus invoking the virtual lookup mechanism

- additional active polymorphism is used in RTTI
- compiler completes machinery to support virtual function call mechanisms
  - vptr inserted into each object to find virtual table
  - virtual functions are assigned fixed indexes within the virtual function table
- structure of virtual function table
  - instance of all overriding functions defined within class
  - instance of all inherited (not overridden) virtual functions
  - a pure_virtual_called() library instance to serve as placeholder for pure virtual functions and to invoke a runtime exception if a pure virtual function is called
- management of inheritance and adjusting virtual function tables
  - address of function instance within base class is copied into derived class vtable
  - overridden virtual functions replace address of the base class' related function with its own
  - derived class can add new virtual functions of its own, increasing size of virtual table by adding address of new virtual function
- virtual function tables also contain type info for the related class

Virtual Functions under Multiple Inheritance

- adds complexity
- assignment of a derived class to a base pointer causes a compile time adjustment of the pointer to point to the pointer to derived + offset of pointer to base within class hierarchy
- calls to delete are readjusted to account for full size of object
- general rule is that the this pointer adjustment in these cases is accomplished at runtime
- can implement solution to avoid extra overhead using a thunk
  - see vcall thunk for info (https://en.wikipedia.org/wiki/Thunk)
- see diagram here for virtual table layout in multiple inheritance
- some compilers concatenate multiple virtual tables into 1
- TODO: more description here

Virtual Functions under Virtual Inheritance

- material to show how this is handled is too complex to generally describe
- suggestion is to not declare nonstatic data members within virtual base classes

Function Efficiency

- shows tests for various forms of member vs nonmember functions
- can recreate tests to demonstrate results

Pointer-to-Member Functions

- address of nonstatic member function is the result of the actual location of the symbol within the text/code segment of the executable if it is nonvirtual
- address is incomplete without the additional address of the this instance
- must invoke

```
void (Class::*ptr_mem_fun)() = &Class::member;
(instance.*ptr_mem_fun)();
// or
(ptr->*ptr_mem_fun)();

// converted by compiler to
(ptr_mem_fun)(&instance);
```

- practice (when disregarding virtual function mechanism overhead) compiler can provide equivalent performance for a pointer-to-member function call as a direct member function call

Supporting Pointer-to-Virtual-Member Functions

- still supports pointer to member function calls as above
- address is unknown at compile time but is resolved through the vptr and index of member within vtable

```
(ptr->*ptr_mem_fun)();
// becomes
(ptr->vptr[statice_cast<uintptr_t>(ptr_mem_fun)])(ptr);
```

- may have changed over time

Pointer-to-Member Functions under MI

```
// fully general structure to support
// pointer to member functions under MI
struct __mptr {
   int delta;
   int index;
   union {
      ptrtofunc  faddr;
      int        v_offset;
   };
};
```

- calling

```
(ptr->*pmf)()
// becomes
                  // non-virtual      // virtual invocation
(pmf.index < 0) ? (*pmf.faddr)(ptr) : (*ptr->vptr[pmf.index](ptr);
```

- many compiler implementation provide pointer-to-member representations based on the characteristics of the class
  - single inheritance instance (simply holds vcall thunk or function address)
  - multiple inheritance instance (faddr and delta members)
  - virtual inheritance instance (four members)

Pointer-to-Member Efficiency

- test detailed show benchmark results

Inline Functions

- can restrict access but maintain efficiency of direct member action by using inlined setters and getters
- inline keyword and mechanisms are only a request to the compiler to inline
  - based on compiler's ability to reasonably expand the function based on the content of the function
- two phases to inline handling
  - analysis of function to determine "inline-ability"
    - turned into static function in some cases when not inline-able
  - inline expansion of function at point of call when inline-able

Formal Arguments

- each formal argument replaced with actual argument
- handling actual expression arguments that have side effects requires introduction of a temporary
- constant expression arguments are evaluated and then substituted
- if neither constant expression nor expression with side effects, basic substitution is used

Local Variables

- in general
  - each local variable within the inline function must be introduced into the enclosing block of the call as a uniquely named variable
- inline functions provide necessary access control support
- inline functions are good replacements for #define macros
- inline functions can lead to executable bloat

Ch. 5 Semantics of Construction, Destruction, and Copy

- a private/protected data member in an abstract base class (any class with pure virtual functions) must be initialized otherwise any inheriting class
- only way to ensure this member is initialized by inheriting classes is to provide a protected ctor in base that takes an argument of that type with a default value
- in general, the data members of a class should be initialized and assigned to only within the constructor and other member functions of that class
- often better to remove data members from abstract base classes

Presence of a Pure Virtual Destructor

- a pure virtual may be defined and invoked but it must be invoked statically rather than through the virtual resolution mechanism

```
inline void
Abstract_base::interface()
{
    // ...
}

inline void
Concrete_derived::interface()
{
    // ok: static invocation
    Abstract_base::interface();
    // ...
}
```

- this is rarely done except in instance of a pure virtual dtor
- each derived class dtor is internally augmented to statically invoke each of its virtual base and immediate base class destructors
- absence will generally result in link time error

Presence of a Virtual Specification

- compilers may optimize virtual function specifications that are not used but it is a bad design choice to declare all functions virtual and rely on compiler to optimize away calls

Presence of const within a Virtual Specification

- results in problems and author tends towards leaving off const specification in virtual declarations

A Reconsidered Class Declaration

- lifetime of an object is a runtime attribute of that object
- local object's lifetime extends from definition to scope end (or rarely explicit destruction)
- global object lifetime is entire program execution
- heap object lifetime extends from point at which operator new is called to allocate heap memory for the object to the point at which operator delete is called on the object to deallocate it
- internally even a POD has default ctors and a dtor generated and called for lifetime management
- difference between data segment and bss segment in C is not exercised in the same way in C++ due to this fact (all global objects are treated as initialized)

Abstract Data Type

- explicit initialization lists are generally faster
- gives drawbacks here that no longer apply
- defines Point without any virtual mechanism (no inheritance)

Preparing for inheritance

- adds virtual getters and outlines previously discussed internal augmentations to code

Object Construction under Inheritance

- ctors often have a lot of internal compiler augmentation
    i. data members initialized in the member initialization list have to be entered within the body of the constructor in the order of member declaration
    ii. if a member class object is not present in the member initialization list but has an associated default constructor, that default constructor must be invoked
    iii. before that, if there is a virtual table pointer (or pointers) contained within the class object, it (they) must be initialized with the address of the appropriate virtual table(s)
    iv. before that, all immediate base class constructors must be invoked in the order of base class declaration (the order within the member initialization list is not relevant)
        - if the base class is listed within the member initialization list, the explicit arguments, if any, must be passed
        - if the base class is not listed within the member initialization list, the default constructor (or default memberwise copy constructor) must be invoked, if present
        - if the base class is a second or subsequent base class, the this pointer must be adjusted

v. before that, all virtual base class constructors must be invoked in a left-to-right, depth-first search of the inheritance hierarchy defined by the derived class
    - if the class is listed within the member initialization list, the explicit arguments, if any, must be passed
    - otherwise, if there is a default constructor associated with the class, it must be invoked
    - in addition, the offset of each virtual base class subobject within the class must somehow be made accessible at runtime
    - these constructors, however, may be invoked if, and only if, the class object represents the "most-derived class."
    - some mechanism supporting this must be put into place

Virtual Inheritance

- there is an idiom for suppressing the invocation of a virtual base class dtor

The Semantics of vptr Initialization

- within the constructors (and destructor) of a class, the invocation of a virtual function by the object under construction is limited to the virtual functions active within the class
- necessary because of the class order of constructor invocation
    - Classes are built from the bottom up and then the inside out
- result is that the derived instance is not yet constructed while the base class constructor executes
- the virtual table is what actually determines the virtual function set that is active for a class
- the best option for handling vptr initialization is to initialize it after the invocation of the base class ctors but before execution of use-provided code or the expansion of members initialized within the member initialization list within the ctor
- general algorithm of ctor execution
    i. within the derived class constructor, all virtual base class and then immediate base class constructors are invoked
    ii. when 1 is complete, the object's vptr(s) are initialized to address the associated virtual table(s)
    iii. the member initialization list, if present, is expanded within the body of the constructor
        - this must be done after the vptr is set in case a virtual member function is called
    iv. explicit user-supplied code is executed
- two conditions under which vptr must be set

1. When a complete object is being constructed
    - declare a Class object, the Class constructor must set its vptr
2. within the construction of a subobject instance, a virtual function call is made either directly or indirectly

Object Copy Semantics

TODO: more here

Ch. 6 Runtime Semantics

- many transformations occur to source code, thus throwing off expectations about how many instructions may be executed by a given expression/statement/operation etc

Object Construction and Destruction

- ctor/dtor insertion for a simple declaration is as expected
- multiple return blocks from a function require that a dtor be placed for locals at each exit point in which any given locals are alive
- presence of a goto statement may require many dtors to be inserted
- in general, limit the lifetime of an object as much as possible

Global Objects

- language guarantees global objects are constructed before program startup
- all placed in data segment and initialized to 0 or a specified value
- description of older methods for handling statics/globals here
- drawbacks to using statically initialized objects:
  - if exception handling is supported, these objects cannot be placed within try blocks
    - bad statically invoked constructors because any throw will by necessity trigger the default terminate() function within the exception handling library
  - complexity involved in controlling order dependency of objects that require static initialization across modules
- recommendation is to not use global objects that require static initialization and not use global objects at all

Local Static Objects

- guaranteed semantics for local static objects
  - object must have its constructor applied only once, although the containin function may be invoked multiple times
  - object must have its destructor applied only once, although again the function may be invoked multiple times

- compiler implementations use address of local static objects to check if the object has been initialized on first use and initialize if necessary
- dtors are invoked within the static deallocation functions associated with a program text file
- semantics of this may have been changed by standards committee

Arrays of Objects

TODO: more here

Default Constructors and Arrays

TODO: more here

Operators new and delete

- operator new has discrete steps
  - allocate memory (as void ptr) and cast depending on context
  - check for successful allocation
  - assign memory
- delete is handled similarly

The Semantics of new Arrays

TODO: more here

The Semantics of Placement Operator new

TODO: more here

Temporary Objects

TODO: more here

A Temporary Myth

TODO: more here

Ch. 7 On the Cusp of the Object Model

- templates
- exception handling
- runtime type identification

Templates

- powerful feature but can be difficult and error prone
- template support
  i. processing of the template declarations
     - what happens when a template class is declared
     - template class member function
     - etc
  ii. instantiation of the class object and inline nonmember and member template functions
     - instances required within each compilation unit.
  iii. instantiation of the nonmember and member template functions and static template class members
     - instances required only once within an executable
     - where the problems with templates generally arise

Template Instantiation

- must specify explicit instantiations of templates to access nested public types and static members

```
Class<Type>::Nested_type
Class<Type>::static_member

// rather than
Class::Nested_type
Class::static_member
```

- definition of class instance as in

```
Class<Type> instance;
```

results in instantiation of template

- template member functions are only instantiated for a specific type instantiation of the template if the member is used
  - for space and time efficiency
  - avoid errors based on unimplemented functionality

Error Reporting within a Template

- errors in a standard class definition are resolved by the compiler at compiler time
- for templates, type checking for template parameters is deferred until an actual instantiation is encountered

- various compilers instantiate templates in different ways and thus errors are caught at different times TODO: check this in current standard

Name Resolution within a Template

- scope of template definition -> site at which template is defined
- scope of template instantiation -> site at which template is actually instantiated
- an implementation must keep two scope contexts
  i. the scope of the template declaration, which is fixed to the generic template class representation
  ii. the scope of the template instantiation, which is fixed to the representation of the particular instance
- a compiler's resolution algorithm must determine which is the appropriate scope within which to search for the name

Member Function Instantiation

- current (old) implementations provided two instantiation strategies
  - a compile-time strategy in which the code source must be available within the program text file
  - a link-time strategy in which some meta-compilation tool directs the compiler instantiation
- (I think this is all compile time now)
- three questions to answer
  i. how does the implementation find the function definition?
  ii. how does the implementation instantiate only those member functions that are actually used by the application?
  iii. how does the implementation prevent the instantiation of member definitions within multiple .o files?

TODO: more details here and refer to standard for updated information

Exception Handling

- requires tracking the local functions and info about objects local to that function
- requires ability to query type of exceptions (limited type information mechanism)
- requires some clean up mechanism

- requires tightly coupled link between compiler-generated data structures and the runtime exception library
    - to maintain execution speed, the compiler can build up the supporting data structures during compilation
    - adds to the size of the program, but means the compiler can ignore these structures until an exception is thrown
    - to maintain program size, the compiler can build up the supporting data structures "on the fly" as each function is executed
    - affects the speed of the program but means the compiler needs to build (and then can discard) the data structures only as they are needed
- exception handling has historically led to many issues for compilers

A Quick Review of Exception Handling

- consists of
    i. throw clause
        - a throw clause raises an exception at some point within the program
        - the exception thrown can be of a built-in or user-defined type
    ii. one or more catch clauses
        - each catch clause is the exception handler
        - it indicates a type of exception the clause is prepared to handle and gives the actual handler code enclosed in braces
    iii. try block
        - a try block surrounds a sequence of statements for which an associated set of catch clauses is active
- thrown exception causes program control to incrementally be passed back up the stack until an appropriate catch handler is encountered or main is reached
    - when main is reached, the default handler (terminate()) is invoked
- as the stack is unwound, destructors are invoked for each active object within the current scope of the function with control
- each segment of a scope requires the compiler to add additional bookkeeping that tracks the local objects that need to be cleaned up when the stack is unwound through a particular scope
- any function that requires resource clean-up must provide a mechanism for a final action within a scope otherwise nothing will be done as the stack is unwound through a scope and resources will be leaked
- can be done with a default catch clause

```
try {
    // ...
}
catch (...) {
```

```
    // ...
}
```

- (can also be done with a Final_action object)

```
template<typename F>
struct Final_action {
    Final_action(F f) : clean{f} {}
    ~Final_action() { clean(); }
    F clean;
};

template<class F>
Final_action<F> finally(F f) {
    return Final_action<F>(f);
}

auto act1 = finally([&] {
    // do cleanup
});
```

- best approach in general is to use RAII classes (smart pointers, etc) for all resources
- be aware of the exceptional cases for memory allocation (new, malloc, etc)
- (see https://opensource.apple.com/source/libunwind/libunwind-35.3/include/mach-o/compact_unwind_encoding.h for some more info about unwinding)

Exception Handling Support

- compilation system must do the following to handle exceptions:
  i. examine the function in which the throw occurred
  ii. determine if the throw occurred in a try block
  iii. if so, then the compilation system must compare the type of the exception against the type of each catch clause
  iv. if the types match, control must pass to the body of the catch clause
  v. if either it is not within a try block or none of the catch clauses match, then the system must
    - destruct any active local objects
    - unwind the current function from the stack
    - go to the next active function on the stack and repeat items 2–5

Determine if the Throw Occurred within a try Block

- function can be thought of set of regions
  - a region outside a try block with no active local objects
  - region outside a try block but with one or more active local objects requiring destruction
  - a region within an active try block

- compiler marks regions for exception handling system
  - commonly done with program counter-range tables
  - program counter holds address of next instruction (EIP/RIP, etc)
  - range table tracks try blocks with beginning and ending program counter for block and related catch block
- on throw
  - if current program counter is within a try/catch block, related catch blocks are checked for a type match
  - if exception is not handled, current function is popped from program stack, the program counter is modified to point to the entry point of the previously active function and the range tables are checked for a match to current program counter
  - exception handling mechanism is performed recursively until exception is handled or main is reach and the default terminate() is called

Compare the Type of the Exception against the Type of Each Catch Clause

- for each throw expression, the compiler must create a type descriptor encoding the type of the exception
- if the type is a derived type, encoding must include information on all of its base class types (to handle polymorphic catches by reference)
- RTTI is a necessary side effect of exception handling support
- type descriptors are used for catch clauses

What Happens When an Actual Object Is Thrown during Program Execution?

- a thrown exception object needs to be set aside and stored on some sort of exception data stack to be passed through unwinding of the stack
- an object passed by value to a catch clause is
  - initialized by value by the thrown exception object
  - when not caught by reference, an exception will be sliced depending on the relationship of the types
  - any virtual function mechanisms are replaced by the value object
  - the by value object is then only a local object which will be destroyed at the end of the catch clause
  - if rethrown using a standalone throw statement will rethrow the original exception
- see referenced article for size and speed comparisons with and without exception handling (possibly recreate on modern architectures with current compilers)
- exception handling implementations within compilers are the most varied language feature

Runtime Type Identification

- prior implementations did not allow arguments to conversion operators

- const member functions changed this limitation
- casting from a base class to a derived class is often called a downcast because of the convention of drawing inheritance trees growing from the root down
- a cast from a derived class to a base is called an upcast
- a cast that goes from a base to a sibling class is called a crosscast

Introducing a Type-Safe Downcast

- C++ was originally criticized for not having a type-safe downcast
- a type-safe downcast requires a runtime query of the pointer as to the actual type of the object it addresses
- support for a type-safe downcast mechanism brings with it both space and execution time overhead
    - requires additional space to store type information, usually a pointer to some type information node
    - it requires additional time to determine the runtime type since determination can be done only at runtime
- conflict
    i. heavy use of polymorphism causes a legitimate need for a type-safe downcast mechanism
    ii. use of built-in data types and nonpolymorphic facilities causes means a programmer should not to be penalized with the overhead of a mechanism that is not necessary
- C++ RTTI mechanism provides a type-safe downcast facility but only for those types exhibiting polymorphism
    - types that make use of inheritance and dynamic binding
- done by distinguishing classes that have virtual methods (declared or inherited)
- places address of class-specific RTTI object within the virtual table
    - overhead is only one pointer per class and type information object itself
    - pointer need only be set once and can be set statically by the compiler

A Type-Safe Dynamic Cast

- the dynamic_cast operator determines at runtime the actual type being addressed
- if the downcast is safe (if the base type pointer actually addresses an object of the derived class) the operator returns the appropriately cast pointer
- if the downcast is not safe
    - if the cast fails and the type to which the cast was attempting is a pointer type, it returns a null pointer of that type
    - if the cast fails and new_type is a reference type, it throws an exception that matches a handler of type std::bad_cast.

- NOTE: a downcast can also be performed with static_cast, which avoids the cost of the runtime check, but it's only safe if the program can guarantee (through some other logic) that the downcast is actually safe
- type_info is the name of the class defined by the Standard to hold the required runtime type information

References Are Not Pointers

- the dynamic_cast of a class pointer has two possible paths of execution
  - a return of an actual address means the dynamic type of the object is confirmed and type-dependent actions may proceed.
  - a return of 0, the universal address of no object, means alternative logic can be applied to an object of uncertain dynamic type
- dynamic_cast when a applied to a reference works differently
  - if the reference is actually referring to the appropriate derived class or an object of a class subsequently derived from that class, the downcast is performed and the program may proceed
  - if the reference is not actually a kind of the derived class, then because returning 0 is not viable, a bad_cast exception is thrown

Typeid Operator

- can achieve pointer-like two path execution with references by checking with typeid (Stroustrup suggests to just handle exception over this)
- typeid returns const ref to a type_info object with overloaded operator ==, before method to check ordering in inheritance, hash_code method, and name for mangled name
- also has related type_index wrapper object
- minimum information required for an implementation if actual name class, ordering of type_info objects, some type descriptor representing the explicit class and any subtypes of the class
- previously said that RTTI is available only for polymorphic classes
  - in practice, type_info objects are also generated for both built-in and nonpolymorphic user-defined types
  - necessary for EH support
  - type_info is retrieved statically for built-in and nonpolymorphic types

Efficient, Inflexible?

- C++ generally has efficient runtime support of the object paradigm
- the object model is inflexible in certain areas
  - dynamically shared libraries
  - shared memory

    - distributed objects

Dynamic Shared Libraries

- drop in replacement of shared libraries can cause breaking changes if object layouts change
    - size of class and offset location of each of its direct and inherited members is fixed at compile time (except for virtually inherited members)
- (there are many more modern methods for handling a variety of dynamic loading approaches)

Shared Memory

- shared memory relocation of dynamic libraries (by runtime linker) can result in a number of issues with addressing (seg faults, etc)

NOTE: this book was published in 2003 and focuses on the classic C++ object model

- many of these notes may be outdated