How C++ Works

Overview

- Constructors and destructors
- Virtual functions
- Single inheritance
- Multiple inheritance
- RTTI
- Templates
- Exceptions
- Operator overloading

Motivation

- C++ is a complicated language
  - Some central features have bizarre implementation quirks
- A clear understanding of how a compiler implements language constructs is important when designing large C++ systems
- We learned a lot about these topics in our line of work

Assumptions

- Familiarity with the high-level behaviour of C++ constructs
  - Virtual functions
  - Inheritance
  - RTTI
  - Exceptions
  - Templates
  - Etc.

Constructors

- Constructors are called when an object:
  - ... enters scope
  - ... is created with operator new
- What about global variables?
  - Constructed before main() by C++ startup code
  - Can’t assume the order of creation
  - Be careful with global constructors
    - The system might not be fully “set up” at this time

Object Construction

- When an object is instantiated:
  - operator new is called by the compiler to allocate memory for the class
  - or it is allocated on the stack
- For each class in the inheritance hierarchy, starting with the base class:
  - the vtable pointers are initialized
  - the initialization list is processed
    - in the order objects are declared!
  - default constructor calls are added as needed by compiler
  - the constructor for that class is called

Object construction pitfalls

- Pay attention to member order
- Calling virtual functions in constructors is dangerous:

```
class A
{
    A()
    {
        foo(); // calls A::foo(), even if object is a B
    }
    virtual foo();
};

class B : public A
{
    B();
    virtual foo();
};
```

Destructors

- Destructors are called when an object:
  - ... leaves scope
  - ... is destroyed with operator delete
- Global objects are destroyed after main()
  - Same pitfalls as global construction
- Operator delete[] informs the compiler to call the destructor for each object in an array
  - The compiler has no way of knowing if a pointer refers to an array of objects, or just a single object. You have to tell it. Memory leaks otherwise.

Object destruction

- Similar to constructors but backwards
  - Only works in hierarchy if the destructor is virtual
  - Otherwise:

```
class A
{
    ~A();
};

class B : public A
{
   ~B() { ImportantStuff(); }
};

A* foo = new B;
delete foo;   // B's destructor isn't called!
```

Virtual functions

- What is a vtable?
  - Array of function pointers representing a class's virtual members
  - Stored in the application’s static data
  - Used for virtual function dispatch
- Virtual functions must be “looked up” in vtable before calling
  - a few cycles slower than a regular function call
  - can incur a cache miss
  - can incur a branch target mispredict
  - can’t be inlined

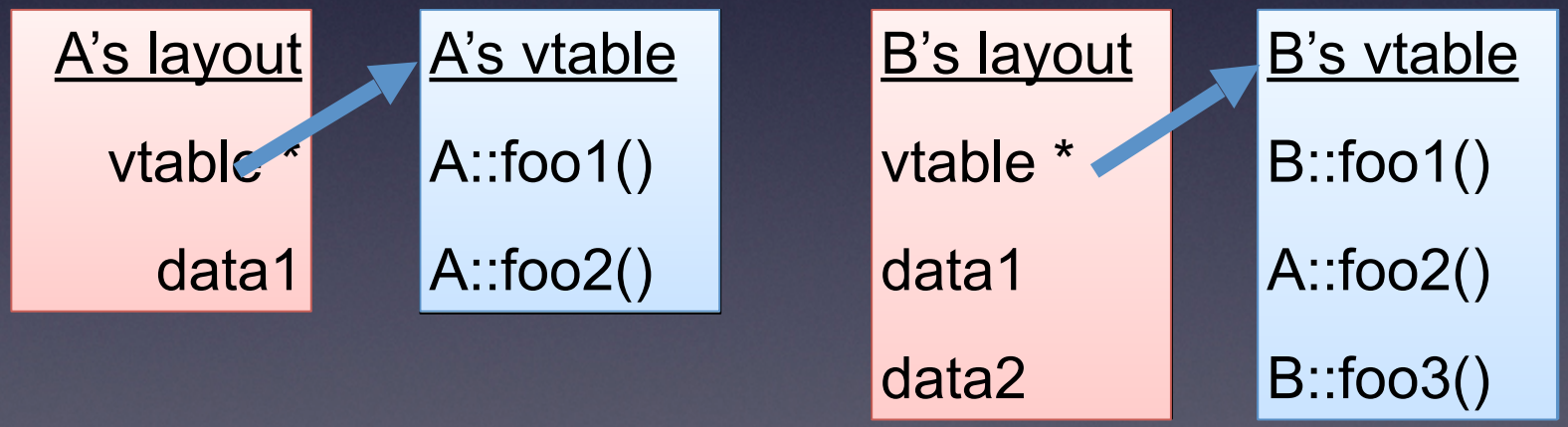
Single inheritance

- Implemented by concatenating layout of base classes together
  - except for the base class vtable pointers
  - only one vtable pointer regardless of inheritance depth
- Cost of single inheritance:
  - one global vtable per class
  - one vtable pointer per object
  - one vtable lookup per virtual call

Single inheritance example

```
class A
{
   virtual foo1();
   virtual foo2();
   int data1;
};
```

```
class B : public A
{
   virtual foo1();
   virtual foo3();
   int data2;
};
```

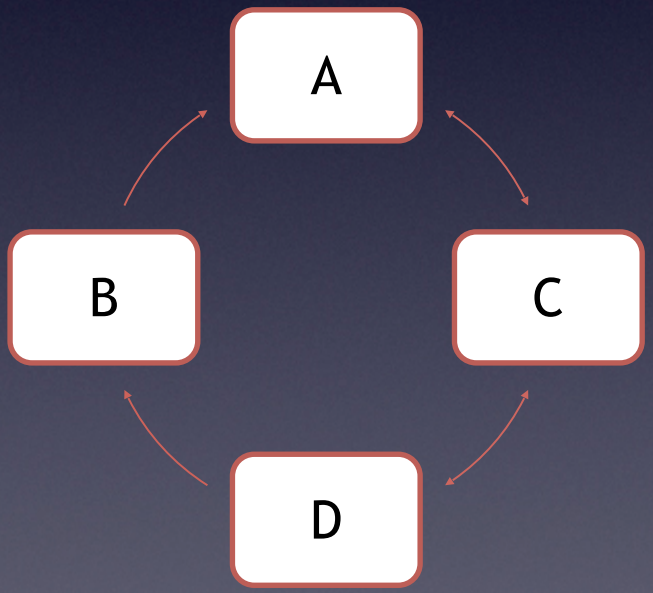


Multiple inheritance

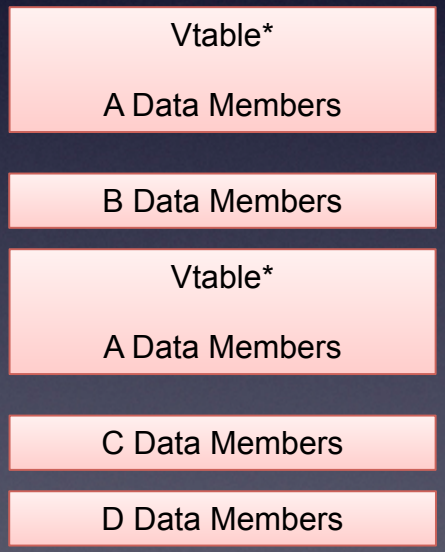
- Implemented by concatenating layout of base classes together
  - Including vtable pointers
  - If two functions in base classes share signatures, compiler can't always disambiguate
  - Pointers to base classes of the same object are not always the same
- Cost of multiple inheritance:
  - one vtable per class
  - one vtable pointer per parent class per object
  - one virtual base class pointer per use of virtual base class
  - a virtual base class adds an extra level of indirection
    - affects virtual and non-virtual calls
  - normal virtual function calls are the same as single inheritance

Regular multiple inheritance

```
class A { … };
class B : public A { … };
class C : public A { … };
class D : public B, public C { … };
```

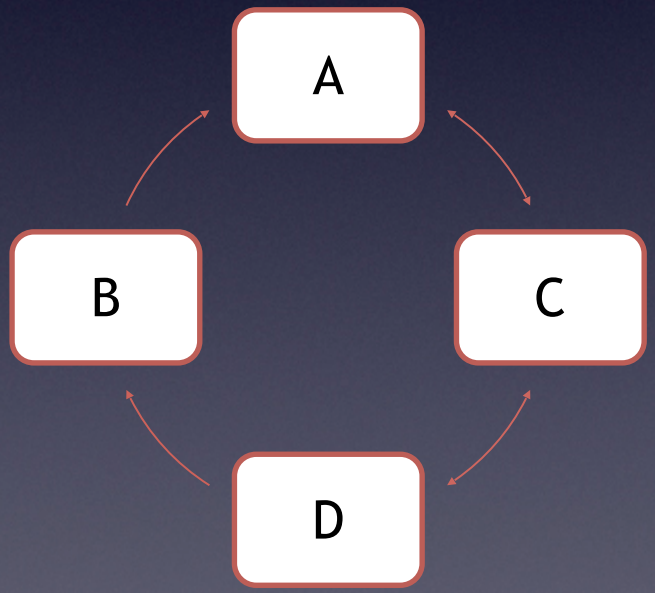


D's footprint:

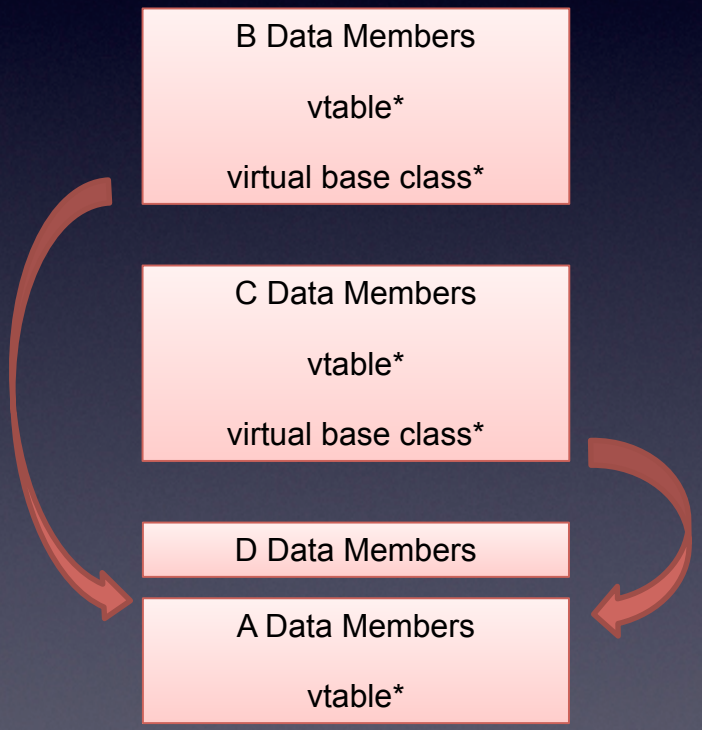


Virtual multiple inheritance

```
class A { … };
class B : virtual public A { … };
class C : virtual public A { … };
class D : public B, public C { … };
```



D's footprint:



(Note how I've used a different possible class layout here.
Class layouts are compiler-dependent, **not** prescribed by the language itself.)

Run Time Type Information (RTTI)

- RTTI relates to two C++ operators:
  - dynamic_cast<>
  - typeid()
- How does RTTI work?
  - Compiler inserts an extra function into a class's vtable
  - Memory hit is per class, not per instance
  - Only pay the speed hit when you use RTTI operators
    - Maximum single inheritance cost is the same as a virtual function times depth of inheritance hierarchy for that class
    - Multiple inheritance is slower

RTTI implementation

User Code:

```
class A
{
   virtual ~A();
};

class B : public A
{
};

A* foo = SomeFoo();
B* bar = dynamic_cast<B*>(foo);
```

Compiler generated casting function:

```
void* cast(void* obj, type dest)
{
  return mytype == dest ? obj : 0;
}

void* siCast(void* obj, type dest)
{
   if (mytype == dest)
      return obj;
   else
      return base->cast(obj, dest);
}
```

dynamic_cast<> in multiple inheritance

```
void* __class_type_info::
dcast (const type_info& desired, int is_public, void
*objptr,
       const type_info *sub, void *subptr) const
{
  if (*this == desired)
    return objptr;

  void *match_found = 0;
  for (size_t i = 0; i < n_bases; i++)
    {
      if (is_public && base_list[i].access !=
PUBLIC)
        continue;

      void *p = (char *)objptr +
base_list[i].offset;
      if (base_list[i].is_virtual)
        p = *(void **)p;
      p = base_list[i].base->dcast (desired,
is_public, p, sub, subptr);
      if (p)
        {
          if (match_found == 0)
            match_found = p;
          else if (match_found != p)
            {
              if (sub)
                {
                  // Perhaps we're downcasting from
*sub to desired; see if
                  // subptr is a subobject of
exactly one of {match_found,p}.
                  const __user_type_info &d =
                    static_cast <const
__user_type_info &> (desired);

                  void *os = d.dcast (*sub, 1,
match_found);
                  void *ns = d.dcast (*sub, 1, p);

                  if (os == ns)
                    /* ambiguous -- subptr is a
virtual base */;
                  else if (os == subptr)
                    continue;
                  else if (ns == subptr)
                    {
                      match_found = p;
                      continue;
                    }
                }

              // base found at two different
pointers,
              // conversion is not unique
              return 0;
            }
        }
    }

  return match_found;
}
```

Operator overloading

- Most operators in C++ can be overloaded
  - Can't overload: . ?: :: .* sizeof typeid
  - Shouldn’t overload: , && ||
  - Principle of Least Astonishment

- Operators have function signatures of form “operator <symbol>”, example :
  - Foo& operator + (Foo& a, Foo& b);

- If you ever implement your own operators (I.e. write a math library) need to understand C++11 move semantics to avoid redundant copies

Templates

- Macros on steroids
  - Evaluated in a similar fashion to macros, but are type-safe.
  - Can be templatized on types or values
- Code is generated at each template instantiation
  - Everything must be defined inline
  - Templatized class is parsed by compiler and held
  - When a template class is instantiated, compiler inserts actual classes into parse tree to generate code.

These two examples will generate identical code

```
template <class T> class foo
{
    T Func(void)
    { return bar; }
    T bar;
};

foo<int> i;
foo<char> c;
```

```
class fooInt
{
    int Func(void)
    { return bar; }
    int bar;
};


class fooChar
{
    char Func(void)
    { return bar; }
    char bar;
}
```

Templated Code Bloat

- Not one, but two ways to bloat code!
  - Because templates must be defined inline, code may be inlined unintentionally
  - Each instantiation of new templatized type causes the creation of a large amount of code
- Combating code bloat
  - Separate non-type-dependent functions into non-templatized functions or base class
  - Use templates as type-safe wrappers for unsafe classes
- When templates are not inlined, duplicate symbols are generated which the linker must strip out

Templates (cont'd)

- Templates can interact fine with derivation hierarchy and virtual functions
  - But the specializations are not naturally related in any way
- Templates cannot be exported from libraries because no code exists
  - Instantiated or fully-specialized template classes can

Exceptions

- Provide a way to handle error conditions without constant checking of return values
- Problems to be solved by exception handling implementation:
  - Finding correct exception handler
  - Transferring control to exception handler
  - Destroying objects on the stack

Finding the correct exception handler

- Table of handlers is kept
  - one per try/catch block
  - also stores reference to the next (parent) try/catch frame
- Global pointer to current try/catch frame is stored

Passing control to exception handler

- At the beginning of each try/catch block the current stack state is stored (setjmp)
- If an exception occurs the runtime searches the try/catch frame for an appropriate handler, resets the stack frame and passes control (longjmp)

Destroying objects on the stack (x86)

- For each function an unwinding table of all stack allocated objects is kept
  - Current initialization state is kept for each object
  - When an exception occurs, current unwind table and all above it but below the handler's frame have all valid objects destroyed
- The table is created even for functions with no try/catch or throw statements
  - Extra work per stack allocation/deallocation
  - Extra work at start and end of a function

Exceptions Example (X86)

C++ Code

```
void Test(void)
{
Foo a;
Foo b;
}
```

No Exceptions

```
?Test@@YAXXZ PROC NEAR
push    ebp
mov     ebp, esp
sub     esp, 72
push    ebx
push    esi
push    edi
lea     ecx, DWORD PTR _f$[ebp]
call    ??0Foo@@QAE@XZ
lea     ecx, DWORD PTR _g$[ebp]
call    ??0Foo@@QAE@XZ
lea     ecx, DWORD PTR _g$[ebp]
call    ??1Foo@@QAE@XZ
lea     ecx, DWORD PTR _f$[ebp]
call    ??1Foo@@QAE@XZ
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     0
?Test@@YAXXZ ENDP
```

With Exceptions

```
?Test@@YAXXZ PROC NEAR
push    ebp
mov     ebp, esp
push    -1
push    __ehhandler$?Test@@YAXXZ
mov     eax, DWORD PTR fs:__except_list
push    eax
mov     DWORD PTR fs:__except_list, esp
sub     esp, 72
push    ebx
push    esi
push    edi
lea     ecx, DWORD PTR _f$[ebp]
call    ??0Foo@@QAE@XZ
mov     DWORD PTR __$EHRec$[ebp+8], 0
lea     ecx, DWORD PTR _g$[ebp]
call    ??0Foo@@QAE@XZ
lea     ecx, DWORD PTR _g$[ebp]
call    ??1Foo@@QAE@XZ
mov     DWORD PTR __$EHRec$[ebp+8], -1
lea     ecx, DWORD PTR _f$[ebp]
call    ??1Foo@@QAE@XZ
mov     ecx, DWORD PTR __$EHRec$[ebp]
mov     DWORD PTR fs:__except_list, ecx
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     0
_TEXT   ENDS
;       COMDAT text$x
text$x  SEGMENT
__unwindfunclet$?Test@@YAXXZ$0:
lea     ecx, DWORD PTR _f$[ebp]
call    ??1Foo@@QAE@XZ
ret     0
__ehhandler$?Test@@YAXXZ:
mov     eax, OFFSET FLAT:__ehfuncinfo$?
Test@@YAXXZ
jmp     ___CxxFrameHandler
text$x  ENDS
?Test@@YAXXZ ENDP
```

Exceptions (x86)

- This behaviour means that exception handling costs even when you don’t actually use it
  - Most compilers have a flag to turn on/off stack unwinding for exception handling
  - This makes exception handling basically useless though
- Exceptions are one of the few C++ constructs that have fully deserved their bad reputation

- But...

Destroying Objects on the Stack (x64)

- For each function a static unwinding table of stack allocated objects is generated by the compiler
  - Current initialization state for each object is calculated based on the program counter
  - When an exception occurs current unwind table and all above it but below the handler's frame have all valid objects destroyed
- The table is created even for functions with no try/catch or throw statements
  - But it's done statically by the compiler with no runtime overhead
- It does mean that throwing an exception is considerably more expensive, but they should be rare, and if you don't use them, there's no cost

Points to take with you

- Most of this is covered in “Effective C++” and “More Effective C++” by Scott Meyers
- All of it is covered in the GCC source code
  - Harder to read though (but comments are hilarious)
- Stroustrup and Ellis “Annotated C++ Reference Manual” describes in detail how C++ concepts can be implemented