Class Layout

Steve Dewhurst
Semantics Consulting, Inc.
www.stevedewhurst.com

 1

1

C++ Classes

- Only non-static data declarations in a class definition add anything to the size of class objects.

```
class HR {
public:
    ~~~
    void terminator(Salary *, Hourly *, Temp *);
private:
    enum { max = 16383 };
    static size_t num_dispatched;
    Employee *emps[max];
    size_t size;
};
```

 7

7

sizeof(Class Object)

- sizeof applied to the HR class yields the same result as sizeof applied to this structure:

```
enum { HR_max = 16383 };

extern size_t HR_num_dispatched;

struct HR_type {
    Employee *emps[HR_max];
    size_t size;
};
```

- You can declare an enumeration nested inside a C structure.
- However, C doesn't consider nested enumeration types and constants to be members of the structure. They have the same scope as the structure name.
- Storage for static member data is allocated with other static data (in .data).

 8

8

Class Members That Don't Affect Layout

- Static data members do not affect class layout.
- They have static lifetime or thread storage duration.
- Therefore, we won't be discussing static data members.
- Type member (included nested classes) do not affect class layout.
- Non-virtual member functions have no effect on class layout.
- However, let's have a few words about them…

 9

9

Non-Virtual Member Functions

- Non-virtual member functions do not occupy any space in the object and don't affect data member layout.

```
class HR {
public:
    void terminate(Salary *);
    void terminate(Hourly *);
    void terminate(Temp *);
    void terminator(Salary *s, Hourly *h, Temp *t);
    ~~~
};

void HR::terminator(Salary *s, Hourly *h, Temp *t) {
    terminate(s);
    terminate(h);
    terminate(t);
}
```

 10

10

Non-Member Equivalent

- The calling sequence for an n-argument non-virtual member function may be the same as that for an (n+1)-argument free function.
- Some platforms use a special calling sequence for the this pointer that may be marginally more efficient.

```
void terminate(HR *, Salary *);
void terminate(HR *, Hourly *);
void terminate(HR *, Temp *);
void terminator(HR *, Salary *s, Hourly *h, Temp *t);

void terminator(HR *hr, Salary *s, Hourly *h, Temp *t) {
    terminate(hr, s);
    terminate(hr, h);
    terminate(hr, t);
}
```

 11

11

Identical Results, Up To Renaming

- Member Function:

```
_ZN2HR10terminatorEP6SalaryP6HourlyP4Temp:
        PUSH    {R4-R6,LR}
        MOV     R4,R0
        MOV     R5,R2
        MOV     R6,R3
        BL      _ZN2HR9terminateEP6Salary
        MOV     R1,R5
        MOV     R0,R4
        BL      _ZN2HR9terminateEP6Hourly
        MOV     R1,R6
        MOV     R0,R4
        POP     {R4-R6,LR}
        B       _ZN2HR9terminateEP4Temp
```

- Non-Member Function:

```
_Z10terminatorP2HRP6SalaryP6HourlyP4Temp:
        PUSH    {R4-R6,LR}
        MOV     R4,R0
        MOV     R5,R2
        MOV     R6,R3
        BL      _Z9terminateP2HRP6Salary
        MOV     R1,R5
        MOV     R0,R4
        BL      _Z9terminateP2HRP6Hourly
        MOV     R1,R6
        MOV     R0,R4
        POP     {R4-R6,LR}
        B       _Z9terminateP2HRP4Temp
```

 12

12

Non-Static Data Member Layout

- What do we know about the order of non-static data members?

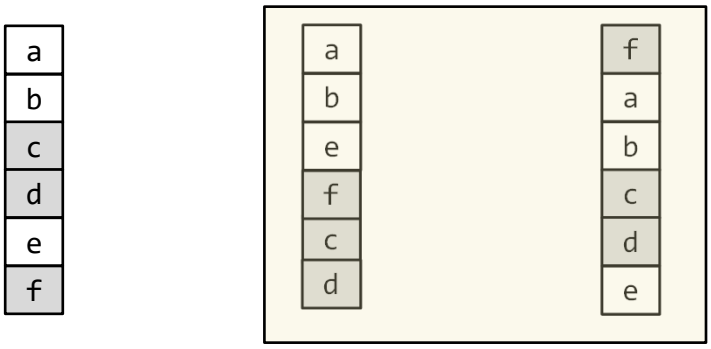
```
class C {
public:      // access specifier
    int a;
    int b;
private:     // access specifier
    int c;
    int d;
public:      // access specifier
    int e;
private:     // access specifier
    int f;
};
```

 13

13

Non-Static Data Member Layout

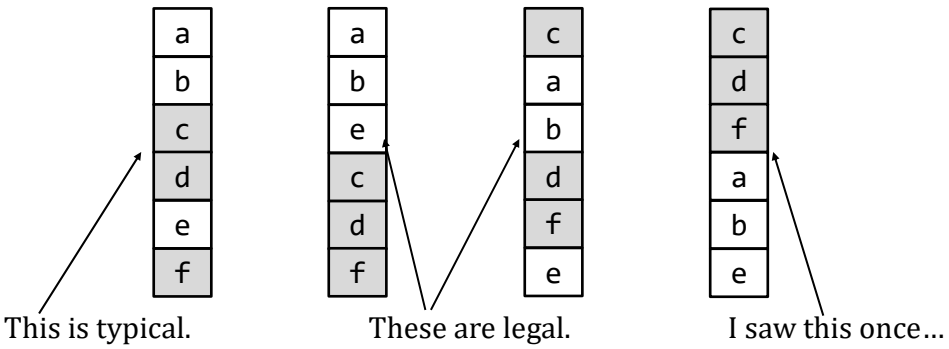
- In traditional C++, the only ordering guarantee was that the members *between access specifiers* had to be laid out in the same relative order.
- Here are some traditionally-valid member orderings:



- The rightmost two orderings are not valid in modern C++.
- In practice, compilers did not (or very rarely) reorder member layout.

Non-Static Data Member Layout

- In modern C++, that ordering guarantee was modified to state that members *with the same access* had to be laid out in the same relative order.
- "Nonstatic data members of a (non-union) class with the same access control…are allocated so that later members have higher addresses within a class object."



Base Class Layout

- "The order in which the base class subobjects are allocated in the most derived object…is unspecified."
- It is *typical* that storage for a base class subobject precedes storage for derived class data members.
- It is *typical* that multiple base class subobjects are laid out in the order they appear on the base class list.
- ✓ *However, a compiler may elect to optimize storage use by permuting the base class subobject order.*

 16

16

Empty Class Members Occupy Space…

- An empty class has no non-static data members, no virtual functions, and no virtual base classes.
- We often call these types/objects/closures "stateless."
- However, even an empty class must occupy some space.

```
class Empty {};  // sizeof(Empty) is probably 1
Empty e;         // sizeof(e) is probably 1

class Empties {  // sizeof(Empties) is probably 3
    Empty e1_;   // offset 0
    Empty e2_;   // offset 1
    Empty e3_;   // offset 2
};
```

 17

17

…Until C++20!

- The `no_unique_address` attribute *allows* an empty non-static data member to share space with another subobject (base class or member) of a different type.

```
class Empty {};      // sizeof(Empty) is probably 1
Empty e;             // sizeof(e) is probably 1

class Empties {      // sizeof(Empties) is now probably 1
    [[no_unique_address]] Empty e1_;
    [[no_unique_address]] Empty2 e2_;
    [[no_unique_address]] Empty3 e3_;
};
```

 18

18

STL

- This is a real boon for implementers of STL containers! For example:

```
template<typename Key, typename Value, // copied from C++20 working draft...
         typename Hash, typename Pred, typename Allocator>
class hash_map {
    [[no_unique_address]] Hash hasher;
    [[no_unique_address]] Pred pred;
    [[no_unique_address]] Allocator alloc;
    Bucket *buckets;
    ~~~
};
```

- `Hash`, `Pred`, and `Allocator` are often stateless.
- In the past, we'd have to do a lot of metaprogramming to make a compile-time decision to use an alternative implementation to *potentially* get this optimization through the empty base optimization (EBO).

 19

19

Empty Base Classes

- The "Empty Base Class Optimization," also known as the EBCO or EBO (if you prefer TLA's) is a common compiler optimization.
- In component design, base classes are often simply collections of typedefs, static members, enumerators, and other class members that do not occupy storage inside the class object.
- The EBO allows the compiler to optimize away storage for such a base class that it could not optimize away for a standalone object or a data member.

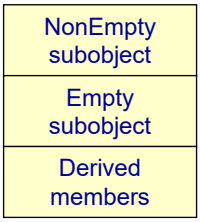
 20

20

Using The EBO/EBCO

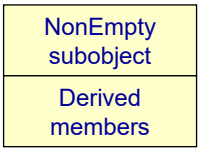
- With respect to multiple inheritance, it often makes sense to rearrange the order of base classes so that empty base classes appear first on the base class list.

```
class Derived : public NonEmpty, public Empty {
    ~~~
};
```



- But the compiler is not required to optimize away storage for an empty base class.
- Sometimes it makes sense to help the compiler to perform the optimization by permuting the base classes.

```
class Derived : public Empty, public NonEmpty {
    ~~~
};
```



 21

21

Aggressive Compilers

- Compilers are allowed to (and some compilers do) perform aggressive EBO optimization by permuting the layout order of base classes.

```
class Derived : public NonEmpty1, public NonEmpty2, public Empty {
    ~~~
};
```

- In this case, some compilers will locate the `Empty` base class subobject at offset `0`, before the storage for the `NonEmpty2` subobject.

 22

22

Aggressive Compilers

- Some compilers will optimize storage for only the first of a sequence of empty base classes.

```
class Derived : public Empty1, public Empty2, public Empty3 { // size == 3
    char member_;
};
```

- Others will optimize all empty base class subobjects.

```
class Derived : public Empty1, public Empty2, public Empty3 { // size == 1
    char member_;
};
```

- It is difficult the predict—in a portable way—the layout of a complex C++ class.
- Wouldn't it be nice to have some standard layout guarantees? Just wait…

 23

23

Lexical vs. Physical Ordering

- Note that language constructs that enforce an ordering on members are defined lexically, not physically.
- For example:
  - A member initialization list initializes in the order of declaration of the members.
  - Base class subobjects are initialized and destroyed based on their lexical position on the base class list, not the order in which they're allocated in memory.
  - A defaulted `operator <=>` compares members in their declared order.

Lexical Comparison

- The implementation compares the data members in the lexical order `x`, `y`, and `z` even if `z` is physically placed at offset `0`.

```
class Flatland {
public:
    auto operator <=>(const Flatland &) const = default;
    int x;
    int y;
private:
    int z;
};
```


What Could Possibly Go Wrong?

- Just be careful about making layout assumptions.

```
class Flatland {
public:
    auto operator <=>(const Flatland &rhs) const
        { return memcmp(this, &rhs, sizeof(Flatland)); }
    int x;
    int y;
private:
    int z;
};
```

✓ *This is the kind of code that will work for decades and then fail unexpectedly.*

 26

26

Act Like a Person

- It's probably better to be normal and put all the data at the same access level.

✓ *And if you want to stay on Scott Meyers's good side, make the data private.*

```
class Flatland {
public:
    auto operator <=>(const Flatland& rhs) const
        { return memcmp(this, &rhs, sizeof(Flatland)); }
    auto x() const { return x_; }
    auto y() const { return y_; }
private:
    int x_;
    int y_;
    int z_;
};
```

✓ *Unfortunately, this version might not work either…for a different reason.*

 27

27

Ensuring Proper Alignment

- Consider a memory-mapped UART that consists of six special registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

 28

28

Ensuring Proper Alignment

- How can we ensure that our representation of this device matches reality?

```
using special_register = uint32_t volatile;
~~~
class UART {
    ~~~
private:
    special_register ULCON;   // line control
    special_register UCON;    // control
    special_register USTAT;   // status
    special_register UTXBUF;  // transmit
    special_register URXBUF;  // receive
    special_register UBRDIV;  // baud rate divisor
};
```

 29

29

Standard-Layout Types

- How can you be sure that each `UART` member has the proper offset within the structure?
- C++ provides layout guarantees only for standard-layout types…
- A ***standard-layout type*** is essentially a C type:
  - a scalar type (arithmetic, enumeration, or pointer type)
  - an array with elements of standard-layout type
  - a standard-layout class (possibly declared as a struct or union)…

 30

30

Standard-Layout Classes

- A ***standard-layout class*** can have:
  - no virtual functions or virtual base classes
  - zero or more base classes of standard-layout class types
  - no two base classes of the same type
  - zero or more non-static data members of standard-layout types, such that:
    - all members have the same access control, and
    - all members are declared in the most derived class or in the same base class.
- A standard-layout class can have non-virtual and static member functions, static data members, and nested types.
- `UART` is a standard-layout class (assuming it has no virtual functions).

✓ *Standard layout classes make reasoning about their layout tractable.*

 31

31

Aside: POD Types

- Standard-layout types are defined in C++11.
- The 2003 Standard didn't mention standard-layout types — it describes layout guarantees in terms of *POD* types.
  - POD stands for "Plain Old Data." I kid you not. (There's also a POF.)
- A POD is a standard-layout class that also lacks:
  - private or protected non-static data members
  - user-declared constructors
  - base classes
- In practice, POD classes are too restrictive for many uses, like modeling devices (the `is_pod` type trait is deprecated in C++20).
- Programmers have been using standard-layout types, even before the concept of standard-layout existed.
- Modern C++ simply legitimizes what programmers have been doing already.

Layout Guarantees and Padding

- Regarding the layout of standard-layout classes, C++ guarantees only that:
  - The first non-static data member is at offset zero.
    - This means that it's reasonable to `reinterpret_cast` a pointer to a standard-layout class object to a pointer to its first non-static data member.
  - Each subsequent non-static data member has an offset greater than the offset of the non-static data member declared before it.
  - The storage for objects of the class are in contiguous memory.
- Thus, you can be sure that `ULCON` will be at offset zero in the `UART` structure.
- However, you can't be quite as sure about any other data member.

✓ *Any class, even a standard-layout class, may have padding bytes (also called "slack bytes") after any non-static data member.*

Layout Guarantees

- Most compilers offer non-standard extensions to help you control structure layout.
- For example, some compilers offer pragmas, as in:

```
#pragma pack(push, 4)
class UART {
    ~~~
};
#pragma pack(pop)
```

- Some dialects of GNU C++ support type attributes such as:

```
class UART __attribute__(packed) {
    ~~~
};
```

 34

34

Using Static Assertions to Verify Layout

- Misaligned members in device classes such as UART almost always lead to run-time failures.
- The best defense against misaligned members is to catch them at compile time using static assertions, as in:

```
class UART {
    ~~~
private:
    special_register ULCON;
    special_register UCON;
    special_register USTAT;
    ~~~
};
static_assert(offsetof(UART, UCON) == 4,
              "UCON member must be at offset 4 within UART");
~~~
```

 35

35

Using Static Assertions to Verify Layout

- By the way, `offsetof` is a standard macro from *<cstddef>*.
- It's guaranteed to work only for standard-layout types.
- For other types, it's a "conditionally-supported" feature.
- Calling `offsetof(C, M)` returns the offset in bytes of non-static data member `M` from the beginning of class type `C`.
- In this assertion:
  - If the expression `offsetof(UART, UCON) == 4` is true, the assertion does nothing.
  - If it's false, the compiler displays a message containing the text

```
UCON member must be at offset 4 within UART
```

 36

36

It's OK to be Nervous

- Static assertions don't cost anything at runtime, so be safe!

```
static_assert(sizeof(special_register) == 4,
    "special_register must be 4 bytes in size");

static_assert(alignof(special_register) == 4,
    "special_register must be word-aligned");

static_assert(is_unsigned_v<special_register>,
    "special_register must be unsigned");

static_assert(is_standard_layout_v<UART>,
    "UART must be standard layout");

static_assert(offsetof(UART, UCON) == 4,
    "UCON member must be at offset 4 within UART");
```

 37

37

Detecting Alignment Problems

- Here's another memory-mapped device that's even more frightening than our UART:

```
struct timer {
    uint8_t MODE;    // offset 0
    uint32_t DATA;   // offset 4, or people will die!
    uint32_t COUNT;  // offset 8, or toast will burn!
};
```

- We can use static assertions to detect problems at compile time.

```
static_assert(offsetof(timer, DATA) == 4, "DATA must be at offset 4");
```

 38

38

Enforcing Alignment

- Alternatively, we can tell the compiler what the alignment of an object or member must be:

```
struct timer {
    uint8_t MODE;
    alignas(4) uint32_t DATA;
    uint32_t COUNT;
};
```

- ...and it never hurts to check!

```
static_assert(offsetof(timer, DATA) == 4, "DATA must be at offset 4");
```

- Доверяй, но проверяй.

 39

39

It's a Good Idea to Check

- Let's go overboard! (Though you don't have to specify alignment to this degree.)

```
struct timer {
    alignas(4) uint8_t MODE;    // offset 0
    alignas(4) uint32_t DATA;   // offset 4
    alignas(4) uint32_t COUNT;  // offset 8
};
static_assert(offsetof(timer, DATA) == 4, "DATA must be at offset 4");
```

 40

40

Really, It's a Good Idea to Check

- It's mid-May, so the following change is made:

```
#pragma pack(push, 1)
                                // compiler A | compiler B
struct timer {                  // ---------- | ----------
    alignas(4) uint8_t MODE;    //  offset 0  |  offset 0
    alignas(4) uint32_t DATA;   //  offset 4  |  offset 1
    alignas(4) uint32_t COUNT;  //  offset 8  |  offset 5
};
static_assert(offsetof(timer, DATA) == 4, "DATA must be at offset 4");

#pragma pack(pop)
```

✓ *Unless you're 1) certain your layout is guaranteed on all platforms of interest, 2) that future maintainers won't introduce bugs, and 3) that future versions of your compilers will behave the same way,* ***use static assertions****.*

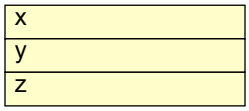
 41

41

What's a Few Slack Bytes Among Friends?

- Here's a simple class:

```
class NarrowLand {
    unsigned long long x;  // offset 0
    unsigned long long y;  // offset 8
    unsigned long long z;  // offset 16
    friend bool operator ==(NarrowLand const &lhs, NarrowLand const &rhs);
};
```



- Let's optimize equality comparison:

```
bool operator ==(NarrowLand const &lhs, NarrowLand const &rhs) {
    return !memcmp(&lhs, &rhs, sizeof(NarrowLand));
}
```

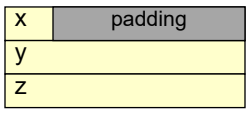
 42

42

What's a Few Slack Bytes Among Friends?

- But we don't need that much precision for the x-axis in NarrowLand:

```
class NarrowLand {
    unsigned char x;       // offset 0
    unsigned long long y;  // offset 8 (still!)
    unsigned long long z;  // offset 16
    friend bool operator ==(NarrowLand const &lhs, NarrowLand const &rhs);
};
```



- Unfortunately, we may now get some surprising results from our operator ==.

 43

43

The Appearance of Normality…

- Things may appear to be normal.

```
NarrowLand global { 'x', 12345ULL, 54321ULL };
~~~
bool g() {
    NarrowLand local { 'x', 12345ULL, 54321ULL };
    return local == global;
}
~~~
int main() {

    if (g())
        cout << "normal" << endl; // we're normal
    else
        cout << "???" << endl;
}
```

 44

44

…Suddenly Vanishes!

- But any number of small changes could provoke an unexpected result.

```
NarrowLand global { 'x', 12345ULL, 54321ULL };
~~~
bool g() {
    NarrowLand local { 'x', 12345ULL, 54321ULL };
    return local == global;
}
~~~
int main() {
    f();
    if (g())
        cout << "normal" << endl;
    else
        cout << "???" << endl;    // now we're not normal
}
```

 45

45

It's OK to be Normal

- One fix is to be normal.

```
bool operator ==(NarrowLand const &lhs, NarrowLand const &rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z;
}
```

- If the optimization is both real and necessary, you can ask a few questions at compile time with the help of <type_traits>:

```
bool operator ==(NarrowLand const &lhs, NarrowLand const &rhs) {
   if constexpr (has_unique_object_representations_v<NarrowLand>)
        return !memcmp(&lhs, &rhs, sizeof(NarrowLand));
   else
        return lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z;
}
```

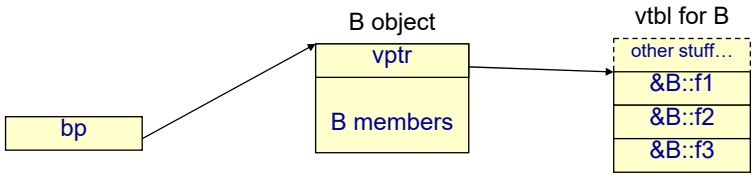

46

An Implementation Of Single Inheritance

- If a class has a virtual function, every object of that type contains a pointer to a shared virtual function table.

```
class B {
public:
    virtual int f1();
    virtual void f2(int);
    virtual int f3(int);
    ~~~
};

B *bp = new B;
```



47

Other Stuff?

- The vtable is a dumping ground for most type-related information about a class.
- The "other stuff" is usually
  - an offset to the start of the complete object,
  - a pointer to the typeinfo for the object, and
  - offsets to virtual base class subobjects in the complete object
- Additionally, we know that a proper polymorphic base class has a public, virtual destructor.
  - We're going to ignore virtual destructors for simplicity.
  - A single destructor will be often represented by two separate entries in the virtual function table; one is a "destroy and delete" destructor, while the other only destroys.

 48

48

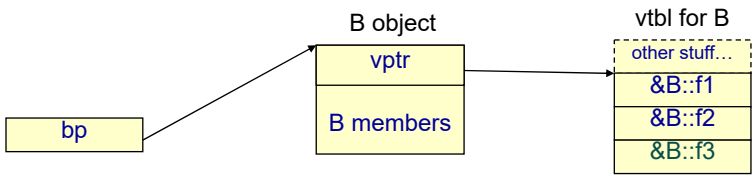
Virtual Calling Sequence

- The virtual calling sequence is indirect through the object's virtual function table. The call

```
bp->f1();
```

- Is translated to something like

```
(*(bp->vptr)[0])(bp)
```



 49

49

Virtual Calling Sequence

- The call

```
bp->f1();
```

- Is translated to something like

```
(*(bp->vptr)[0])()
```

- Or, in other words,

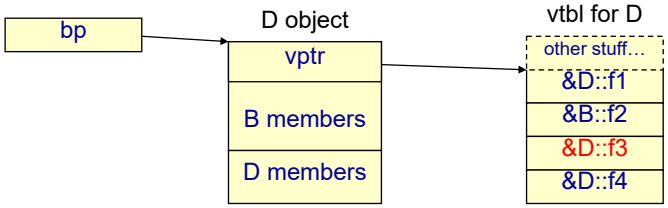
```
MOV     R0,R4           # get bp into "this" register
LDR     R1,[R4, #+0]    # get bp->vptr
LDR     R1,[R1, #+0]    # get vptr[0] (member function address)
MOV     LR,PC           # set return link
BX      R1              # call member function
```


Implementation Of Overriding

- A derived class has its own virtual table.
- Overriding derived class functions replace those of the base class.

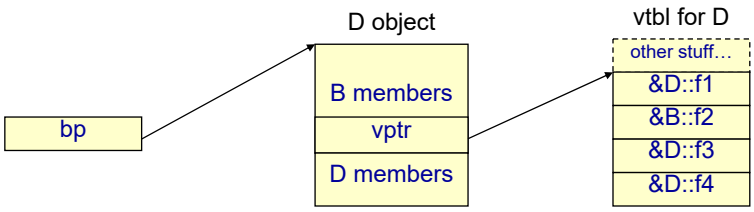
```
class D: public B {
public:
    int f1() override;       // overrides B::f1
    virtual void f4();       // new virtual
    int f3(int) override;    // overrides B::f3
    ~~~
};

B *bp = new D;
bp->f3();
```



Alternative Facts

- Note that while this is a typical implementation, the standard does not specify a particular mechanism.
- Some compilers would locate the virtual pointer somewhere other than at the beginning of an object, typically at the end of a base class subobject.



- It's conceivable that a compiler might not use the vptr/vtbl mechanism at all. (I'm not aware of any such compiler.)

 52

52

Implementation of Multiple Inheritance

- If there's more than one base class, it's usually the case that not all base classes can occupy the same location within a derived class object.

```
class Shape
    { ~~~ };
class Subject
    { ~~~ };
class ObservedShape: public Shape, public Subject
    { ~~~ };
```

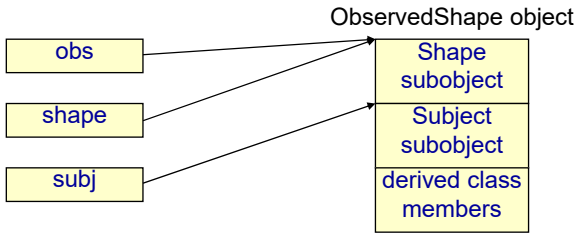
- The derived class object will have more than a single base class subobject.

 53

53

Implementation of Multiple Inheritance

```
ObservedShape *obs = new ObservedShape;
Shape *shape = obs;                    // safe, predefined conversion
Subject *subj = obs;                   // safe, predefined conversion
```

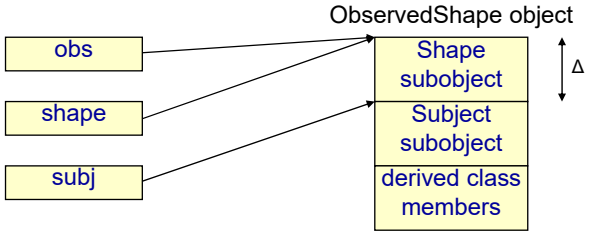


 54

54

Objects With Multiple Addresses

- Suppose we refer to the same `ObservedShape` object through each of its types:



- Some subobjects occur at a non-zero delta from the start of the "complete" object.
- Both addresses are valid addresses of the derived class object.

✓ *An object can have more than one valid address.*

 55

55

Comparing Object Addresses

- When we compare pointers in C++, we are not asking a question about addresses.
- We are asking a question about object identity.
- If the pointers refer to the same object, they compare equal. In most cases the comparison amounts to a simple address comparison:

```
if (obs == shape)
```

 56

56

Comparing Object Addresses

- In other cases, a base class subobject is offset within the derived class object.

```
if (obs == subj)
```

- In that case, we have to perform an address adjustment prior to the comparison.
- The amount of the adjustment, or "delta," is known at compile time.

```
(obs ? obs + delta : 0) == subject
```

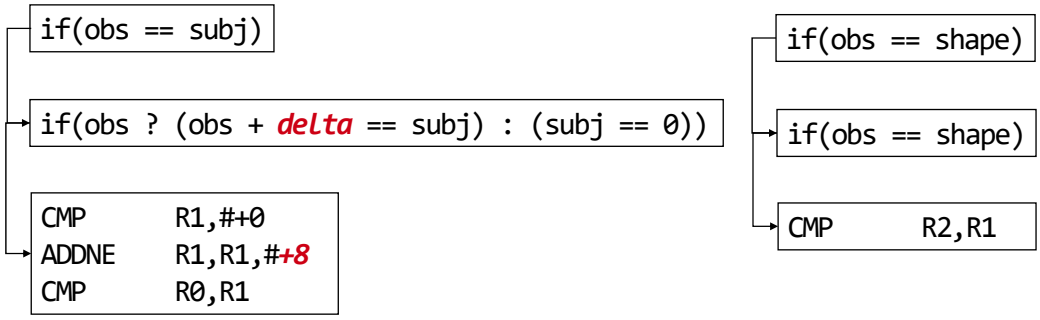
- This delta is usually small enough to be represented as immediate data in an instruction; the address adjustment is fast.

 57

57

Cost of Address Comparisons Under MI

- Most architectures do a good job with this; the delta adjustment is usually an immediate operand.



Requires Delta Arithmetic

Does Not Require Delta Arithmetic

Static Casts Under Multiple Inheritance

- The same address adjustment takes place for a static cast.

```
ObservedShape *obs = new ObservedShape;
Shape *shape = obs;                              // no delta
Subject *subj = obs;                             // delta

obs = static_cast<ObservedShape *>(shape);       // no delta
obs = static_cast<ObservedShape *>(subj);        // delta

obs = (ObservedShape *) shape;                   // no delta
obs = (ObservedShape *) subj;                    // delta
```

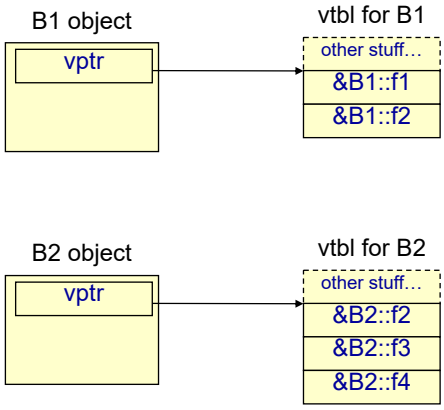
✓ *The `dynamic_cast` operator usually uses a different mechanism and is more expensive.*

Base Class Layouts

- Consider a couple of (base) class implementations.

```
class B1 {
public:
    virtual void f1();
    virtual void f2();
};

class B2 {
public:
    virtual void f2();
    virtual void f3(int);
    virtual void f4();
};
```



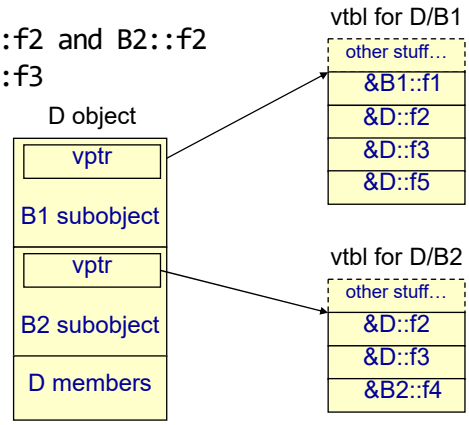
 60

60

Derived Class Layout Under Multiple Inheritance

- Under multiple inheritance, an object may have more than one vptr.

```
class D: public B1, public B2 {
public:
    void f2() override;     // overrides B1::f2 and B2::f2
    void f3(int) override;  // overrides B2::f3
    virtual void f5();
};
```



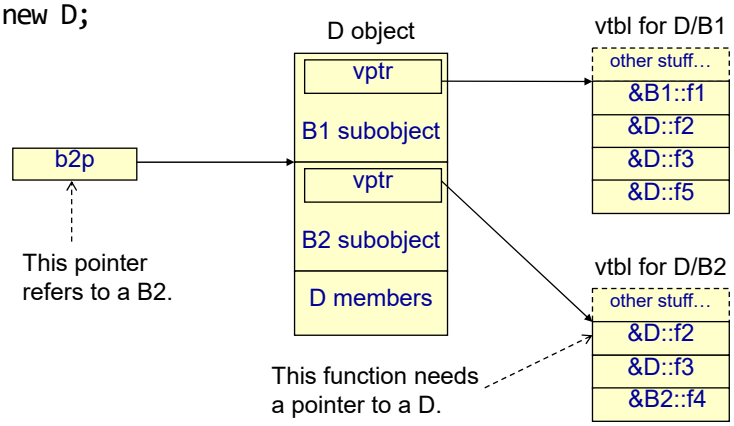
 61

61

Problems with this

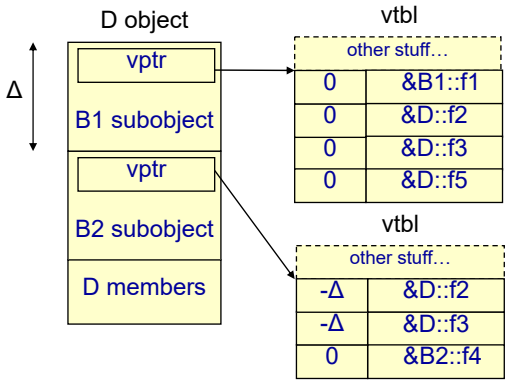
- The existence of multiple addresses introduces a problem: If a pointer to `B2` is used to call an overriding `D` member function, the value of the `this` pointer will be incorrect.

```
B2 *b2p = new D;
B2p->f2();
```



A Traditional Implementation

- One early implementation stored delta adjustments for the `this` pointer in the virtual table itself.
- This approach has the unfortunate property of having to perform the delta arithmetic even when it's not needed.



A Superannuated Calling Sequence

- This approach imposed a space and time overhead on all virtual calls even if multiple inheritance is not used.
  - Every virtual table entry stores a delta, even if it's zero
  - Every virtual call adds a delta to this, even if it's zero
- For example, the call

```
B2 *b2p = get_me_a_B2();
b2p->f3();
```

- Was translated as

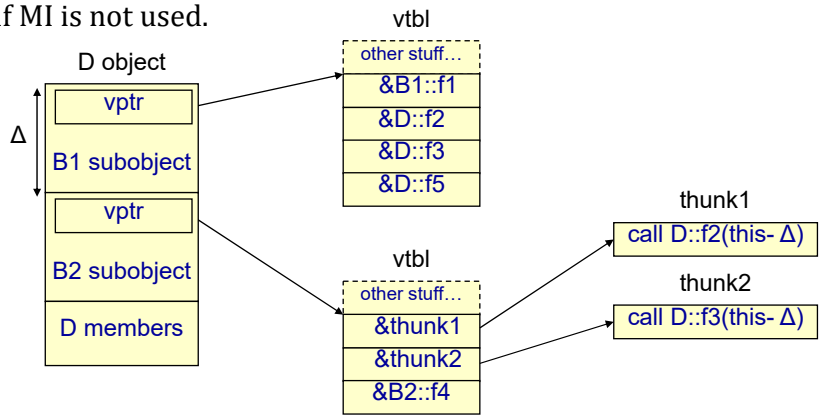
```
(*b2p->vptr[1].funcaddr)(b2p+vptr[1].delta)
```

 64

64

"Thunk" Implementation

- A more common approach is to use a small snippet of code (mis-named a "thunk") to perform the delta adjustment if necessary.
- The calling sequence is the same as it is under single inheritance, and imposes no penalty if MI is not used.

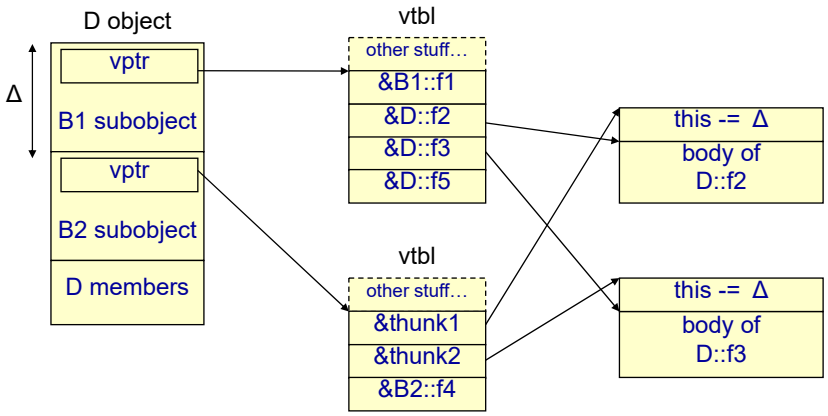


 65

65

Thunk Refinement

- Often the thunk can be implemented as a small section of code contiguous with the overriding function.

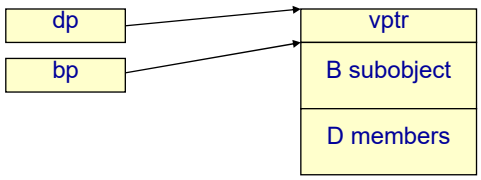


Multiple Addresses Under Single Inheritance

- It's rare but possible to create objects with multiple addresses under single inheritance.

```
class B { /* no virtual functions */ };
class D: public B { /* has a virtual function */ };
D *dp = new D;
B *bp = dp;
```

- On many platforms, the layout of a `D` object will look like this:

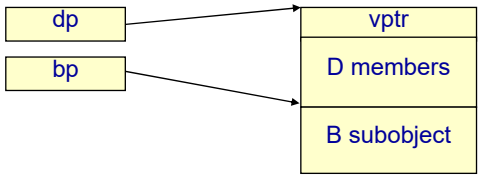


Multiple Addresses Under Single Inheritance

- You can even manage to do this without a virtual function.

```
class B { /* no virtual functions */ };
class D: public virtual B { /* no virtual functions */ };
D *dp = new D;
B *bp = dp;
```

- On many platforms, the layout of a D object will look like this:



 68

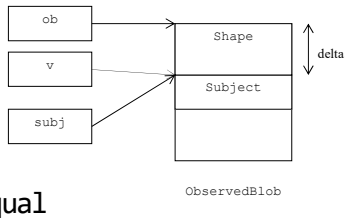
68

void * Intermediaries

- The correct offset adjustment is known statically from the pointer types.
- If an object address is passed through a `void *` intermediary, that required type information is lost.

```
ObservedBlob *ob = new ObservedBlob;
Subject *subj = ob; // safe, predefined conversion
void *v = subj;     // dangerous conversion

if(ob == subj)      // different addresses compare equal
~~~
if(ob == v)         // different addresses don't compare equal!
```



 69

69

Improper Use of void *

- Does this ever happen? Yes.
- Consider a "framework" for setting and getting a `Widget`.

```
typedef void *Widget;     // improper use of typedef...
void setWidget(Widget);
Widget getWidget();
```

- The standard guarantees that this mechanism will work only if the `void *` is cast back to precisely the same type as the pointer that was copied to the `void *`.

```
ObservedBlob *obp = new ObservedBlob;
setWidget(obp);
~~~
Shape *sp = static_cast<ObservedBlob*>(getWidget()); // works...
```

 70

70

Problems with void * and Hierarchies

- Legacy code often assumes that the base class part of a derived class object has the same address as the complete object.

```
ObservedBlob *obp = new ObservedBlob;
setWidget(obp);
~~~
Shape *sp = static_cast<Shape *>(getWidget()); // will often work...
```

- While this would almost always "work," it was never guaranteed to, and is flagged as undefined behavior in the standard.

 71

71

Being Badiwad

- However, if a correct and conventional use of multiple inheritance is employed elsewhere, this code will no longer "work."

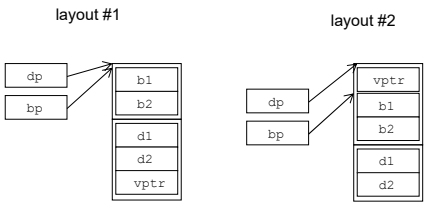
```
ObservedBlob *obp = new ObservedBlob;
setWidget(obp);
~~~
Subject *sp = static_cast<Subject*>(getWidget());        // bug!
Subject *sp = static_cast<ObservedBlob*>(getWidget());   // OK
```

- To be clear: the change occurred elsewhere, and this code was not modified. It still broke.

Single Inheritance and void *

- This code was *never* guaranteed to work and could fail under single inheritance as well.

```
class B { int b1, b2; };
class D : public B {
  virtual void f();
  int d1, d2;
};
~~~
setWidget(new D);
~~~
B *bp = (B *) getWidget();    // an old-fashioned cast for old-fashioned code
```



A House Built on Sand

- It was never correct to cast from a `void *` to anything but the original type.
- It so happened that the cast often worked.
- But long-term changes to the language undermined assumptions, and the shifting language brought the house down.
- How many of us do this?

```
T *p = nullptr;
~~~
delete p;     // ok.
~~~
delete p;     // undefined behavior that probably "works" for now.
```

 74

74

A House Built on Standard

- Not only does our code outlive our hardware, it often *outlives the language in which it was written*.
- Therefore:

✓ *Never ass|u|me.*

✓ *The standard is there for a reason.*

✓ *Follow it.*

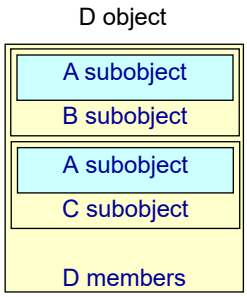
 75

75

Duplicate Subobjects

- What happens if the same base class appears more than once in a class lattice?

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

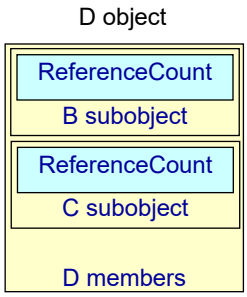


- You get two copies of the subobject.
- Sometimes, this is correct.

Duplicate Subobjects

- Usually, it's not.

```
class ReferenceCount { ~~~ };
class B: public ReferenceCount { ~~~ };
class C: public ReferenceCount { ~~~ };
class D: public B, public C { ~~~ };
```



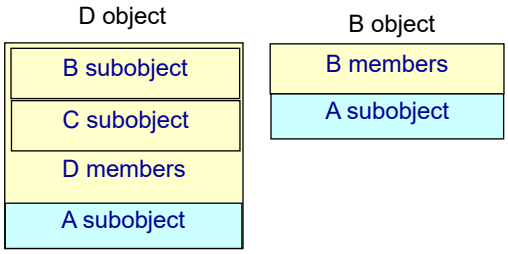
- Generally, the best thing to do is redesign.

```
class B { ~~~ };
class C { ~~~ };
class D: public B, public C { ~~~ };
~~~
auto p = make_shared<D>();
```

Virtual Base Classes

- If redesign is not appropriate, and having duplicated subobjects is not correct, then you may be forced to consider virtual base classes.

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



- Any duplicate virtual subobjects will share the same storage.
- Yes, you can have virtual and non-virtual A subobjects in the same complete object.
- Let's not go there…

 78

78

Virtual Bases Are Problematic

- Virtual base classes are complex.
- They introduce problems with initialization and default initialization.
- They "complexify" constructors and destructors.
- They may cause compiler-generated mechanism to be inserted into each object of the class.
- They break layering of abstractions.
- They also cause inefficiency. That's what we'll look at now.

✓ *Note, however, that virtual base classes are in the C++ language for a reason. Sometimes they're just what you need. But not often.*

 79

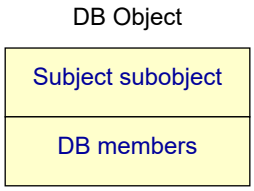
79

Non-Virtual Base Classes

- Let's look at a very simple hierarchy:

```
class Subject {
public:
    virtual ~Subject();
    virtual void notify();
    ~~~
};

class DB: public Subject {
public:
    void notify();
    ~~~
};
```



 80

80

Efficient Member Access

- Access to the base class's members is straightforward and cheap.

```
void useDB(DB *p) {
    p->mem_ = 12;
}

_Z5useDBP2DB:
        MOV     R1,#+12
        STR     R1,[R0, #+4]
        MOV     PC,LR
```

 81

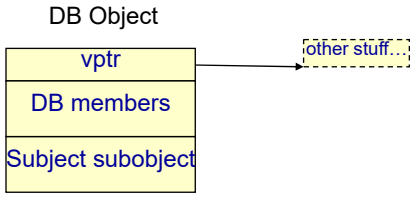
81

Introducing Virtual Bases

- Suppose design constraints force us to use virtual inheritance instead of non-virtual inheritance.

```
class Subject {
public:
    virtual ~Subject();
    virtual void notify();
    ~~~
};

class DB: public virtual Subject {
public:
    void notify();
    ~~~
};
```

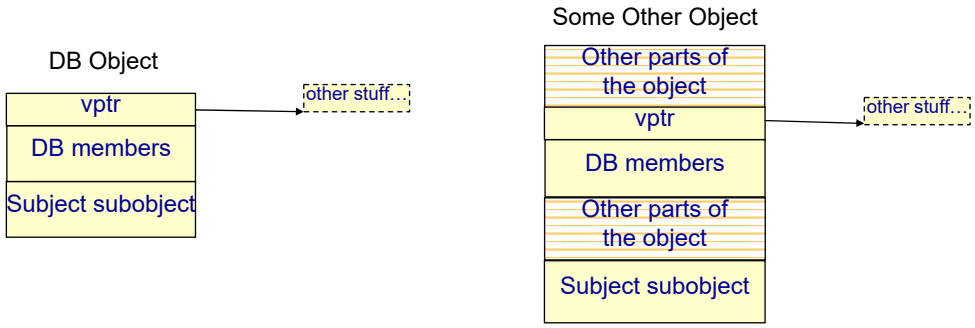


 82

82

Where's The Subobject?

- There is no guarantee that the storage for the `Subject` subobject is adjacent to that for `DB`.
- If `DB` is itself used as a base class in another object, its virtual `Subject` subobject may be shared and located elsewhere.



- Most of the expense (and complexity) of using virtual base classes lies in finding out just where the virtual base class subobjects are.

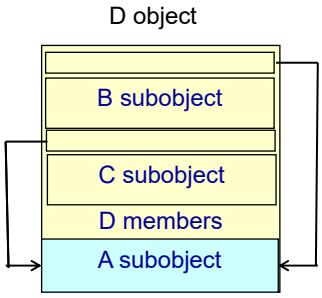
 83

83

Implementation

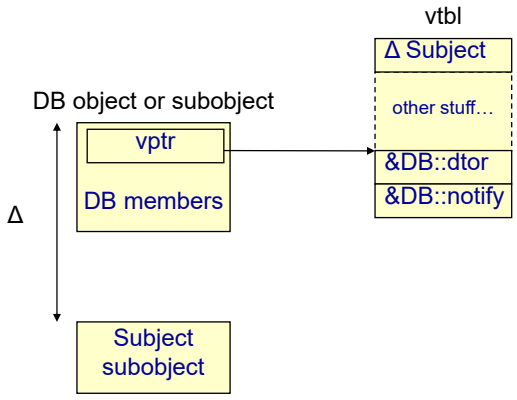
- There are a variety of implementations of virtual base classes.
- The original mechanism stored pointers to virtual base subobjects.

✓ *Such a class is not bitwise-copyable.*



Typical Implementation

- The most common mechanism stores an offset in the virtual table of the class.



Inefficient Member Access

- Now the "simple" access to a base class member is not so simple.

```
void useDB(DB *p) {
    p->mem_ = 12;
}

_Z5useDBP2DB:
        LDR     R1,[R0, #+0]
        LDR     R1,[R1, #-12]
        ADD     R0,R1,R0
        MOV     R1,#+12
        STR     R1,[R0, #+4]
        MOV     PC,LR
```


Avoid Virtual Inheritance

- Avoid virtual base classes unless your design really insists on them.

```
void useDB(DB *p) {
    p->mem_ = 12;
}
```

without virtual base

```
_Z5useDBP2DB:
        MOV     R1,#+12
        STR     R1,[R0, #+4]
        MOV     PC,LR
```

with virtual base

```
_Z5useDBP2DB:
        LDR     R1,[R0, #+0]
        LDR     R1,[R1, #-12]
        ADD     R0,R1,R0
        MOV     R1,#+12
        STR     R1,[R0, #+4]
        MOV     PC,LR
```

Use Interface Classes

- For a variety of design reasons (outside the scope of this presentation) it is advantageous for virtual base classes to contain no non-static data members.
- For example:

```
class Subject {
public:
    virtual ~Subject() {}
    virtual void notify() = 0;
};

class DB : public virtual Subject {
public:
    void notify();
    ~~~
};
```

- In this case, there can be no inefficiency in data member access.

 88

88

The End!

 89

89

Unless…

A Miracle Occurred and We Have More Time

Annex: Are Virtual Functions Slow?

- There is an ongoing assumption that virtual functions are slower than non-virtual functions.
- Near the end of a well-reasoned talk, one experienced researcher brought out the following statement:
- "Virtual functions are the real issue. If C++ is avoided throughout a project's code base the inefficiencies of virtual functions will not rear their ugly head."
- This opinion was common in the past, and unfortunately persists.
- However, the reality is that virtual functions, when used appropriately, shrink the size of your code while improving efficiency, safety, and maintainability.

A Simple Comparison

- Let's look at a simple example.

```
class Operation {
public:
  virtual void exec();
};

void exec(Operation *);
~~~
Operation *op = getNextOperation();
exec(op);      // non-virtual call
op->exec();    // equivalent virtual call
```


Comparing Virtual and Non-Virtual

- A quick look seems to support that conclusion.

```
exec(op);     // non-virtual call

MOV      R0,R4
BL       _Z4execP9Operation

op->exec();  // equivalent virtual call

MOV      R0,R4
LDR      R1,[R4, #+0]
LDR      R1,[R1, #+0]
MOV      LR,PC
BX       R1
```

Parochial Reasoning

- We observe that a virtual call is both bigger and slower than a non-virtual call.
- However, that observation considers only the local context of the call.
- It doesn't consider what happens after the call is executed.

A Larger Example

- Let's drag out the omnipresent shape hierarchy.

```
class Shape {
public:
    virtual ~Shape();
    virtual void draw() const = 0;
    virtual void rotate(int) = 0;
private:
    short x_, y_;
};
class Circle : public Shape {
public:
    void draw() const;
    void rotate(int);
private:
    int radius_;
};
class Square : public Shape { ~~~ };
class Triangle : public Shape { ~~~ };
class Blob : public Shape { ~~~ };
```


Operations on Unknown Shapes

- Simple manipulations of unknown types of shapes is straightforward…

```
void manipulate(Shape *sp) {
    sp->draw();
    sp->rotate(12);
}
```

- …but do indeed generate a significant amount of code.

```
_Z10manipulateP5Shape:
        MOV     R4,R0
        LDR     R1,[R4, #+0]
        LDR     R1,[R1, #+8]
        MOV     LR,PC
        BX      R1
        MOV     R1,#+12
        MOV     R0,R4
        LDR     R2,[R4, #+0]
        LDR     R2,[R2, #+12]
        POP     {R4,LR}
        BX      R2
```

 96

96

A C-Like Alternative

- Let's recast our shape design using non-virtual operations: type-based conditionals.
- We'll proceed in the usual C-like way, embedding a type code in the object…

```
enum Type { CIRCLE, SQUARE, TRIANGLE, BLOB };

struct Shape {
    Type type_;
    short x_, y_;
};

void draw(Shape *);
void rotate(Shape *, int);
```

- Here, we're using structs and non-member functions.
- We could just as well use classes and static member functions.

 97

97

Type Codes

- Derived classes set the code in the base class.

```
struct Circle : public Shape {
    Circle() : Shape(CIRCLE) {}
    int radius_;
};
```

- Note that we could have used a more traditional C union of structs rather than derived classes.

Using Non-Virtuals

- The manner in which we manipulate shapes is similar...

```
void manipulate(Shape *sp) {
    draw(sp);
    rotate(sp, 12);
}
```

- ...but the code is a lot smaller and faster.

```
_Z10manipulateP5Shape:
        PUSH    {R4,LR}
        MOV     R4,R0
        BL      _Z4drawP5Shape
        MOV     R1,#+12
        MOV     R0,R4
        POP     {R4,LR}
        B        _Z6rotateP5Shapei
```


Things are looking bad for virtual functions…

```
_Z10manipulateP5Shape:
        MOV     R4,R0
        LDR     R1,[R4, #+0]
        LDR     R1,[R1, #+8]
        MOV     LR,PC
        BX      R1
        MOV     R1,#+12
        MOV     R0,R4
        LDR     R2,[R4, #+0]
        LDR     R2,[R2, #+12]
        POP     {R4,LR}
        BX      R2
```

```
_Z10manipulateP5Shape:
        PUSH    {R4,LR}
        MOV     R4,R0
        BL      _Z4drawP5Shape
        MOV     R1,#+12
        MOV     R0,R4
        POP     {R4,LR}
        B       _Z6rotateP5Shapei
```

 100

100

Type-Based Conditionals

- …until we recognize that the virtual function call has already deduced the type of shape it's manipulating.
- The non-virtual function still has to find out.

```
void draw(Shape *sp) {   // aka _Z4drawP5Shape
    switch(sp->type_) {
    case CIRCLE:
        cdraw(static_cast<Circle *>(sp));
        break;
    case SQUARE:
        sdraw(static_cast<Square *>(sp));
        break;
    case TRIANGLE:
        tdraw(static_cast<Triangle *>(sp));
        break;
    case BLOB:
        bdraw(static_cast<Blob *>(sp));
        break;
  }
}
```

 101

101

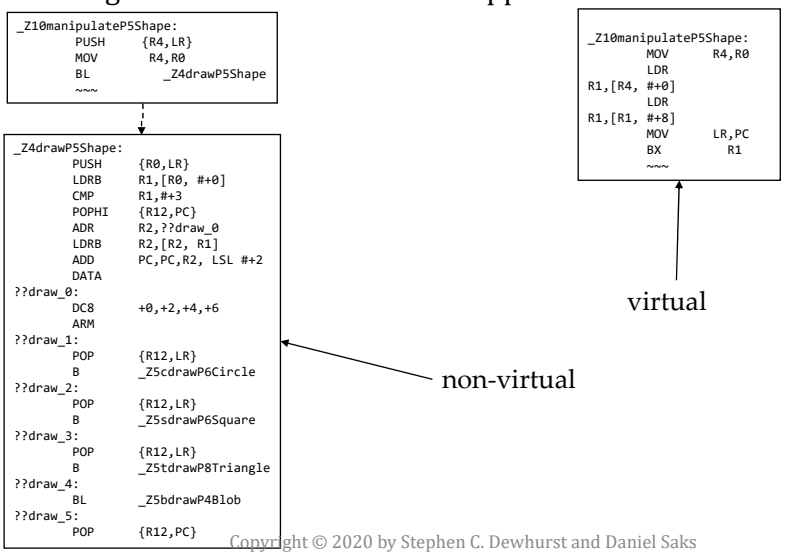
Type Based Conditionals

- The generated code is pretty high quality...

```
_Z4drawP5Shape:
        PUSH    {R0,LR}
        LDRB    R1,[R0, #+0]
        CMP     R1,#+3
        POPHI   {R12,PC}
        ADR     R2,??draw_0
        LDRB    R2,[R2, R1]
        ADD     PC,PC,R2, LSL #+2
        DATA
??draw_0:
        DC8     +0,+2,+4,+6
        ARM
??draw_1:
        POP     {R12,LR}
        B       _Z5cdrawP6Circle
??draw_2:
        POP     {R12,LR}
        B       _Z5sdrawP6Square
??draw_3:
        POP     {R12,LR}
        B       _Z5tdrawP8Triangle
??draw_4:
        BL      _Z5bdrawP4Blob
??draw_5:
        POP     {R12,PC}
```


Virtuals In Context

- ...but it's not as good as the virtual function approach.



Virtual Efficiencies For Type-Based Code

- The virtual mechanism generates only about 33% of the code that the non-virtual mechanism does (usual wafflewords apply).
- If you factor in the size of the readonly data employed by virtuals (virtual function tables, primarily), then the mechanism generates about 75% of the readonly code and data that the non-virtual mechanism does.
- Additionally, each new type-based switch adds to the code space, but there's only one virtual function table per class (if the class has virtual functions), no matter how many objects of the class there are.
- There are additional pros and cons to each approach.

✓ *For non-virtuals, it's possible to inline code within the switch, which may improve speed over a statically-forwarded call.*

✓ *For virtuals, there is no need for a "default" case, since that case simply cannot occur. (This implies that there is—or should be if the user is coding responsibly—an additional cost to each user of the non-virtual implementation to check for a "does not recognize" error.)*

Use The Right Tool For The Job

- Virtual functions are not intended to replace statically-bound functions.

✓ *For statically-bound functions, C++ provides non-member functions (aka "free" functions), non-virtual member functions, and static member functions.*

- Virtual functions are meant to augment your coding and design toolkit with a safe, maintainable replacement for type-based conditional code.
- If you design carefully, employing virtual and non-virtual functions appropriately, your code is likely to be smaller, faster, more correct, and more easily maintained.

✓ *But be aware of newer tools in the toolkit. In context, use of a variant and variant visitor may be an appropriate substitute for a virtual function.*

- Observation: A language feature cannot generally be evaluated in isolation. A valid evaluation requires consideration of the context of the feature's intended use.

The End! Really!