C++ Inheritance II, Casts

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 14 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say

Administrivia

- Quick poll: extra time for Exercise 14?

- Exercise 14a out today, due Friday
  - Practice with dynamic dispatch in C++

- HW3 due next Thursday 😱😱😱
  - Remember to use `hw3fsck` to check your index file!

Lecture Outline

- C++ Inheritance
  - **Static Dispatch**
  - Dynamic Dispatch, Two Perspectives
  - Abstract Classes
  - Constructors and Destructors
- C++ Assignment, 🔪Slicing🔪, and Casts
  - 🔪 Slicing 🔪
  - New-style Casts

- Reference: *C++ Primer*, Chapter 15

What happens if we omit "virtual"?

- By default, without virtual, methods are dispatched *statically*
  - At <u>compile time</u>, the compiler writes in a `call` to the address of the class' method in the `.text` segment
    - Based on the compile-time promised type of the callee
  - This is *different* than Java

```
class Derived : public Base { ... };

int main(int argc, char** argv) {
  Derived d;
  Derived* dp = &d;
  Base* bp = &d;
  dp->foo();
  bp->foo();
  return EXIT_SUCCESS;
}
```

```
Derived::foo()
...
```

```
Base::foo()
...
```

Static Dispatch Example

- Removed `virtual` on methods:

Stock.h

```
double Stock::GetMarketValue() const;
double Stock::GetProfit() const;
```

```
DividendStock dividend;
DividendStock *ds = &dividend;
Stock *s = &dividend;

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes Stock::GetMarketValue()
s->GetMarketValue();

// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes Stock::GetMarketValue().
ds->GetProfit();

// invokes Stock::GetProfit().
// Stock::GetProfit() invokes Stock::GetMarketValue().
s->GetProfit();
```

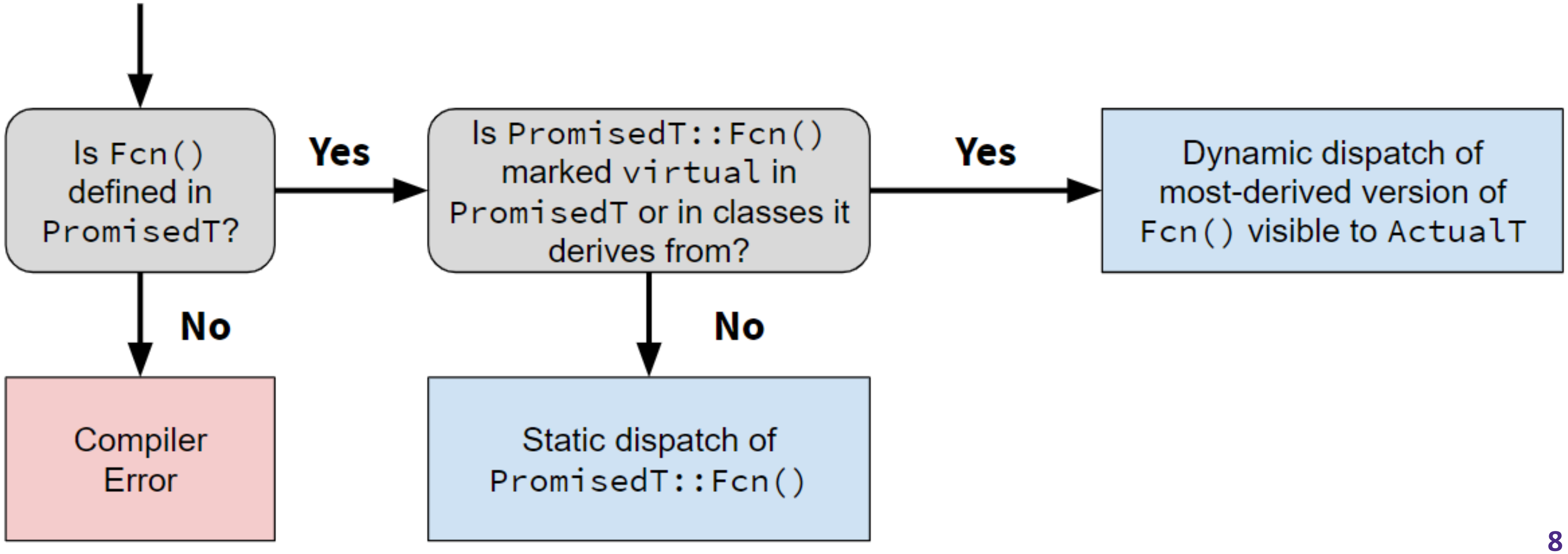
Why Not Always Use `virtual`?

- Two (fairly uncommon) reasons:
  - Efficiency:
    - Non-virtual function calls are a tiny bit faster (no indirect lookup)
    - A class with no virtual functions has objects without a `vptr` field
  - Control:
    - If `f()` calls `g()` in class X and `g` is not virtual, we're guaranteed to call `X::g()` and not `g()` in some subclass
      - Particularly useful for framework design
- In Java, all methods are virtual, except `static` class methods, which aren't associated with objects
- In C++ and C#, you can pick what you want
  - Omitting virtual can cause obscure bugs

Mixed Dispatch

- Which function is called is a mix of both compile time and runtime decisions as well as *how* you call the function
  - If called on an object (*e.g.* `obj.Fcn()`), usually optimized into a hard-coded function call at compile time
  - If called via a pointer or reference:

```
PromisedT *ptr = new ActualT;
ptr->Fcn();  // which version is called?
```



Mixed Dispatch Example

mixed.cc

```
class A {
 public:
  // m1 will use static dispatch
  void m1() { cout << "a1, "; }
  // m2 will use dynamic dispatch
  virtual void m2() { cout << "a2"; }
};

class B : public A {
 public:
  void m1() { cout << "b1, "; }
  // m2 is still virtual by default
  void m2() { cout << "b2"; }
};
```

```
void main(int argc,
          char **argv) {
  A a;
  B b;

  A *a_ptr_a = &a;
  A *a_ptr_b = &b;
  B *b_ptr_a = &a;
  B *b_ptr_b = &b;

  a_ptr_a->m1();  //
  a_ptr_a->m2();  //

  a_ptr_b->m1();  //
  a_ptr_b->m2();  //

  b_ptr_b->m1();  //
  b_ptr_b->m2();  //
}
```

Mixed Dispatch Example

mixed.cc

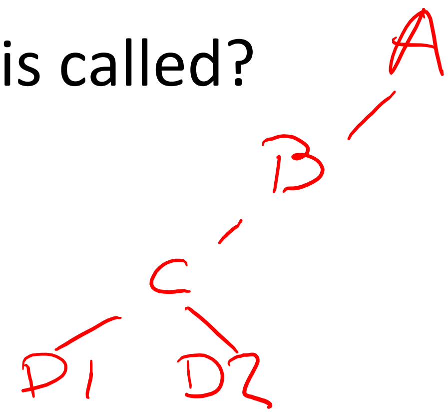
```
class A {
 public:
  // m1 will use static dispatch
  void m1() { cout << "a1, "; }
  // m2 will use dynamic dispatch
  virtual void m2() { cout << "a2"; }
};

class B : public A {
 public:
  void m1() { cout << "b1, "; }
  // m2 is still virtual by default
  void m2() { cout << "b2"; }
};
```

```
void main(int argc,
          char **argv) {
  A a;
  B b;

  A *a_ptr_a = &a;
  A *a_ptr_b = &b;
  B *b_ptr_a = &a;
  B *b_ptr_b = &b;

  a_ptr_a->m1();  // a1,
  a_ptr_a->m2();  // a2

  a_ptr_b->m1();  // a1,
  a_ptr_b->m2();  // b2

  b_ptr_b->m1();  // b1,
  b_ptr_b->m2();  // b2
}
```

Poll Everywhere

pollev.com/cse333

- Whose `Foo()` is called?



	Q1	Q2
A.	A	A
B.	A	B
C.	D1	A
D.	D1	B
E.	I'm not sure…	

```
void Bar() {
  D1 d1;
  D2 d2;
  A *a_ptr = &d1;
  C *c_ptr = &d2;

  // Q1:
  a_ptr->Foo();

  // Q2:
  c_ptr->Foo();
}
```

test.cc

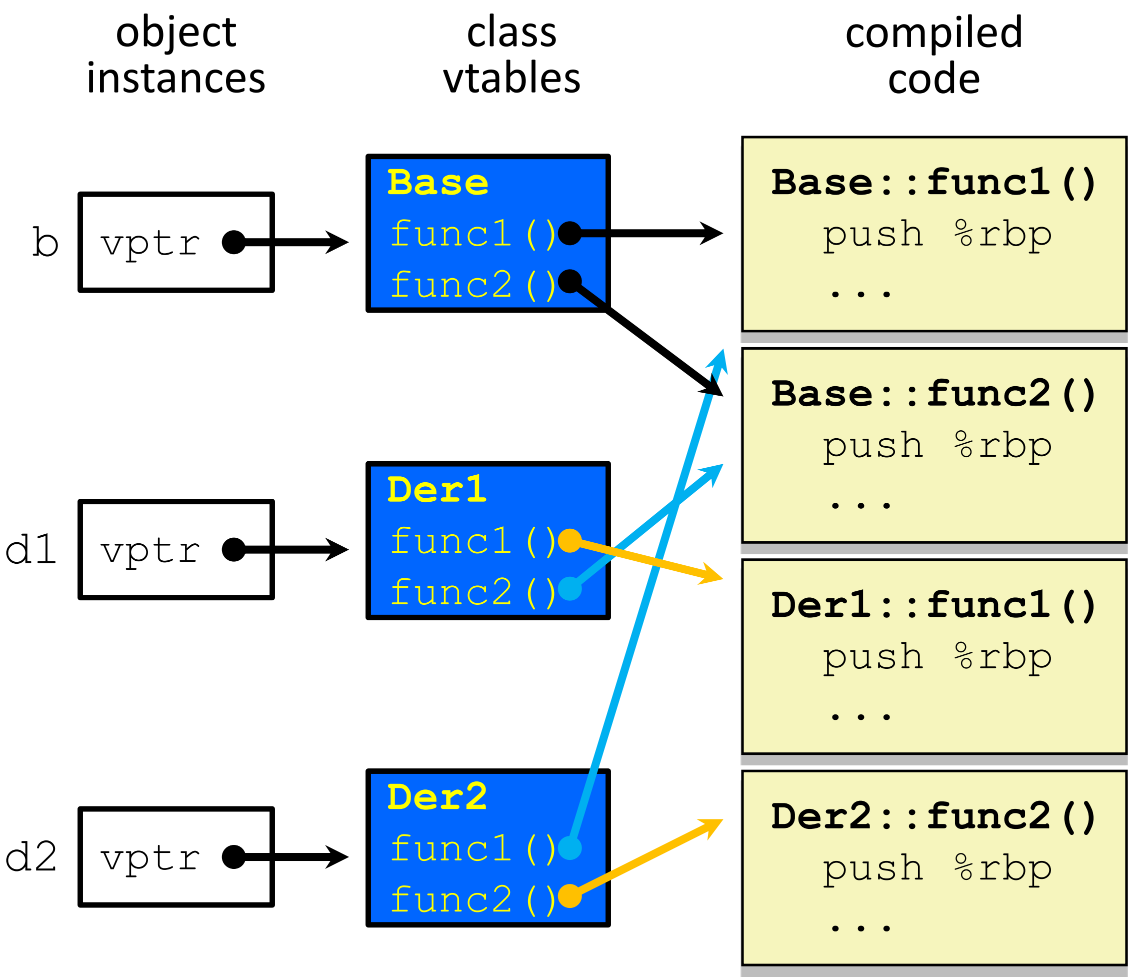
```
class A {
 public:
   void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D1 : public C {
 public:
  void Foo();
};

class D2 : public C {
};
```

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - **Dynamic Dispatch, Two Perspectives**
  - Abstract Classes
  - Constructors and Destructors
- C++ Assignment, 🔪Slicing🔪, and Casts
  - 🔪 Slicing 🔪
  - New-style Casts

- Reference: *C++ Primer*, Chapter 15

Review: vtable/vptr




```
Base b;
Der1 d1;
Der2 d2;

Base *bptr = &d1;

bptr->func1();
// bptr -->
// d1.vptr -->
// Der1.vtable.func1
//    -->
// Base::func1()

bptr = &d2;

bptr->func1();
// bptr -->
// d2.vptr -->
// Der2.vtable.f1 -->
// Base::f1()
```

Two Perspectives on the Same Thing

- In the STL,“the spec” implies “the implementation”
  - Eg, “fast random access” => array impl for `std::vector`
- In dynamic dispatch, “the implementation” implies “the spec”
- This gives you two options for understanding dynamic dispatch
  - Though in both cases, you must access the object via indirection (eg, pointer or reference)

Perspective #1: Memorizing the Rules

1. `virtual` starts at the "highest" point in the inheritance tree, and applies to any of its descendents
   - Even if you "skip a level" or omit the `virtual` keyword

2. "virtualness" is decided by the compile-time `PromisedType`

3. The invoked method is decided by the runtime `ActualType`, found by walking up the tree until a method is found

```
A
  /* non-virtual */
  void Foo()

B
  virtual void Foo()

C
  /* No Foo(), but virtual */

D
  /* implicitly virtual */
  void Foo()
```

```
B b;
D d;
A *ap = &d;   // ap->Foo(): A
B *bp = &d;   // bp->Foo(): D
C *cp = &d;   // cp->Foo(): D
ap = &b;      // ap->Foo(): A
bp = &b;      // bp->Foo(): B
```

Perspective #2: Understanding the Implementation

1. Once `Foo()` has been declared `virtual`, it will always have an entry in its vtable and all of its descendents' vtables.
   - If you "skip a level", the address of the parent's entry is copied
2. The *compiler* decides to use the vtable based on whether `Foo()` has an entry in `PromisedType`'s vtable
3. The actual `method()` is decided at runtime by using the instance's vptr, which points to `ActualType`'s vtable

A	
`void Foo()`	`/* empty vtable */`

B	
`virtual void Foo()`	`Foo: &B::Foo()`

C	
`/* No Foo() */`	`Foo: &B::Foo()`

D	
`void Foo()`	`Foo: &D::Foo()`

```
B b;
D d;
A *ap = &d;   // ap->Foo(): A
B *bp = &d;   // bp->Foo(): D
C *cp = &d;   // cp->Foo(): D
ap = &b;      // ap->Foo(): A
bp = &b;      // bp->Foo(): B
```

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Dynamic Dispatch, Two Perspectives
  - **Abstract Classes**
  - Constructors and Destructors
- C++ Assignment, Slicing, and Casts
  - Slicing
  - New-style Casts

- Reference: *C++ Primer*, Chapter 15

Abstract Classes

- Sometimes we want to include a function in a class but *only* implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a "pure virtual" function
    - Example: `virtual string noise() = 0;`
- A class containing *any* pure virtual methods is abstract
  - You can't create instances of an abstract class
  - Extend abstract classes and override methods to use them
- A class containing *only* pure virtual methods is the same as a Java interface
  - Pure type specification without implementations

Lecture Outline

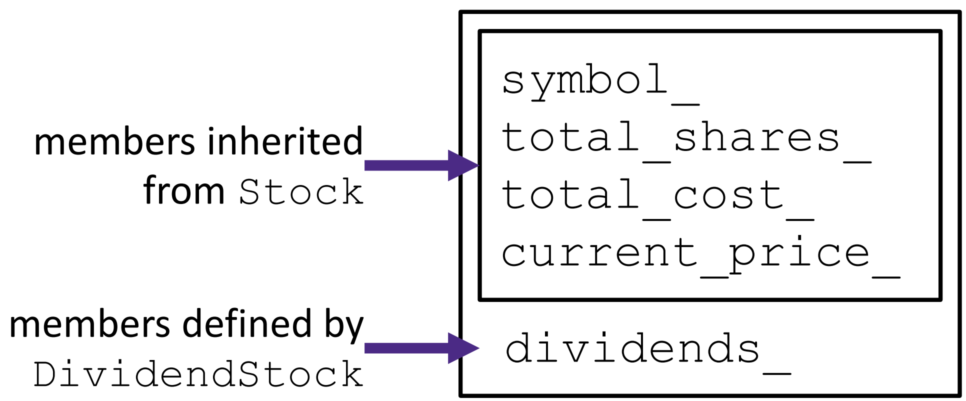
- C++ Inheritance
  - Static Dispatch
  - Dynamic Dispatch, Two Perspectives
  - Abstract Classes
  - **Constructors and Destructors**
- C++ Assignment, 🔪Slicing🔪, and Casts
  - 🔪 Slicing 🔪
  - New-style Casts

- Reference: *C++ Primer*, Chapter 15

Derived-Class Objects

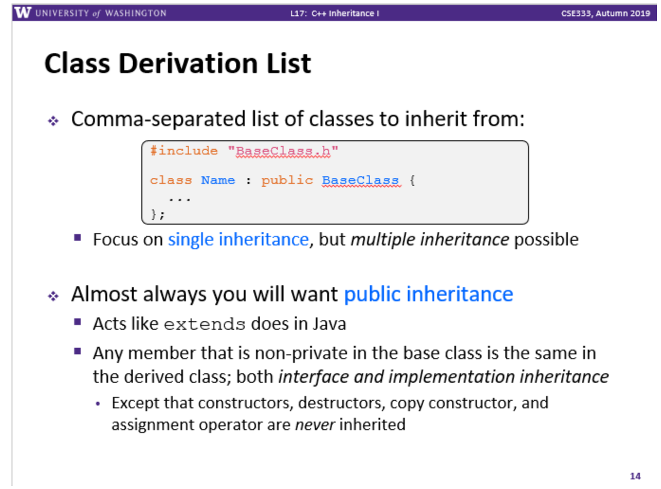
- A derived object contains "subobjects" corresponding to the data members inherited from each base class
  - No guarantees about how these are laid out in memory (not even contiguousness between subobjects)

- Conceptual structure of `DividendStock` object:

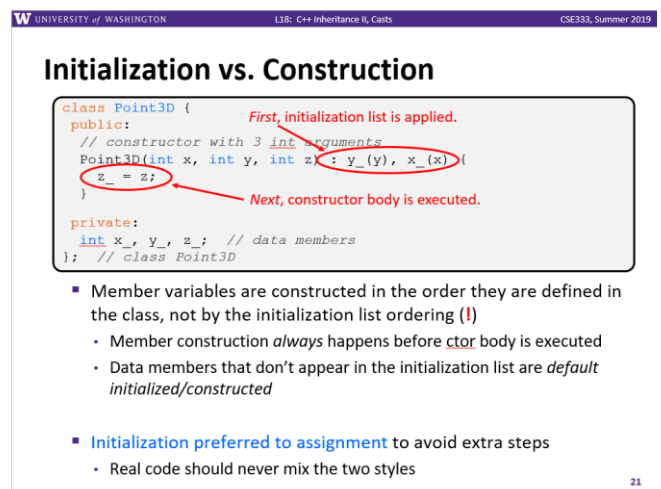


Initializing Sub-objects

- *L17: "Except that constructors, destructors, copy constructor, and assignment operator are never inherited"*



- *L11: "Member variables are constructed in the order they are defined in the class … [and] before [the] ctor body is executed*
  - *Data members that don't appear in the initialization list are default initialized/constructed"*



Constructors and Inheritance

- A derived class **does not inherit** the base class' ctor
  - The derived class must have its own ctor (possibly synthesized)

- The base class ctor is invoked *before* the derived's ctor
  - By default, the base class's default ctor is called
    - Compiler error if the base class doesn't have a default constructor!
  - Use the derived class's initialization list to specify which base class constructor to use

- Then the derived class' member variables are constructed

- Finally, the body of derived's ctor is invoked

Constructor Examples

badctor.cc

```
class Base {  // no default ctor
 public:
  Base(int yi) : y(yi) { }
  int y;
};

// Compiler error when you try to
// instantiate a Der1, as the
// synthesized default ctor needs
// to invoke Base's default ctor.
class Der1 : public Base {
 public:
  int z;
};

class Der2 : public Base {
 public:
  Der2(int yi, int zi)
    : Base(yi), z(zi) { }
  int z;
};
```

goodctor.cc

```
// has default ctor
class Base {
 public:
  int y;
};

// works now
class Der1 : public Base {
 public:
  int z;
};

// still works
class Der2 : public Base {
 public:
  Der2(int zi) : z(zi) { }
  int z;
};
```

Destructors and Inheritance

- Destructor of a derived class:
  - *First* runs body of the dtor
  - *Then* invokes of the dtor of the base class
- Static dispatch of destructors is almost always a mistake!
  - Good habit to always define a dtor as virtual
    - Empty body if there's no work to do

baddtor.cc

```
class Base {
 public:
  Base() { x = new int; }
  ~Base() { delete x; }
  int *x;
};

class Der1 : public Base {
 public:
  Der1() { y = new int; }
  ~Der1() { delete y; }
  int *y;
};

void foo() {
  Base *b0ptr = new Base;
  Base *b1ptr = new Der1;

  delete b0ptr;  //
  delete b1ptr;  //
}
```

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Dynamic Dispatch, Two Perspectives
  - Abstract Classes
  - Constructors and Destructors
- C++ Assignment, Slicing, and Casts
  - **Slicing**
  - New-style Casts

Assignment and Inheritance

- C++ allows you to assign the value of a derived class to an instance of a base class
  - Known as object slicing
    - It's legal since `b = d` passes type checking rules
    - But `b` doesn't have space for any extra fields in `d`

slicing.cc

```
class Base {
 public:
  Base(int xi) : x(xi) { }
  int x;
};

class Der1 : public Base {
 public:
  Der1(int yi) : Base(16), y(yi) { }
  int y;
};

void foo() {
  Base b(1);
  Der1 d(2);

  d = b;  //
  b = d;  //
}
```

STL and Inheritance: Problem

- Recall: STL containers store **copies of values**
  - What happens when we want to store mixes of object types in a single container? (*e.g.* `Stock` and `DividendStock`)
  - You get sliced ☹

```
#include <list>
#include "Stock.h"
#include "DividendStock.h"

int main(int argc, char **argv) {
  Stock s;
  DividendStock ds;
  list<Stock> li;

  li.push_back(s);    // OK
  li.push_back(ds);   // OUCH!

  return EXIT_SUCCESS;
}
```

STL and Inheritance: Solution

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! ☺
  - `sort()` does the wrong thing ☹
  - You have to remember to `delete` your objects before destroying the container ☹
    - Smart pointers!

Lecture Outline

- C++ Inheritance
  - Static Dispatch
  - Dynamic Dispatch, Two Perspectives
  - Abstract Classes
  - Constructors and Destructors
- C++ Assignment, 🔪Slicing🔪, and Casts
  - 🔪 Slicing 🔪
  - **New-style Casts**

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type
    - Don't change the data, but treat differently
  - Forcibly convert a primitive type to another
    - Actually changes the representation

- You *can* still use C-style casting in C++, but sometimes the intent is not clear

Casting in C++

- C++ provides an alternative casting style that signals the programmer's intent explicitly:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

staticcast.cc

static_cast

- static_cast can convert:
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - Non-pointer conversion
    - *e.g.* float to int
- static_cast is checked at compile time

```
class A {
 public:
  int x;
};

class B {
 public:
  float x;
};

class C : public B {
 public:
  char x;
};
```

```
void foo() {
  B b; C c;

  // compiler error
  A *aptr = static_cast<A*>(&b);
  // OK
  B *bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C *cptr = static_cast<C*>(&b);
}
```

dynamiccast.cc

dynamic_cast

- dynamic_cast can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- dynamic_cast is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type

```
class Base {
 public:
  virtual void foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base *bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1 *dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- `const_cast` adds or strips const-ness
  - Dangerous (!)

```
void foo(int *x) {
  *x++;
}

void bar(const int *x) {
  foo(x);                         // compiler error
  foo(const_cast<int*>(x));       // succeeds
}

int main(int argc, char **argv) {
  int x = 7;
  bar(&x);
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.* storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is “wide” enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
    - This is used (carefully) in hw3

Implicit Conversion

- The compiler tries to infer some kinds of conversions
  - When types are not equal and you don't specify an explicit cast, the compiler looks for an acceptable implicit conversion

```
void bar(const std::string &x);

void foo() {
  int x = 5.7;   // conversion, float -> int
  bar("hi");     // conversion, (const char*) -> string
  char c = x;    // conversion, int -> char
}
```

Sneaky Implicit Conversions

- (`const char*`) to `string` conversion?
  - If a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions
  - At most, one user-defined implicit conversion will happen
    - Can do `int` → `Foo`, but not `int` → `Foo` → `Baz`

```
class Foo {
 public:
  Foo(int x) : x(x) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char **argv) {
  return Bar(5);  // equivalent to return Bar(Foo(5));
}
```

Avoiding Sneaky Implicits

- Declare one-argument constructors as `explicit` if you want to disable them from being used as an implicit conversion path
  - Usually a good idea

```
class Foo {
 public:
  explicit Foo(int x) : x(x) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char **argv) {
  return Bar(5);  // compiler error
}
```

Extra Exercise #1

- Design a class hierarchy to represent shapes
  - *e.g.* Circle, Triangle, Square
- Implement methods that:
  - Construct shapes
  - Move a shape (*i.e.* add (x,y) to the shape position)
  - Returns the centroid of the shape
  - Returns the area of the shape
  - `Print()`, which prints out the details of a shape

Extra Exercise #2

- Implement a program that uses Extra Exercise #1 (shapes class hierarchy):
  - Constructs a vector of shapes
  - Sorts the vector according to the area of the shape
  - Prints out each member of the vector

- Notes:
  - Avoid slicing!
  - Make sure the sorting works properly!