Introduction to C++ Casting

Joshua Lehrer
Vice President - Principal Software Engineer
FactSet Research Systems, Inc.

Who is Joshua Lehrer

- 20+ years C++ experience
- FactSet Research Systems, NYSE:FDS
  - Principal software engineer
  - Advisor & trainer to >1,000 developers
  - Gatekeeper of >12M line core codebase
-  Brown University computer science
- Contact
  - training@LehrerFamily.com
  - twitter: @jsl_13
  - http://www.lehrerfamily.com/

Review C-style Casts

- What does this do? "(T)value"
  - Changes between integral types
    - `(int) floatVal`
  - Add / remove cv-qualification[1]
    - `((Object*)this)->NonConstMethod();`
  - Up / down inheritance tree
    - `(SubClass*) ptrBase`
  - Custom cast operator
    - `(float) fixed_point`
  - Conversion constructor
    - `(fixed_point) float`
  - Interpret pointer differently
    - `(const char *) intPtr`

[1] const-volatile qualification, e.g. "const int" "volatile void *"

Example #1

```
void func(const foo &arg) {
 //
 // lots of code here
 //

 // lib author failed to make print const
   ((foo&)arg).print();
}
```

Example #1 cont…

```
void func(const bar &arg) {
 //
 // lots of code here
 //

 // lib author failed to make print const
   ((foo&)arg).print();
}
```

Example #1 cont…

```
void func(const bar &arg) {
 //
 // lots of code here
 //

 // lib author failed to make print const
   ((foo&)arg).print(); //format HD
}
```

What Happened?

- Intent: cast off const qualification
- Reality: changed the type
- Compiler complied
  - “You, dear developer, are smarter than I”

const_cast<>

- Adds or removes cv-qualification
  - Only cast that allows removal of cv-qualification
- No more, no less
- Safe to add, unsafe to remove
- Declare intention to compiler
  - Compiler enforces
- Declare intention to developers
  - Easy to write code compiler understands
  - Hard to write code developers understand

Example #1 Revisited

```
void func(const foo &arg) {
  //
  // lots of code here
  //

  // lib author failed to make print const
    const_cast<foo&>(arg).print();
}
```

Example #1 Revisited cont…

```
void func(const bar &arg) {
  //
  // lots of code here
  //
  foo.cxx: In function ‘void func(const bar&)’:
  foo.cxx:10:21: error: invalid const_cast from type ‘const bar*’ to type ‘foo*’
  /
}
```

const_cast<> uses

- `const_cast<foo*>(this)->m_value=3`
  - Don't do this
- Parameter fixing
  - `unsigned long hash(char *str)`
- Example of adding const
  - None, compiler will handle automatically

```
size_t strlen(const char*);
char * p = … ;
strlen(p);
```

Will This Compile?

```
void func() {
    buffer<char, 256> b(…);
    const char *p = const_cast<const char *>(b);
}
```

```
foo.cxx: In function ‘void func()’:
foo.cxx:10:45: error: invalid const_cast from type
‘buffer<char, 32u>’ to type ‘const char*’
```

Will This Compile?

```
void func(const char ** p) {
  *p = "Hello";
}

int main() {
  char* foo;
  func(&foo);

}
```

Adding *const* Sometimes Dangerous

```
void func(const char ** p) {
  *p = "Hello";
}

int main() {
  char* foo;
  func(const_cast<const char**>(&foo));
  foo[0]='Y'; //yikes, write to immutable location
}
```

Cast Operators

- C++ style casts follow this format
  - **[keyword]_cast<target_type>(value)**
- Valid keywords
  - **const**
  - **static**
  - **dynamic**
  - **reinterpret**
- Values
  - Null ☐ Null
  - Value returned can differ

static_cast<>

- Between statically related types
- Common, duplicates most c-style casts
- Built-in
  - int □□ float
- Custom casts / conversion operators
  - fixed_point □□ float
- Up / down inheritance tree
  - Unchecked
- Pointers
  - void* □□ T*

Limitations of static_cast<>

```
int j = 0;

// error:  'int'      'int*'
int* pi = static_cast<int*>(j);

// error:  'int*'      'int'
int i = static_cast<int>(&j);
```

Limitations of static_cast<>

```
void func(int&);

const int i=0;
// error:  ‘const int’      ‘int&’
func(static_cast<int&>(i));
```

Limitations of static_cast<>

```
struct A {};
struct B {};

A* pA = …;
// error:  ‘A*’        ‘B*’ [1]
B* pB = static_cast<B*>(pA);
```

[1] use reinterpret_cast<> or jsl::pointer_cast<>

reinterpret_cast<>

- Avoid this wicked witch of casting
  - "using [reinterpret_cast] is almost always evil" – Herb Sutter
  - "reinterpret_cast is not a subject for polite discussion. Use it to alert readers that your code is impolite." – KAI Software
- Usually implementation defined
- Use on unrelated pointer/reference types

reinterpret_cast<> Uses

- Predefined memory locations

```
char* const p = reinterpret_cast<char*>(0x14);
```

- Decoding binary streams

```
template <typename T>
inline const T& decode(const char *&p)[1] {
   return *reinterpret_cast<const T*&>(p)++;
}
i = decode<int>(stream_p);
f = decode<float>(stream_p);
```

[1] Don't do this

dynamic_cast<>

- Acts on dynamic type of pointer/reference
- Queries inheritance tree
  - Safely go down/across
- Failure conditions
  - Pointers: null
  - Reference: throws std::bad_cast
- Static type must have vtable

dynamic_cast<>

- Runtime cost
  - Use, if necessary, until proven inefficient
  - May be optimized away
- Respects inheritance protections
  - Public / protected / private / friend
- Special case
  - dynamic_cast<cv-void*>
  - Pointer to most derived base class

dynamic_cast<> Example

```
struct shape { virtual void draw() const = 0; };
struct square : shape { virtual void draw() const; };
struct circle : shape {
    virtual void draw() const;
    void draw_special() const;
};

void draw(const shape *p) {
    if (const circle *c = dynamic_cast<const circle*>(p)) {
        c->draw_special();
    } else {
        p->draw();
    }
}
```

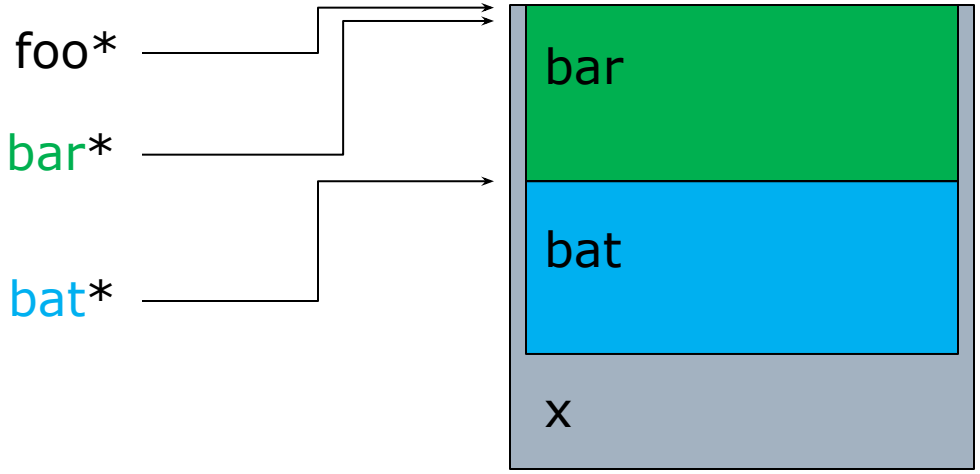
dynamic_cast<> cont…

- When to use
  - Never?
  - Use proper polymorphism
- static_cast<> vs. dynamic_cast<>
  - static_cast<*> w/o null-test
  - dynamic_cast<*> w/ null-test

Pointer Values

- Usually unchanged, can change
  - Implementation defined
  - Applies to static & dynamic casts

```
struct foo : public bar, public bat { int x; };
```



std::*_pointer_cast

- Language vs. library
- Three routines

```
shared_ptr<T>  static_pointer_cast (const shared_ptr<U>&)
shared_ptr<T> dynamic_pointer_cast (const shared_ptr<U>&)
shared_ptr<T>   const_pointer_cast (const shared_ptr<U>&)
```

- Return shared_ptr<T>
  - Appropriately casted pointer from <U>
  - Reference count increased
  - Reference to Deleter
- Example:

```
if (shared_ptr<derived> p = dynamic_pointer_cast<derived>(pBase)) {
} else {
}
```

jsl::pointer_cast<>

- Pointer □□ pointer via dual void* casts
- Full implementation at http://www.lehrerfamily.com/pointer_cast.h
- Example:

```
template <typename T> inline T pointer_cast(void* p) {
   return static_cast<T>(p);
 }

 int * p1 = pointer_cast<int*>(pchar);
```

Why Use C++ Casts?

- Safer, catches bugs @ compile vs. run time
- Easily recognized
  - Searchable
  - Calls attention
- Precise intention conveyed
  - To compiler
  - To reviewers – auto commented code!
- "Perhaps making casts ugly and hard to read[/type] is a *good* thing." – Scott Meyers

Credits

- “More Effective C++”
  - By Scott Meyers
  - Item #2
- “Exceptional C++”
  - By Herb Sutter
  - Forward by Scott Meyers
  - Item #44
- Additional Input From
  - Hillel Sims @ Bloomberg
  - Adam Midvidy @ MongoDB