
CS11 – *Advanced C++*

Fall 2009-2010

Lecture 7

Today's Topics

- All about casting in C++
 - Implicit casting
 - **explicit** keyword on constructors
 - Explicit casting in C++
 - **mutable** keyword and **const**
-

Implicit Type-Conversions

- C++ has *implicit* type-conversions for primitive types
 - You don't explicitly cast from one type to the other
 - They just happen, without warning.
 - Promotions
 - Value is preserved, no information is lost
 - Examples: `char` → `int`, `bool` → `int`
 - Conversions
 - Value may be changed or become invalid
 - Examples: `double` → `char`, `double` → `float`
 - Beware! Compiler doesn't give you much help!
-

Conversion of User-Defined Types

- Can define implicit conversion ops for user-types

```
class Rational {  
public:  
    Rational(int num, int denom);  
    ...  
    // Convert from Rational to double  
    operator double() const;  
};
```

- Provides conversion from **Rational** to **double**

```
Rational r(1, 2);    // r is 1/2  
double d = 0.5 * r;
```

- **r** is converted to **double**, then multiplication is performed
-

Unexpected Results!

- You want to print **Rational** values with `<<`
 - Print them as “num/denom”
 - ...but, you forgot to implement `<<`

- You write this code:

```
Rational r(1, 2);  
cout << r;
```

- You expect this to print `1/2`
 - (Or, actually, to not compile since you didn't implement `<<`)
 - But it *does* compile, and it prints out `0.5`
 - Not too surprising, just subtle.
-

Crafty Compilers

- Compiler sees no `<<` for `Rational`
 - “But `Rational` can be converted to `double`, and `double` can be output with `<<...`”
 - Problem:
 - Implicit conversion operations can produce unexpected or undesirable results!
 - Violates the “Law of Least Surprise”
 - Moral:
 - Be *very* careful with implicit conversion operations
 - Better yet, don’t use them:

```
double Rational::asDouble() { ... }
```
-

More Implicit Conversion Options

- Single-argument constructors *also* enable implicit conversions in C++

- Example:

```
Rational(int num = 0, int denom = 1);
```

- Defines default values for arguments
- Also allows `ints` to be converted to `Rationals`

```
Rational r1(3); // r1 = 3/1
```

```
Rational r2 = 5; // r2 = 5/1
```

```
r1 = 6; // r1 = Rational(6)
```

- Compiler figures out the conversions to use!
-

Another Implicit Conversion Example

- An integer-array class:

```
class Array {  
    ...  
public:  
    Array(int lowBound, int highBound);  
    Array(int size);  
  
    int & operator[](int index);  
  
    bool operator==(const Array &) const;  
    bool operator!=(const Array &) const;  
};
```

More Unexpected Results

- Want to compare two arrays:

```
Array a(10), b(10);  
...  
for (int i = 0; i < 10; i++) {  
    if (a == b[i]) {  
        ... // Do stuff!  
    }  
}
```

- Oops; meant to type `a[i] == b[i]`
 - But, the code compiles!
 - What happens:
 - Compiler guesses this: `a == Array(b[i])`
 - Wrong, not to mention *terribly* inefficient!
-

Disallowing Implicit Conversions

- Compiler should complain in these cases
 - Don't want it to make up stuff that compiles, but that you didn't mean!
 - Enter the **explicit** keyword
 - Added to C++ *specifically* because of these issues
 - Can declare constructors to be **explicit**
 - C++ won't use them for implicit conversions
 - Example:

```
explicit Array(int size);
```
-

Default Parameters and **explicit**

- If constructor can take only a single argument it will be used as an implicit conversion
 - ...even multi-arg constructors with default values
 - **Rational** example again:

```
Rational(int num = 0, int denom = 1);
```

 - Defines default values for arguments
 - *Also* provides implicit conversion from **int**
 - Can also use **explicit** here:

```
explicit Rational(int num = 0, int denom = 1);
```

 - No longer allows implicit conversions from **int**
-

Explicit Casts in C and C++

- C has one explicit cast operator for everything
 - Can convert between related types
 - e.g. `double` to `int`
 - Can cast pointers to different types
 - e.g. `void*` to `float*`, or `float*` to `char*`
 - Can cast pointers to `int`, and vice versa
 - Can cast away `const`-ness
 - Many unsafe or potentially unsafe scenarios!
 - C++ provides four explicit casting operators
 - Breaks down different kinds of casts into explicit operations
 - Provides more type-safety checks at compile-time, run-time
 - Easier for programmers to understand, too
-

C++ Cast Operations

- **const_cast**
 - Casts away `const`-ness
 - **static_cast**
 - Performs safe conversions between related types, using static (compile-time) type information
 - **dynamic_cast**
 - Uses runtime type information to safely cast down or across class hierarchies
 - **reinterpret_cast**
 - For all the dangerous stuff.
-

Removing **const** Constraints

- `const_cast` removes `const` constraints
 - `T const_cast<T>(const T value)`
- Not for changing the type of `value` !
 - Using `const_cast` to change types will not compile
- Examples:

```
void output(SpecialWidget *psw);
```

```
const SpecialWidget csw;
```

```
output(&csw);
```

```
// COMPILE ERROR
```

```
output(const_cast<SpecialWidget*>(&csw));
```

```
// OK
```

```
Widget *pw = new SpecialWidget();
```

```
output(const_cast<SpecialWidget*>(pw));
```

```
// COMPILE ERROR
```

Cached Values and **const**

- **const_cast** sometimes used when objects cache temporary values that are expensive to compute

```
class Date {  
    ...  
    string cache;  
    bool cacheValid;  
    void updateCacheVals();    // sets cache value  
public:  
    ...  
    string stringRep() const;  
};
```

- **stringRep()** returns a **string** version of the date value
 - Cache the result, as long as date value doesn't change
-

Cached Values and **const** (2)

■ Implementation of **stringRep()**

```
string Date::stringRep() const {
    if (!cacheValid) {
        // Type of this is "const Date *", so
        // cast to non-const Date *
        Date *mut = const_cast<Date *>(this);
        mut->updateCacheVals();
        mut->cacheValid = true;
    }
    return cache;
}
```

- **this** is **const** because the member function is **const**
 - Must *cast away const* to update cached values!
-

Cached Values and **const** (3)

- That solution isn't very elegant.
- Also, it might not actually work!
 - If original variable is declared as `const`, casting away `const` is not guaranteed to work on all implementations
 - e.g. compiler or OS might store variable in read-only memory
- Example:

```
Date d1;  
const Date d2;  
string s1 = d1.stringRep();  
string s2 = d2.stringRep(); // ???
```

- `s2` is not guaranteed to have a valid result across all implementations
-

Mutable Data-Members

- Better solution is to use the **mutable** keyword

```
class Date {  
    ...  
    mutable string cache;  
    mutable bool cacheValid;  
    void updateCacheVals() const;  
public:  
    ...  
    string stringRep() const;  
};
```

- Compiler will ensure that **cache** can be changed, even on variables declared **const**
 - **mutable** means “can never be **const**”
-

Mutable Data-Members (2)

- Implementation of `stringRep()` gets simpler:

```
string Date::stringRep() const {
    if (!cacheValid) {
        // Only mutable data-members are changed.
        updateCacheVals();
        cacheValid = true;
    }
    return cache;
}
```

- And this also works now:

```
Date d1;
const Date d2;
string s1 = d1.stringRep();
string s2 = d2.stringRep(); // OK!
```

Proper Uses of `const_cast`

- `const_cast` is best for situations when a function doesn't use `const`, but it ought to!
 - The function doesn't change the argument, but argument wasn't declared `const` (usually by mistake)

- Example: dealing with bad standard library impls.

```
size_t strlen(char *s); // should be const char *
```

```
const char *prompt = "Shall we play a game?";  
size_t length = strlen(const_cast<char*>(prompt));
```

- *Very uncommon situation!*
 - You shouldn't need `const_cast` very often
-

Static Casts

- **static_cast** converts between related types
 - `static_cast<T>(value)`
 - Converts `value` to type `T`
 - Only uses static type information (compile-time check)
 - Example use-cases:
 - Convert from an integer type to an enumeration
 - Convert from a floating-point type to an integer type
 - Convert from one pointer-type to another
 - Typically, within the same class hierarchy
 - Example:

```
int *p = static_cast<int*>(malloc(100 * sizeof(int)));
```

 - `malloc()` returns a `void*` which needs to be casted
-

Dynamic Casts

- **dynamic_cast** converts between types in a class hierarchy, using run-time type information
 - **dynamic_cast<T>(value)**
 - **T** must be a pointer or reference to a polymorphic type (a class with **virtual** member-functions)
 - Performs a run-time type-check to see if **value** can be cast to **T**
 - If so, conversion takes place and result has type **T**
 - If not, **dynamic_cast** evaluates to 0 !
 - Mainly used to cast a base-class pointer/reference to a derived class.
 - Has very complex behavior with multiple inheritance!
-

Static and Dynamic Casts

- Static casts are faster than dynamic casts
 - Compile-time check vs. a runtime check
- Static casts can't move down a class hierarchy
 - Requires run-time information about objects
- Example:

```
class Widget { };
class SpecialWidget : public Widget { };
Widget *pw = new SpecialWidget();

// COMPILER ERROR: static type of pw is Widget*
SpecialWidget *spw = static_cast<SpecialWidget*>(pw);

// OK: pw points to a SpecialWidget
SpecialWidget *spw =
    dynamic_cast<SpecialWidget*>(pw);
```

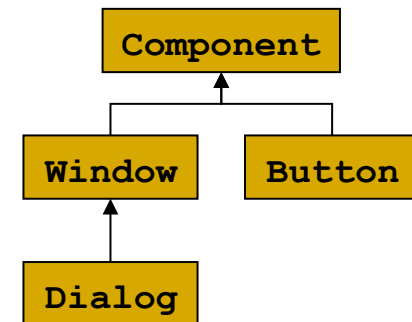
Static and Dynamic Cast Examples

```
Dialog *d = new Dialog();  
Window *w = static_cast<Window *>(d);  
// OK, compile-time check (fast)
```

```
Button *b = new Button();  
Component *c = dynamic_cast<Component *>(b);  
// OK, runtime check (slower)
```

```
Button *b2;  
b2 = static_cast<Button *>(c); // COMPILE ERROR  
b2 = dynamic_cast<Button *>(c); // OK, b2 == b
```

```
Dialog *d2;  
d2 = static_cast<Dialog *>(b); // COMPILE ERROR  
d2 = dynamic_cast<Dialog *>(b); // OK, d2 == 0
```



Reinterpreting Values

- **`reinterpret_cast<T>(value)`**
 - Converts between unrelated types
 - Again, converts `value` to type `T`
 - Example use-cases:
 - Convert an integer to a pointer, or vice versa
 - Convert a pointer to a completely unrelated pointer type
 - If destination type is same bit-width, the result's bit-pattern is same as the source's bit-pattern
 - Should need **`reinterpret_cast`** very rarely
 - If you do need it, encapsulate this functionality inside a class, and don't expose it to others!
 - (Make sure you really *do* need it!)
-

Reinterpreting Values (2)

■ Example:

- An API that represents windows and other GUI components with handles

```
/** Clears the specified window. */  
void clearWindow(int handle) {  
    Window *pWnd = reinterpret_cast<Window*>(handle);  
    ...  
}
```

- API users aren't exposed to implementation details

■ A safer solution?

- The API could use an STL `map` (or `hash_map`) to associate integer handles with GUI components
 - Slower, but less likely to fail spectacularly!
-

Smart Pointers and Casting

- Important to cast smart-pointers properly!
 - Heap-allocated object may become owned by two non-cooperating smart-pointer objects

- Boost example:

```
typedef boost::shared_ptr<Widget> SPWidget;  
typedef boost::shared_ptr<SpecialWidget>  
    SPSpecialWidget;
```

```
SPWidget spw(new SpecialWidget(35));
```

- To get at the actual T^* inside, use
 T^* `shared_ptr<T>::get()`
-

Smart Pointers and Casting (2)

- Casting our Boost shared-pointer, take 1.

```
SPWidget spw(new SpecialWidget(35));  
  
...  
// Dynamic-cast to a special widget, then  
// wrap with a smart-pointer.  
SPSpecialWidget spsw(  
    dynamic_cast<SpecialWidget *>(spw.get()));
```

- Problems?
 - The smart-pointer wrappers don't know about each other!
 - Each one will try to delete the object when its reference-count goes to zero. One will crash.
-

Smart Pointers and Casting (3)

- Good smart-pointer classes will provide casting operations for you to use
 - The functions return new smart-pointer objects themselves
 - The returned smart-pointer knows that multiple smart-pointers are managing the object
- Casting our Boost shared-pointer, take 2:

```
SPWidget spw(new SpecialWidget(35));  
...  
// Safely dynamic-cast the smart-pointer.  
SPSpecialWidget spsw(  
    boost::dynamic_pointer_cast<SpecialWidget>(spw));
```

This Week's Assignment

- Add reflection and cylinders to the raytracer
 - Reflection is pretty easy to implement
 - Can generate some cool pictures!
 - Can use default arguments to make it easier
 - New scene-object: cylinders
 - Can be oriented along an arbitrary axis
 - Reuse some of the sphere computations
 - Update the scene description language
-