G52CPP
C++ Programming
Lecture 18

Dr Jason Atkin

Last lecture

- Final comments about virtual functions

- Automatically created methods:
  - Default Constructor
  - Copy Constructor
  - Assignment operator
  - Destructor

A ‘default’ implementation

```
class MyClass
{
public:

   // Constructor
   MyClass()
   {
   }

   // Destructor
   ~MyClass()
   {
   }

   // Copy constructor
   MyClass( const MyClass& rhs )
      // Initialise each member
      : i( rhs.i )
   {
   }

   // Assignment operator
   MyClass& operator=(
      const MyClass& rhs )
   {
      // Copy each member
      return *this;
   }
};
```

C++11 Move Functions

- Since C++11 you have also been able to create a ‘Move Assignment Operator’ and ‘Move Constructor’
- Will be used to do the copy if and only if the compiler knows that the thing you are copying from will be lost anyway (e.g. it is a temporary variable)
- It moves the contents from the old object to the new object rather than copying them
- Ideal if object’s contents would take a long time to copy
- Define it using a && and no const for the type, e.g.:

```
MyClass& operator=( MyClass&& rhs )
{
    /* Move members, e.g. std::move() or std::swap() */
    return *this;
}
```

- **I will not expect you to know the details for an exam**, but know that these exist in case you see them after you leave

This Lecture

- Conversion operators and constructors

- Casting
  - static cast
  - dynamic cast
  - const cast
  - reinterpret cast

Conversion constructors

Summary: implicit functions

```
class MyClass
{
private:
   int i;

public:
   // Constructor
   MyClass()
   { }

   // Destructor
   ~MyClass()
   { }

   // Copy constructor
   MyClass( const MyClass& rhs )
      // Initialise each member
      : i( rhs.i )
   {
   }
   // Assignment operator
   MyClass& operator=(
      const MyClass& rhs )
   {
      // Copy each member
      i = rhs.i;
      return *this;
   }
};
```

Conversion constructor

- A conversion constructor is **a constructor with one parameter**. e.g. Constructor for **`MyClass`**:

```
MyClass( char c )
{ … do something with c … }
```

  - **Then you can use the following code:**

```
MyClass ob = 'h';
```

- Conversion constructor converts from one type of object to another
  - Can be used **implicitly** to convert between types (unless you say otherwise)
- The conversion constructor is very similar to the copy constructor, i.e. has one parameter, e.g.:

```
MyClass( const MyClass& rhs )
{ … Copy the members … }
```

Conversion constructor

```
class Converter
{
public:
   // Conversion constructor
   // Convert INTO this class
   Converter( int i = 4 );

private:
   int _i;
};
```

```
int main()
{
   int i = 4;
   // Construction from int
   Converter c1(5);
   Converter c2 = i;
}
```

```
// Conversion constructor
Converter::Converter(int i)
   : _i(i) // Set value
{
   cout << "Constructing from int\n";
}
```

Forcing explicit construction

- Providing a **one-parameter** constructor provides a conversion constructor
- This allows compiler to use it to convert to the type whenever it wants/needs to do so
- To avoid this, use the keyword explicit
  - Constructor can then ONLY be used explicitly

```
class MyClass
{
public:
   explicit MyClass( int param );
};
```

Example of ‘explicit’

```
struct MyClass
{
   MyClass( int );
};

MyClass::MyClass( int i )
{
   cout << "Constructor M "
        << i << endl;
}

struct ExplicitClass
{
   explicit
        ExplicitClass( int );
};
```

struct defaults to **public**

```
ExplicitClass::
   ExplicitClass( int i )
{
   cout << "Constructor E "
        << i << endl;
}

int main()
{
   // Call constructor
   MyClass m1(1);
   MyClass m7 = 5;
   // Call constructor
   ExplicitClass e1(100);
   // Cannot do this:
   ExplicitClass e7 = 300;
}
```

Conversion operators

Conversion operator

- Convert **from** a class into something else
- Uses operator overloading syntax
  - See later lecture on operator overloading
- Instead of an operator symbol, the new type name and `()` are used
- e.g. convert to float:

```
operator float() { return …; }
```

- This allows the compiler to convert to the class any time it wants to (without a cast)

Conversion constructor and operator

```
class Converter
{
public:
   // Conversion constructor
   // Convert INTO this class
   Converter( int i = 4 );

   // Conversion operator
   // Convert FROM this class
   operator int();
private:
   int _i;
};
```

```
// Conversion operator
Converter::operator int()
{
   printf( "Converting to int\n" );
   return _i;
}
```

```
// Conversion constructor
Converter::Converter(int i)
{
   _i = i;
}
```

```
int main()
{
   int i = 4;
   // Construction from int
   Converter c1(5);
   Converter c2 = i;
   // Conversion to int:
   int j = (int)c2;
   int k(c2);
   int m = k + c2;
}
```

Breaking the rules

Casting

Unchangable values?

- Here we have constant references passed in
- Can we change x and y?

```
void foo(
    const int& x,
    const int& y )
{
    x = 5;
    y = 19;
}
```

- Can we add anything to allow us to be able to change them?

C++ style casts

Casting away the `const`-ness

- Remove the `const`ness of a reference or pointer

```
void foo( const int& x, const int& y )
{
   int& xr = (int&)(x);
   // Since we cast away const-ness we CAN do this
   xr = 5;
   // or this
   int& yr = (int&)(y);
   yr = 19;
}
```

WARNING!
Do not actually do this unless there is a REALLY good reason!
Casting away `const`-ness is usually very bad

```
void const_cast_example()
{
   int x = 4, y = 2; foo( x, y );
   printf( "x = %d, y = %d\n", x, y );
}
```

const_cast <type> (var)

- Remove the **const**ness of a reference or pointer

```
void foo( const int& x, const int& y )
{
   int& xr = const_cast<int&>(x);
   // Since we cast away const-ness we CAN do this
   xr = 5;
   // or this
   int& yr = const_cast<int&>(y);
   yr = 19;
}
```

WARNING AGAIN
Do not actually do this unless there is a REALLY good reason!
Casting away **const**-ness is usually very bad

```
void const_cast_example()
{
   int x = 4, y = 2; foo( x, y );
   printf( "x = %d, y = %d\n", x, y );
}
```

Four new casts

- **`const_cast<newtype>(?)`**
  - **Get rid** of '**`const`**'ness (or **`volatile`**-ness)
  - No cast needed to **add** '**`const`**'ness (or **`volatile`**)
- **`dynamic_cast<newtype>(?)`**
  - Safely cast a pointer or reference from base-class to sub-class
  - Checks that it really IS a sub-class object
- **`static_cast<newtype>(?)`**
  - Cast between types, converting the type
- **`reinterpret_cast<newtype>(?)`**
  - Interpret the bits in one type as another
  - Mainly needed for low-level code
  - Effects are often platform-dependent
  - i.e. 'treat the thing at this address as if it was a…'

Why use the new casts?

- This syntax makes the **presence** of casts more obvious
  - Casts mean you are '***bending the rules***' somehow
  - It is useful to be able to find all places that you do this
- This syntax makes the **purpose** of the cast more obvious
  - i.e. casting to remove 'const' or to change the type
- Four types give more control over what you mean, and help you to identify the effects
- Sometimes needed: **`dynamic_cast`** provides **run-time** type checking
- Note: Casting a pointer will not usually change the stored address value, only the type. This is **NOT** true with multiple inheritance

static_cast <type> (var)

- **static_cast<newtype>(oldvariable)**
  - Commonly used cast
  - **Attempts** to convert **correctly** between two types
  - Usually use this when **not** removing **const**-ness **and** there is **no** need to check the sub-class type at runtime
  - Works with multiple inheritance (unlike reinterpret!)

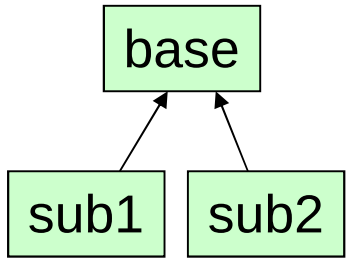
```
void static_cast_example()
{
  float f = 4.1;
  // Convert float to an int
  int i = static_cast<int>(f);
  printf( "f = %f, i = %d\n", f, i );
}
```

`dynamic_cast <type> (var)`

- Casting from derived class to base class is easy
  - Derived class object IS a base class object
  - Base class object **might not** be a derived class object
- **`dynamic_cast<>()`**
  - **Safely** convert **from** a **base-class pointer** or reference **to** a **sub-class pointer** or reference
  - Checks the type at **run-time** rather than compile-time
  - Returns NULL if the type conversion of a **pointer** cannot take place (i.e. it is not of the target type)
  - **There is no such thing as a NULL reference**
    If reference conversion fails, it throws an exception of type **`std::bad_cast`**

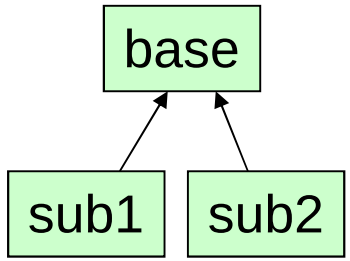
`static_cast` example

```
sub1 s1;
sub1* ps1 = &s1;

// Fine: treat as base class
base* pb1 = ps1;

// Treat as sub-class
sub2* ps2err = static_cast<sub2*>(pb1);
// Static cast: do conversion.
ps2err->func();
// This is an BAD error
// Treating sub1 object as a sub2 object
```



`dynamic_cast` example

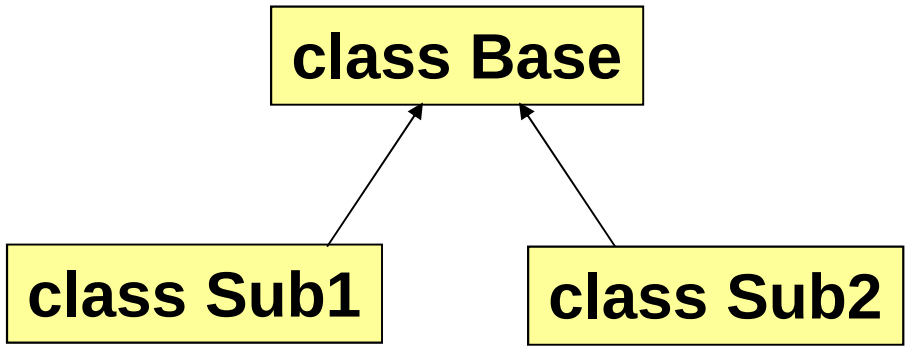
```
sub1 s1;
sub1* ps1 = &s1;

// Fine: treat as base class
base* pb1 = ps1;

// Treat as sub-class
sub2* ps2safe = dynamic_cast<sub2*>(pb1);
// Dynamic cast: runtime check
if ( ps2safe == NULL )
  printf( "Dynamic cast on pb2 failed\n" );
else
  ps2safe->func();
```



Exception thrown by dynamic_cast

```
void foo()
{
   Sub1 s1;
   Base& rb = s1;
   Sub2& rs2 = dynamic_cast<Sub2&>(rb);
   cout << "No exception was thrown by foo()" << endl;
}
int main()
{
   try
   {
       foo();
   }
   catch ( bad_cast )
   { cout << "bad_cast exception thrown" << endl; }
   catch ( ... )
   { cout << "Other exception thrown" << endl; }
}
```

Dynamic cast on a reference



Note: s1 is destroyed properly when stack frame is destroyed

reinterpret_cast<type>(var)

- **reinterpret_cast<>()**
  - Treat the value as if it was a different type
  - Interpret the bits in one type as another
  - Including platform dependent conversions
  - **Hardly ever needed, apart from with low-level code**
  - Like saying “Trust me, you can treat it as one of these”
  - e.g.:

```
void reinterpret_cast_example()
{
  int i = 1;
  int* p = & i;
  i = reinterpret_cast<int>(p);
  printf( "i = %x, p = %p\n", i, p );
}
```

A Casting Question

- Where are casts needed, and what sort of casts should be used?

(Assume **`BouncingBall`** is a sub-class of **`BaseEngine`**)

```
BouncingBall game;
BaseEngine* pGame = &game; // ?
BouncingBall* pmGame = pGame; // ?

BouncingBall game;
BaseEngine& rgame = game; // ?
BouncingBall& rmgame = rgame; // ?
```

Answer : pointers

```
BouncingBall game;
BaseEngine* pGame = &game; // No cast
BouncingBall* pmGame =
    dynamic_cast<BouncingBall*>(pGame);
if ( pGame==NULL ) { /* Failed */ }
```

No cast needed to go from sub-class to base class.

In this case, because the game object really is a **`BouncingBall`**, a **`static_cast`** would have worked.
But would not have checked this – would have been BAD!

Answer : references

```
BouncingBall game;
BaseGameEngine& rgame = game; // No cast
try
{
      BouncingBall& rmgame =
            dynamic_cast<BouncingBall&>(rgame);
}
catch ( std::bad_cast b )
{
      // Handle the exception
      // Happens if rgame is NOT a BouncingBall
}
```

Need to check for any exceptions being thrown for references

Again, in this case, because the **`rgame`** really is a **`BouncingBall`**, a **`static_cast`** would have worked. But would have been BAD!

Repeat: `dynamic_cast`

- **Safely** converts **from** a **base-class pointer** or reference **to** a **sub-class pointer** or reference
  - Checks the type at **run-time** rather than compile-time, to verify it really is a sub-class
- Returns **NULL** if the type conversion of a **pointer** cannot take place
  - i.e. it is not of the target type
- If **reference** conversion fails it **throws an exception** of type **`std::bad_cast`**
  - There is no such thing as a NULL reference

Other casts questions

- When would you use a **`const_cast`**?
- What is the difference between a **`reinterpret_cast`** and a **`static_cast`**?
- When would you use a **`static_cast`**?

Answers

- When would you use a **`const_cast`**?
  - To remove **`const`** or **`volatile`** qualifier
  - This is the only C++ style cast that can do that

- What is the difference between a **`reinterpret_cast`** and a **`static_cast`**?
  - **`reinterpret_cast`** says change the type of the pointer. i.e. keep the bits/bytes that it points to, but treat it as the new type. e.g. **`float*`** to **`int*`**
  - **`static_cast`** says attempt to actually do the conversion between types (e.g. **`float`** to **`int`**)

- When would you use a **`static_cast`**?
  - When none of the others apply
  - i.e. unless casting from base to sub-class, wanting to keep the bits or removing const/volatile

Next lecture

- Exceptions and exception handling
- RAII (Resource Acquisition Is Initialisation)