G52CPP
C++ Programming
Lecture 15

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

IMPORTANT

- No optional demo lecture at 2pm this week
- **Please instead use the time to do your coursework**
- **I will send you something about collision detection (don't worry)**
- We will have the 'using app wizard and creating fully featured programs fast' (with MFC) demo lecture after the Easter holidays

Last Lecture

- Inheritance and constructors
  - Virtual destructors
- Namespaces and scoping
- Some standard class library classes
  - String
  - Input and output

Scoping/using clarification

```
#include <string>
#include <iostream>
using namespace std;

namespace cpp
{
   void MyPrintFunction1()
   {
      // Do something
   }
}

void MyPrintFunction1()
{
   // Do something
}
```

```
using namespace cpp;
```

```
int main()
{
   MyPrintFunction1();
}
```

```
int main()
{
   ::MyPrintFunction1();
}
```

```
int main()
{
   cpp::MyPrintFunction1();
}
```

Scoping/using clarification

```
#include <string>
#include <iostream>
using namespace std;

namespace cpp
{
    void MyPrintFunction1()
    {
        // Do something
    }
}

void MyPrintFunction1()
{
    // Do something
}
```

```
using namespace cpp;
```

```
int main()
{
    MyPrintFunction1();
}
```

g++ namespace.cpp

namespace.cpp: In function “int main()”:

namespace.cpp:22:19: error: call of overloaded “MyPrintFunction1()” is ambiguous

namespace.cpp:22:19: note: candidates are:

namespace.cpp:13:6: void MyPrintFunction1()

namespace.cpp:7:7: void cpp::MyPrintFunction1()

Scoping/using clarification

```
#includ
#includ
using n

namespa
{
    void MyPrintFunction1()
    {
        // Do something
    }
}

void MyPrintFunction1()
{
    // Do something
}
```

$ g++ namespace.cpp
$

The other two work correctly, compiling with no errors
They are unambiguous

General rule: If there is ambiguity it will NOT compile

```
p;
```

```
1();
```

```
int main()
{
    ::MyPrintFunction1();
}
```

```
int main()
{
    cpp::MyPrintFunction1();
}
```

This Lecture

- Standard template library overview
  - By example – NOT VITAL
- Conversion operator
- Friends
- Casting
  - static cast
  - dynamic cast
  - const cast
  - reinterpret cast

Standard Template Library

(You need lectures 17 and 18 to understand how this is implemented)

A large library of template classes and algorithms

Gives speed guarantees for the speed

STL container classes

vector
string
map
list
set
stack
queue
deque
multimap
multiset

- In **std** namespace
- Know that Standard Template Library exists
  - If you go for C++ job interview, learn basics
- These are template classes

  e.g. **vector<int>** for **vector** of **int**s

  Unlike Java, C++ vector class will check types
- Also have iterators
  - Track position/index in a container
  - e.g. to iterate through a container
- And algorithms (over 70 of them)
  - Apply to containers
  - e.g. min(), max(), sort(), search()

Example of using **vector**

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
   vector<char> v(10);
   // 10 elements

   int size = v.size();

   cout << "Size " << size
        << endl;

   // Set each value
   for( int i=0 ; i < size ; i++ )
        v[i] = i;

   // Iterate through vector
   vector<char>::iterator p
                 = v.begin();

   for( ; p != v.end() ; p++ )
        *p += 97;

   // Output the contents
   for( int i=0 ; i < size ; i++ )
        cout << v[i] << endl;

   return 0;
}
```

Conversion operators

Conversion operator

- Convert **from** a class into something else
- Uses operator overloading syntax
  - See lecture 17 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;
}
```

Friends

Accessing private data

```
#include <cstdio>
class TheFriend
{
public:
   void DoSomething(
        Friendly& dest,
        const Friendly& source )
   {    // Copy _i member
        dest._i = source._i + 1;
   }
};

class Friendly
{
public:
   Friendly( int i = 4 )
        : _i(i) { }
private:
   int _i;
};
```

Data is private!

```
void FriendFunc(
   const char* message,
   const Friendly& thing )
{  // Access _i member
   printf( "%s : _i = %d\n",
        message, thing._i );
}


int main()
{
   Friendly d1(2), d2;
   TheFriend f;
   f.DoSomething(d2,d1);
   FriendFunc("d2",d2);
}
```

Accessing private data

```
#include <cstdio>
class TheFriend
{
public:
   void DoSomething(
         Friendly& dest,
         const Friendly& source )
   {     // Copy _i member
         dest._i = source._i + 1;
   }
};

class Friendly
{
public:
   Friendly( int i = 4 )
         : _i(i) { }
private:
   int _i;
};
```

Do something to 'Friendly'

Data is private!

```
void FriendFunc(
   const char* message,
   const Friendly& thing )
{  // Access _i member
   printf( "%s : _i = %d\n",
       message, thing._i );
}

int main()
{
   Friendly d1(2), d2;
   TheFriend f;
   f.DoSomething(d2,d1);
   FriendFunc("d2",d2);
}
```

`friend`s

- Classes can grant access to their **`private`** member data and functions to their **`friends`**
- The class still maintains control over which classes and functions have access
- The **`friends`** of a class are treated as class members **for access purposes** – although they are **not** members
- **Declare** your friends within your class body and use the keyword **`friend`**

friend function

```
class Friendly
{
// Make function a friend
friend void FriendFunc(
   const char* message,
   const Friendly& thing );

public:
   Friendly(int i=4) : _i(i)
   {}

private:
   int _i;
};
```

```
void FriendFunc(
   const char* message,
   const Friendly& thing )
{
   printf(
       "%s : _i = %d\n",
       message,
       thing._i );
}
```

```
int main()
{
   Friendly d1(2), d2;
   FriendFunc( "d1", d1 );
   FriendFunc( "d2", d2 );
}
```

friend class

.h file:

```
class Friendly;

class TheFriend
{
public:
   void DoSomething(
        Friendly& dest,
        const Friendly& source);
};
```

Forward declaration of class

```
class Friendly
{
friend class TheFriend;
public:
   Friendly(int i=4) : _i(i){}
private:
   int _i;
};
```

.cpp file:

```
void
   TheFriend::DoSomething(
   Friendly& dest,
   const Friendly& source )
{
   dest._i =
       source._i + 1;
}
```

Note: Could make this a static member function since it does not need to access or alter any member data

```
int main()
{
   Friendly d1(2), d2;
   TheFriend f;
   f.DoSomething( d2, d1);
}
```

Breaking the rules

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;
}


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

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

`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;
}


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

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

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)

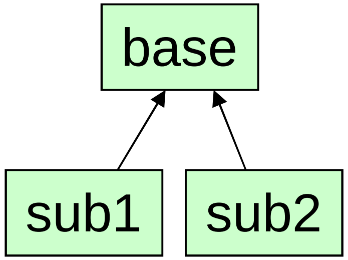
- **`static_cast<newtype>(oldvariable)`**
  - Commonly used cast
  - **Attempts** to convert **correctly** between two types
  - Usually use this when **not** removing **`const`**-ness **and** there is **no** need to check the sub-class type at runtime
  - Works with multiple inheritance (unlike reinterpret!)

```
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`** (see Thursday lecture)

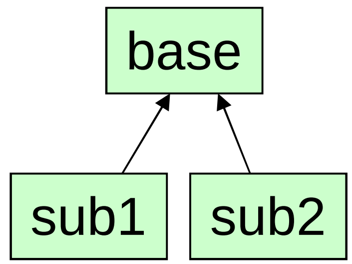
`static_cast` example




```
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

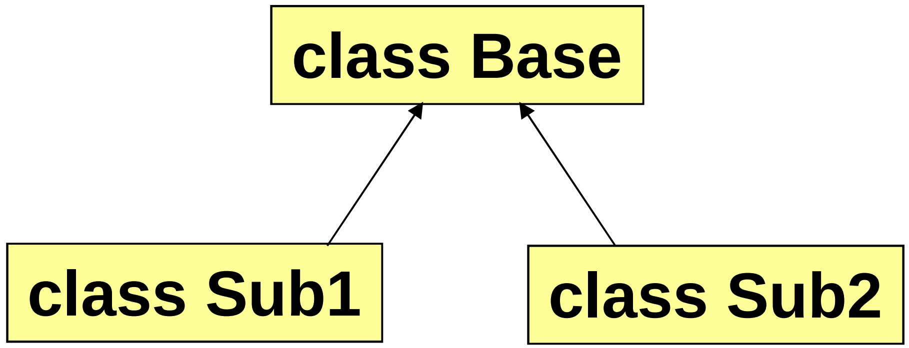
```
sub1 s1;
sub1* ps1 = &s1;

// 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`

Dynamic cast on a reference

```
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; }
}
```



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)