Part III

C++11/14/17 continued

Outline

Delegated constructor

- In C++03, if there is common code between constructors, you have to create an `init()` or similar function:

```
class Car{
  Car(){
    allocateSpace();
  }
  Car(const Car& c){
    allocateSpace();
    // Now copy values from c
  }
  void allocateSpace(){ ... }
};
```

You cannot call a constructor from another constructor.

Delegated constructor

- In C++11, we can do the following:

```
class Car{
  Car(){
    // Allocate space ...
    // Throw any necessary exceptions
  }
  Car(const Car& c) : Car(){
    // Space already allocated by default constructor
    // Now copy values from c
  }
};
```

- That is, we call the constructor of an empty `Car` from the copy-constructor.
- This is now somewhat cleaner.

Disabling default methods

- Recall that C++ defines default constructors, copy-constructors, copy-assignments, move operators, move-assignment operators, and destructors as needed, for any class you define.
- In some cases this is undesired behaviour as it permits unexpected code.
- In C++11 you can disable the creation of these:

```
class Rational{
public:
  // Initialise to n/1
  Rational(int n) : num(n), denom(1) { }
  // Initialise to n/d
  Rational(int n, int d) : num(n), denom(d) { }
  // Uninitialized Rational makes no sense.
  Rational() = delete;
private:
  int num; int denom;
};
int main(void){
  Rational a; // Invalid
}
```

Explicitly enabling default methods

- Conversely, you may have written a non-default constructor (or other method), but want the default constructor behaviour as well:

```
class B{
public:
  B(int x) : data(x) {}
  B() = default;
private:
  int data;
}
B b; // Only legal because of = default line.
```

- This makes it explicitly obvious that you are relying on the default behaviour, not anything subtlely different.
- Without the `B() = default;` the compiler would not define this constructor.

Virtual functions

- Virtual functions are necessary for polymorphic classes.
- We can specify in the base class that a function is virtual and then functions in derived classes are marked as `override`:

```
class Vehicle{
public:
  virtual void turnIgnition(bool)const;
};

class Car : public Vehicle{
public:
  void turnIgnition(bool)const override;
};
```

- It is an error to specify `override` for a function that is not overriding another one.
- The main reason for this syntax is clarity for the developer about the intent of the class/function.

Final functions

- Sometimes we want to prevent virtual functions from being overridden.

```
class Car : public Vehicle{
public:
  virtual void turnIgnition(bool)const final;
};
class FordPrefect : public Car{
public:
  void turnIgnition(bool)const override; // Error
};
```

- We have prevented any further derived classes from `Car` from overriding the `turnIgnition` function.
- This *may* provide some performance improvement, because the compiler knows that `car->turnIgnition(true)` always calls `Car::turnIgnition`, never any overridden version.
- This improvement is unlikely to be important in practice, though; measure if you think it is important.

Final classes

- Sometimes we want to prevent classes from being derived from.

```
class Car final : public Vehicle{
  ...
};
```

- Now, no class can derive from `Car`.
- For both uses of `final`, only use it if it makes sense from a design perspective, i.e. if there is a logical reason why no one should ever derive from the class, or override a function further.
- See `Examples/final.C`

Part IV

C++11/14/17

Outline

Static assert

- When developing complex templated classes, you will often make assumptions on the templated-over types that need to be checked.
- If they are not checked, they will either lead to screeds of compiler-errors or weird run-time behaviour.
- Use `static_assert`: (See `Examples/static_assert.C`)

```
template<int D>
class A{
  static_assert(D >= 0, "D must be positive");
};
int main(void){
  A<+1> a;
  A<-1> b;
}
```

- This will cause a compile error:

```
static_assert.C: In instantiation of  class A<-1>   :
static_assert.C:10:9:   required from here
static_assert.C:4:3: error: static assertion failed:
                     D must be positive
```

Static assert ctd

- The expression for `static_assert` must be capable of being evaluated at compile-time.
- If it is not, the compiler will complain.
- For example, the following is not valid:

```
int e = 0;
template<int D>
class A{
  static_assert(e >= 0, "e must be positive");
};
```

(although using `const int e` would be OK).

- The previous example is very simple; more complex tests can check that a template parameter is an arithmetic type, for example.

Outline

Lambda functions

- In C++03 we had to create functors, which were classes with an `operator()` overload, and could therefore act as a function.
- In C++11 we can create functors in-place, called *lambda functions*.

```
std::vector<int> a{ -1, 5, 10, -9, 12, 3  };
int cutoff = 5;
std::for_each(a.begin(), a.end(),
  [cutoff](int x){if(x < cutoff) std::cout << x << ",";}
);
```

  will only display values in `a` that are less than the cut-off 5.
- To unpack the lambda function:
  - Variables from the external scope needed in the lambda function have to be captured: `[cutoff]`
  - If we do not need to capture any variables, specify `[]`.
  - The list-member has to be passed to the lambda function: `(int x)`
  - (The syntax of `for_each` requires that the functor take a single parameter, of the element type.)
  - The remainder of the function body is in {}.

Lambda functions ctd

- So far, this looks overly complicated; the same could be achieved with a `for` loop.
- However, we can use a different algorithm:

```
std::transform(a.begin(), a.end(), b.begin(),
  [cutoff](int x){return (x < cutoff) ? x : 0;}
  );
```

  which copies `a` into `b`, except that it replaces values larger than `cutoff` with zeros.
- Or:

```
std::sort(a.begin(), a.end(),
          [](int a, int b){return (a % 10 < b % 10);}
         );
```

  to sort `a` according to the units-digits of its elements.
- See `Examples/lambda.C`

Lambda functions ctd

- In some cases lambda functions can make the code more compact and easy to read.
- In some cases they can make it substantially more complicated to read.
- A few extra syntax notes:
  - The capture list can be given as
    - `[&]`: all variables captured by reference, or
    - `[&, a, b]`: captures all local variables other than `a` and `b` by reference, or
    - `[=]`: all variables captured by value, or
    - `[=, &a, &b]`: captures all local variables by value except for `a` and `b` which are captured by reference.
  - If there are no parameters to pass to the lambda function, the `()` can be omitted.
  - Parameter values are captured at the point where the lambda function is created.

Functors

- In the first lecture series we discovered function pointers and user-defined functors, but never combined the two.
- C++11 makes this easier with `std::function`
- This allows us to create functors with particular signatures from existing functions.

Simple functionality:

```
#include <functional>
double operate(double x, double y){
  return x + 2*y;
}

std::function<double(double, double)> op = operate;
std::cout << "operate(3.2, 4.3) = " << op(3.2, 4.3);
```

Functors

- We can also bind some of the parameters to fixed values:

```
std::function<double(double)> op2 =
std::bind(operate, std::placeholders::_1, 4.5);

std::cout << "operate(1, 4.5) = " << op2(1) << std::endl;
```

- `op2` is now a function that takes a single parameter `x`, and evaluates `operate(x, 4.5)`.
- We can also repeat placeholders, to form a functor of a different type:

```
std::function<double(double)> op3 =
std::bind(operate, std::placeholders::_1,
                   std::placeholders::_1);

std::cout << "operate(1, 1) = " << op3(1) << std::endl;
```

Functors for member functions

- We can even do something similar for member functions:

```
struct Object{
  double func(double x, double y)const {
    return x + y * data;
  }

  double data;
};

Object o;
o.data = 10;

std::function<double(double)> op4 =
       std::bind(std::mem_fn(&Object::func),
                 &o, std::placeholders::_1, 3.0);

std::cout << "o.func(4, 3) = " << op4(4) << std::endl;
```

See **Examples/function.C**

Functors for member functions

- Note that the first parameter is a pointer to an `Object`. This is the object that will be acted on.
- The class pointer could also be a placeholder.
- Note that once you have a `std::function<double(double)>`, it doesn't matter what the contained function is; it can be copied around arbitrarily.
- However, any pointers to objects are stored as pointers, so if the object changes, the action of the functor could also change.
- Further, passing around an object pointer inside a functor may lead to surprising side-effects. (See `Examples/function.C`, and the `updateData()` and `func()` calls.
- Using functors introduces an extra level of overhead; if using them makes your code clearer, then do so unless/until you discover that they are a bottleneck.

Outline

Shared pointers

- In some cases you may allocate memory that needs to be referred to by multiple objects, any of which may be deleted at any time.
- In order not to leak memory, the last object to be deleted should also free the memory.
- For example, consider an `Array` object that allows shallow copies to be made, and/or sub-`Array`s to be created:

```
Array shrinkArray(const Array& a){
  Box r = a.extent();
  Array b = a(shrink(region, 1));
  return b;
}
```

- In order to avoid pointless copying of data, line 3 makes `b` refer to the same block of memory as `a`.
- (Subject to `b` covering a smaller grid than `a`, i.e. clever indexing has to be employed within element access to `b`).

Shared pointers

- You can use a **std::shared_ptr** to handle the allocated memory.
- This includes a reference counter that ensures the memory pointed to is freed when its last instance goes out of scope.

```
struct A{
  std::shared_ptr<int> data;
  A(){
    data = std::shared_ptr<int>(new int[10],
                                std::default_delete<int[]>());
  }
  A(const A& a){
    data = a.data;
  }
  ~A(){
    data.reset();
    data = nullptr;
  }
  int& operator[](int i){
    return data[i];
  }
};
```

Shared pointers

- A `std::shared_ptr` will use the `delete` operator on its contained type by default; if a different destructor is required, supply it at construction time, hence the `std::default_delete<int[]>()` above.
- Strictly, the code in the destructor is not needed; it just causes the pointer to be freed (if it's the last instance holding the pointer), and then set to the null pointer.
- (However, it is needed for the example code `Examples/shared_ptr.C`, which calls the destructor explicitly.)
- Detailed explanation of what happens in various cases can be found in the example code.
- The `shared_ptr` implements the operations you would expect from a normal pointer: `[] -> *` conversion to `(bool)`

Shared pointers

- If `new int[10]` throws an exception, the code given may leak. However, there is no simple solution until C++17.
- At C++17, the following works correctly:

```
std::shared_ptr<int[]> a(new int[10])
```

as the `delete[]` operator is used when it goes out of scope.

Outline

Regular expressions

- You may have used regular expressions within Bash, Emacs, vi(m), etc.
- They are now available in C++.
- On the whole, you should not be using regular expressions in scientific programs; settings files should be parsed using an external library.
- Various regular expression notations are available, the default is a variant of ECMA-262 (similar to that used in JavaScript).
- Alternatives are those used by `awk`, `grep`, POSIX, Extended POSIX.
- A single, reasonably complex, example will suffice.

Regular expressions

Examples/regex.C:

```
#include <regex>
int main(void){
  std::string text = "It was the best of times; it was the
    worst of times.";

  std::regex pat("([[:alpha:]]*)st ");
  std::smatch sm;
  while(std::regex_search(text, sm, pat)){
    std::cout << sm.str() << " sub-expression " << sm[1] <<
    std::endl;
    text = sm.suffix();
  }
}
```

This will produce output:

```
best  with sub-expression = be
worst  with sub-expression = wor
```

sm[0] represents the text matched by the full regular expression.
Note: This example does not work in g++-4.8; versions $\geqslant$ 5.0 do.

Outline

Type traits

- When using templated functions, we sometimes want different functionality based on what form a type takes.
- Simple example:

```
template<typename T>
void print(const T& s){
  if(std::is_arithmetic<T>::value){
    std::cout << "Number: " << s << std::endl;
  }
  else if(std::is_pointer<T>::value){
    std::cout << "Pointer " << std::hex << s << std::endl;
  }
}
```

- These are known as *type traits* and there is a long list of possible traits which allow inspection of a type.
- They may be useful in conjunction with `static_assert`.
- See `Examples/type_traits.C`

Type traits

- `is_void<X>`
- `is_integral<X>`
- `is_floating_point<X>`
- `is_array<X>`
- `is_fundamental<X>`
- `is_scalar<X>` (not class or function)
- `is_member_pointer<X>`
- `is_const<X>`
- `is_abstract<X>` Does `X` have a pure virtual function?
- `is_default_constructible<X>` Can `X` be constructed with no parameters?
- Many others are available...

enable_if

- Sometimes we want certain templated functions only to be compiled if certain conditions hold.
- The construct:

```
std::enable_if<bool cond, typename T = void>
```

has a member called `type` (of type `T`) iff `cond` is `true`
- This is usually used in a SFINAE context (see later lecture) to provide different versions of a function depending on the type being passed.
- Consider a templated `Vector<T>` which should work with the following:

```
Vector<double> a(9.6);
Vector<int> b(10);
Vector<int> c(a);
```

- The second line should initialize all elements of `b` to be 10.
- The third line should copy values from `a` into `c` (Note that one contains `double` and the other `int`).

enable_if ctd

We end up with two templated functions in `Vector`.

See `Examples/enable_if.C`

```
template<typename S>
Vector(const S& s,
       typename std::enable_if<std::is_arithmetic<S>::value,
                               int>::type = 0){
  for(unsigned int i=0 ; i < 10 ; i++){
    m_data[i] = s;
  }
}
template<typename S>
Vector(const S& s,
       typename std::enable_if<!std::is_arithmetic<S>::value,
                                int>::type = 0){
  for(unsigned int i=0 ; i < 10 ; i++){
    m_data[i] = s[i];
  }
}
```

enable_if ctd

- If `S` is an arithmetic type, then `enable_if<...>::type` is an integer parameter, with default value 0.
- If `S` is not an arithmetic type, then `enable_if<...>::type` is not a type, and the function is ill-defined.
- However, SFINAE means that this templated function does not raise an error but the compiler merely discards it from the set of available functions that it considers.
- The opposite logic works for a `Vector<int>` for the second function.
- Thus, the first function is called if an arithmetic type is passed, and the second is called if a non-arithmetic type is used.