CSCI251/CSCI851 Autumn-2020
Advanced Programming (**SGPc**)

C++ good practice:

Part C

On we go …

- The last set finished on page 248 so we will pick up from there…

Page 255 : Who are the users?

- Users can be:
  - People who use/run the application we write.
  - Programmers who make use of the code/classes we write in developing their own applications.
    - This may be the class developer, but it might not be.
- It's usually going to be clear from the context which user type we are referring to.
- Just as application developers should think about the needs of their target market, so the (application) users, code/class developers should think about the needs of their target market, the code users.

Pages 258-259: `const` calls

- `const` member functions cannot change the object on which they are called.
  - The `this` pointer is, by default, a `const` pointer to the non`const` version of the class type.
  - The `const` modifier for member functions changes the type of `this` to being a `const` pointer to `const` type.
- So objects that are `const` may only call `const` member functions.
- Similarly calls through references or pointers to `const` objects can only be to `const` member functions.

Page 261: Nonmember interface functions → same header as the class

- Sometimes when we develop classes we develop auxiliary functions that work with the class, so are art of the interface, but aren't in the class themselves.
- It makes sense that these should be declared in the same header as the class itself.

Pages 263-265: Some notes on constructors

- A default constructor, this is the synthesized default one, is automatically generated only if the class has no constructors declared, default or non-default.
- Classes with data members of built-in or compound types should usually only rely on the synthesized default constructor if all such data members have in-class initializers.
- Constructors shouldn't override in-class initializers except to use a different initial value.
  - If there aren't in-class initializers, each constructor should explicitly initializer every data member of built-in type.

Page 269: `class` vs `struct`

- The only difference between these keywords in declaring an abstract data type is the default access level.
- Style suggests if all of the members are to be `public`, we use `struct`, and if anything in there is to be `private` we use `class`.

Page 270: Where are my `friends`?

- It's usually a good idea to list all your `friend` declarations together, and usually at the end or at the beginning of the class definition.
- The `friend` declaration is only about access, there needs to be a general declaration too, outside the class, prior to the function being used.
  - Compilers won't necessarily enforce this rule though ☹
- To make the friends visible to users of our class we usually declare the friends outside the class but in the same header as the class itself.
- Note, from page 280, each class determines who it's friends are.

Page 270: Why encapsulate?

- Two primary advantages:
  - The code of the (class) user cannot (easily) corrupt the state of an encapsulated object.
  - The implementation of the encapsulated class can change without needing to change the user level code.
    - Unless we change factors like the names of public functions, arguments to public functions, meaning of arguments/functions.
- Changes to code within the class still require recompilation to take effect.

Page 274: In-class initializer syntax

- In-class initializers are C++11 functionality.
- There are two forms for using an in-class initializer:
- In curly braces:

```
vector<Screen> screens{Screen(24, 80, ' ')};
vector<Screen> screens{24, 80, ' '};
```

- Or following an = sign.

```
vector<Screen> screens = {24, 80, ' ')};
```

Page 276: Returning `*this` ... Overloading const differ

- A `const` member function that returns `*this` as a reference should have a return type that is a reference to `const`.
- In the textbook example this is used in illustrating overloading with `const` being the difference.

```
Screen &display(std::ostream &os)
{ do_display(os); return *this; }   // non-const version

const Screen &display(std::ostream &os) const
{ do_display(os); return *this; }   // const version
```

Page 277: Private utility functions

- The `do_display()` function on the previous slide is a private function, we call it from within the two display functions defined within the class.

```
private:
        void do_display(std::ostream &os) const {os << contents;}
```

- Several reasons to do this…
  - Writing it once, particularly since while initially `do_display()` may be minimal as the class grows it saves rewriting code in several places.
  - Easier for debugging.
  - No overhead in the call since it's implicitly inline.
    - The code is organised without the function call overhead.

Page 278: Different classes

- Classes with identical data members and member functions are still different types.

```
struct One {
   int X, Y;
   float Z;
};
```

```
struct Two {
   int X, Y;
   float Z;
};
```

- This means, for example, that you cannot just directly copy/assign an element of one to the other.

```
One obj1;
Two obj2;
obj1=obj2;
```

Pages 284-285: Some scope notes

- Member function definitions are processed after the compiler processes all of the declarations in the class.
- This means we don’t have to worry about carefully ordering definitions so as to only use functions listed earlier in our file.
- Typically we would define type names at the start of a class though, so members using that type know what it means.

Page 286: Un-hiding

- Hiding is when we use a member function parameter name that is the same as the name of a data member.
- We can avoid the hiding using the scope resolution operation or the `this` pointer.

```
class X
{
private:
        int number;
public:
        void function(int);
}

void X::function(int number){
       cout << number * X::number << endl;
       cout << number * this->number << endl;
 }
```

Pages 288-289: Constructor initializers

- Constructor initializer lists have to be used to provide values for members that are const, reference variables, or of a class type that does not have a default constructor.

```
class ConstRef {
public:
        ConstRef(int ii);
private:
        int i;
        const int ci;
        int &ri;
}
```

```
ConstRef::ConstRef(int ii)
{
        i=ii;
        ci=ii; // error: cannot assign to const.
        ri=i; // error: ri was never initialized.
}
```

```
ConstRef::ConstRef(int ii) : i(ii), ci(ii), ri(i) { }
```

Pages 289-290: More on constructor initializers

- Using constructor initializers improves efficiency.
  - Rather than initialising and then assigning, all of it is being done at once.
- Furthermore, as on the previous slide, we avoid the problem of not initialising something that must be initialised.
- One cautionary note is that the order of construction is as per the declaration order in the class, not as per the order in the initialisation list.
  - It's a good idea to keep those orders consistent.
  - And, if there are dependencies you need to be careful about what that order is; so generally it's also a good idea to avoid using members to initialise other members if you can.

Pages 290-291: Default arguments in constructors

- If you provide a constructor that has default arguments for all of its parameters then this is also defining the default constructor.
- We don’t want ambiguity and having a default constructor synthesized by the compiler would result in ambiguity.

Pages 291-292: Constructor delegation

- This can be thought of as a constructor calling another constructor within the same task.
  - One constructor delegates the responsibility of part of the object initialization to another constructor.
- To use delegation you include the delegated-to constructor in the initialisation list of the delegating constructor.

```
Sales_data(string s, unsigned count, double price) :
bookNo(s), units_sold(count), revenue (cout*price){}

Sales_data(): Sales_data(“”, 0, 0){}
Sales_data(string s): Sales_data(s, 0, 0) {}
```

Pages 293-294: Default constructor role

- The default constructor is needed in various situations.
- It's almost always right to provide a default constructor when you define some other constructor.
- Warning:

```
MyClass object();   // wrong
MyClass object;     // right
```

Pages 294-297: Implicit ADT conversions

- If you have a constructor for a class `X` and if takes a single argument of type `Y`, then that constructor is carrying out an implicit conversion from `Y` to `X`.
- We may be throwing data away though.
- For `Sales_data` we had the constructor, …

```
Sales_data(string s): Sales_data(s, 0, 0) {}
```

- If `Sales_data` has a member function `combine` we could do something like

```
string word="Null item";
item.combine(word);
```

- I guess we could use something like …

```
string word="Tigger is a 10 year old fluffy cat";
Cat cat1(word);
```

- … to provide a conversion from a string to a `Cat`, with parsing used to extract the data members for the Cat.

- A more detailed example may be needed on implicit ADT conversions.

Pages 298: Aggregate classes

- A class characterised by 4 properties:
  - The data members are all public.
  - No defined constructors.
  - No in-class initializers, i.e. setting values in class definition.
  - No base classes or virtual functions.
- Initialisation uses a braced list of member initializers.

```
struct Data{
   int value;
   string thing;
   };
```

```
Data obj={ 0, "Bob"};
```

Pages 299-300: Literal classes

- These can be used in defining new types that can be used in `constexpr`.
- A class characterised by 4 properties:
  - The data members all have literal type.
  - There must be at least one `constexpr` constructor.
  - If a data member has an in-class initializer either:
    - The initializer is a constant expression,
    - Or, for an ADT data member, the initializer using that ADT's `constexpr` constructor.
  - The default definition must be used for the destructor.

Page 300-304: `Static` Class members

- The `static` keyword is used only in the declaration in the class body, it isn't needed outside.
- Data members that are `static` aren't defined when we create objects of the class, so aren't initialized by the class' constructors.
- Typically we cannot initialize `static` members inside the class, so they are defined outside.
- To help ensure the object is defined exactly once, the definition of `static` data members is best put in the same file as the definitions of the class noninline member functions.

- The book describes best practice as being that even when a `const static` data member is initialized in the class body, that member should ordinarily be defined outside the class definition.

```
class Account{
   …
private:
   static constexpr int period = 30;
}
constexpr int Account::period;
```

- This ends Part I of the textbook.
- The next set of the C++ good practice guide will start looking at the C++ library.