COMP110: Principles of Computing

Smart Pointers

Statics and singletons

Static members

- Class members (methods and fields) can be marked as `static`

Static members

- Class members (methods and fields) can be marked as `static`
- They are then **shared** across **all instances** of the class

Static members

- Class members (methods and fields) can be marked as `static`
- They are then **shared** across **all instances** of the class
- Static members can be accessed from **outside** the class using `::` notation

The singleton pattern

- A **singleton** class has a **single static instance**

The singleton pattern

- A **singleton** class has a **single static instance**
- A **static** `getInstance` function returns a **reference** to the instance

The singleton pattern

- A **singleton** class has a **single static instance**
- A **static** `getInstance` function returns a **reference** to the instance
- There is **one and only one** instance — creating new instances is not allowed, neither is destroying the existing instance

In TextureManager.h

```
class TextureManager
{
public:
    static TextureManager& getInstance()
    { return instance; }

    Texture* getTexture(const std::string& name);

private:
    TextureManager() { };
    static TextureManager instance;
};
```

In `TextureManager.h`

```
class TextureManager
{
public:
    static TextureManager& getInstance()
    { return instance; }

    Texture* getTexture(const std::string& name);

private:
    TextureManager() { };
    static TextureManager instance;
};
```

In `TextureManager.cpp`

```
#include "TextureManager.h"

TextureManager TextureManager::instance;
```

Key points:

Key points:

- Constructor is **private** to prevent creation of extra instances

Key points:

- Constructor is **private** to prevent creation of extra instances
- `instance` and `getInstance` are **static**, everything else is **not static**

Key points:

- Constructor is **private** to prevent creation of extra instances
- `instance` and `getInstance` are **static**, everything else is **not static**
- Static variables must be **instantiated** in a `.cpp` file
  - Optionally with constructor parameters

Key points:

- Constructor is **private** to prevent creation of extra instances
- `instance` and `getInstance` are **static**, everything else is **not static**
- Static variables must be **instantiated** in a `.cpp` file
  - Optionally with constructor parameters

Example usage:

```
Texture* texture = TextureManager::getInstance().
    getTexture("player.png");
```

Statics and initialisation

- Static members are initialised at the **very start of the program**

Statics and initialisation

- Static members are initialised at the **very start of the program**
  - I.e. **before** `main()` is entered

Statics and initialisation

- Static members are initialised at the **very start of the program**
  - I.e. **before** `main()` is entered
- The order of initialisation is **undefined**

Statics and initialisation

- Static members are initialised at the **very start of the program**
  - I.e. **before** `main()` is entered
- The order of initialisation is **undefined**
  - **Do not** make your static initialisation code rely on other statics having been initialised already — this **will** cause obscure bugs!

Statics and initialisation

- Static members are initialised at the **very start of the program**
  - I.e. **before** `main()` is entered
- The order of initialisation is **undefined**
  - **Do not** make your static initialisation code rely on other statics having been initialised already — this **will** cause obscure bugs!
- Further reading: `https://isocpp.org/wiki/faq/ctors#static-init-order`

Statics and initialisation

- Static members are initialised at the **very start of the program**
  - I.e. **before** `main()` is entered
- The order of initialisation is **undefined**
  - **Do not** make your static initialisation code rely on other statics having been initialised already — this **will** cause obscure bugs!
- Further reading: `https://isocpp.org/wiki/faq/ctors#static-init-order`
- Consider using the **construct on first use** idiom if your singleton class has a complex constructor

Smart pointers

Motivation

- Pointers in C++ are powerful, but with great power comes great responsibility!

Motivation

- Pointers in C++ are powerful, but with great power comes great responsibility!
- In particular, responsibility for the **lifetime** of object instances, i.e. remembering to call `delete` at the right time

Motivation

- Pointers in C++ are powerful, but with great power comes great responsibility!
- In particular, responsibility for the **lifetime** of object instances, i.e. remembering to call `delete` at the right time
- C++11 introduced **smart pointers** to try and make this easier

Unique pointer

- Defined in standard header file `<memory>`

Unique pointer

- Defined in standard header file `<memory>`
- `std::unique_ptr` has **sole ownership** of the instance to which it points

Unique pointer

- Defined in standard header file `<memory>`
- `std::unique_ptr` has **sole ownership** of the instance to which it points
- I.e. two `unique_ptr`s **cannot** point to the same instance

Unique pointer

- Defined in standard header file `<memory>`
- `std::unique_ptr` has **sole ownership** of the instance to which it points
- I.e. two `unique_ptr`s **cannot** point to the same instance
- Use `std::make_unique<T>` instead of `new T`

Unique pointer

- Defined in standard header file `<memory>`
- `std::unique_ptr` has **sole ownership** of the instance to which it points
- I.e. two `unique_ptr`s **cannot** point to the same instance
- Use `std::make_unique<T>` instead of `new T`
- The instance is destroyed when the `unique_ptr` goes **out of scope**, or when it is assigned to point to a different instance

Unique pointer example

```
#include "stdafx.h" // --> #include <memory>

class MyClass { };

int main()
{
    std::unique_ptr<MyClass> p
        = std::make_unique<MyClass>();
    return 0;
    // p is automatically deleted here
}
```

Aside: automatic type inference

- When declaring a **local variable** with an initial value, can use `auto` in place of the type

Aside: automatic type inference

- When declaring a **local variable** with an initial value, can use `auto` in place of the type
- The compiler **infers** the type from the type of the initial value

Aside: automatic type inference

- When declaring a **local variable** with an initial value, can use `auto` in place of the type
- The compiler **infers** the type from the type of the initial value

```
// The following are equivalent:
std::unique_ptr<Duck> duck = std::make_unique<Duck>();
auto duck = std::make_unique<Duck>();
```

Aside: automatic type inference

- When declaring a **local variable** with an initial value, can use `auto` in place of the type
- The compiler **infers** the type from the type of the initial value

```
// The following are equivalent:
std::unique_ptr<Duck> duck = std::make_unique<Duck>();
auto duck = std::make_unique<Duck>();
```

- Careful — this can make your code **more readable** or **less readable** depending on how and where you use it!

Vectors of objects revisited

From last time:

Vectors of objects revisited

From last time:

- `std::vector<Duck>`: when the vector changes size, instances are **copied** and the old ones **destroyed**

Vectors of objects revisited

From last time:

- `std::vector<Duck>`: when the vector changes size, instances are **copied** and the old ones **destroyed**
- `std::vector<Duck*>`: no copying and destroying, but you must remember to `delete` elements on removal

Vectors of objects revisited

From last time:

- `std::vector<Duck>`: when the vector changes size, instances are **copied** and the old ones **destroyed**
- `std::vector<Duck*>`: no copying and destroying, but you must remember to `delete` elements on removal

But now:

Vectors of objects revisited

From last time:

- `std::vector<Duck>`: when the vector changes size, instances are **copied** and the old ones **destroyed**
- `std::vector<Duck*>`: no copying and destroying, but you must remember to `delete` elements on removal

But now:

- `std::vector<std::unique_ptr<Duck>>`: class instances stay put, but are automatically destroyed when removed from the `vector`!

Unique pointers can't be copied

```
auto p = std::make_unique<MyClass>();
auto q = p;
```

Unique pointers can't be copied

```
auto p = std::make_unique<MyClass>();
auto q = p;
```

- This won't compile!

Unique pointers can't be copied

```
auto p = std::make_unique<MyClass>();
auto q = p;
```

- This won't compile!

C2280 'std::unique_ptr<MyClass,std::default_delete<_Ty>>::unique_ptr(const std::unique_ptr<_Ty,std::default_delete<_Ty>> &)': attempting to reference a deleted function

Unique pointers can't be copied

```
auto p = std::make_unique<MyClass>();
auto q = p;
```

- This won't compile!

```
C2280 'std::unique_ptr<MyClass,std::default_delete<_Ty>>::unique_ptr(const std::unique_ptr<_Ty,std::default_delete<_Ty>> &)': attempting to reference a deleted function
```

- Translation: assigning one `unique_ptr` from another is forbidden

Aside: default and deleted functions

- As standard, C++ classes come with constructors and methods for **copying** and **moving** instances

Aside: default and deleted functions

- As standard, C++ classes come with constructors and methods for **copying** and **moving** instances
- You (the class developer) can **override** these methods, make them **protected** or **private**, or **delete** them entirely (e.g. to forbid copying)

Aside: default and deleted functions

- As standard, C++ classes come with constructors and methods for **copying** and **moving** instances
- You (the class developer) can **override** these methods, make them **protected** or **private**, or **delete** them entirely (e.g. to forbid copying)
- Further reading:
  - `http://www.stroustrup.com/C++11FAQ.html#default`
  - `http://en.cppreference.com/w/cpp/language/classes`

Unique pointers can be moved

```
auto p = std::make_unique<MyClass>();
auto q = std::move(p);
```

Unique pointers can be moved

```
auto p = std::make_unique<MyClass>();
auto q = std::move(p);
```

- Now `q` has **ownership** of the instance

Unique pointers can be moved

```
auto p = std::make_unique<MyClass>();
auto q = std::move(p);
```

- Now `q` has **ownership** of the instance
- `p` no longer refers to this instance

Unique pointers can be moved

```
auto p = std::make_unique<MyClass>();
auto q = std::move(p);
```

- Now `q` has **ownership** of the instance
- `p` no longer refers to this instance
- In fact `p` becomes null

Unique pointers can be moved

```
auto p = std::make_unique<MyClass>();
auto q = std::move(p);
```

- Now `q` has **ownership** of the instance
- `p` no longer refers to this instance
- In fact `p` becomes null
- Another way you are allowed to move `unique_ptr`s is by **swapping**:

```
auto p = std::make_unique<MyClass>("hello");
auto q = std::make_unique<MyClass>("world");
std::swap(p, q);
```

Shared pointers

- Unique pointers have their uses, but uniqueness is a big restriction

Shared pointers

- Unique pointers have their uses, but uniqueness is a big restriction
- C++ also has `std::shared_ptr`, which does allow multiple pointers to the same instance

Shared pointers

- Unique pointers have their uses, but uniqueness is a big restriction
- C++ also has `std::shared_ptr`, which does allow multiple pointers to the same instance
- **Reference counting** ensures the instance is destroyed when **all** **`shared_ptr`s** to it have gone away (and not before!)

```
int main()
{
    bool flag = true;
    std::shared_ptr<MyClass> p = nullptr;

    if (flag)
    {
        auto q = std::make_shared<MyClass>();
        p = q;
    }
    /* q is out of scope, but p still holds a
        reference to the instance so it stays alive */

    if (p)
        p->doSomething();

    return 0;
    /* p is out of scope now, so the instance is
        destroyed */
}
```

Reference counting

- Objects managed by `shared_ptr` have a **reference count**, initially 0

Reference counting

- Objects managed by `shared_ptr` have a **reference count**, initially 0
- When a `shared_ptr` is made to point to the instance, its reference count is **incremented**

Reference counting

- Objects managed by `shared_ptr` have a **reference count**, initially 0
- When a `shared_ptr` is made to point to the instance, its reference count is **incremented**
- When a `shared_ptr` no longer points to the instance (because it goes out of scope or is reassigned), the reference count is **decremented**

Reference counting

- Objects managed by `shared_ptr` have a **reference count**, initially 0
- When a `shared_ptr` is made to point to the instance, its reference count is **incremented**
- When a `shared_ptr` no longer points to the instance (because it goes out of scope or is reassigned), the reference count is **decremented**
- When the reference count is decremented to 0, the instance is destroyed

FALMOUTH UNIVERSITY

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**
- They can also be used as **fields** within classes

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**
- They can also be used as **fields** within classes
- When the instance containing them is destroyed:

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**
- They can also be used as **fields** within classes
- When the instance containing them is destroyed:
  - `unique_ptr` destroys the instance to which it points

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**
- They can also be used as **fields** within classes
- When the instance containing them is destroyed:
  - `unique_ptr` destroys the instance to which it points
  - `shared_ptr` decrements the reference count on the instance to which it points, which may result in it being destroyed

Smart pointers and composition

- We have seen `unique_ptr` and `shared_ptr` used as **local variables**
- They can also be used as **fields** within classes
- When the instance containing them is destroyed:
    - `unique_ptr` destroys the instance to which it points
    - `shared_ptr` decrements the reference count on the instance to which it points, which may result in it being destroyed
- ... just as you would expect (hopefully)!

```cpp
class Duck
{
public:
    Duck(const std::shared_ptr<Pond>& pond)
        : pond(pond),
          bill(std::make_unique<Bill>())
    { }

private:
    std::unique_ptr<Bill> bill;
    std::shared_ptr<Pond> pond;
};
```

```
class Duck
{
public:
    Duck(const std::shared_ptr<Pond>& pond)
        : pond(pond),
          bill(std::make_unique<Bill>())
    { }

private:
    std::unique_ptr<Bill> bill;
    std::shared_ptr<Pond> pond;
};
```

When a `Duck` instance is destroyed:

```cpp
class Duck
{
public:
    Duck(const std::shared_ptr<Pond>& pond)
        : pond(pond),
          bill(std::make_unique<Bill>())
    { }

private:
    std::unique_ptr<Bill> bill;
    std::shared_ptr<Pond> pond;
};
```

When a `Duck` instance is destroyed:

- Its `Bill` is also destroyed;

```
class Duck
{
public:
    Duck(const std::shared_ptr<Pond>& pond)
        : pond(pond),
          bill(std::make_unique<Bill>())
    { }

private:
    std::unique_ptr<Bill> bill;
    std::shared_ptr<Pond> pond;
};
```

When a `Duck` instance is destroyed:

- Its `Bill` is also destroyed;
- Its `Pond` is destroyed **if and only if** there are no other `shared_ptr`s to it (e.g. in other `Duck` instances)

```cpp
class Bar;

class Foo
{
public:
    std::shared_ptr<Bar> bar;
};

class Bar
{
public:
    std::shared_ptr<Foo> foo;
};

int main()
{
    auto p = std::make_shared<Foo>();
    p->bar = std::make_shared<Bar>();
    p->bar->foo = p;
}
```

Reference cycles

- In the example on the previous slide, the instances are **not** destroyed properly at the end of `main()`

Reference cycles

- In the example on the previous slide, the instances are **not** destroyed properly at the end of `main()`
- `Foo` still holds a reference to `Bar`, but `Bar` holds a reference to `Foo`

Reference cycles

- In the example on the previous slide, the instances are **not** destroyed properly at the end of `main()`
- `Foo` still holds a reference to `Bar`, but `Bar` holds a reference to `Foo`
- This prevents both objects' reference counts from reaching 0

Reference cycles

- In the example on the previous slide, the instances are **not** destroyed properly at the end of `main()`
- `Foo` still holds a reference to `Bar`, but `Bar` holds a reference to `Foo`
- This prevents both objects' reference counts from reaching 0
- Solution: make one of the references a **weak pointer**

Weak pointers

- `std::weak_ptr` is similar to `std::shared_ptr`, except it does **not** alter the reference count of the instance

Weak pointers

- `std::weak_ptr` is similar to `std::shared_ptr`, except it does **not** alter the reference count of the instance
- A `weak_ptr` must be **locked** in order to access the instance to which it points

Weak pointers

- `std::weak_ptr` is similar to `std::shared_ptr`, except it does **not** alter the reference count of the instance
- A `weak_ptr` must be **locked** in order to access the instance to which it points

```
std::shared_ptr<MyClass> lockedInstance
    = weakPtr.lock();
```

Weak pointers

- `std::weak_ptr` is similar to `std::shared_ptr`, except it does **not** alter the reference count of the instance
- A `weak_ptr` must be **locked** in order to access the instance to which it points

```
std::shared_ptr<MyClass> lockedInstance
    = weakPtr.lock();
```

- Locking creates a temporary `shared_ptr` to the instance — this ensures that the reference count stays above 0 while the instance is being used

Smart pointers — Summary

- `std::shared_ptr` uses **reference counting** to manage the lifetime of objects

Smart pointers — Summary

- `std::shared_ptr` uses **reference counting** to manage the lifetime of objects
- `std::weak_ptr` is useful for breaking **reference cycles** when using `shared_ptr`s for composition

Smart pointers — Summary

- `std::shared_ptr` uses **reference counting** to manage the lifetime of objects
- `std::weak_ptr` is useful for breaking **reference cycles** when using `shared_ptr`s for composition
- `std::unique_ptr` can be used when you want to enforce that there is **only one** pointer to a particular instance

Smart pointers — Summary

- `std::shared_ptr` uses **reference counting** to manage the lifetime of objects
- `std::weak_ptr` is useful for breaking **reference cycles** when using `shared_ptr`s for composition
- `std::unique_ptr` can be used when you want to enforce that there is **only one** pointer to a particular instance
- NB: all smart pointer types allow **polymorphism**, e.g.

```
class Bird {};
class Duck : public Bird {};

std::shared_ptr<Bird> bird = std::make_unique<Duck>();
```

What about “dumb” pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers

What about "dumb" pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers
- I.e. use `std::shared_ptr<T>` instead of `T*`, and `std::make_shared` instead of **`new`**, and then you never have to worry about calling **`delete`**

What about “dumb” pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers
- I.e. use `std::shared_ptr<T>` instead of `T*`, and `std::make_shared` instead of `new`, and then you never have to worry about calling `delete`
- So why did we bother teaching you about normal C++ pointers?!

What about “dumb” pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers
- I.e. use `std::shared_ptr<T>` instead of `T*`, and `std::make_shared` instead of `new`, and then you never have to worry about calling `delete`
- So why did we bother teaching you about normal C++ pointers?!
  - Because C++ code written before 2011 (and much code written after) uses them extensively

What about “dumb” pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers
- I.e. use `std::shared_ptr<T>` instead of `T*`, and `std::make_shared` instead of `new`, and then you never have to worry about calling `delete`
- So why did we bother teaching you about normal C++ pointers?!
  - Because C++ code written before 2011 (and much code written after) uses them extensively
  - Because sometimes you need to use them, e.g. when working with SDL or other APIs / libraries

What about "dumb" pointers?

- In modern C++, it is **almost always preferable** to use smart pointers instead of normal C++ pointers
- I.e. use `std::shared_ptr<T>` instead of `T*`, and `std::make_shared` instead of `new`, and then you never have to worry about calling `delete`
- So why did we bother teaching you about normal C++ pointers?!
  - Because C++ code written before 2011 (and much code written after) uses them extensively
  - Because sometimes you need to use them, e.g. when working with SDL or other APIs / libraries
  - Because once you can write a nontrivial program that uses pointers and doesn't crash or leak memory, you can call yourself a Real Programmer ☺

Reminders

This Wednesday (9th March)

Your proposal for **COMP110 Coding Task 2** is due **in the lecture**

- Bring it with you for us to check in class

This Wednesday (9th March)

Your proposal for **COMP110 Coding Task 2** is due **in the lecture**

- Bring it with you for us to check in class

Your A* pathfinding program for **COMP110 Worksheet 5** is due **by 6pm**

- Submit a pull request on GitHub
- Book a tutor meeting to discuss and sign off

This Wednesday (9th March)

Your proposal for **COMP110 Coding Task 2** is due **in the lecture**

- Bring it with you for us to check in class

Your A* pathfinding program for **COMP110 Worksheet 5** is due **by 6pm**

- Submit a pull request on GitHub
- Book a tutor meeting to discuss and sign off

Wednesday's lecture is a **COMP140** lecture

- Heuristic analysis of your game controllers