COMP26020
Programming Languages and Paradigms

Lecture 41: Smart pointers

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

Traditional C++ still needs *new*

~~1. Creating and returning large objects~~

2. Dynamic lifetime patterns

Dynamic polymorphism

```cpp
class Base {
public: void f();
};

class Derived : public Base {
public: void f(); // overrides f from Base class
};

void Base::f() { std::cout << "Base's f() called\n"; }
void Derived::f() { std::cout << "Derived's f() called\n"; }

int main(int argc, char **argv) {
  Base b;
  Derived d;
  // We use a Base* not because we want to use the object as a Base object
  // but because we want a type that can fit either object
  Base *ptr = (condition) ? &b : &d;
  ptr->f(); // Base::f but that’s probably not what we want

  return 0;
}
```

Lecture 30

Pointers for handling related types dynamically

Plain variables have definite types and memory footprints

Creating objects with dynamic type

```
{
  Base b;
  Derived d;
  Base *ptr = (condition) ? &b : &d; // Creates both objects
}

{
  Base *ptr = (condition) ? Base() : Derived(); // ERROR, pointer to rvalue
}

{
  Base *ptr = (condition) ? new Base : new Derived; // Correct but manual new
}
```

Creating objects with dynamic type

Object needs to be dynamically allocated

RAII

- Lifetime associated with an automatic object
- Encapsulated?

Smart pointers

Encapsulating a raw pointer to heap memory

Similar syntax: * and ->

Custom constructors & destructors to manage lifetime

std::unique_ptr<T>

std::shared_ptr<T>

std::weak_ptr<T>

Smart pointer operations

	std::unique_ptr
Create	make_unique<T>(args)
Out-of-scope	Delete object
Reassigned	Delete object
Copy	Illegal
Move	Transfer ownership

Smart pointer operations

	std::unique_ptr	std::shared_ptr
Create	make_unique<T>(args)	make_shared<T>(args)
Out-of-scope	Delete object	Decrement count
Reassigned	Delete object	Decrement count
Copy	Illegal	Increment count
Move	Transfer ownership	Transfer ownership

make_unique / make_shared

Initialise the smart pointer with a ptr returned by new

Or do both with make_unique / make_shared

```
{
  Base *ptr = (condition) ? new Base : new Derived; // Correct but manual new
}

{
  Base *ptr = (condition) ? new Base : new Derived;
  auto sptr = std::unique_ptr<Base>(ptr); // Better
}

{ // Even Better, no new statement
  auto sptr = (condition) ? std::make_unique<Base>() : std::make_unique<Derived>();
}
```

Traditional C++ still needs *new*

~~1. Creating and returning large objects~~

~~2. Dynamic lifetime patterns~~

Modern C++ does not need *new**

~~1. Creating and returning large objects~~

~~2. Dynamic lifetime patterns~~

*apart from the code implementing these smart pointers as well as highly optimised library code. But application code does not need *new*.

Recap

- Heap pointers
  - Still needed
  - But encapsulated in smart pointers
  - Automatically track usage
  - Delete heap pointer when no more usage

Up Next

- Other language improvements
  - Type inference
  - Safe NULL
  - Compile-time evaluation
  - Structured binding