Inlining

There's no excuse for not using accessors, there is no performance loss.

inline considerably improves the opportunities for optimization, however excessive inlining causes code bloat.

Inlining is not automatical for compilers, it is just a way of suggesting something to the compiler.

Callable entities

Lambda

Lambas can be used as predicates for standard algorithms, example :

```
1 sort(v.begin(), v.end(), [](int a, int b) { return a > b;});
```

We can then call procedures without naming them directly in the codes.

Lambdas are more than Functions :

```
1 auto delta = 2;
2 auto incr = [delta](int x) { return x + delta;};
3 std::cout << incr(2) << std:endl;
4 // 4
```

We can set local variables, or use auto.

Lambdas demystified

Pointers

There are three iportant types:

- shared ownership
- no ownership at all
- unique ownership

We have:

- shared_ptr shares ownership

- weak_ptr is a non-owning pointer
- unique_ptr is a transfer of ownership pointer

Unique pointers

unique_ptr has a nice constructor :

```
auto u = std::make_unique<int>(2);
```

Cannot copy a unique pointer but can move a unique pointer.

```
auto u = std::make_unique<int>(2);
auto u2 = std::move(u);
assert(u == nullptr);
```

We can :

```
auto p = std::make_shared<int>{};
p = std::make_unique<int>(12);
```

The temporary unique pointer gave (moved) its content before dying, however:

```
auto p = std::make_shared<int>{};
auto u = std::make_unique<int>(12);
p = u; // KO cause *not* unique
```

But :

```
auto p = std::make_shared<int>{};
auto u = std::make_unique<int>(12);
p = std::move(u); // transfer
assert(!u);
```

Weak pointers

shared_ptr are nice but:

- We just need some local temporary pointers
- or we can have circular references
- or we can also have asymmetrical relationships
- or we want to avoid dangling pointer problems

a weak_ptr is great because it does not count.

Consider:

```
auto p = new int(10);
auto p2 = p;
delete p;
// p2 is around
```

versus

```
auto p = std::make_shared<int>(10);
auto p2 = std::weak_pttr<int>{p};
p.reset();

//p2 is around but:
assert(p2.expired == true);
```

We cannot have a weak pointer pointing at the memory address of a unique pointer.

Within class space

RTTI

dynamic_cast => very powerful tool

```
shape s = shape{1, 1};

auto c = dynamic_cast<circle*>(s);

if (c)
  return true;
```

Conclusion about C++

C++ is a very powerful tool but riddled with historical idioms

Go see boost

“Effective Modern C++”, 2014