Tampereen yliopisto
Tampere University

Memory management and object ownership

Smart pointers

Object lifetime

- Objects are complicated ⇒ creation and deletion may require special actions
- Static vs. dynamic lifetime

Ownership

- Object is always owned by a program structure or by another object
- Owner is responsible to delete the object (created dynamically)

Fig: Kyle Luce's blog

Interfaces and passing objects

- Modules/objects call each other via encapsulated interfaces
- Calls pass objects from place to place
- It is a good thing, if the ownership (= responsibility to delete) of an object remains always in the same place

Interfaces and passing objects

- However, it is usually necessary to create an object in one side of an interface and to delete it in another side ⇒ ownership of the object moves across the interface
- Movement must be written in the interface documentation!

Ownership

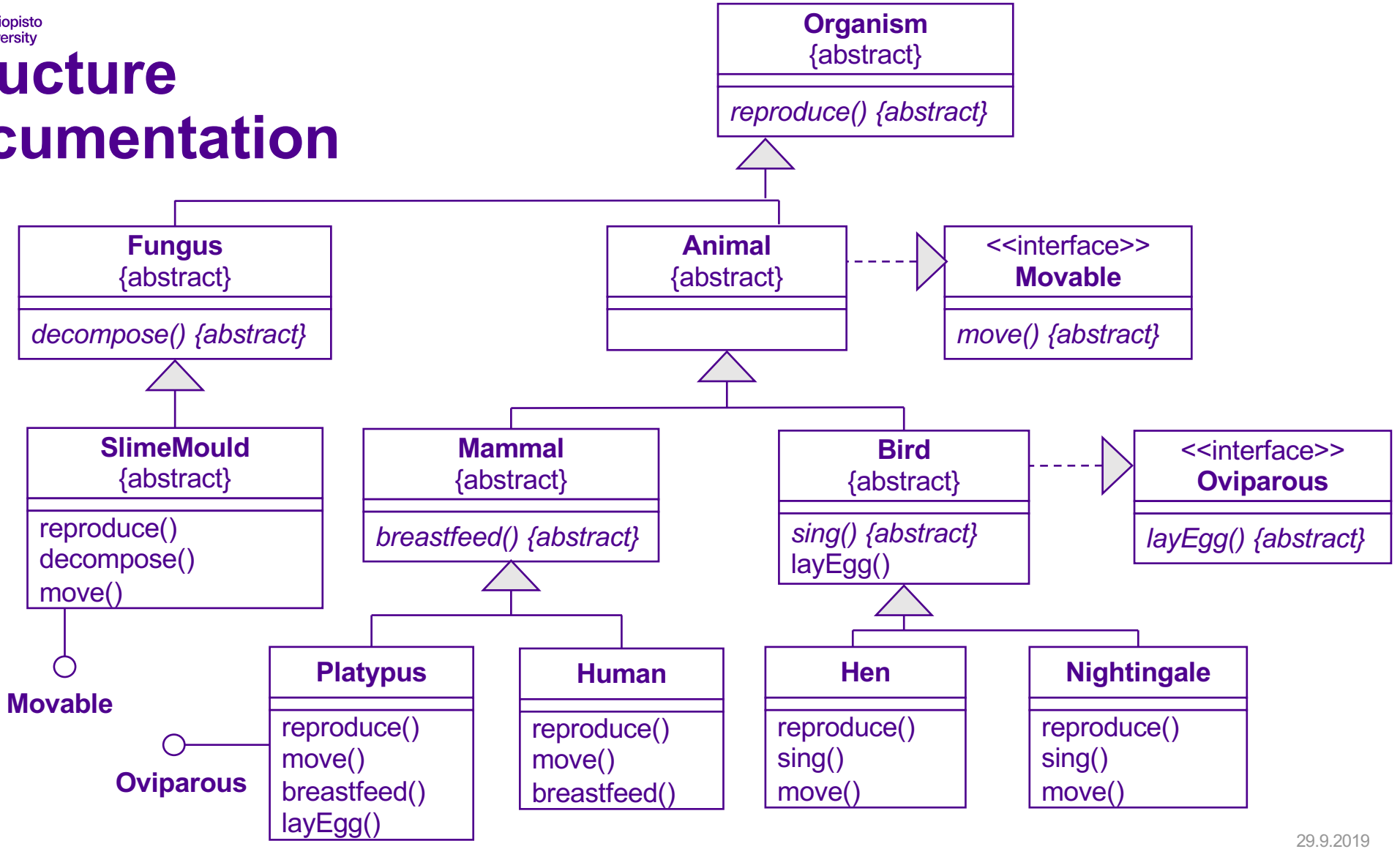
- Ownership documentation is important:
  - Responsibility of object deletion
  - Required dispose actions
- Automatic garbage collection (e.g. Python) takes care of the first one (of above) but not the latter one
- C++ has not (yet) garbage collection, but the destructor takes care of dispose actions when an object is deleted

Ownership: deletion

- Programmer is responsible to **delete** an object that has been created by **new** command
- C++11 provides smart pointers to manage ownerships:
  - `std::shared_ptr`
  - `std::weak_ptr`
  - `std::unique_ptr`

Tampereen yliopisto
Tampere University

Structure documentation



Ownership documentation

- UML has different associations (bi-directional, aggregation, composition, ...)
- Associations have effects on the ownership relations among objects

Ownership documentation

- Nowadays C++ has several ways to refer to an object (reference, pointer, automatic pointer, shared pointer, weak pointer)

⇒ These choices can be used to document the design decisions of UML. In addition, the reference type supports the implementation of the association in question. How clever!

(Smart ?!) pointers

Fig: DwarfVader (CC BY-NC-ND 2.0)

Ownership documentation in C++

Reference (&)

- No ownership, cannot be NULL (0)
- Target remains the same during the lifetime of a reference
- Not suitable for an element of STL containers (Assignable)

Pointer (∗)

- No ownership (or ownership management by hand)
- Suitable for an element of STL containers (Assignable)

Ownership and smart pointers

Shared ownership

- Several objects can own the same resource
- Resource does not depend on the lifetime of a single object

Unique ownership

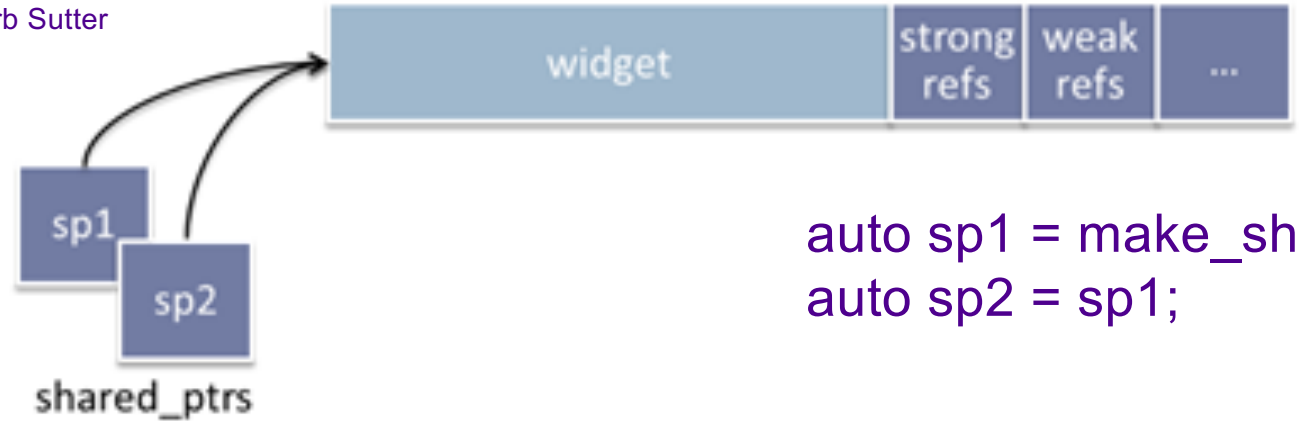
- One object owns a resource

Shared pointer `std::shared_ptr`

- Smart pointer with a reference counter: shared ownership between several shared pointers
- Resources will be released, when the reference counter is 0
  - Last shared pointer destroys the resource
  - Be careful with cycles!
  - Created as `std::make_shared<X>(...)`

Shared pointer `std::shared_ptr`

Fig: Herb Sutter




```
auto sp1 = make_shared<widget>(…);
auto sp2 = sp1;
```

Others:

- Raw pointer: `spw.get()`, (no release)
- To find out sharing situation: `use_count(), unique()`

Shared_ptr: using get()

```
void output(const std::string& msg, int* pInt) {
    std::cout << msg << *pInt << std::endl;
}

int main() {
    int* pInt = new int(42);
    std::shared_ptr<int> pShared = make_shared<int>(42);
    output("Raw pointer ", pInt);
    // output("Shared pointer ", pShared); // compiler error
    output("Shared pointer with get() ", pShared.get());
    delete pInt;
    return 0;
}
```

```
Raw pointer 42
Shared pointer with get() 42
```

Weak pointer `std::weak_ptr`

- Indicates an interest to a shared object, not strong enough to keep the object alive
- Useful for breaking cycles of shared pointers
- Useful for checking, if the object has already been destroyed

Weak pointer `std::weak_ptr`

- Last shared pointer deletes the referenced object, although there are weak pointers left (`wp.expired()`)
- Cannot access the referenced object directly, but produces a shared pointer (wp.lock())

Unique pointer `std::unique_ptr`

- Unique ownership
- As cheap as a usual pointer (no reference counters)
- Ownership can be moved or released explicitly (original becomes empty)
- Creating an object pointed by a unique pointer (C++14): `std::make_unique<X>(...)`

Tampereen yliopisto
Tampere University

Unique pointer example

```
std::unique_ptr<Thing> p1;
std::unique_ptr<Thing> p2 (std::make_unique<Thing>(…)
);
// p1 = p2;                          // Error! Can’t copy
unique ptr
p1 = std::move(p2);                  // p2.get() == nullptr
Thing* tp = p1.release();            // p1.get() == nullptr
…
p1.reset(new Thing(…));     // p1.get() != nullptr
```

Function pointers

- Passing functionality as parameter
  - function cannot be passed as a parameter, but a pointer to a function can be passed
  - e.g. to STL algorithms (and associative containers)
- Another way: function objects

Function pointers

```
bool isLessThan5(int i) {
    return i < 5;
}

void printLessThan5(vector<int> const& v) {
    vector<int>::const_iterator i = v.begin();
    while( (i = find_if(i, v.end(), &isLessThan5) ) != v.end() ) {
        cout << *i << ' ';
        ++i;
    }
    cout << endl;
}
```