DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

[CS 225] Advanced C/C++

Lecture 2: Layout of Objects in Memory

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Relationships between object types
- Impact of relationships on sizes of objects
  - Non-class types
  - Classes
    - Empty classes
    - Classes with data members
    - Classes with `virtual` members
    - Derived classes
- Reuse of implementation and interface

Relationships between Object Types

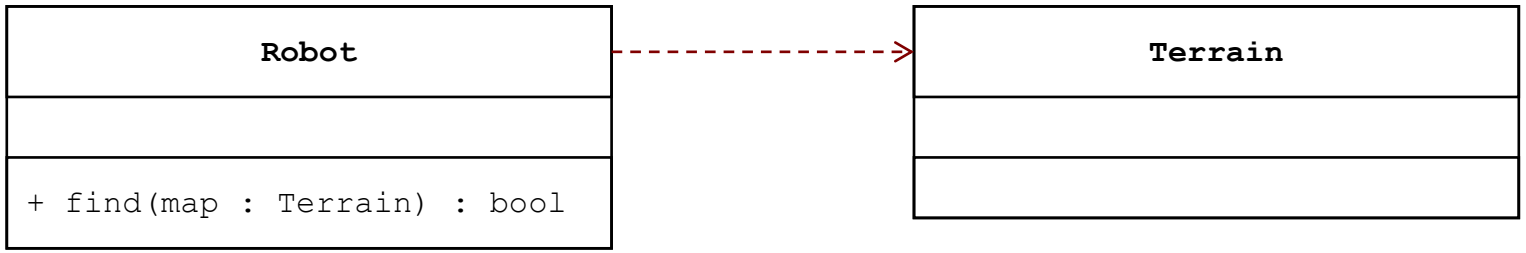
Commonly recognized of class relationships:

- Dependency
- Association
- Aggregation
- Composition
- Inheritance
  - Generalization
  - Specialization
  - Realization

Relationships between Object Types

Dependency is the weakest type of a relationship between two classes.

It does not impact the layout of the classes, but one or both classes can accept references to objects of the opposite class as *arguments in their member functions*.



Relationships between Object Types

Association means that classes exist independently, but one or both can reference the opposite class through its data members.

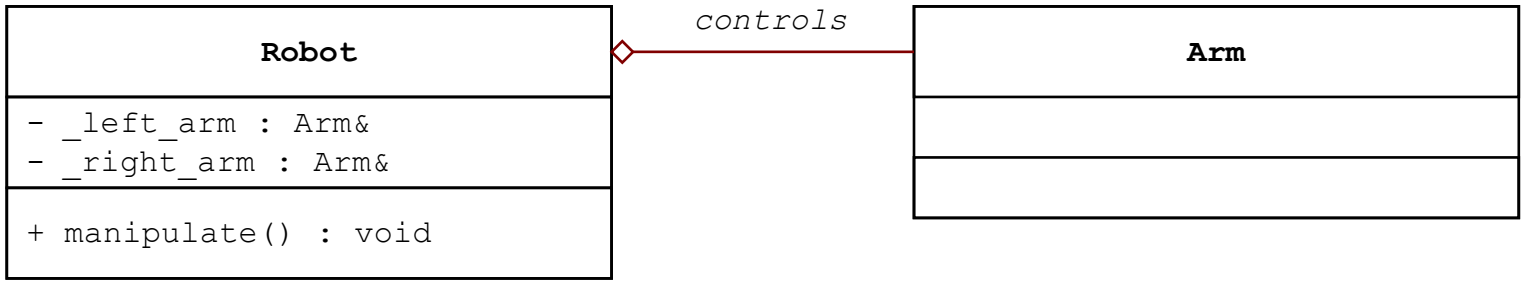
In C++, association is typically implemented as a *pointer* to a target class.

Relationships between Object Types

***Think**: "Robot aggregates a tool, but it can leave it behind; the tool still exists."*

Aggregation is a one-sided binary association, where one object references the target object, and many other objects can reference the target too. The target object may live longer than any object it references.

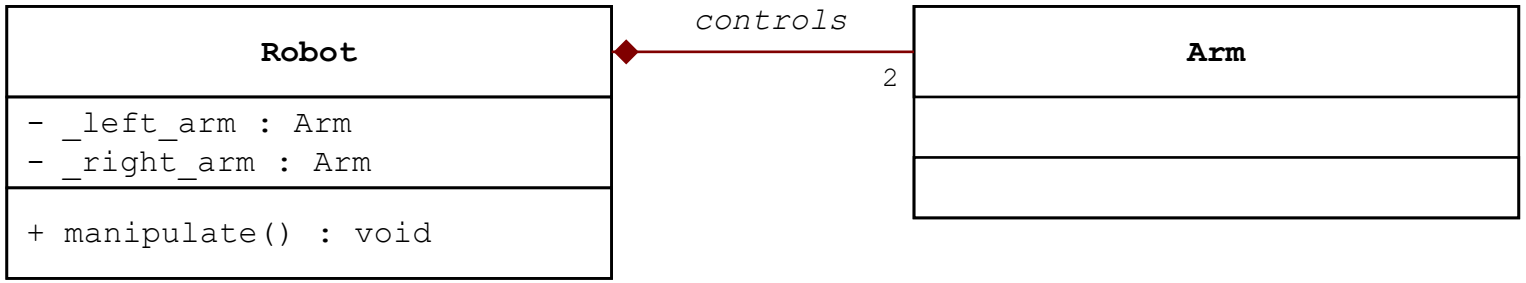
In C++, aggregation is typically implemented as a *pointer* or a *reference* to a target class.



Relationships between Object Types

Composition is a strong aggregation, where one object owns the target. The target object is destroyed with the destruction of the parent; it cannot exist independently.

In C++, composition is typically implemented as a target class *data member*.



***Think**: "Robot is composed of a non-detachable leg that has its own properties; the leg is an integral part of the robot."*

Relationships between Object Types

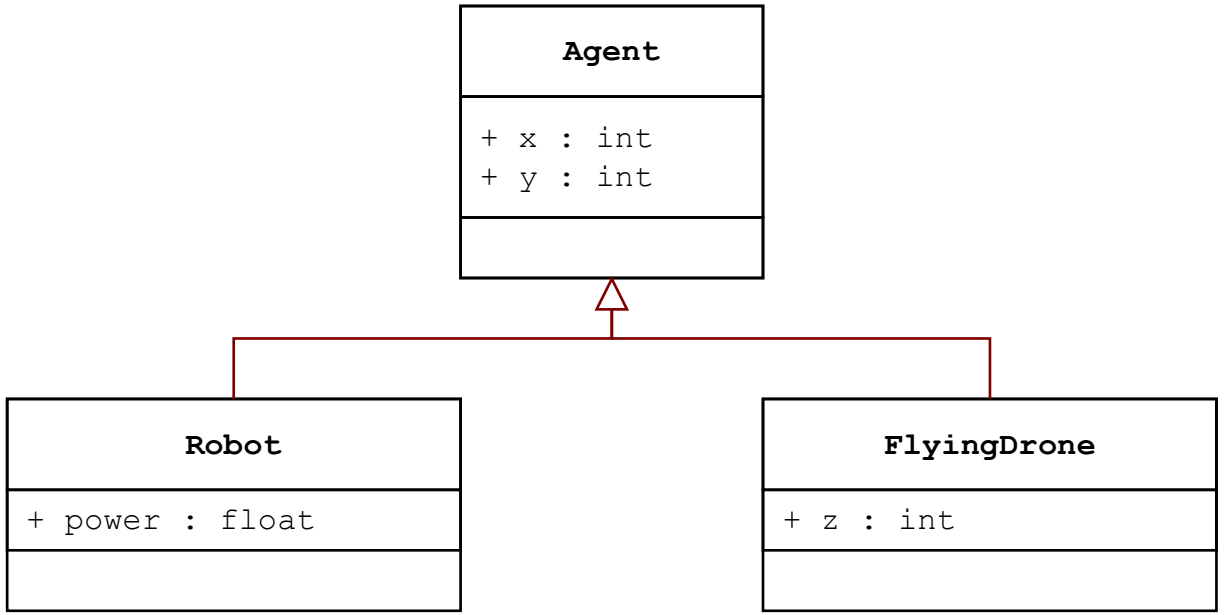
Inheritance is a category of relationships where a *base* class (“*parent*”, “*superclass*”) is reused by a definition of a *derived* class (“*child*”, “*subclass*”).

Inheritance enables three relationships:

- *Generalization* – we extract common parts of multiple classes to a more generalized base class.
- *Specialization* – we take a class and inherit from it to introduce more specialized derived classes.
- *Realization* – we inherit from an interface class to provide implementation for all required functionality.

Relationships between Object Types

In C++, generalization and specialization are implemented as class inheritance.



Sizes of Objects

- Impact of relationships on sizes of objects
  - Non-class types
  - Empty classes
  - Classes with data members
  - Classes with `virtual` members
  - Derived classes

Sizes of Objects – Non-class Types

- Fundamental data types – as per the specification, e.g.
  - `unsigned char` – 1 byte.
  - `int` – at least 16 bits, depending on the platform.
  - `uint32_t` – exactly 32 bits, fixed width.
  - Operator `sizeof()` works as expected.

- Static arrays – multiplicity of an element size
  - `uint32_t x[10]` – 40 bytes (assuming a byte is 8 bits)
  - Operator `sizeof()` works differently on a static array and an array that has decayed to a pointer.

Sizes of Objects – Non-class Types

- Pointers – size depending on the platform
  - `int*` – 8 bytes on a 64-bit platform with 8 bit bytes.
  - Operator `sizeof()` works as expected.

- References – 0 or more bytes, typically:
  - References with the same scope as a referenced object can be optimized out; the program can work on the original object.
  - Data members, function params, references with a different scope – a compiler can use pointers under-the-hood.
  - Operator `sizeof()` shows the size of a bound object. To check the size of the reference object, make it a data member.

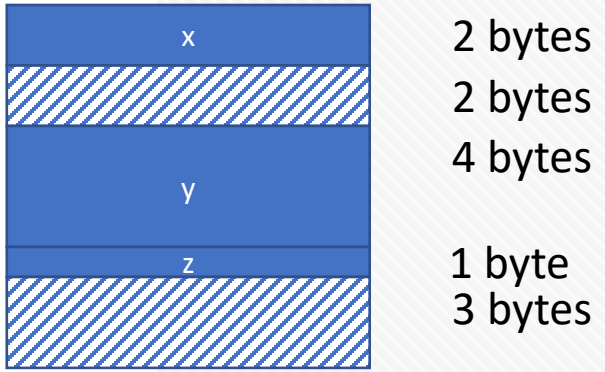
Sizes of Objects – Classes

- Empty class
  - A class has 0 bytes.
  - An object has 1 byte (of padding)
    to prevent having two distinct objects with the same address.
  - Operator `sizeof()` returns the size of the object, not the type.

Sizes of Objects – Classes

- Classes with data members
  - Total size is the sum of members and paddings.
  - Data members are laid out in order of declarations.
  - Data members may be preceded by padding to align them to an address required by a member.
  - The last data member may be followed by padding to align the first data member of a consecutive object of the same type to the required address.
  - **Empty class** data members occupy 1 byte (in C++20 you can use `[[no_unique_address]]`).

```
class MyClass
{
  signed short int x;
  float y;
  unsigned char z;
};
```

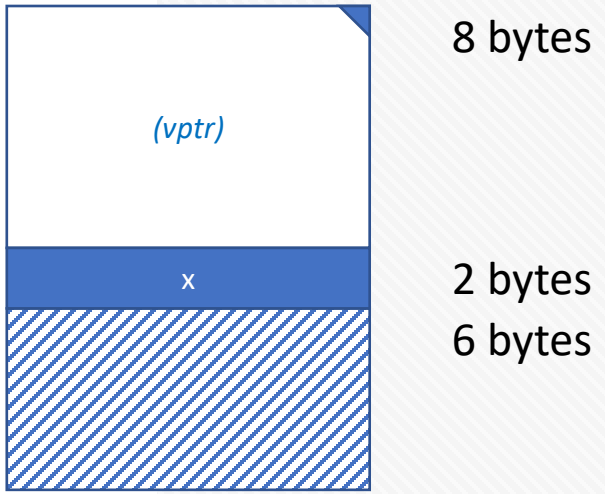


Sizes of Objects – Classes

```
class MyClass
{
  signed short int x;

  virtual ~MyClass();
};
```

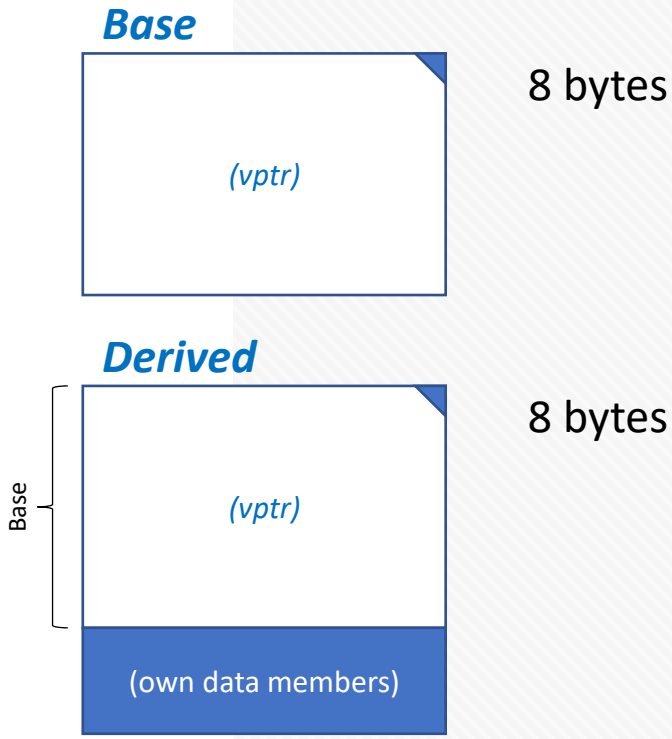
- Classes with virtual members
  - If a class has anything `virtual`, a hidden pointer (*vptr*) will exist at the beginning of an object; it points to its **virtual method table** (*vtable*).
  - The size of the class grows by the size of *vptr* and resulting padding.



Sizes of Objects – Classes

```
class Base
{
  virtual ~Base();
};
class Derived : Base
{
  // Own data members
};
```

- Derived classes
  - A base class' data members reside in memory before data members of a derived class.
  - A class derived from an **empty class**, inherits 0 bytes (*EBO – empty base optimization*).
  - A class derived from a class with **vptr** can reuse it, without adding another one.
  - Exception: *virtual inheritance (we will cover it soon*).



Reuse of Implementation and Interface

- Reuse of implementation
- Reuse of implementation and interface
- Reuse of interface

Reuse of Implementation and Interface

- Purpose: Reuse of implementation
  - The current object is made of another object, or it uses a base class as the foundation of its implementation.
  - Base and derived objects are *business objects* (**“has-a” idiom**).
  - Base or derived objects are *implementation objects* (**“is-implemented-in-terms-of” idiom**).

- Execution:
  - Private or protected inheritance.
  - Data member composition.

Reuse of Implementation and Interface

- Purpose: Reuse of implementation and interface
  - The current object is made of another object, or it uses a base class as the foundation of its implementation.
  - The code outside of the class should be able to use the current object where the base class object is expected as the interface is reused too (**“is-a” idiom**).
  - This is a generalization/specialization relationship; C++ (unlike many other languages) supports multiple inheritance for this purpose.

- Execution:
  - Public inheritance.

Reuse of Implementation and Interface

- Purpose: Reuse of interface
  - Another class declares expected behavior (*the interface*), but it does not define it (*the implementation*); everything there is pure virtual, no data members, etc.
  - The current object is an instance of a class that reuses that interface and implements the functionality (**“is-a” idiom**).
  - This is a realization relationship; most OOP languages support multiple inheritance for this purpose.

- Execution:
  - Public inheritance of an **interface class** (*an interface*).