Object-Oriented Programming: Inheritance

What is “Object-Oriented Programming?”

Definitions

One of our first slides

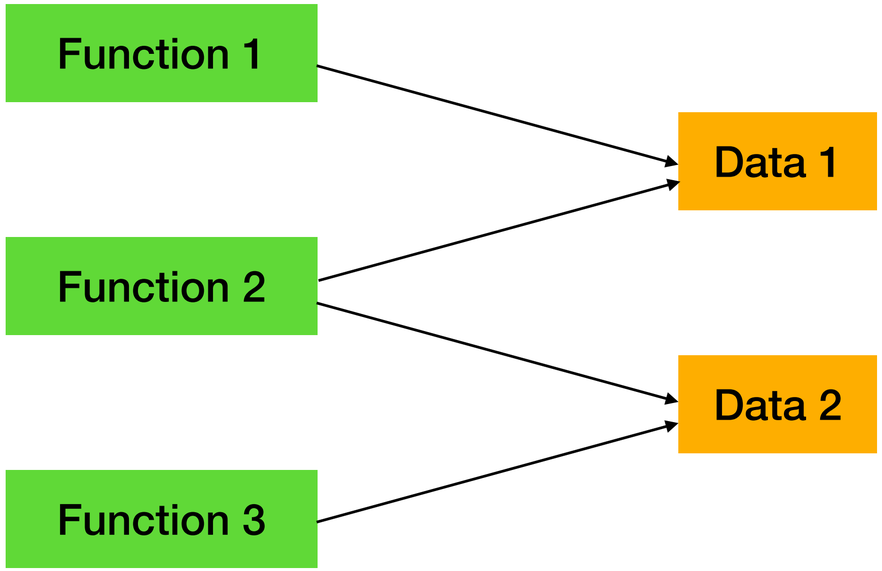
- ✓ Objects are software units modelled after entities in real life
  - ✓ They have attributes: e.g., length, density, color
  - ✓ They have a behaviour and provide functionalities: e.g., a door can be opened/shut, a nucleus can decay
- ✓ Object-oriented programming means writing code in terms of objects, i.e., well defined units which have attributes and offer functionalities
  - ✓ A program hence consists in interactions among objects using methods offered by each one of them
  - ✓ Objects are “smart” data structures: they are data with behaviour

What is “Object-Oriented Programming?”

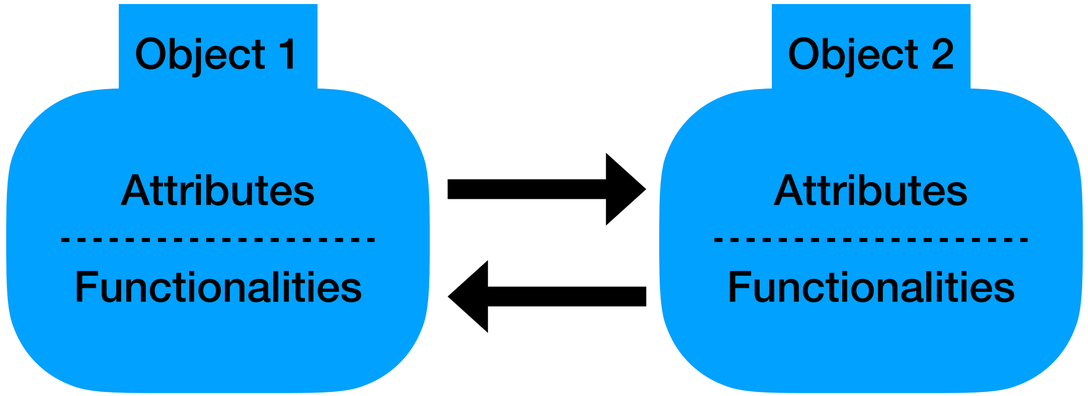
In Pictures

One of our first slides

Traditional concept



✓ Object-oriented concept



Characteristics of C++

One of our first slides

From object-oriented programming

✓ Data abstraction – the creation of classes to describe objects

✓ Data encapsulation for controlled access to object data

- Inheritance by creating derived classes (including multiple derived classes)
- Polymorphism – the implementation of instructions that can have varying effects during program execution

From C

- Universal
- Efficient
- Close to the machine
- Portable

Extensions

- Templates – allow the construction of functions and classes based on types not yet stated

See slides on `std::map` and `std::pair`

- Exception handling

What is Inheritance?

- Powerful way to reuse software without too much re-writing
- Often several types of object are actually special cases of a basic object
  - Keyboard and files are different types of an input stream
  - Screen and file are different types of output stream
  - Resistors and capacitors are different types of circuit elements
  - Circle, square, ellipse are different types of 2D shapes
  - In StarCraft, engineers, builders, soldiers are different types of units
- Inheritance allows to define a base (or parent) class that provides basic functionalities to derived (or child) classes
  - Derived classes extend the base class by adding new data members and functions

Inheritance: `Student` “is a” `Person`

Based on `examples/08/Inheritance1.cpp`

Base class

```
#include <string>
#include <iostream>
using namespace std;


class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for " << name_ << endl;
    }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is " << name_ << endl;
    }


  private:
    string name_;
};
```

Derived class

```
class Student : public Person {
  public:
    Student(const string& name, int id) : Person(name) {
      id_ = id;
      cout << "Student(" << name << ", " << id << ") called" << endl;
    }

    ~Student() {
      cout << "~Student() called for name: "
           << name() << " and id: " << id_ << endl;
    }

    int id() const { return id_; }

  private:
    int id_;
};
```

Example of Inheritance in Use

Based on `examples/08/Inheritance1.cpp`

```
int main() {
  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);
  susan->print();
  cout << "name: " << susan->name() << " id: " << susan->id() << endl;

  delete john;
  delete susan;

  return 0;
}
```

Application

```
$ g++ -Wall -o Inheritance1 Inheritance1.cpp
$ ./Inheritance1
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am a Person. My name is Susan
name: Susan id: 123456
~Person() called for John
~Student() called for name: Susan and id: 123456
~Person() called for Susan
```

`Student` “Behaves as” `Person`

Based on `examples/08/Inheritance1.cpp`

```
int main() {
  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);
  susan->print();
  cout << "name: " << susan->name() << " id: "
       << susan->id() << endl;

  delete john;
  delete susan;

  return 0;
}
```

- `print()` and `name()` are methods of `Person`
- `id()` is a method of `Student`

- Methods of `Person` can be called with an object of type `Student`
  - Functionalities implemented for `Person` available for free
  - No need to re-implement the same code over and over again
  - If a functionality changes, the change happens in a single place: makes code maintenance simpler and less bug prone

`Student` is an “Extension” of `Person`

Based on `examples/08/Inheritance1.cpp`

```
class Student : public Person {
  public:

    // Constructor(s) and Destructor

    int id() const { return id_; }

  private:
    int id_;
};
```

```
int main() {
  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);
  susan->print();
  cout << "name: " << susan->name() << " id: "
       << susan->id() << endl;

  delete john;
  delete susan;

  return 0;
}
```

- `Student` provides all functionalities of `Person` **and more**
  - Has additional data members and member functions
  - Is an extension of `Person` but not limited to be the same
- No need to access source code of a class to inherit from it: use public interface and add new data members and functions

Inheritance is a One-way Process/Relation

Based on `examples/08/InheritanceBad1.cpp`

- You cannot use methods of `Student` on a `Person` object
  - `Student` knows to be derived from `Person`
  - `Person` does not know what could be derived from it
- You can treat a `Student` object (`*susan`) as a `Person` object

```
int main() {

  Person* susan = new Student("Susan", 123456);
  cout << "name: " << susan->name() << endl;
  // cannot call id() on a Person pointer
  cout << "id: " << susan->id() << endl;

  delete susan;

  return 0;
}
```

`susan` is a pointer to `Person` but initialized by a `Student` constructor!

OK… because a `Student` is also a `Person` (elements of polymorphism)

```
$ g++ -Wall -o InheritanceBad1 InheritanceBad1.cpp
InheritanceBad1.cpp:53:28: error: no member named 'id' in 'Person'
  cout << "id: " << susan->id() << endl;
                    ~~~~~  ^
1 error generated.
```

`public`, `protected`, and `private` Inheritance

```
class Person {
  public:
  private:
  protected:
    int age_;
};
```

```
class Student : public Person {
};
```

Three possible access specifiers and flavours of inheritance:

1. public inheritance makes all `public` members (data and methods) of the base class `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class [access to `private` members provided only via public methods (getters)]

2. protected inheritance makes all `public` and `protected` members (data and methods) of the base class `protected` in the derived class

3. private inheritance makes all `public` and `protected` members (data and methods) of the base class `private` in the derived class

`protected` Members

Based on `examples/08/Inheritance2.cpp`

```
class Person {
  public:
    Person(const string& name, int age) {
      name_ = name;
      age_ = age;
      cout << "Person(" << name << ", "
           << age << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for " << name_ << endl;
    }

    string name() const { return name_; }
    int age() const { return age_; }
    void print() {
      cout << "I am a Person. My name is " << name_
           << ". My age is " << age_ << endl;
    }

  private:
    string name_;

  protected:
    int age_;
};
```

Base class

```
class Student : public Person {
  public:
    Student(const string& name, int age, int id) :
      Person(name,age) {
      id_ = id;
      cout << "Student(" << name << ", " << age
           << ", " << id << ") called" << endl;
    }

    ~Student() { //Protected members can be used in derived class!
      cout << "~Student() called for name: “ << name()
           << " age: " << age_ << " and id: " << id_ << endl;
    }

    int id() const { return id_; }

  private:
    int id_;
};
```

Derived class

- `protected` members become `protected` members of derived classes
  - `protected` is somehow between `public` and `private`

Constructors of Derived Classes

- Compiler calls default constructor of base class in constructors of derived class ***unless*** you explicitly call a specific constructor
- Necessary to ensure memory is ***always*** allocated for data members of the base class and they are initialised when creating instance of derived class

```
class Student : public Person {
  public:
    Student(const string& name, int id) /*: Person(name)*/ {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

  private:
    int id_;
};
```

- Bad programming: `Student` constructor does not call `Person` constructor
  - Compiler is forced to call `Person()` to ensure memory for `name_` is allocated and initialised
  - Do not rely on default constructor to do the “right thing”

Common Error: Missing Base Constructors

Based on `examples/08/InheritanceBad2.cpp`

```
class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for " << name_ << endl;
     }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is " << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) /*: Person(name)*/ {
      id_ = id;
      cout << "Student(" << name << ", " << id << ") called" << endl;
    }

    ~Student() {
      cout << "~Student() called for name: " << name() << " and id: " << id_ << endl;
    }

    int id() const { return id_; }

  private:
    int id_;
};
```

```
$ g++ -Wall -o InheritanceBad2 InheritanceBad2.cpp
InheritanceBad2.cpp:31:5: error: constructor for 'Student' must explicitly initialize the base class 'Person' which does not have a default constructor
    Student(const string& name, int id) {
    ^
InheritanceBad2.cpp:6:7: note: 'Person' declared here
class Person {
      ^
1 error generated.
```

Bad Working Example

Based on `examples/08/InheritanceBad3.cpp`

```
class Person {
  public:
    Person() { } // default constructor provided
  // ...
};

class Student : public Person {
  public:
    Student(const string& name, int id) {
      // We do not call Person(name) to initialise the name
      // the compiler will call the default constructor Person()
      // which does not give a name to person
      // so our student will have no name
      // this example compiles and runs but has a wrong behavior
      id_ = id;
      cout << "Student(" << name << ", " << id << ") called" << endl;
    }
  // ...
};
```

```
int main() {

  Student* susan = new Student("Susan", 123456);
  susan->print();

  delete susan;

  return 0;
}
```

```
$ g++ -Wall -o InheritanceBad3 InheritanceBad3.cpp
$ ./InheritanceBad3
Student(Susan, 123456) called
I am a Person. My name is
~Student() called for name: and id: 123456
~Person() called for
```

Code compiles, links, and runs, but shows sign of bad behavior

Overloading Methods from the Base Class

Based on `examples/08/Inheritance3.cpp`

- Derived classes can also overload functions provided by the base class
  - Same signature but different implementation

```
class Person {
  public:
    // ...
    void print() {
      cout << "I am a Person. My name is " << name_ << endl;
    }

 private:
    string name_;
};
```

```
class Student : public Person {
  public:
    //...
    void print() {
      cout << "I am Student "
           <<  name()
           << " with id " << id_
           << endl;
    }
  private:
    int id_;
};
```

Overloading Methods from the Base Class

Based on `examples/08/Inheritance3.cpp`

```
int main() {
  Person* john = new Person("John");
  john->print();  // Person::print()

  Student* susan = new Student("Susan", 123456);
  susan->print(); // Student::print()
  susan->Person::print(); // Person::print()

  Person* p2 = susan;
  p2->print(); // Person::print()

  delete john;
  delete susan;

  return 0;
}
```

```
$ g++ -Wall -o Inheritance3 Inheritance3.cpp
$ ./Inheritance3
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan
I am a Person. My name is Susan
~Person() called for John
~Student(  called for name: Susan an  id: 123456
~Person() called for Susan
```

Compiler calls the correct version of `print()` for `Person`

Compiler calls the correct version of `print()` for `Student`

We can use `Person::print()` implementation for a `Student` instance by specifying its scope

Remember: a function is uniquely identified by its namespace and class scope

Undesired Limitation

Based on `examples/08/Inheritance4.cpp`

```
int main() {
  Person john("John");
  john.print();  // Person::print()

  Student susan("Susan", 123456);
  susan.print(); // calls Student::print()
  susan.Person::print(); // explicitly call Person::print()

  // using base class pointer
  cout << "-- using base class pointer" << endl;
  Person* p2 = &susan;
  p2->print(); // calls Person::print()

  //using derived class pointer
  cout << "-- using derived class pointer" << endl;
  Student* sp = &susan;
  sp->print(); // calls Student::print()

  // using base class reference
  cout << "-- base class reference" << endl;
  Person& p3 = susan;
  p3.print(); // calls Person::print()

  // behavior of print() depends on the type of pointer determined
  // at compilation time

  return 0;
}
```

```
$ g++ -Wall -o Inheritance4 Inheritance4.cpp
$ ./Inheritance4
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan
-- using base class pointer
I am a Person. My name is Susan
-- using derived class pointer
I am Student Susan with id 123456
-- base class reference
I am a Person. My name is Susan
~Student() called for name: Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

Choosing Type at Runtime

Based on `examples/08/Inheritance5.cpp`

- Call to method `print()` is resolved base on the type of the pointer
  - `print()` method determined by type of pointer not type of object
- Desired feature: use generic `Person*` pointer but call appropriate `print()` method for objects based on their **actual type**

Choosing Type at Runtime

Based on `examples/08/Inheritance5.cpp`

```
int main() {

  int itype;
  do{
    cout << "Choose class to allocate a dynamic object. "
         << "1: Person 2: Student" << endl;
    cin >> itype;
  } while (itype != 1 && itype!=2);

  Person* p;


  if(itype==1) {
    p = new Person("john");
  } else {
    p = new Student("Susan", 123456);
  }

  // calling print()
  cout << "calling print() on a Person* pointer" << endl;
  p->print();

  cout << "delete the object" << endl;
  delete p;

  return 0;
}
```

```
$ g++ -Wall -o Inheritance5 Inheritance5.cpp
$ ./Inheritance5
Choose class to allocate a dynamic object. 1: Person 2: Student
1
Person(john) called
calling print() on a Person* pointer
I am a Person. My name is john
delete the object
~Person() called for john
$ ./Inheritance5
Choose class to allocate a dynamic object. 1: Person 2: Student
2
Person(Susan) called
Student(Susan, 123456) called
calling print() on a Person* pointer
I am a Person. My name is Susan
delete the object
~Person() called for Susan
```

Watch out!

Destructors of Derived Classes

- Similar to constructors: compiler calls the default destructor of base class in destroying an instance of a derived class
- No compilation error if destructor of base class not implemented
  - Default will be used but… bad things can happen!
- Extremely important to implement correctly the destructors to avoid **memory leaks**
- We will get back to this with `virtual` destructors

"Hey! Your application has a memory leak."