Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-2 CS213
2018/2019

# Inheritance

Dr. Amin Allam

## 1 Software engineering overview

The amount of effort required to plan, develop, maintain, understand, and test a program containing $N$ instructions is proportional to $N^{1.5}$. For example, if a $100$-instructions program requires $100^{1.5} = 1,000$ effort units, a $500$-instructions program will require $500^{1.5} \approx 11,000$ effort units. Therefore, if we managed to divide a $500$-instructions program into $5$ independent $100$-instructions programs, the amount of required effort will decrease from $11,000$ to $5 \times 1,000 = 5,000$.

Practically, it is impossible to divide a program into completely independent subprograms, but it can be divided into highly independent subprograms. So, there are two basic software measures:

- Cohesion: the amount of relatedness of the steps of the same subprogram.
- Coupling: the amount of relatedness of different subprograms.

A good software design should maximize cohesion and minimize coupling of subprograms (i.e. classes).

Testing consumes half of the software engineering process time. Therefore, modifying any part of a tested program is much costly, since the program should be fully retested after such modification. Programmers should avoid modifying any part of a tested program.

A good software design should follow the five SOLID principles. Each of these principles is going to be explained in appropriate places during the following lectures.

## 2 Inheritance

Aggregation generally models the has-a relationship. Inheritance generally models the is-a relationship. Moreover, they are used for several other purposes. Aggregation is usually preferred over inheritance. However, inheritance has clear advantages in specific situations, such as polymorphism which is going to be explained in the following lectures.

Given the following definition of class `Student`:

```
class Student
{
protected:      // Allow access from this class and derived  classes  only
    int ID; double GPA; string name;
public:
    void Input() {cin>>ID>>GPA>>name;}
    void Output() {cout<<ID<<" "<<GPA<<" "<<name;}
};
```

Suppose we need to define a class `MasterStudent` that behaves similarly to class `Student` and contains additional functionality as follows:

```
class MasterStudent : public Student   // Inherit MasterStudent from Student
{
private:
    bool passedQuals;
public:
    void Output() {cout<<ID<<" "<<GPA<<" "
                           <<name<<" "<<passedQuals;}
};
```

The first line of the previous code inherits `MasterStudent` from `Student`, which means that `MasterStudent` is a `Student` with some additional functionality. We call `Student` the parent or base class of `MasterStudent`, while we call `MasterStudent` a child, derived, or inherited class form `Student`. It can also be called a subclass from `Student`.

Inheritance (creating derived classes from an existing class) is also called specialization, while its inverse (creating a base class to existing classes) is called generalization.

The reason of using `protected` instead of `private` is to allow class `MasterStudent` (and any class derived from `Student`) to access the member variables of class `Student`. Still, they can not be accessed from anywhere else.

Suppose in any function, we create an instance (object) from class `MasterStudent`:

```
MasterStudent m;
```

The object `m` will be allocated in stack memory. The object `m` contains data members declared in `Student` plus all data members declared in `MasterStudent`. So, the object `m` contains the 4 members: `m.ID, m.GPA, m.name, m.passedQuals`.

A similar effect can be done by:

```
MasterStudent* p = new MasterStudent;
```

The statement `new MasterStudent` creates an unnamed object of type `MasterStudent` in the heap memory, and returns its address which is stored in the pointer `p`.

Calling a method (function) from an object of type `MasterStudent` will search for it first in the class `MasterStudent` and executes it if found. Otherwise, it will search for it in the base class `Student` and executes it if found. Otherwise, it flags a compiler error. This is a type of overloading where the function call is associated to a specific function during compile-time:

```
MasterStudent m;
MasterStudent* p = new MasterStudent;
m.Input();       // Calls Input() of Student class
p->Input();      // Calls Input() of Student class
m.Output();      // Calls Output() of MasterStudent class
p->Output();     // Calls Output() of MasterStudent class
delete p;
```

It is possible to define `Output()` of `MasterStudent` to reuse the already written code in the `Output()` of `Student` as follows:

```
class MasterStudent : public Student
{
private:
    bool passedQuals;
public:
    void Output()
    {
        Student::Output();     // Student:: is important to avoid recursion
        cout<<" "<<passedQuals;
    }
};
```

Consider these modified implementations of `Student` and `MasterStudent` which contains one argument constructors as follows:

```
class Student
{
protected:
    int ID; double GPA; string name;
public:
    Student(){}
    Student(int _ID) {ID=_ID;}
    void Input() {cin>>ID>>GPA>>name;}
    void Output() {cout<<ID<<" "<<GPA<<" "<<name;}
};

class MasterStudent : public Student
{
protected:     // Ready to be inherited
    bool passedQuals;
public:
    MasterStudent(){}
    MasterStudent(int _ID) : Student(_ID) {} // Specify Student ctor
    void Output() {Student::Output(); cout<<" "<<passedQuals; }
};
```

Whenever an object of type `MasterStudent` is constructed, the constructor of `Student` is called, then the constructor of `MasterStudent` is called. By default, the empty constructor of `Student` is called, unless the constructor is explicitly specified as in the commented line above.

Whenever an object of type `MasterStudent` is destroyed, the destructor of `MasterStudent` is called, then the destructor of `Student` is called. Remember that a stack object is destroyed when it goes out of scope, while a heap object is destroyed by calling `delete` on a pointer containing its address.

Up to this point, we have only used public inheritance. The following example illustrates the difference between various inheritance modifiers:

```
class A
{
private:    void A1() {}
protected:  void A2() {}
public:     void A3() {}
};
void TestA() {A a;  a.A1();✖  a.A2();✖  a.A3();✔}

class B : public A
{
private:    void B1() {A1();✖  A2();✔  A3();✔}
protected:  void B2() {}
public:     void B3() {}
};
void TestB() {B b;  b.A1();✖  b.A2();✖  b.A3();✔}

class C : public B
{
private:    void C1() {A1();✖  A2();✔  A3();✔}
protected:  void C2() {}
public:     void C3() {}
};
void TestC() {C c;  c.A1();✖  c.A2();✖  c.A3();✔}

class Q : private A
{
private:    void Q1() {A1();✖  A2();✔  A3();✔}
protected:  void Q2() {}
public:     void Q3() {}
};
void TestQ() {Q q;  q.A1();✖  q.A2();✖  q.A3();✖}

class Z : public Q
{
private:    void Z1() {A1();✖  A2();✖  A3();✖}
protected:  void Z2() {}
public:     void Z3() {}
};
void TestZ() {Z z;  z.A1();✖  z.A2();✖  z.A3();✖}
```

```
class P : protected A
{
private:    void P1() {A1();✖  A2();✔  A3();✔}
protected:  void P2() {}
public:     void P3() {}
};
void TestP() {P p;  p.A1();✖  p.A2();✖  p.A3();✖}

class W : public P
{
private:    void W1() {A1();✖  A2();✔  A3();✔}
protected:  void W2() {}
public:     void W3() {}
};
void TestW() {W w;  w.A1();✖  w.A2();✖  w.A3();✖}
```

Note that the inheritance modifier never affects how the derived class accesses the functions of its immediate parent class. However, when the functions of the immediate parent are accessed through the derived class from external function, or from another class, the inheritance modifier comes to work: `public` does not have any effect, `private` modifies the access of all base functions to `private`, `protected` modifies the access of `public` base functions to `protected`.

The default inheritance modifier is `private` for classes, and `public` for structs. The most common use is the `public` modifier. It is advised to write the modifier explicitly for readability.

It is possible to derive a class from several classes (multiple inheritance) as follows:

```
class A                             class B
{                                   {
public:                             public:
    void F() {}                         void F() {}
    void G() {}                         void H() {}
};                                  };

class C : public A, public B       // Inherits C from both A and B
{
public:
    void R()
    {
        G();  H();              // calls A::G() and B::H()
        F();              ✖     // Ambiguous. Do one of the following instead:
        A::F();  B::F();
    }
};
```

## 3 The open/closed principle

The open/closed principle is one of the SOLID principles stated as follows (We use the word module in our lectures to mean class or function):

"Modules should be open for extension but closed for modification."

The main benefit of this principle is that modifying a working class after the testing phase is much costly as stated at the beginning of this document. Inheritance is a mechanism that allows to extend a class (by creating a derived class from it). The derived class includes all features of the base class, plus some more possible features, which can be viewed as extending the base class. Also, it allows reusing the base class instead of rewriting its code in the derived class.