Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-2 CS213
2018/2019

Classes

Dr. Amin Allam

1 Access specifiers

Access specifiers can be applied to data members and function members of a class. There are three different access specifiers: `public`, `private`, and `protected`. `public` members can be accessed from anywhere. `private` members can be accessed only from member functions of the same class, `friend` functions, and member functions of `friend` classes. `protected` members can be accessed from derived class members (which will be explained in the inheritance lecture) and from whatever can access `private` members.

The default access specifier for a `class` is `private`, while the default one for a `struct` is `public`. This is actually the only difference between `class` and `struct` in C++. Access specifiers apply to all members listed after them, until a new access specifier appears.

```
class A
{
    int r;      // private by default
    void F() {this->r=1; this->b=2; this->G();}   ✔   // private
private:
    void G() {r=1; d=2; c=3;}   ✔   // private
    // same as:  void G() {this->r=1; this->d=2; this->c=3;} because there
    //                are no local variables or parameters with the same names
    int b;    // private
public:
    int c;    // public
    void H() {r=5; b=6; c=7; G();}   ✔   // public
    // same as:  void H() {this->r=5; this->b=6; this->c=7; this->G();}
private:
    int d;    // private

friend int foo(A*);     // foo() is friend of A, and not member in A
friend class C;    // class C is friend of A
};
```

`public` members of `class A` can be accessed from anywhere. Member functions of `class A` can access its `private` members. The global function `int foo(A*);` can access `private` members of `class A` because it is declared as `friend` in `class A`. For the same reason, all member functions of `class C` can access `private` members of `class A`.

The following code illustrates the previous discussion:

```
void fun(A a)
{
   A t;
   a.F(); a.d=7; ✖
   a.H(); a.c=9; ✔
   t.F(); t.d=7; ✖
   t.H(); t.c=9; ✔
}

int foo(A* a)
{
   A t;
   a->F(); a->d=7; ✔
   a->H(); a->c=9; ✔
   t.F(); t.d=7; ✔
   t.H(); t.c=9; ✔
   return 0;
}

class B
{
   A t;
   void P(A a) {a.H(); t.c=9;} ✔
   void Q(A& a) {t.H(); a.c=2;} ✔
   void W(A* a) {a->H(); t.c=5;} ✔
   void E(A a) {t.G(); a.b=3;} ✖
};

class C
{
   A t;
   void W(A* a) {a->H(); t.c=5;} ✔
   void P(A a) {t.G(); a.b=3;} ✔
};

int main()
{
   A* a=new A;
   a->F(); a->r=7; ✖
   a->H(); a->c=9; ✔
   delete a;
   return 0;
}
```

2 Static members

A class may contain static data members. In contrast to ordinary data member, a static data member is allocated once before creating any object, and shared between all objects of this class. A static data member is the same as a global variable, except that it is accessed only through the class name, or through any object of the class. Similarly to global variables, static data members must be defined outside class, possibly initialized.

A class may contain static function members. In contrast to ordinary function member, a static function member is not related to a specific object of this class. Hence the keyword `this` can not be used inside such functions, and static member functions can not access non-static data members or non-static function members. A static function member is the same as a global function, except that it is accessed only through the class name, or through any object of the class.

```
class S
{
private:
    int a;
    static int b;

    void F() {a=b+x+y+G();}   ✔
    static int G() {return b+y;}  ✔
    static int U() {return H()+x;}  ✖

public:
    int x;
    static int y;

    void H() {a=b+x+y+G();}   ✔
    static int P() {return G()+b+y;}  ✔
    static int V() {return H()+x;}  ✖
};
int S::b=4;          // Static data members must be defined outside class
int S::y;            // Initialized to zero by default (as global variables )

int main()
{
    S::b=5;  S::G();  ✖          S::y=7;  S::P();  ✔
    S s,t;
    cout<<S::y<<s.y<<t.y<<endl;  // Prints: 777
    s.b=8;  s.G();  ✖            s.P();  ✔
    s.x=5; s.y=5;  t.x=9; t.y=9;  ✔
    cout<<s.x<<s.y<<" "<<t.x<<t.y<<endl;  // Prints: 59 99
    return 0;
}
```

3 Constructors and destructors

A constructor of a class is a member function which is called whenever an object is created. A constructor is distinguished from other functions by having the same name of the class, and not having a return data type (not even `void`). A constructor may be overloaded, just as any other function. If the programmer does not define any constructor, the compiler provides a default empty constructor which does not take any parameters and has empty body. If the programmer defines at least one constructor, the compiler does not provide the default empty constructor.

When an object is created (as stack or heap object), its data members are constructed first, then the body of its constructor is called. It is possible to construct data members using non-zero argument constructors as will be shown in the following examples. A copy constructor is a one-argument constructor whose parameter is a reference to the same class (the parameter is possibly modified by `const`). A copy constructor is invoked whenever an object is passed or returned by value. If the programmer does not define a default copy constructor, the compiler provides a default copy constructor which calls the copy constructors of the data members.

All basic data types except references (such as `int` and `int*`) have two constructors: empty constructor and copy constructor. A reference data type (such as `int&`) has only a one-argument constructor whose parameter is the type being referenced (such as `int`).

A destructor of a class is a member function which is called whenever an object is destroyed (by going out-of-scope for stack objects, and calling `delete` for heap objects). A destructor is distinguished from other functions by having the same name of the class preceded by a tilde, and not having a return data type (not even `void`). A destructor does not take any parameters and can not be overloaded. If and only if the programmer does not define any destructor, the compiler provides a default destructor which has empty body. When an object is destroyed, the body of the destructor is called, then the destructors of its data members are called.

The following example illustrates the usage of constructors and destructors:

```
class A
{
public:
   int v;
   A() {v=0; cout<<"A empty constructor"<<endl;}
   A(int x) {v=x; cout<<"A constructor (int)"<<endl;}
   A(A& a) {v=a.v; cout<<"A copy constructor"<<endl;}
   ~A() {cout<<"A destructor"<<endl;}
};
class B
{
public:
   A a;
   B() {cout<<"B empty constructor"<<endl;}
   ~B() {cout<<"B destructor"<<endl;}
};
```

```
int main()
{
    A x;        // Prints: A empty constructor
    A r(4);             // Prints: A constructor (int)
    A z(x);             // Prints: A copy constructor
    A* w=new A;         // Prints: A empty constructor
    A* p=new A();       // Prints: A empty constructor
    p->v=5;
    A* q=new A(*p);     // Prints: A copy constructor
    cout<<q->v<<endl;    // Prints: 5
    delete w;    // Prints: A destructor
    delete p;    // Prints: A destructor
    delete q;    // Prints: A destructor
    B s;     // Prints: A empty constructor  B empty constructor
    B t(s);    // Prints: A copy constructor ( calls the default copy constructor of B)
    return 0;// Prints: B destructor A destructor B destructor A destructor (for t,s)
}              //        A destructor A destructor A destructor (for z,r,x)
```

The following example illustrates how to invoke the constructors of data members:

```
class C
{
public:
    int a;  float b;

    C(int x) : a(x), b(0) {}    // Similar to: C(int x) {a=x; b=0;}
    C(int x, float y) : a(x), b(y) {}    // Similar to: C(int x) {a=x; b=y;}
};

class D
{
public:
    C c;  int a;

    D(int i) : c(i), a(0) {}
    D(int i, float j) : c(i, j), a(0) {}
    D(int i, float j, int k) : c(i, j), a(k) {}
};
```

It is possible to use `C(int x){a=x; b=0;}` in place of `C(int x):a(x),b(0){}`, because there is almost no difference between calling the copy constructor on `int` variables (by the statement `int a(x);` or `int a=x;`) and calling the assignment operator on `int` variables (by the statements `int a; a=x;`). But it is not possible to use `D(int i){c=i; a=0;}` in place of `D(int i):c(i),a(0){}` because the first definition attempts to call the empty constructor and the assignment operator of `class C` which are not defined.