Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 15

- C++ Class Definition
- Access Restrictions
- Member Functions
- Interface and Implementation
- Constructors, Destructor
- Copy Constructor, Assignment Operator

Class Definition

```
class Pair {
  public:  // access qualifier

  // data members
  int x, y;

  // function members
  void print(ostream &os) {
    os << '(' << x << ',' << y << ')';
  }
  void init() { x = y = 0; }
};

Pair p;                     // define class variable
p.init(); p.print(cout);    // call member functions
```

Class bodies consist of declarations and definitions of data and function members

Access Restrictions

- public: the data/function member is accessible to all member functions and the owner of the class variable
- private: data/function is only accessible to member functions but not to the object owner
- protected: similar to private, used with class inheritance. Function members of derived class have access, but the object owner does not.
- default access type is private

Access Examples

```
class A {
public:

  int x;
  void foo() { x++; y--; }

private:

  int y;
  void bar() { x--; y++; }
};

A a;

a.x = 0;  // OK, public data member
a.foo();  // OK, public function member
a.y = 0;  // NOT OK, private data member
a.bar();  // NOT OK, private function member
```

Member Functions

```
Point p;

p.init();       // initialize coordinates in p
p.print(cout);  // write point p to cout
```

- Act on local data members
- Defined in class body (or outside, later)
- Can be called by the variable owner if public
- Call syntax:
  `<class-variable>.<function-name>(<param-list>);`

Member Function Implementation

C++ programs can be translated into equivalent C programs
(in fact, the first C++ compilers did just that)
How can class member functions be implemented?

- Member functions access local data
- Need object address $\rightsquigarrow$ add one parameter: pointer to object
- `Class::func(<param-list>)` $\rightsquigarrow$
  `Class_func(Class *p, <param-list>)`

```
void Point::init() { x = y = 0; }
Point a; a.init();

=> possible translation into C:

void Point_init(Point *p) { p->x = p->y = 0; }
Point a; Point_init(&a);
```


Member Function Examples

```
class String {
public:
  void set(const char *s);
  int  length() const; // const -> function can't change data
  void print(ostream &os = cout) const;
  bool palindrome() const;
  void reverse();

private:
  ... // internal data members
};

String str;
str.set("foo");
str.reverse(); // "oof"
str.print();
int l = str.length();
```


Separating Interface and Implementation

A class user does not need to know its implementation details. Knowing the public members is sufficient

Suggestions (1):

- Use a header file for each class
- Put a comment on top of the class definition describing its purpose. Briefly comment each member. The class users look at the header files to get concise documentation

Suggestions (2)

- Consider `#include` directives to incorporate private declarations into the class definition or put them at the end of the class definition. Users don't need to see them.
- Small functions that are often called should be defined in the class body. The compiler can then replace function calls by the function body (inline functions)
- Use member functions to acess data members (e.g. `set_x`, `get_x`). It simplifies debugging and is more flexible w.r.t. later implementation changes.
- Otherwise, refrain from implementations in the class body — it makes reading your code easier

Foo.H: Interface

```
#ifndef Foo_H
#define Foo_H

// What is Foo good for? ...

class Foo {

public:

  // access functions
  int get_x() const { return x; }

  void set_x(int xnew) { x = xnew; }

  // initialization
  void init();

  // print x to cout
  void print() const;

private:
  int x;
};
#endif
```

Foo.C: Implementation

```
#include "Foo.H"
#include <iostream>

void Foo::init()
{
  x = 0;
}

void Foo::print() const
{
  std::cout << x;
}
```

main.C: Application

```
#include "Foo.H"

int main()
{
  Foo a;

  a.init();  a.set_x(5);
  a.print();
  return 0;
}
```


Constructors

```
class Foo {
public:
  Foo() { x = 0; }          // constructor 1
  Foo(int x_) { x = x_; }   // constructor 2
  int x;
};

Foo a;                      // constructor 1 called
Foo b();                    // NO! - declares function b!
Foo c(10), *p = new Foo(1); // constructor 2 called
Foo d[100];                 // constr.1 called 100 times
```

- Class variables are automatically initialized by constructors
- NICE! No uninitialized struct variables anymore!
- If not defined, the (default) constructor does nothing
- Declaration syntax for class `X`: `X(<parameter-list>);`

Destructors

```
class Foo {
public:
  Foo()  { p = new int[100]; }
  ~Foo() { delete [] p; } // clean up
  int *p;
};
```

- Is called whenever a class variable leaves the scope or is deleted. NICE: automatic cleanup!
- No parameters – only one destructor.
  The default destructor does nothing
- Must be defined whenever the class object allocates resources (memory, files ...)
- Declaration syntax for class `X`: `~X();`

Copy Constructor

```
class Foo {
public:
  Foo() { x = 0; }
  Foo(const Foo &y) { x = y.x; } // copy constructor
  int x;
};

void g(Foo x) { };

Foo a;     // Constructor is called
Foo b = a; // Copy Constructor is called
g(b);      // -"-, not called if void g(Foo &x)
```

- Is called when a class variable is passed by value or a class value is assigned in a class variable declaration
- Default: direct copy (pointers: watch out!)
- Declaration syntax for class `X`: `X(const X &x);`

Assignment Operator (1)

```
class Foo {
public:
  int x;
  Foo() { x = 0; }
  Foo &operator=(const Foo &y) {
    x = y.x;
    return *this; // returns a reference to the object
  }               // itself. this points to the object and
};                // is implicitely known in member funcs.

Foo a, b;         // calls constructor
a = b;            // assignment operator called
Foo c = a;        // copy constructor called in declaration
```


Assignment Operator (2)

- The assignment operator can be overloaded for classes
- Prototype for class `X`:
  `X &operator=(const X &x);`
- Default assignment: member-by-member copy (perhaps not what you want if class has pointer members!)
- Operator = should return reference to variable – this makes `a = b = 0` is possible!


```
#include <iostream>              // Complete Example
using namespace std;

class X {
public:
  X()                      { cout << "CONSTR" << endl; }
  X(const X &x)            { cout << "COPY"   << endl; }
  X &operator=(const X &x) { cout << "ASSIGN" << endl; return *this; }
  ~X()                     { cout << "DESTR"  << endl; }
};

void g(X x) { cout << "g" << endl; }

int main()                          output:
{
  X u;                              CONSTR

  X v(u);                           COPY

  X w = v;                          COPY

  v = u;                            ASSIGN

  g(v);                             COPY
                                    g
                                    DESTR
}                                   DESTR x3
```

```
#include <iostream>       // Vector class that requires ctor,cctor,aop,dtor
using namespace std;

class V {
public:
  V(int n_)                { alloc(n_); } // creates vector of n_ elements
  V(const V &x)            { copy(x); }
  V &operator=(const V &x) { free(); copy(x); return *this; } // BUGGY!
  ~V()                     { free(); }

  int size() const { return n; } // return #elements in vector

private:
  int n;        // number of elements
  int *p;       // vector has its own array, thus shallow copy does not work!

  void alloc(int n_) { n = n_; p = new int[n]; } // allocates array

  void free() { delete [] p; }                   // releases array

  void copy(const V &x) {                        // copies array
    alloc(x.size());
    for (int i=0; i < n; ++i) p[i] = x.p[i];
  }
};
```


Shallow vs. Deep Copy

- If object only contains simple types or pointers that are shared among objects, direct (=shallow) copy is OK – no need to define the copy constructor and assignment operator
- Otherwise, use deep-copy: define cctor and aop and recursively clone data members
- Make sure there are no resource leaks and no self-assignments!

```
class X
{
public:
  X &operator= (const X &x) {
    if (this == &x) return *this; // self-assignment!
    ... // release current resources and copy x
    return *this
  }
};
```