Lecture 14

- C++ Classes
  - Intro
  - Member functions
  - Constructors, destructors
  - Copy constructor

C++ Classes

Overview: Classes vs. Structures

- Structures are **special cases** of classes
- Structures don't impose any overhead
- Structures are not initialized
- Manual structure clean-up when no longer needed

Classes

- provide additional functionality
  (some introduce run-time overhead):
  - **Access restrictions**
  - **Member functions**
  - **Automatic initialization, destruction**
  - Separation of interface and implementation
  - **Inheritance** (modeling isA relationship & more)
- also called “objects” = data + associated functions

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; }
};
```

- Syntax:
  ```
  class <class_name> {
     <class_body>
  };
  ```
- The body consists 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 (later)
- 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

```
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);
```

- 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 => add one parameter: pointer to object
  - Class::func(<param-list>) =>
    Class_func(Class *p, <param-list>)

Member Function Example

```
class String {
   public:
   void init(const char *s);
   int  length() const;
   void print(ostream &os = cout) const;
   bool palindrome() const;
   void reverse();
   private:
   ... // internal data members
};

String str;
str.init(”foo”);
str.reverse(); // “oof”
str.print();
int l = str.length();
```

const after the function declaration prevents the implementation from changing data members - safeguard!

Separating Interface and Implementation

- A class user does not need to know its implementation details. Knowing the **public members** is **sufficient**
- Suggestions:
  - 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. Should be inline functions (speed).
- 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() {
  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:
   int x;

   Foo() { x = 0; }             // constructor 1
   Foo(int x1) { x = x1; }      // constructor 2
};

Foo a;        // constructor 1 called
Foo b();      // NO! - declares function b!
Foo c(10);    // constructor 2 called

Foo *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>**);**

Destructor

```
class Foo {
   public:
   int *p;

   Foo()  { p = new int[100]; }

   ~Foo() { delete [] p; } // clean up
};
```

```
{
  Foo x;
  // calls Foo()

} // ~Foo called here
```

- 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:
   int x;

   Foo() { x = 0; }
   Foo(const Foo &y) { x = y.x; } // copy constr.
};

void g(Foo x) {};

Foo a;
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: bit-copy! (define own c.c. if pointers are used!)
- Declaration syntax for class X: X(const X &x);