Lecture Three: User-Defined Types

A declaration is a statement that introduces a name into a scope.

Before a name can be used in a C++ program, it must be declared. Most declarations are found in headers. A declaration that (also) fully specifies the entity declared is called a definition. There are many kinds of entities that a programmer can define in C++. The most interesting are: variables, constants, functions, namespaces, types (classes and enumerations) and templates.

A class is a user-defined type. It is composed of built-in types, other user-defined types, and functions. The parts used to define the class are called members. A class has zero or more members. Members can be of various types. Most are either data members, which define the representation of an object of the class, or function members, which provide operations on such objects. We access members using the object.member notation. Usually we think of a class as having an interface plus an implementation.

```cpp
#include <iostream>

using namespace std;

struct Point {
    int x, y;
};

void initPoint(Point& p, int x, int y) {
    p.x = x;
    p.y = y;
}

void displayPoint(const Point& p) {
    cout << "Point (" << p.x << ", " << p.y << ")\n";
}

int main() {
    Point p;
    displayPoint(p);
    initPoint(p, -1, 2);
    displayPoint(p);
}
```

Class members are private by default. A struct is a class where members are public by default.
The C++ language provides you with some built-in types, such as char, int, and double. A type is called built-in if the compiler knows how to represent objects of the type and which operations can be done in it (such as + and *) without being told by declarations supplied by a programmer in source code.

Types that are not built-in are called user-defined types. They can be standard library types – available to all C++ programmers as part of every ISO Standard C++ implementation – such as string, vector and ostream, or types that we build for ourselves, such as Point, Line and Triangle. The standard library types are as much a part of the language as the built-in types, but we still consider them user-defined because they are built from the same primitives and with the same techniques as the types we build ourselves; the standard library builders have no special privileges or facilities that you don’t have. Like the built-in types, most user-defined types provide operations; for example vector has [ ] and size(), ostream has <<, Shape has add(Point) and set_color() as you will see, and so on.

Why do we build types? Types are good for directly representing ideas in code. There are two aspects to a type: a) representation and b) operations. Many ideas follow this pattern: something has data to represent its current value – sometimes called the current state – and a set of operations that can be applied to it. C++ provides two kinds of user-defined types: classes and enumerations. The class is by far the most general and important, so we first focus on classes. In C++ (as in most modern languages) a class is the key building block for large programs – and very useful for small ones as well.

```cpp
#include <iostream>

using namespace std;

class Point {
private:
    int x, y; // the two coordinates of the point
public:
    Point(int x, int y); // initialize the point
    void report(); // show this point to the world
};

Point::Point(int a, int b) : x(a), y(b) {
}

void Point::report() {
    cout << "Point (" << x << ", " << y << ")\n";
}

int main() {
    Point p(-1, 2);
    p.report();
}
```

Defining a simple Point class. Notice the constructor and the instance method (an accessor).

Notice also how the constructor is defined. The : x(a), y(b) notation is how we initialize members.
Point::Point(int a, int b) {
    x = a;
    y = b;
}

This definition would have worked equally well for the Point constructor. But the distinction between
this version and the one used is that the previous approach more directly initializes the members. In the
example above we have two separate steps (first assign with default value, then initialize as desired).

Point::Point(int x, int y) {
    x = x;
    y = y;
}

Notice that this version leaves the point members (x and y) uninitialized.

Point::Point(int x, int y) {
    this->x = x;
    this->y = y;
}

We can use the this pointer if we truly want to call the arguments by the same name as the members.

#include <iostream>

using namespace std;

class Point {
private:
    int x, y; // the two coordinates of the point
public:
    Point(int x, int y); // initialize the point
    void report() const; // show this point to the world
    void translate(int dx, int dy); // mutator, changes the point
};

Point::Point(int x, int y) : x(x), y(y) { }

void Point::report() const {
    cout << "Point (" << x << ", " << y << ")\n";
}

void Point::translate(int dx, int dy) {
    this->x += dx;
    this->y += dy;
}

int main() {
    Point p(-1, 2);
    p.report();
    Point c = p;
    p.translate(2, -1);
    p.report();
    c.report();
    // Point d;
}

Note the difference between mutators and accessors, c’s copy of p and how d can’t be instantiated.
Define a class of objects of type Point, then one of type Line. Points have two coordinates and should be able to calculate their distance to other points, while lines should be able to report their lengths.

```cpp
#include <cmath>
#include <iostream>

using namespace std;

class Point {
public:
    Point(double x, double y) : x(x), y(y) { }
    double getX() const { return x; }
    double getY() const { return y; }
private:
    double x, y;
};

double Point::distanceTo(Point other) {
    double dx = x - other.getX(), dy = y - other.getY();
    return sqrt(pow(dx, 2) + pow(dy, 2));
}

class Line {
public:
    Line(Point a, Point b) : a(a), b(b) { }
    double length() { return a.distanceTo(b); }
private:
    Point a, b;
};

int main() {
    Point a(3, 0), b(0, 4), c(1, 0), d(0, 1);
    Line u(c, d);
    cout << a.distanceTo(b) << endl;
    cout << u.length() << endl;
}
```

A Point is a pair of two numbers. A Line is a pair of two Points.