Lecture Twelve: Track I Wrap-Up

In the on-line library we have two books:

- Ivor Horton’s Beginning Visual C++ 2008
- Big C++

This set of notes is summarizing the contribution of Horstmann’s book (second, on right) to this semester’s Introductory Track. There’s no cumulative, separate “Homework” section below because we will review 8 chapters and discuss 8 different problems (one from each chapter) from the book. For each problem we list the relevant section to read from the book. Solutions are provided at the end, for reference only. Give problems an honest try first.

This set of notes is the equivalent of the first six sets of notes posted earlier. Wherever possible I strongly recommend reading the whole chapter/book. In what follows I will state a problem and focus on those parts in the book that provide the solution.

Ch. 2: Numbers and Objects

Exercise P2.16

Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in boldface.

Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes

Extra credit if you can deal with the case that the first time is later than the second time:

Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes

Note that you can’t use if statements since they were not covered in the chapter (or anywhere in the book for that matter) thus far. If statements are covered in the next chapter. You can use them in this problem if you think you have absolutely no other way of solving it.
Now: what is needed to solve this problem?

Answer: simple arithmetic.

For the how: read at least 2.5.

Also, 2.7 provides a major conceptual clue. I’m not saying you need to use Time objects, I am saying that you should pay attention to what they provide (like being able to convert a time to a number of seconds from the beginning of the day).

Here’s my program in action:

```
Please enter the first time: 0858
Please enter the second time: 1002
1 hours and 4 minutes
Press any key to continue . . .
When the second time is bigger than the second
Please enter the first time: 0900
Please enter the second time: 0858
23 hours and 58 minutes
Press any key to continue . . .
When the second time is before the first one.
```

It’s all about reading int values and working with int operators such as / and % and *.

Chapter Two Highlights

Besides what’s needed in the sample problem discusses above here’s what else you can read about in this chapter: comments, good code layout, integer and floating-point numbers, arithmetic expressions and types, strings, variables, simple input and output and using objects and invoking instance methods on them. Note that all programs are discussed in detail and generic syntactic constructs are discussed in the context of specific solutions. Even in this web version diagrams are very clear (see Fig. 3, 4, for example). There is some discussion of casting in section 2.2, and of constants in the next section (which also talks about enumerated types in Advanced Topic 2.4. The main section for the posted assignment is 2.5 (Arithmetic). Pay attention to what / means as integer division and to the meaning of the % operator. Reading strings is different than reading lines (as strings). Strings are objects, like Time objects. Objects can do things for you, if you ask properly; for example, objects of type string can report their length, create a copy of (a part of) themselves, etc. Operations with strings includes concatenation. A special section is 2.6.4, which deals with formatted output. Regarding Time objects please note you need to use some of the code provided with the book. Other than that everything he says there is correct and useful and the diagrams are wonderful. If you use his library you can also produce graphics, but that’s optional in the chapter and uninteresting in the context of this class. There is also a nice summary and a link to the C++ documentation site.

Other good problems to think about in this chapter: R2.11, P2.3 (no ifs but you can use abs), P2.9, P2.10.

---

1 #insert <iomanip> with cout << setprecision(2) << setw(8) << x; and cout << setprecision(2) << setw(8) << x;
Here’s my program in action in successive instances:

There are two solutions that I present. The first one is identical with the statement. The second one only contains one test based on which the answer is determined. Read 3.1, 3.5 to understand my solutions.

Chapter Three Highlights

Branches, loops, booleans, nested branches and loops, comparing numbers and strings, infinite loops. The selection operator is sometimes useful (advanced topic 3.1). Loops: while, for, do. Advanced topic 3.5 brings up cin.fail() and cin.eof(). If you find boolean algebra confusing the review exercises are a must. Good problems to practice with: P3.4, P3.5, P3.8, P3.14, P3.22 (other than the one discussed).

Loops are very important. Take a look at this review exercise from chapter 1. Can you solve it now?

Exercise R1.16

Write an algorithm to settle the following question: A bank account starts out with $10,000. Interest is compounded monthly at 6 percent per year (0.5 percent per month). Every month, $500 is withdrawn to meet college expenses. After how many years is the account depleted?
Ch. 4: Functions

Exercise P4.21

Write a function

    bool is_inside(Point p, Circle c)

that tests if a point is inside a circle. (You need to compute the distance between p and the center of the circle, and compare it to the radius.) Write a test program that asks the user to click on the center of the circle, then asks for the radius, then asks the user to click on any point on the screen. Display a message that indicates whether the user clicked inside the circle.

For this I will provide the classes as follows:

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

using namespace std;

class Point {
public:
    Point(double x, double y) { this->x = x; this->y = y; }
    double distanceTo(Point other) {
        return sqrt(pow(this->x - other.x, 2) + pow(this->y - other.y, 2));
    }
private:
    double x, y;
};

class Circle {
public:
    Circle(Point p, double radius) : center(p) { this->radius = radius; }
    Point getCenter() { return Point(center); }
    double getRadius() { return this->radius; }
private:
    Point center;
    double radius;
};

bool isInside(Point p, Circle c) {
    // you need to provide this!
}

int main() {
    Point p(3, 4), q(-2, 1);
    Circle c(p, 10);
    cout << isInside(q, c);
}
```
Alternatively, if you don’t feel very comfortable doing this define a procedure that takes five arguments, two doubles for the point, and three doubles for the circle (two of them the coordinates of the center, and the third one the circle’s radius) and returns true or false as the coordinates of the point are inside or outside the circle.

Chapter Four Highlights

Sections 4.1, 4.2 are very basic and clear. Section 4.3 is mostly on ergonomics and reliability, while 4.4 clarifies the purpose of the return keyword and return type of a function. Parameters are discussed in section 4.5 along with the concept of declaration of a function. After a short section 4.6 on side effects (not taken in their most general meaning) the next section gives an example of a procedure as opposed to a function (a procedure does not return any value, merely producing side-effects). Section 4.8, next, becomes the first most important section thus far. Note Syntax 4.4 and Advanced Topic 4.2 and we point out that without this section it would be less clear that parameters are being passed by value.

```cpp
#include <iostream>

using namespace std;

void fun(int x) {
    x = x + 1;
    cout << x << endl;
}

int main() {
    int x = 5;
    cout << x << endl;
    fun(x);
    cout << x << endl;
}
```

Parameters are passed by value. Any changes to x in fun are local, and not visible in the caller (main).

```cpp
#include <iostream>

using namespace std;

void fun(int& x) { // notice this change here
    x = x + 1;
    cout << x << endl;
}

int main() {
    int x = 5;
    cout << x << endl;
    fun(x); // fun now can change the x in the caller
    cout << x << endl;
}
```

Passing a parameter by reference. Let’s revisit this as soon as we discuss classes and objects.
Section 4.9 is really the last important technical section of this chapter. (The rest sections are important in general, even if we were not to program in this particular programming language, that is, C++). In this section variable scope and global variables are discussed. Here’s an example of the latter:

```cpp
#include <iostream>

using namespace std;

int x;

void fun() {
    x = x + 1;
    cout << x << endl;
}

int main() {
    x = 5;
    cout << x << endl;
    fun();
    cout << x << endl;
}
```

Using global variables is like passing parameters by reference (to a certain extent).

Sections 4.10 and 4.11 develop a useful function and sections 4.12 and 4.13 discuss testing. Part of this has to do with how we manage correctness and for that we have assert. Sections 4.14 and 4.15 further discuss test suites, unit testing and the debugger.

Review exercises are recommended. Exercise 4.9 is the basic example the context where the definition of a function is traditionally useful. Once we move to next chapter we will have plenty of opportunities to write functions.

---

**Ch. 5: Classes**

---

**Exercise P5.8**

Implement a class Car with the following properties. A car has a certain fuel efficiency (measured in miles/gallon or liters/km—pick one) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0. Supply a function drive that simulates driving the car for a certain distance, reducing the fuel level in the gas tank, and functions get_gas, to return the current fuel level, and add_gas, to tank up. Sample usage:

```cpp
Car my_beemer(29); // 29 miles per gallon
my_beemer.add_gas(20); // Tank 20 gallons
my_beemer.drive(100); // Drive 100 miles
cout << my_beemer.get_gas() << "\n"; // Print fuel remaining
```
Please start by re-reading section 2.7 (Using Objects) in Chapter 2.

Section 5.1 tries to explain the process of modeling. The next section immediately delves into concrete development. Sections 5.3 and 5.4 further clarify the idea of encapsulation and discuss the definition of member functions declared in an interface. Most of what you need to solve the problem for this chapter is presented now. Notice the many examples including class Time used already in Chapter 2.

Take a look at Syntax 5.2 and Common Error 5.3 in this section. Sections 5.5 and 5.6 cover constructors with and without parameters. Notice Advanced Topic 5.1, Syntax 5.4 and Advanced Topic 5.2 which tells us that Operator Overloading is also discussed in Chapter 14. So we have, for now, the following:

```cpp
#include <iostream>
using namespace std;

class Point
{
  public:
    Point (double xval, double yval);
    void move(double dx, double dy);
    double get_x() const;
    double get_y() const;
    void report() const {
      cout << "Point at (" << x << ", " << y << ") " << endl;
    }
  private:
    double x;
    double y;
};

Point::Point(double xval, double yval) : x(xval), y(yval) { }

void Point::move(double dx, double dy) { x += dx; y += dy; }

double Point::get_x() const { return x; }

double Point::get_y() const { return y; }

int main() {
  Point p = Point(2, 3);
  p.report();
  // Point q; // won't work because no default constructor
  Point q(-1, 1);
  q.report();
}
```

Basic object-oriented modeling.

Sections 5.7 and 5.8 discuss accessing data members and compare member and non-member functions mainly through the implicit parameter that member functions receive. The end of section 5.8 discusses the components of a basic program and the next section 5.9 shows how to do separate compilation. A summary ends the chapter. Suggested problems: P.5.3, 4, 5, 6, 7, 9, 10, 17, 19.
In the context of the previous program let’s discuss this:

```cpp
#include <iostream>

using namespace std;

class Point {
public:
    Point (double xval, double yval);
    void move(double dx, double dy);
    double get_x() const;
    double get_y() const;
    void report() const {
        cout << "Point at (" << x << ", " << y << ")" << endl;
    }
private:
    double x;
    double y;
};

Point::Point(double xval, double yval) : x(xval), y(yval) { }

void Point::move(double dx, double dy) { x += dx; y += dy; }

double Point::get_x() const { return x; }

double Point::get_y() const { return y; }

void fun(Point p) {
    p.report();
    p.move(2, 2);
    p.report();
}

int main() {
    Point p = Point(2, 3);
    p.report();
    fun(p);
    p.report();
}
```

Call by value means a copy of the object is in fact passed. Not a reference like in Java.

```cpp
... void fun(Point& p) {
...
```

The only necessary change if you want to pass the points by reference.
In section 6.1 note Syntax 6.1 and 6.2 and the following code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    cout << v.size(); // size zero
}
```

Passing vectors as a parameter: same thing as passing an object. By default the arguments are passed by value. If you pass them by reference you save the time it takes to copy. If you want to pass them without changing them the use of const is recommended in conjunction with &,

```cpp
#include <iostream>

using namespace std;

void fun(const vector<int>& v) {
    v[1] = 100;
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << endl;
}

int main() {
    vector<int> v;
    v.push_back(3);
    v.push_back(2);
    v.push_back(1);
    fun(v);
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << endl;
}
```

This code obviously doesn’t compile. Remove const, recompile, run. Remove &, do the same.

Section 6.4 wraps up the vector part of this chapter. Note that you can easily have vector of objects.

Vectors have automatic memory management, unlike arrays, which are being discussed next.

Section 6.5 is a long section but it contains a lot of useful information for your problem P6.18 (which has been posted on the website a while ago). Note that you can’t allocate arrays with Syntax 6.3 unless you have a constant that is known at compile time. Note that this limitation is specific to Syntax 6.3 only.
Arrays of characters are the same as strings. Both review problems and programming exercises are recommended if you’re not very sure about the syntax. The problem you need to solve is mainly about two dimensional arrays rather than memory management.

---

**Exercise P6.18**

Implement the following algorithm to construct magic $n \times n$ squares; it works only if $n$ is odd. Place a 1 in the middle of the bottom row. After $k$ has been placed in the $(i, j)$ square, place $k + 1$ into the square to the right and down, wrapping around the borders. However, if you reach a square that has already been filled, or if you reach the lower right corner, then you must move one square up instead. Here is the $5 \times 5$ square that you get if you follow this method:

```
 11 18 25  2  9
 10 12 19 21  3
  4  6 13 20 22
 23  5  7 14 16
 17 24  1  8 15
```

Write a program whose input is the number $n$ and whose output is the magic square of order $n$ if $n$ is odd.

---

In the next chapter we learn how name of arrays are just addresses to the starting point of a contiguous area of memory. To solve the problem above just assume you are being given an array of a certain size, already allocated, ready to be filled with values.

```cpp
#include <iostream>

using namespace std;

void fun(int a[], int size) {
    a[0] = 100;
    for (int i = 0; i < size; i++)
        cout << a[i] << endl;
    cout << "------------------" << endl;
}

int main() {
    int a[] = {1, 2, 3}, size = 3;
    for (int i = 0; i < size; i++)
        cout << a[i] << endl;
    cout << "------------------" << endl;
    fun(a, size);
    for (int i = 0; i < size; i++)
        cout << a[i] << endl;
    cout << "------------------" << endl;
}
```

By default arrays seem to be passed by reference. We’ll see next chapter why.
#include <iostream>

using namespace std;

const int SIZE = 3;

void show(int a[][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++)
            cout << a[i][j] << " ";
        cout << endl;
    }
    cout << "-------------------\n";
}

void magic(int a[][SIZE]) {
    int count = 0;
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++) {
            a[i][j] = count;
            count += 1;
            show(a);
        }
}

int main() {
    int a[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
    magic(a);
}

Note that when passing a two-dimensional array to a function we must specify the number of columns as a constant with the parameter type. The number of rows can be variable. You can define magic now.

---

Ch. 7: Pointers

Here’s the problem for this chapter:

**Exercise P7.2**

Implement a class `Person` with two fields `name` and `age`, and a class `Car` with three fields:

- the model
- a pointer to the owner (a `Person*`)
- a pointer to the driver (also a `Person*`)

Write a program that prompts the user to specify people and cars. Store them in a `vector<Person*>` and a `vector<Car*>`. Traverse the vector of `Person` objects and increment their ages by one year. Finally, traverse the vector of cars and print out the car model, owner’s name and age, and driver’s name and age.
These other two problems are also worth looking at:

---

Exercise P7.5

Write a function that computes the average value of an array of floating-point data:

double average(double* a, int a_size)

In the function, use a pointer variable, and not an integer index, to traverse the array elements.

---

Exercise P7.6

Write a function that returns a pointer to the maximum value of an array of floating-point data:

double* maximum(double a[], int a_size)

If a_size is 0, return NULL.

---

In this chapter we learn to deal with memory locations better. Already 7.1 has a lot of material.

Let’s start from this:

```cpp
#include <iostream>

using namespace std;

class Point {
public:
    Point(int x, int y) : x(x), y(y) { }
    void report() { cout << "Point (" << x << ", " << y << ")" << endl; }
    int x, y;
};

void fun(Point* p) {
    p->x += 1;
    p->y += 1;
    p->report();
}

int main() {
    Point* p;
    p = new Point(2, 3);
    (*p).report();
    fun(p);
    p->report();
}
```

Basic relationship between pointers and references.

When we pass a pointer as an argument we actually pass the value the pointer points to by reference.

Deallocating memory with delete, and the address operator (already used) are in section 7.2.
No garbage collector. Allocate memory in a function, delete it. Otherwise it persists.

NULL is a keyword indicating absence of value to point to.

A reference is a pointer in disguise.

```cpp
#include <iostream>

using namespace std;

void show(int a[], int size) {
    for (int i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
}

int main() {
    int* a = new int[10];
    for (int i = 0; i < 10; i++)
        a[i] = 10 - i;
    show(a, 10);
}
```

There is an intimate connection between arrays and pointers in C++.

Stack vs. heap allocation.

Careful.

Arrays of objects.

Pointers to character strings and to functions. Pointer arithmetic.

```cpp
#include <iostream>
#include <string>

using namespace std;

class Student {
public:
    string name;
    int age;
    Student(string name, int age) : name(name), age(age) { }
    void report() { cout << name << " " << age << endl; }
};

int main() {
    Student a("Larry Bird", 56);
    Student b = a;
    a.age = 57;
    a.report();
    b.report();
}
```

In which we have two Larry Birds.
int main() {
    Student* a = new Student("Larry Bird", 56);
    Student* b = a;
    (*a).age = 57;
    a->report();
    b->report();
}

In which there is only one Larry Bird.

#include <iostream>
#include <vector>
using namespace std;

void show(vector<vector<int>*>* v) {
    for (int i = 0; i < v->size(); i++) {
        for (int j = 0; j < (*v[i]).size(); j++)
            cout << (*v[i])[j] << " ";
        cout << endl;
    }
    cout << "------------------\n";
}

int main() {
    int size;
    cin >> size;
    vector<vector<int>*>* a(size);
    for (int i = 0; i < a->size(); i++) {
        a[i] = new vector<int>(size);
        for (int j = 0; j < size; j++)
            (*a[i])[j] = 0;
    }
    show(a);
    int col = size / 2, row = size - 1;
    int k = 1;
    while (k <= size * size) {
        (*a[row])[col] = k;
        k += 1;
        int u = (row + 1) % size;
        int v = (col + 1) % size;
        if ((*a[u])[v] == 0) {
            row = u;
            col = v;
        } else {
            row -= 1;
        }
    }
    show(a);
}

Implementing the magic square program with two-dimensional vectors.
Ch. 8: Inheritance

G Exercise P8.11

Implement a base class shape and derived classes Rectangle, Triangle, and Square. Derive Square from Rectangle. Supply virtual functions double area() and void plot(). Fill a vector of shape pointers with a mixture of the shapes, plot them all, and compute the total area.

This is one of the problems suggested for this chapter.
Small fonts just help you ignore these answers as long as possible.

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Please enter the first time: ";
    int timeOne;
    cin >> timeOne;
    cout << "Please enter the second time: ";
    int timeTwo;
    cin >> timeTwo;
    int t1 = timeOne / 100 * 60 + timeOne % 100;
    int t2 = timeTwo / 100 * 60 + timeTwo % 100;
    int diff = (t2 - t1 + 24 * 60) % (24 * 60);
    cout << diff / 60 << " hours and " << diff % 60 << " minutes" << endl;
}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Please enter the year: ";
    int year;
    cin >> year;
    if (year % 4 == 0) {
        if (year < 1582) {
            cout << year << " is a leap year.\n";
        } else {
            if (year % 100 == 0) {
                if (year % 400 == 0) {
                    cout << year << " is a leap year.\n";
                } else {
                    cout << year << " not a leap year.\n";
                }
            } else {
                cout << year << " is a leap year.\n";
            }
        }
    } else {
        cout << year << " is not a leap year.\n";
    }
}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Please enter the year: ";
    int year;
    cin >> year;
    if ((year <= 1582 && year % 4 == 0) || (year > 1582 && (year % 4 == 0 && year % 100 != 0 ||
        year % 4 == 0 && year % 100 == 0 && year % 400 == 0))) {
        cout << year << " is a leap year.\n";
    } else {
        cout << year << " is not a leap year.\n";
    }
}
```

```cpp
bool isInside(Point p, Circle c) {
    return p.distanceTo(c.getCenter()) <= c.getRadius();
}
```
This is actually extremely easy.