Comp151

Introduction

Background Assumptions

- This course assumes that you have taken COMP102/103, COMP104 or an equivalent. The topics assumed are:
  - Basic loop constructs, e.g., for, while, repeat, etc.
  - Functions
  - Arrays
  - Basic I/O
  - Introduction to classes
  - Abstract Data Types
  - Linked Lists
  - Recursion
  - Dynamic Objects

Why Take This Course

You all know how to program, so why take this course?

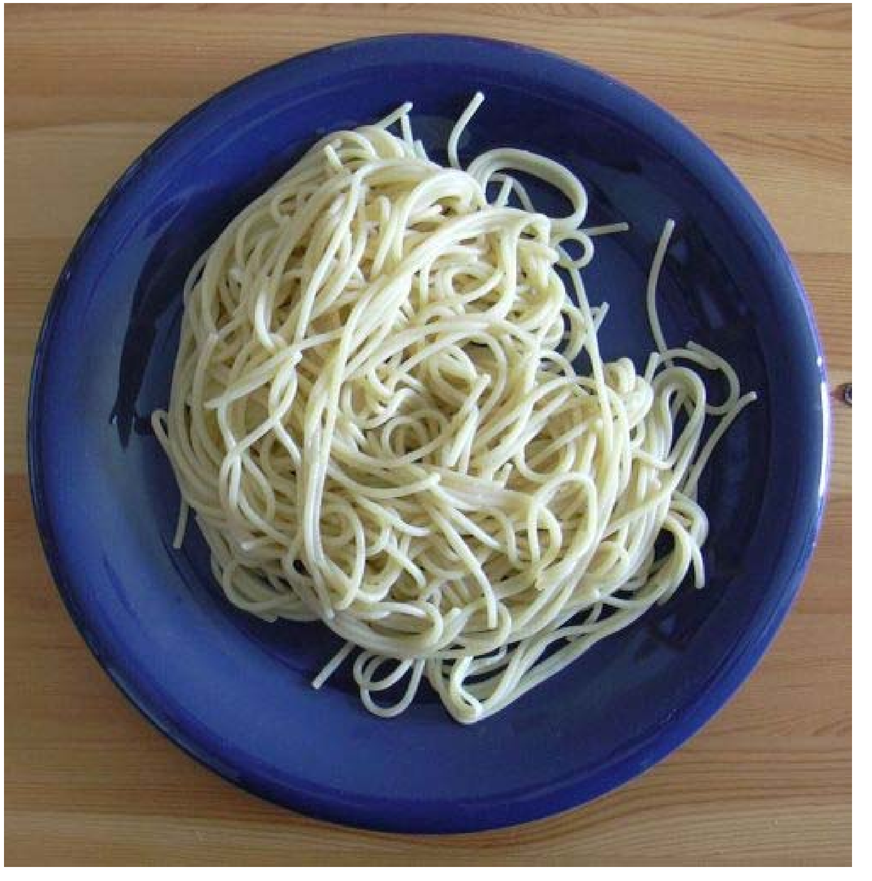
- In COMP104 you essentially only learned “the C part” of C++ and can write “small” C++ programs.
- Most of the time you write code that is (almost) the same as code that’s been written many times before. How do you avoid wasting time “re-inventing the wheel”? How do you re-use coding effort?
- What if you need to write a large program and/or work with a team of other programmers? How do you maintain consistency across your large program or between the different coders?
- In this course you will learn the essence of Object Oriented Programming (OOP). The goal is to teach you how to design and code large software projects.

A Short "History" of Computing

- Early programming languages (e.g., **Basic**, **Fortran**) were **unstructured**. This allowed "spaghetti code" – code with a complex and tangled structure - with **goto**'s and **gosub**'s jumping all over the place. Almost impossible to understand.

Spaghetti Code

```
10 k =1
20 gosub 100
30 if y > 120 goto 60
40 k = k+1
50 goto 20
60 print k, y
70 stop
100 y = 3*k*k + 7*k -3
110 return
```



- Hard to follow the flow.
- Difficult to modify.

A Short "History" of Computing

- Early programming languages (e.g., **Basic**, **Fortran**) were **unstructured**. This allowed "spaghetti code" – code with a complex and tangled structure - with **goto**'s and **gosub**'s jumping all over the place. Almost impossible to understand.

- The abuses of spaghetti code led to **structured** programming languages to support **procedural programming** (**PP**) – e.g., **Algol**, **Pascal**, **C**.

- While well-written C is easier to understand, it's still hard to write large, consistent code. This inspired researchers to borrow AI concepts (from knowledge representation, especially semantic networks) resulting in **object-oriented programming** (**OOP**) languages – e.g., **Smalltalk**, **Simula**, **Lisp** (CLOS), **Eiffel**, **Objective C**, **C++**, **Java**).

- Even with OOP, typically programmers still re-invent code instead of re-using it, often because OOP creates less efficient code. This led to **generic programming** (**GP**) – e.g., **Lisp** (CLOS), **C++**, and (sort of) the latest **Java** versions.

Procedural Programming (PP)

```
int func(int j)
{
    return (3*j*j + 7*j -3);
}

int main()
{
    int k = 1;
    while (func(k) <= 120)
    k++;
    printf("%d\t%d\n", k, func(k));
    return (0);
}
```

- Loop constructs: for, while, repeat, do-while
- Program = a set of procedures/functions
- Focus is on code: how to get things done

Problems of PP

```
const int MAX ALTITUDE = 11000;
const int MAX SPEED = 960;
struct Airplane
{
    int altitude;
    int speed;
};
void takeoff(Airplane B747);
void descend(Airplane B747, int feet);
```

- Data and code are separated. Data is passive, code is active.
- There are usually some constraints on variables, e.g., altitude of airplane must be >= 0 but less than some constant. Also, not all speeds are possible at all altitudes.

Example: Problems of PP

```
const int MAX_ALTITUDE = 11000;
const int MAX_SPEED = 960;
const MAX_RUNWAY_SPEED = 400;
const MIN_FLY_SPEED = 350;
struct Airplane
{   int altitude;
    int speed;
};
void takeoff(Airplane B747)
{
    // initial state: speed == 0, altitude = 0
    B747.speed = (MAX_RUNWAY_SPEED + MIN_FLY_SPEED) / 2;
    // accelerate and climb to 1000 ft
    B747.altitude += 1000;
    B747.speed += 200;
    // cruising speed and altitude
    B747.altitude = MAX_ALTITUDE;
    B747.speed = MAX_SPEED;
}
void descend(Airplane B747, int feet);
```

- Question: How can constraints be enforced?

How to Maintain Data Consistency

- Data/State Consistency: Every time we change the value of a member of an Airplane structure we must check to make sure that the new value is valid with respect to the values of all of the other data items (members).

- A snapshot of the values of all of the data members of an object represents the **state** of the object.

- Ensuring data consistency is one of the major challenges in a (large) software project.

- This challenge becomes even more difficult when the program is modified and new constraints are added.

Example: Adding a New Data Member

```
struct Airplane
{
    int altitude;
    int speed;

    bool flaps_out;
            // Flaps must be extended below a certain
            // speed to gain lift, but they must be
            // retracted before the speed gets too high;
            // otherwise they will be damaged
}
```

- How to ensure that flaps are not changed at wrong speed?

Solution(?): Maintain Data Consistency

```
struct Airplane
{   int altitude;
    int speed;
    bool flaps;
};

void set_speed(Airplane A, int new_speed)
{
    // make sure we don't violate any constraints
    // when changing the speed of the airplane
    // e.g., if altitude == 0 then Speed <= MAX_RUNWAY_SPEED
    //     and flaps are not extended/retracted at inappropriate speeds
}
```

- One solution: Define a restricted set of functions that access the data members (of Airplane) to ensure data consistency (of speed, altitude and flaps).

Maintain Data Consistency (cont)

- For this to work, the rest of the program <u>must</u> use <u>only</u> these functions to change an `Airplane`'s state, rather than changing the members directly.

- If we can enforce this, then modifying the `Airplane` structure is easier. We know that we only need to modify the restricted set of functions that <u>directly</u> access the members of the `Airplane` structure and make sure that they do not violate the new constraints.

- Since the rest of the program does not directly access the `Airplane` structure's members we do not have to worry about keeping the data consistent.

Maintain Data Consistency (cont)

- Question: How can we ensure that the `Airplane` structure’s members are only accessed by the restricted set of functions?

- Answer: In standard procedural programming languages we can't....

Object-Oriented Programmming (at last)

- It would be very useful if we could make it <u>impossible</u> to access the `Airplane` structure directly, i.e., to force the rest of the program to to use the restricted set of functions to access it.

- This observation leads to **OOP**.
  - In OOP the fundamental entity is the <u>class</u>.
  - In contrast with PP, in OOP objects are “alive”; they take care of their own internal state and “talk” with other objects.
  - In OOP there is little code outside of the class.
  - Instead of focusing on how to do things (implementation), classes tell the world what they can do (interface).

Example OOP Class + Function

```
class Airplane
{
public:
    void set_speed(int new_speed);
    void set_altitude(int new_altitude);
private:
    int altitude;
    int speed;
};

void some_function()
{
    Airplane marys_B747;
    marys_B747.set_speed(340);
    marys_B747.set_altitude(1500);
    marys_B747.speed = 5000; // ERROR: speed is private!!
}
```

Classes and Objects

- A **class** is a user-defined type representing a set of items with the same structure and behavior, e.g., `Airplane`.

- An **object** is a variable of a class type, e.g., `marys_747`. We also say `marys_747` is an **instance** of the `Airplane` class.

- **Instantiation**: The process of creating an object (an instance of a class type) is called **instantiating** an object, e.g., `Airplane marys_B747;`

- Each object of a class has its own data members with their own values.

- All objects in the same class share a common set of member functions. Example: `Airplane marys_B747, joes_DC10;`
  `marys_B747` and `joes_DC10` have different speed and altitude values but share the `set_speed` and `set_altitude` functions.

Classes and Objects (cont)

- When calling a procedure in PP, we are simply
  “calling function X” or
  “calling X”.

- In OOP, we are formally
  “invoking method/operation/function X on object Y
  (where object Y is of class Z)”

  – e.g., invoking function `set_altitude` on object `marys_B747` of class `Airplane`.

OOP Does Not Magically Impose Good Design

```
struct Airplane
{
    int speed;
    int altitude;
};

// … is the same as  …

class Airplane
{
public:
    int get_altitude { return altitude;}
    int get_speed() { return speed;}
    void set_speed(int x) { speed = x;}
    void set_altitude(int x) { altitude = x;}
private:
    int altitude;
    int speed;
};
```

- In fact, it can make things <u>worse</u>!!

Things That OOP Supports

- Data Abstraction (Abstract Data Type)
- Data Encapsulation (Data Hiding)
- Inheritance (Hierarchy Among Classes)

- With enough self-discipline, most OOP programming can be done in C or other PP languages.
- What makes an OOP language different is the support and <u>enforcement</u> of the above three concepts built into it.
- We will spend much of this semester learning and mastering the above concepts.

The Holy Grail: Reusable Code

- Designing reusable code in which modules can be reused as basic building blocks and plugged in where necessary has long been a dream.

- Code is reusable if:
  - It is easy to understand
  - It can be assumed to be correct
  - Its interface is clear
  - It requires no changes to be used in a new program

- When properly applied OOP and GP help us get closer to our goal of reusable code.