Static, Reference, Const

Static member

- Member variables and functions that are shared by all instances belonging to the same class
  - Variables that every instances can commonly share
  - Functions that are independent of member variables

  → **Define them as static**

Sample code with static

```
class Health {
private:
      char  *name;
      float  height, weight;
      static char *heightUnit, *weightUnit; // "cm","m","g","kg",etc.
public:
      Health (char *n, float h, float w); {
            name = n; height = h; weight = w;
      }
      static char* getUnit(char type) { //get unit for height or weight
            if (type == 'h') return heightUnit;
            else if (type == 'w') return weightUnit;
      }
      static void setUnit(char type, char* unit) {
      ...
};
```

Caution in using static (1)

- Static function **cannot** include ordinary member variables and functions
  - They only can be used as arguments of the function

✕
```
static int getBMI () {
    return weight / (height * height);
}
```

○
```
static int getBMI (Health *data) {
    return data->weight/(data->height * data->height);
}
```

Caution in using static (2)

- Static function can NOT be called from an object

  Health data;

  ✕ char *hUnit = data.getUnit ('h');

  – Instead, add the class name as prefix

    char* hUnit = Health::getUnit ('h');

  – prefix can be omitted within Health class

    char* hUnit = getUnit ('h');

Variable as Reference

Reference

- Set the same entity as a argument (variable)
- Argument for function is NOT a copy but its reference
  - In C language, argument is a copy of a variable
  - In C, reference to variables is a copy of pointer value
  - In C++, reference is obtained by adding & notaion after type name
    - Argument is treated as normal variable, NOT as pointer
    - The value of referenced variable can be updated

Sample code using reference

```
void swap(int& x, int& y) {// C lang -> int *x,*y
    int tmp = x; // C lang -> tmp = *x;
    x = y;          // *x = *y;
    y = tmp;        // *y = tmp;
}
int main() {
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d, b = %d¥n",a,b); //a = 2, b = 1
}
```

Usage of reference

- Can avoid the copy of large size object, by which efficiently set the value to a function
  - Copy only the head address of the object
- The value of variable can be updated ← careful monitoring is required for some cases
  - const declaration can be used for restricting the update (as explained later)
- Frequently used in overloading operators (as explained later)

Caution in using reference

- Reference is an alias name of object;
  its definition alone causes an error
  × int &a;
  ○ int &a = n; // if int n; exists
- Cannot be used as reference for pointer
  × int& *ary = new int [10];
  × int& ary[] = new int [10]
  ○ int *ary = new int [10];

Constant values with const modifier

const modifier

- Declare a variable as a constant value
  - The value can be assigned only in initialization
- Enhance optimization by compiler
- Declare const keyword before type definition
- Applicable to the arguments of function
  - Restrict update for referenced variables with &
- Applicable to a whole process of function
  - All objects within the function cannot be updated

Usage of const

```
class Health {
private:
    const char  *name; // name cannot be updated
    float  height, weight;
    const float averageBMI = 20.0; // averaged BMI
public:
    Health (const char *n, float h, float w); {
        name = n; height = h; weight = w;
    }
    float getBMI() const; // member variables cannot
            // be updated within this function
};
```

const modifier for string variables

- Re-assignment is allowed to the variables defined as const char *

```
const char *name = "TUT"; // initialization
name = "CS"; // ◯ you can re-assign
```

- However, you cannot **edit** the string as

```
name[1] = 'O'; // × this causes compile error
```