Chapter 10.3 – Classes, Pointers, and Dynamic Arrays

It is possible to dynamically allocate objects.

```
struct Record {
        int number;
        char grade;
};

int main() {
        Record *p;
        p = new Record;
}
```

The **arrow operator** (->) combines the actions of a dereferencing operator (*) and a dot operator (.) to specify a member variable. Example)

```
        p->number = 2001;
(or)    (*p).number = 2001;
```

The **this** keyword is a pointer that points to the calling object.

```
class Student {
      string name;
      int *grades; // a dynamically allocated array, contains grades
for the student
      int size; // size of the grades array

public:
      Student(string n, int size) {
            name = n;
            this->size = size;
            grades = new int[size];
      }
};
```

The rule of "Big-3": if a class defines one of the following:

1) **Destructor**
2) **Copy constructor**
3) **Assignment operator**

then it should explicitly define all 3 and not rely on their default implementation.

Destructors

A destructor is a member function that is called automatically when an object goes out of scope. Since dynamically allocated variables must be explicitly “deleted” to free memory space, destructors must be defined for member variables using dynamic allocation.

```
class Student {

…
public:
     ~Student();
};

Student::~Student() {
     delete[] grades;
};
```

Overloading the Assignment (=) Operator

- The default assignment operator makes a shallow copy of each member variable from one object to another. It copies the values over.
- The overloading assignment operator function must be a member of a class; it cannot be a friend of the class.

Example)

```
class Student {

…
public:
     Student& operator=(const Student& rtSide);
};

Student& Student::operator= (const Student& rtSide) {

     if (this == &rtSide)
          return *this;

     name = rtSide.name;
     size = rtSide.size;
     delete[] grades;
     grades = new int[size];
     for (int i = 0; i < size; i++)
          grades[i] = rtSide.grades[i];
     return *this;

}
```

Copy Constructors

The copy constructor is called in the following instances:

1. When a class object is being declared, and is initialized by another object of the same type in parenthesis
   ```
   ex)  Student a(“John Smith”, 20);
        Student b(a);
   ```
2. When a function return a value of the class type.
3. When an object is passed-by-value.

```
class Student {

…
public:
     Student (const Student& studentObject);
};

Student::Student(const Student& studentObject) {
     name = studentObject.name;
     size = studentObject.size;
     grades = new int[size];
     for (int i = 0; i < size; i++)
          grades[i] = studentObject.grades[i];
}
```