Week 4-1 : Constructor, Destructor

Part1. Constructor

- 생성자의 외형

클래스의 이름과 함수의 이름이 동일.
반환형이 선언되어 있지 않으며, 실제로 반환하지 않음.
함수의 일종으로 오버로딩과 매개변수에 디폴트 값 설정이 가능.

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

class Constructor
{
int num1;
int num2;

public:
Constructor()
{
num1=0;
num2=0;
}
Constructor(int n)
{
num1=n;
num2=0;
}
Constructor(int n1, int n2)
{
num1=n1;
num2=n2;
}
```

```
/* 디폴트 매개변수 생성자
Constructor(int n1=0, int n2=0)
{
num1=n1;
num2=n2;
}
*/

void ShowData() const
{
cout<<num1<<' '<<num2<<endl;
}
};

int main(void)
{
Constructor sc1;
sc1.ShowData();

Constructor sc2(100);
sc2.ShowData();

Constructor sc3(100, 200);
sc3.ShowData();
return 0;
}
```

sc1, sc2, sc3 객체를 생성하면서 생성자를 거치는데 각각 오버로드 된 생성자를 거침을 확인할 수 있다. 디폴트 매개변수의 생성자를 위해서는 나머지 생성자들을 다 지워야 하며, 실행결과는 동일하다.

- 멤버 이니셜라이저를 통한 초기화

멤버변수로 선언된 객체의 생성자 호출에 활용됨.
몸체부분에서 초기화를 시키는 것이 아니라, 매개변수 옆에서 초기화를 하는 형태

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

```
class Constructor
{
int num1;
int num2;

public:

Constructor(int n1, int n2) : num1(n1),     num2(n2)
{

}

void  ShowData() const
{
cout<<num1<<' '<<num2<<endl;
}
};

int main(void)
{
Constructor sc(100,200);
sc.ShowData();

return 0;
}
```

Part2. Destructor

생성자에서 할당한 리소스의 소멸에 사용
new 연산자를 이용해서 할당해 놓은 메모리 공간이 있다면, 소멸자에서 delete 연산을 이용해서 이 메모리를 소멸

참고>> new와 delete

c언어의 malloc과 free에 대응되는 것으로 c++에서는 new와 delete를 사용한다. 객체를 생성할 때는 반드시 new를 사용하여야 한다.

```
#include <iostream>
```

```
#include <cstring>
using  namespace std;

class Book
{
private:
char * bookName;
int bookNum;
public:
Book(char  *  tempName,  int tempNum)
{
int len=strlen(tempName)+1;
bookName=new char[len];
strcpy(bookName, tempName);
bookNum=tempNum;
}

void  ShowBookInfo() const
{
cout<<"도서명 : "<<bookName<<endl;
cout<<"도서번호  : "<<bookNum<<endl;
}

~Book()
{
delete []bookName;
cout<<"destructor"<<endl;
}
};

int main(void)
{
Book book1("Computer Programming", 2001001);
Book book2("This is C++", 400010);
book1.ShowBookInfo();
book2.ShowBookInfo();
return 0;
}
```

Week 4-2 : Copy Constructor, Vector, Polymorphism

Part3. Copy Constructor

- **복사 생성자**

매개변수에 생성된 이름을 다시 호출하면 객체를 복사함.
복사 생성자를 정의하지 않으면, 멤버 대 멤버의 복사를 진행하는 디폴트 복사 생성자가 자동 삽입됨.

- **변환을 통한 복사 생성자의 종류**

묵시적 변환 : = , 명시적 변환 : (객체)
묵시적 변환을 막기 위해서 explicit을 사용

- **복사 생성자의 호출 시점**

1. 기존에 생성된 객체를 이용해서 새로운 객체를 초기화하는 경우

```
Point x2(x1);
```

2. Call-by-value 방식의 함수호출 과정에서 객체를 인자로 전달하는 경우

```
Point  copyFunc(Point obj)
{
        return obj;
}
```

3. 객체를 반환하되, 참조형으로 반환하지 않는 경우

```
Point  copyFunc(Point obj)
{
        return obj;
}
```

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

```
class Book
{
private:
char * bookName;
int bookNum;
public:
Book(char  *  tempName,  int tempNum)
{
int len=strlen(tempName)+1;
bookName=new char[len];
strcpy(bookName, tempName);
bookNum=tempNum;
}

void  ShowBookInfo() const
{
cout<<"도서명 : "<<bookName<<endl;
cout<<"도서번호  : "<<bookNum<<endl;
}

~Book()
{
delete []bookName;
cout<<"destructor"<<endl;
}
};

int main(void)
{
Book book1("Computer Programming", 2001001);
Book  book2("This  is  C++", 400010);
Book book3(book2);
book1.ShowBookInfo();
book2.ShowBookInfo();
book3.ShowBookInfo();
return 0;
}
```

복사생성자를 따로 정의하지 않았으며, 때문에 디폴트 복사 생성자가 멤버 대 멤버를 단순히 복사한다. 위 코드의 경우 에러가 발생하는데, 디폴트

복사 생성자의 경우 같은 책이름에 대한 문자열을 가리키게 되고, book2에서 소멸자를 수행한 뒤, book3 소멸자를 진행해야 하나 벌써 소멸된 문자열에 대해서 소멸을 진행하려 하기 때문에 에러가 발생한다.

이를 해결하기 위하여, 책이름에 대한 문자열을 다른 메모리에 복사하는 것이 필요하며, 이러한 복사 생성자를 사용하는 것을 '깊은 복사'라고 한 다.

```
Book(const Book& copy) : bookNum(copy.bookNum)
{
bookName = new char[strlen(copy.bookName)+1];
strcpy(bookName, copy.bookName);
}
```

- **Vector**

임의 접근 반복자(Random Access Iterator)를 지원하는 배열 기반 컨테이너.
하나의 메모리 블록에 원소가 연속되어 저장됨.

```
template<typename T, typename Allocator = allocator<T>>
class vector
```

v.pop_back() : v의 마지막 원소를 제거한다.
v.push_back() : v의 끝에 원소를 추가한다.

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

using namespace std;

int main(void)
{
vector<int> v;
v.push_back(5);
v.push_back(2);
v.pop_back();
}
```

- Polymorphism

문장을 같으나 결과가 다름

상속 : 상위 클래스가 가지고 있는 모든 멤버를 물려받도록 구현하는 방식, 하위 클래스에는 하위 클래스에 선언된 멤버와 함께 상위 클래스에 선언된 멤버도 존재하게 된다. Is a 관계일 때 성립한다.

- 상속의 형태

```
class Person
{
private:
        int age;
        char name[50];
public:
        Person(int myage, char * myname) :    age(myage)
        {
        strcpy(name, myname);
        }
        void  ShowName() const
        {
        cout<<"My  name is"<<name<<endl;
        }
        void  ShowAge() const
        {
        cout<<"My  age is"<<age<<endl;
        }
};

class  Student  :  public Person
{
private:
```

```
        char major[50];
public:
        Student(char * myname, int myage, char * mymajor) : Person(myage,
myname)
        {
        strcpy(major, mymajor);
        }
        void  ShowStudent() const
        {
        ShowName();
        ShowAge();
        cout<<"My  major is"<<major<<endl<<endl;
        }
};
```

학생은 사람이라는 is a 관계가 성립하며, 위 코드는 학생이 사람의 특성을 상속받는 형태로 구현되었다. 학생은 public Person의 코드 부분을 통해서 상속이 구현된 것이고, 이 통해서 Person의 멤버들도 상속받는다.