기초 C++ 프로그래밍

목 적 : C++는 기존의 C언어로부터 객체 지향 프로그래밍을 지원하기 위해 만들어진 언어로서 현재 가장 중요한 프로그래밍 언어 중 하나로 자리 잡은 언어이다. C언어의 특성을 그대로 물려받으면서도 객체 지향 프로그래밍이 가능한 C++ 프로그래밍의 기초적인 작성 방법 및 응용을 살펴보고 객체 지향 프로그래밍의 특성을 공부해본다.

목차

1. 기본 개념

C++는 기존의 C언어로부터 객체 지향 프로그래밍을 지원하기 위해 만들어진 언어로 1990년대부터 가장 중요한 프로그래밍 언어로 자리 잡았으며 앞으로도 계속 발전할 것으로 보인다. C++는 효율적이면서도 크기가 작고 실행속도가 빠르며 호환성이 뛰어난 프로그램을 생성하는 C언어의 특성을 그대로 물려받으면서도 현대 프로그래밍에서 가장 중요하게 여기고 있는 개념인 **객체 지향 프로그래밍**의 특성을 사용하여 날로 복잡해지는 현대적인 프로그래밍 작업을 구조화하여 사용할 수 있도록 해준다.

이번 단원에서는 C++프로그래밍의 기초적인 작성 방법 및 응용을 다루며 객체 지향 프로그래밍의 특성에 대해 공부해 본다.

1-1. C++의 특징

C++는 다음 세 가지 프로그래밍 방식을 모두 포함하고 있다.

- ☛ 절차 언어(Procedural Language) 방식
- ☛ 객체 지향 언어(Object-Oriented Language: OOP) 방식
- ☛ 템플릿(Template)을 이용한 일반화 프로그래밍(Generic Programming) 방식

C++는 C언어와 동일하게 절차 언어 방식의 프로그래밍 작성이 가능하다. 여기에 몇 가지 기능을 더 추가하여 객체 지향 언어 방식을 지원한다. 그러나 중요한 점은 C++가 기존의 절차 언어인 C와 유사한 문법을 지니고 있다고 해서 단지 몇 가지 키워드나 구문만 더 배우면 끝나는 간단한 문제가 아니라는 것이다. 기존의 절차 언어 방식의 프로그래밍 구조와는 전혀 다른 방식의 객체 지향 프로그래밍 구조를 학습하여야 한다. C++에 **클래스(Class)**라는 특성을 추가함으로써, C++는 객체 지향적인 언어 방식도 표현 가능하게 되었다. 이번 실험에서는 C++를 이용한 객체 지향 프로그래밍에 중점을 둔다. C++를 사용하는 주된 이유는 객체 지향 프로그래밍을 지원하기 때문이다.

C++가 지원하는 또 다른 프로그래밍 기법인 일반화 프로그래밍의 목적은 데이터 타입과 무관한 코드를 작성하는 것이다. 이를 위해 C++에서는 **템플릿(Template)**을 사용하여 일반형을 통해 함수나 클래스를 정의한다. 또한 유용한 템플릿을 모아 놓은 **표준 템플릿 라이브러리(STL: Standard Template Library)**를 사용할 수 있다. 최근 프로그래밍 추세에서는 이러한 부분을 강조하고 있다.

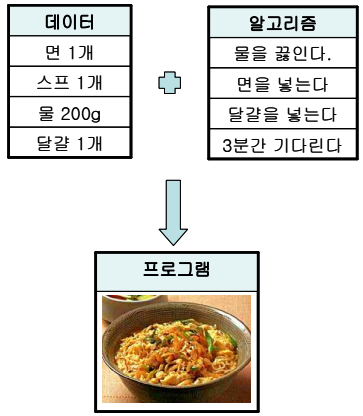
1-2. 간략한 역사

1-2-1. C언어

C언어는 1969년 AT&T 연구소의 *Ken Thompson*과 *Dennis Ritchie*에 의해서 작성된 언어로서 UNIX 운영체제를 개발하기 위한 프로젝트의 일환으로 만들어진 구조화된 프로그래밍을 지원하는 시스템 프로그래밍 언어이다. 운영체제는 컴퓨터의 자원을 관리하고 사용자와 컴퓨터를 연결시켜 주는 프로그램이므로 하드웨어를 효율적으로 제어하기 위해, 크기가 작고 빠르게 실행되는 프로그램을 만들 수 있으며 간결한 언어가 필요하다. C언어 이전에는 어셈블리 언어로 이러한 운영체제를 개발하였으나, 어셈블리의 단점은 해당 시스템에 알맞은 어셈블리 언어를 사용해야 했으므로 특정 컴퓨터에서 사용하기 위해서는 프로그램을 처음부터 다시 작성해야 하는 어려움이 있었다. 이러한 문제를 해결하기 위해 그 당시 **고수준 언어(High-level Language)**로 C언어가 등장하게 되었다. 고수준 언어는 특정 언어로 작성된 프로그램을 여러 종류의 컴파일러를 사용하여 특정 컴퓨터에 맞는 내부 언어로 바꿔 사용하는 기능을 제공한다. *Ritchie*는 저수준 언어의 효율성과 하드웨어 접근 능력, 고수준 언어의 일반성과 이식성이 결합된 언어를 원했고, 따라서 C언어를 탄생시켰다.

1-2-2. C언어의 철학

일반적으로 컴퓨터 언어는 데이터와 알고리즘이라는 두 가지 개념을 다루게 된다. 데이터는 프로그램이 사용하고 처리하는 정보로서 대개 사용자가 입력을 하게 되는 정보를 의미하고, 알고리즘은 이러한 데이터를 처리하는 방법을 의미한다. 이 두 가지 요소를 결합하여 하나의 프로그램이 완성된다.



[그림 1] 데이터 + 알고리즘 = 프로그램

C언어는 개발될 당시의 주요 다른 언어들과 마찬가지로 알고리즘에 더 치중하는 **절차적(구조적) 프로그래밍** 방식을 다룬다. 절차적 프로그래밍이란 컴퓨터가 수행해야 할 동작들을 명확히 구분하고 그 구분된 동작들을 프로그래밍을 통해 구현하는 것이다. 이를 위해 컴퓨터 과학자들은 세련된 구조적 프로그래밍 방식을 제안하였고 C언어는 이러한 특성을 받아들여 if-else, for, while 등의 분기문과 반복문들을 사용하여 구조적 프로그래밍을 작성한다.

1-2-3. C++ 언어

C++는 1979년 *Bjarne Stroustrup*이 C언어를 발전시키기 위해 시작한 **"C with Classes"**의 작업으로부터 시작되었다. 이후 이는 1983년 C++(++는 증가 연산자를 뜻함)로 개명되었다.

그는 Simula(처음으로 객체 지향을 도입한 프로그래밍 언어)가 큰 규모의 소프트웨어 개발에 도움이 되는 중요한 특성들을 많이 갖고 있음을 알고 있었다. 그러나 Simula를 실제로 소프트웨어 개발에 사용하기에는 너무 느렸다. 그래서 그는 C언어의 범용성, 빠른 속

도, 높은 보급률에 주목하여, C를 Simula같은(Simula-like) 특성들을 통해 확장하고자 하였다. 그리하여 그는 C에 클래스, 파생 클래스(Derived Class), 강한 타입 체크 등등을 추가하고, 이 언어를 C로 변환하는 컴파일러인 *Cfront*를 만들었다. 그리고 1985년 8월 C++의 최초의 상업적 버전을 발표하였다.

이후 *Stroustrup*에 의해 C++는 지속적으로 변화되고 확장되었으며, 1998년 표준화를 위해 국제 표준화 기구(ISO)에서 C++의 국제 표준(ISO/IEC 14882:1998, 보통 C++98로 알려짐)이 제정된 이후 지속적으로 산업계의 새로운 필요들을 충족시키고 더 나은 언어를 만들기 위한 표준화 작업이 진행되고 있다. 현재 2011년 8월에 제정된 **C++11**이 가장 최신의 개정된 C++ 표준이다.

C++의 초기 이름에서 살펴볼 수 있듯이, C에서 객체 지향을 지원 가능하게 하기 위해 가장 중요하게 여겨진 개념이 **클래스(Class)**이다. 따라서 이 클래스를 제대로 이해하고 클래스 중심의 프로그래밍 방법을 습득하는 것이, 객체 지향 개념 및 C++의 습득에 가장 중요한 영향을 끼친다.

유용하고 신뢰성 있는 클래스를 설계하는 것은 어려운 작업이다. 그러나 다행히도 OOP 언어는 이미 만들어진 클래스를 새로운 프로그램에 결합하는 것이 매우 간단하며 많은 소프트웨어 회사들이 다양하고 유용한 클래스 라이브러리들을 개발하여 시판하는 중이다. 이미 누군가 만들어 놓은 신뢰할 수 있는 코드를 재활용하고 적용할 수 있다는 점은 C++의 강점 중 하나라 하겠다.

2. C++ 시작하기

프로그램을 작성하는데 있어서 기본적인 구조를 이해하는 일은 가장 먼저 선행되어야 한다. 이번 장에서는 C++ 프로그램의 가장 기본적인 구조를 전반적으로 훑어보도록 한다. C++가 C언어의 기본 문형을 포함하고 있기 때문에 C언어를 어느 정도 숙지하고 있다는 가정 하에서 두 언어를 비교하며 C++ 구조를 살펴보도록 한다.

2-1. main() 함수의 사용

example1.cpp

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

int main(void)
{
        int i=3;
        cout<<"Number is "<<i<<endl;
        return 0;
}
```

C++ 프로그램의 기본형

```
int main(void)
{
        명령어들;
        return 0;
}
```

[Code 1] 간단한 C++ 함수 및 기본형

[Code 1]로부터 main()이 하나의 함수라는 사실과 이 함수가 어떻게 동작하는지를 쉽게 생각해 볼 수 있다. 우선 함수 머리(Function Heading)인 int main()과 중괄호 {}로 묶인 함수 몸체(Function Body)로 구분되며, main()함수 끝에 있는 return 명령문은 함수의 실행을 종료하고 리턴 값을 반환한다.

C++ 함수의 기본적인 사용법은 다음과 같다.

☞ C언어에서의 함수 사용법과 동일

☞ 함수 머리로부터 호출 함수와 피호출 함수 사이에 넘겨지는 정보들을 파악

ex) int func(int a, double b): func함수는 a, b를 전달인자 리스트로 받고 int형을 반환

이런 일반적인 규칙이 main()함수에 대해서는 다소 혼동되는 점이 있는데 그 이유는 프로그램 내의 어느 부분에서도 main()함수를 호출하지 않기 때문이다. 그러나 사실 프로그램이 처음 시작될 때 main()함수는 운영체제에서 먼저 호출되어진다. 따라서 main()함수는 운영체제와 프로그램 사이의 연결 고리 역할을 한다고 볼 수 있다.

앞의 예제에서 알 수 있듯이 main()함수에서는 전달 인자 리스트가 존재하지 않으며 리턴형은 int형임을 알 수 있다. 여기서 리턴형은 프로그램이 운영체제에게 현재 프로그램 수행 결과가 성공적인지 아닌지를 알려주는 값을 의미한다(main()함수의 return값이 0인 경우 성공적인 수행).

ANSI/ISO C++ 위원회에서는 return 0; 라는 문장을 사용자가 명시적으로 지정하지 않더라도 main() 끝에는 항상 이것이 있다고 간주하도록 결정하였다. 그러나 가급적 위 예제 형태의 main()을 사용하기를 권한다.

2-2. I/O Stream의 활용

C언어에서는 stdio.h 헤더 파일을 프로그램에 상단에 포함시켜 선행처리기(preprocessor)로부터 번역을 수행하여 각종 입출력 함수에 대한 기능 정의 및 함수 선언을 해당 프로그램에 포함시킨다. 마찬가지로 C++에서는 iostream에 입출력에 관련된 모든 라이브러리 함수들이 선언되어 있으며 가장 자주 사용하게 되는 cout, cin 객체에 대한 정보도 이곳에 포함되어 있다. (iostream의 io는 입력(input)과 출력(output)을 나타낸다.) 따라서 C++에서 입출력 관련 라이브러리를 사용할 경우 iostream을 프로그램 상단부에 포함하도록 한다.

여기서 헤더에 .h 확장자가 붙지 않는 것을 확인할 수 있는데, 이는 C++의 표준화가 처음 이루어진 1998년, 기존에 사용하던 비표준 라이브러리들과 표준 라이브러리들을 구별하기 위해서 표준 라이브러리의 모든 요소를 std namespace에 넣고 이를 .h를 제거한 헤더를 별도로 제공하기 시작한 것에서 비롯된 구문이다. **이름 공간(namespace)**은 동시에 여러 라이브러리를 프로그램에 포함하였을 때, 같은 이름의 요소들이 동시에 나타나 충돌하는 경우를 피하기 위해 도입된 것으로, 표준 라이브러리는 모두 std에 속해 있으므로 클래스나 함수 등을 사용할 때 앞에 std::을 명시해 주어야 하는 불편함이 있는데, 이를 해결하기 위해서 프로그램 상단의 헤더 포함 부분 아래에 using namespace std;를 추가한다. 앞으로 표준 라이브러리를 사용할 때에는 관습적으로 이 구문을 포함할 것이다. 만약 이름 공간을 사용하지 않을 경우에는 아래와 같이 코드를 작성한다.

example1.cpp

```
#include <iostream>

int main(void)
{
        int i=3;
        std::cout<<"Number is "<<i<<std::endl;
        return 0;
}
```

2-2-1. Output stream의 활용

C++에서는 output을 위한 표준 출력 객체로 cout을 제공한다. cout은 <<연산자를 이용하여 여러 가지 자료형을 스스로 판단하여 출력하여준다. cout은 사용자 인터페이스가 매우 간단하며 이해하기도 쉽다(<<연산자는 cout쪽으로 흘러들어가는 형태로 볼 수 있다).

example2.c

```
#include <stdio.h>

int main(void)
{
        printf("Hello World!\n");
        return 0;
}
```

example2.cpp

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

int main(void)
{
        cout<<"Hello World"<<endl;
        return 0;
}
```

[Code 2] C와 C++에서의 기본 출력 (1)

[Code 2]에서 <<는 그 문자열을 cout객체에 전달한다는 의미이며 cout은 문자열, 수, 개별적인 문자들을 포함한 여러 가지 정보를 출력하는 방법을 알고 있는 미리 정의된 객체이다. 따라서 "Hello World"라는 문자가 cout객체에 전달되어지며 이것이 화면에 출력되어지게 된다. endl은 개행문자('\n')에 해당한다.

example3.c

```
#include <stdio.h>

int main(void)
{
        int i = 5;
        char ch = 's';
        double pi = 3.14;
        printf("%d,%c,"%f\n",i,ch,pi);
        return 0;
}
```

example3.cpp

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

int main(void)
{
        int i = 5;
        char ch = 's';
        double pi = 3.14;
        cout<<i<<","<<ch<<","<<pi<<endl;
        return 0;
}
```

[Code 3] C와 C++에서의 기본 출력 (2)

C언어의 기본 출력 함수인 printf()는 data type에 따라 출력 형태를 %d %s %f 등으로 구분되어지나, cout 객체는 그런 구분 없이 << 연산자를 사용하여 모든 형태를 제어한다. (option 기능을 통해 printf()가 가지고 있는 세밀한 출력 제어도 cout 객체로 표현 가능)

2-2-2. Input stream의 활용

C++에서는 input을 위한 표준 입력 객체로 cin을 제공한다. cin은 >>연산자를 이용하여 여러 가지 자료형을 입력받는다. cout객체와 마찬가지로 자주 사용되는 입출력 객체이다. (>>연산자는 cin에서 나가게 되는 형태로 볼 수 있다)

example4.c

```
#include <stdio.h>

int main(void)
{
        int i;
        printf("Input number : ");
        scanf("%d",&i);
        printf("Number is %d\n", i);
        return 0;
}
```

example4.cpp

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

int main(void)
{
        int i;
        cout<<"Input number : ";
        cin>>i;
        cout<<"Number is "<<i<<endl;
        return 0;
}
```

[Code 4] C와 C++에서의 기본 입력

[Code 4]에서는 int형의 변수 i에 값을 대입하여 출력하는 프로그램을 작성한 것으로 cin 객체는 이것이 int형임을 스스로 판단한다.

2-3. 선언 명령문과 변수

변수(Variable)란 프로그램 내에서 하나의 값을 저장할 수 있는 기억 장소를 의미하며, 프로그램 실행 시 그 값이 언제나 변경 가능하므로 상수와 대조되는 값이다. C++의 변수 선언 및 자료형은 기본적으로 C와 동일하다. 그러나 example5.cpp에서 보면 알 수 있듯이 C언어와는 달리 C++에서는 함수 중간에 변수를 선언하고 사용할 수 있다.

(※주의: 1983년에 제정된 ANSI C 표준에서는 함수의 처음 부분에서만 변수 선언이 가능하나, **99년에 제정된 C언어 표준인 C99에는 변수 선언이 함수 중간에도 가능하도록 변경**되었다. 그러나 오래 전에 출판된 서적에서나 또는 **몇몇 컴파일러에서는 아직 ANSI C를 따르기 때문에** 문제가 생길 수 있다(Microsoft Visual C++는 2010년 현재에도 C99를 지원할 예정이 없다). C99의 이와 같은 변화 또한 C++의 표준화 작업 등의 결과에서 영향을 받은 것이다.)

example5.c

```
#include <stdio.h>

int main(void)
{
        int i, sum = 0;
        printf("Program run\n");
        for(i=0;i<=10;i++)
                sum += i;
        printf("Sum is %d\n", sum);
        return 0;
}
```

example5.cpp

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

int main(void)
{
        cout<<"Program run"<<endl;
        int sum = 0; // 중간에 변수 선언 가능
        for(int i=0;i<=10;i++)
                sum += i;
        cout<<"Sum is "<<sum<<endl;
        return 0;
}
```

[Code 5] 변수 선언 방법

☛ C 언어(ANSI C)에서는 함수 내에서 사용되는 변수는 맨 위에 선언
☛ C++에서는 프로그램 중간에 변수를 선언 가능함

2-4. 함수의 사용

함수(Function)는 프로그램을 구성하는 단위 모듈(Module)이며 C언어에서 가장 중요한 요소이다. 마찬가지로 C++에서도 OOP 정의에 필수적으로 사용되므로 매우 중요한 역할

을 차지한다. 여기에서는 간단한 함수의 활용과 C++에서 추가된 함수의 특징들을 살펴보도록 한다.

2-4-1. 함수의 선언 및 정의

C++에서 함수의 선언 및 정의는 C언어와 동일하게 사용되며 그 기능 또한 같다.

- ☞ 함수 정의 제공
- ☞ 함수 원형 제공
- ☞ 함수 호출

게다가 C++에서는 문법이 좀 더 엄격해짐에 따라 함수의 원형을 반드시 선언하여야 한다. 함수의 원형 선언 방법은 C언어와 마찬가지로 표준함수인 경우 include를 이용하여 함수의 원형이 포함된 헤더 파일을 포함하고, 사용자가 직접 작성한 사용자정의 함수인 경우 함수 사용 전에 원형을 직접 기술하거나 원형이 정의된 헤더 파일을 작성하고 이를 포함하도록 한다. C언어에서는 함수 원형이 밝혀지지 않은 경우 디폴트 원형을 적용하는 예외사항이 발생하지만 C++에서는 **반드시 함수의 원형이 선언**되어야만 정상적인 컴파일 수행이 가능하다. 또한 함수의 매개 변수 타입은 생략이 불가능하고 다음의 예만 가능하다.

- ☞ 함수의 리턴값과 인수의 타입만을 명시 ex) int func(int, char);
- ☞ 함수의 리턴값과 인수의 타입과 인수의 이름을 명시 ex) int func(int a, char b);

2-4-2. 함수 다중정의(Function Overloading)

C언어에서는 동일한 이름의 함수가 존재할 수 없었다. 이는 함수 호출시, 컴파일러가 함수의 이름을 이용하여 해당 함수를 호출하기 때문에 정한 규칙이었다. 그러나 C++에서는 동일명의 함수에 인수(전달인자 리스트)의 개수나 인수의 자료형이 다른 함수를 사용하는 것을 가능하게 하였다. 이는 같은 기능을 수행하는 함수에 있어서 인수(전달인자 리스트)가 다른 경우에도 함수를 좀 더 유연하게 사용하기 위한 수단으로 C++에서는 매우 중요한 개념이다. 이것을 함수 다중정의라고 한다.

example6.c

```
#include <stdio.h>

int i_add(int, int);
double d_add(double, double);

int main(void) {
   printf("Result 1 : %d\n", i_add(5,10));
   printf("Result 2 : %f\n", d_add(5.2,10.3);
}

int i_add(int a, int b){
   return a+b;
}

double d_add(double a, double b){
   return a+b;
}
```

example6.cpp

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

int add(int, int);
double add(double, double);

int main(void) {
   cout<<"Result 1 : "<<add(5,10)<<endl;
   cout<<"Result 2 : "<<add(5.2,10.3)<<endl;
   return 0;
}

int add(int a, int b){
   return a+b;
}

double add(double a, double b){
   return a+b;
}
```

[Code 6] 함수의 다형성

[Code 6]에서 보면 알 수 있듯이 C언어에서는 동일 기능을 제공하는 add()함수를 처리하는 자료형에 따라 각각 다른 함수명으로 구현한다. 그러나 C++에서는 동일명의 함수로 구현하여 프로그램을 좀 더 유연하게 한다.

[참고자료] 함수 다중정의(Overloading)와 함수 재정의(Overriding)

C++에서 함수의 다형성을 이야기하는 경우 다중정의와 재정의를 말하는데, 서로 용어가 유사하여 혼동되는 경우가 많다. 이를 위해 간단히 설명하면

함수 다중정의	함수의 이름은 동일하지만 인수(전달인자리스트)의 개수나 인수의 자료형이 다른 경우를 말한다.
	void foo(void);
	void foo(int value);
	void foo(int left, int right);
	만약 foo(1)을 호출하면 두 번째 함수가 호출되고, foo(1,2)를 호출하면 세 번째 함수가 호출된다.

함수 재정의	상속에서 사용되는 개념으로 상속된 원래 함수를 그대로 사용하는 것이 아니라 자기가 새롭게 함수를 바꾸어 다른 기능으로 사용하는 것을 말한다. 자세한 내용은 후반부 객체 지향 프로그래밍에서 자세히 설명한다.

2-5. 기타사항

2-5-1. 주석 처리

주석은 프로그래머가 기록하는 일종의 메모로서 프로그램의 구역 구별이나 내용서술에 이용하며 C++에서는 두 가지 형태의 주석 스타일 사용 가능하다. 주석은 프로그램이 복잡해질수록 값진 역할을 수행하며 프로그램의 가독성을 높인다.

- ☞ C++ 스타일: 겹슬래쉬(//)를 사용하며 // 이후의 그 줄의 끝까지 주석 처리
- ☞ C 스타일: 기존의 C언어(ANSI C)에서 사용되는 주석으로 /* */ 표기를 여러 줄에 걸쳐 사용 가능

2-5-2. 캐스트(Cast) 연산자의 사용

프로그래밍을 하다 보면 자료형을 변환해야 하는 경우가 존재하게 된다. 일반적으로 명시적 형변환과 암시적 형변환만을 고려하며 이 두 가지 방법은 동일한 기능을 한다.

- ☞ 명시적 형변환 d = double(i);
- ☞ 암시적 형변환 d = (double) i;

실제 C++에서는 C언어의 미약한 형변환을 보강하기 위해 여러 캐스트 연산이 존재한다. 좀 더 자세한 내용은 *The C++ Programming Language* (Bjarne Stroustrup)를 참조한다.

3. C++으로의 확장

3-1. 포인터(Pointer)와 C++

3-1-1. 포인터

포인터는 값 자체를 의미하는 변수가 아니라 주소값을 저장하고 있는 변수를 의미하며 메모리를 관리하는데 있어서 필수적인 자료형이다. C++에서 사용되는 포인터는 C언어에

서 사용하는 포인터와 그 개념이 동일하다.

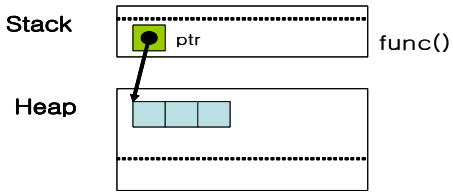
3-1-2. 포인터와 C++의 관계

C++에서는 컴파일 시간(Compile Time)이 아닌 실행 시간(Run Time)에 결정을 내리는 형태를 취하기 때문에 프로그램 중에 메모리를 할당하는 방법이 필요하며 포인터를 이용하며 메모리 연산을 수행하게 된다. 이것은 프로그램 실행 시 사용될 메모리양의 결정이 상황에 적절히 대처할 수 있는 융통성을 제공한다. 예를 들어 배열 선언 시 반드시 그 크기가 명시되어야 하나 크기의 결정을 실행 시간이 될 때까지 미룸으로서 프로그램이 실행되는 시점에 사용자가 프로그램에게 그 크기를 알려줄 수 있다. C++에서는 좀 더 편리한 동적 메모리 할당과 해제를 위해 새로운 키워드(Keyword)로 new, delete를 제공한다.

3-2. 동적 메모리의 할당 및 해제

C언어에서도 동적인 메모리 할당을 위한 방법으로 malloc() 등을 사용한다. 그러나 C++에서는 문법 수준에서 이러한 기능을 제공하므로 편리하게 사용 가능하다. 메모리 할당을 위해 사용되는 new는 할당하고자 하는 데이터의 크기만큼 힙(Heap)으로부터 할당하여, 첫 번째 번지를 리턴한다. 만약 할당이 불가능할 경우 NULL을 반환한다. 메모리 해제를 위한 delete는 사용한 메모리 해제하는 기능으로 new로 할당된 메모리는 delete를 이용하여 해제하여야 한다. 그렇지 않으면 프로그램이 수행되는 동안 메모리를 사용하게 된다. 따라서 할당받은 메모리를 사용할 수 있도록 해당 주소를 포인터로 관리해야 한다.

☞ 할당받은 메모리의 첫 번째 주소를 포인터에 저장하여 사용



[그림 2] 동적 메모리의 할당

모든 실행 프로그램은 각각의 프로그램 스택(Stack)영역과 힙 영역을 가지고 있으며 함수 호출 시에는 해당 함수를 프로그램 스택에 삽입한다. 그리고 함수의 사용이 끝나면 프로그램 스택 내에서 해당 함수를 삭제한다. 따라서 함수 내에서 사용된 변수도 해당 메모리에서 삭제되게 된다. 만약 함수 내에서 사용한 메모리를 함수 종료 후에도 사용된 메모리를 계속 사용하고 싶으면 프로그램 힙 영역에 별도 메모리를 할당한 후에 사용해야 한다([그림 2] 참조).

C++에서는 new와 delete로 힙 영역의 메모리를 사용할 수 있다. [Code 7]의 예제를 통해 C언어에서 자주 사용하던 메모리 할당을 C++에서는 어떻게 사용할 수 있는지 살펴볼 수 있다. C++에서는 C에서 제공되는 기본 자료형 뿐만 아니라 클래스의 힙 영역 할당을 가능하게 하기 위해서 new와 delete를 새롭게 제공한다. 따라서 이는 내부적으로 malloc()과 free()보다 더 복잡한 작업을 수행하지만, 반면 사용 방법은 더 간단하다. 단지 자료형 앞에

new를 붙여 주는 것으로 명시적 형변환과 데이터형의 크기를 지정할 필요 없이 메모리를 할당할 수 있으며, 이렇게 new를 사용하여 할당한 메모리는 delete를 사용하여 해제하면 된다.

C++의 동적 메모리 할당에서의 주의할 점은 다음과 같다.

☞ new로 할당하지 않은 메모리는 delete로 해제하지 않는다.

☞ delete를 사용하여 같은 메모리 블록을 연달아 두 번 해제하지 않는다.

☞ new[]를 사용하여 메모리를 할당한 경우 delete[]를 사용한다(delete를 사용해도 컴파일 에러는 발생하지 않지만, 메모리 누수(Memory Leak)가 발생한다).

☞ new를 대괄호 없이 사용했으면 delete도 대괄호 없이 사용한다.

☞ NULL 포인터에는 delete를 사용하지 않는 것이 안전하다(아무 일도 일어나지 않음).

example7.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct{
   int id;
   char name[10];
}student;

int main() {
   int *t, i, j;
   student *s;

   s = (student*)malloc(sizeof(student));
   t = (int*)malloc(sizeof(int)*10);

   s->id = 1;
   strcpy(s->name, "홍길동");

   printf("id : %d, name %s\n", s->id, s->name);

   for(i=0;i<10;i++)
      t[i] = i;

   for(j=0;j<10;j++)
      printf("%d\n", t[j]);

   free(s);
   free(t);
   return 0;
}
```

example7.cpp

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

struct student{
   int id;
   char name[10];
};

int main() {
   student *s = new student;
   int *t = new int[10];

   s->id = 1;
   strcpy(s->name, "홍길동");

   cout<<"id : "<<s->id<<", name : "<<s->name<<endl;

   for(int i=0;i<10;i++)
      t[i] = i;

   for(int j=0;j<10;j++)
      cout<<t[j]<<endl;

   delete s;
   delete[] t;
   return 0;
}
```

[Code 7] 동적 메모리의 할당 및 해제

3-3. 참조 호출(Call by Reference)

참조 호출이란 이름이 다른 변수가 같은 메모리 번지를 가리키도록 하는 것으로 C언어의 포인터와 유사하나 실제로는 다른 개념이다. 포인터는 변수를 메모리 공간에 유지하지만 참조 호출은 별도 메모리를 할당하지 않는다.

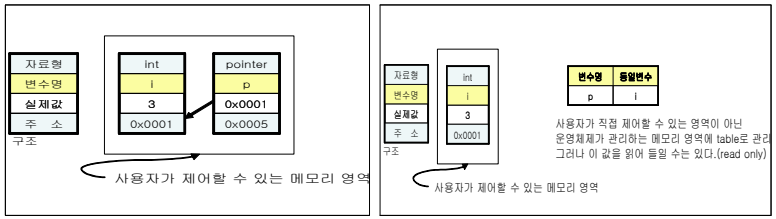
example8_1.cpp

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

int main() {
   int i=3;
   int *p = &i;

   cout<<"i is "<<i<<endl;
   cout<<"*p is "<<*p<<endl;
   cout<<"p is "<<p<<endl;     // p의 주소

   return 0;
}
```

example8_2.cpp

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

int main()
{
   int i=3;
   int &p = i;
   cout<<"i is "<<i<<endl;
   cout<<"p is "<<p<<endl;

   cout<<"&i is "<<&i<<endl; // i의 주소값 출력
   cout<<"&p is "<<&p<<endl; // p의 주소값 출력

   return 0;
}
```

[Code 8] 포인터와 참조 호출



[그림 3] 위의 Code에서의 메모리 사용

참조호출 사용시 주의해야 할 사항들로 참조자 선언시 참조자의 형과 참조할 변수의 형이 서로 일치해야 하며 참조자는 선언과 동시에 초기화하여야 한다. 그리고 한번 초기화되면 다른 대상을 가리킬 수 없다. 이러한 제약은 [그림 3]의 메모리 구조를 살펴보면 명확해진다. 이는 참조호출이 포인터와 같이 사용자의 별도 메모리 영역에 생성되는 것이 아니라 배열이름 등과 유사하게 내부적으로 Table 형태로 관리되므로 서로 비슷한 성질을 지니게 된다.

example9_1.cpp

```
void swap(int *a, int *b)
{
      int tmp;
      tmp = *a;
      *a = *b;
      *b = tmp;
}
```

example9_2.cpp

```
void swap(int &a, int &b)
{
      int tmp;
      tmp = a;
      a = b;
      b = tmp;
}
```

[Code 9] swap 함수의 포인터 구현과 참조 호출 구현

3-4. 구조체(Structure)와 클래스(Class)

3-4-1. 구조체(Structure)

구조체란 서로 연관성 있는 변수들을 하나로 묶어 만들어지는 새로운 사용자 정의 데이터 형으로 구조체의 사용은 두 단계로 구분한다.

- ☛ 데이터형 특성을 정의, 즉 구조체 묘사를 정의하는 단계: 구조 정의
- ☛ 구조체 변수를 생성: 구조체 데이터 객체를 생성

구조체는 순수하게 데이터만을 표현하는 객체를 의미하게 된다. 사실 C++에서는 struct 구문 내에 함수를 포함할 수 있다. 그러나 이러한 경우 클래스를 사용하는 것이 일반적이며, C++에서는 C언어와 다르게 좀 더 간편한 방식으로 구조체를 정의한다. 즉, typedef문의 사용 없이도 변수를 선언하는 방식과 동일하게 사용 가능하다.

[표 1] 구조체 사용의 다양한 예

C 프로그램			C++ 프로그램
①	②	③	④

/* 정의 */
struct Ttag{
int id;
double grade;
}T;
 	
/* 사용 */
struct T t;
	/* 정의 */
	struct Ttag{
	int id;
	double grade;
	};
	
	typedef struct Ttag T;
	
	/*사용*/
	T t;
		/* 정의 */
		typedef struct Ttag{
		int id;
		double grade;
		}T;
		
		/*사용*/
		T t;
			// 정의
			struct Ttag{
			int id;
			double grade;
			}T;
			
			// 사용
			T t;

- ☛ ①의 경우 T의 사용시에 매번 struct 구문을 적어주어야 사용 가능함
- ☛ ②의 경우 typedef 문으로 정의하여 사용
- ☛ ③의 경우 ②에서 구조 정의와 typedef의 형태를 하나로 합쳐서 표현
- ☛ ④는 C++에서 기본적으로 제공하는 형태

3-4-2. 클래스(Class)

클래스는 OOP의 특성을 표현하기 위한 C++의 문법으로 기본적으로 구조체의 기본 속성과 유사하나 확장된 내용과 문법 형식을 지니고 있다. 클래스는 서로 관련이 있는 데이터(멤버 변수)와 이를 조작하기 위한 함수(멤버함수)로 구성되어 있으며 C++에서 OOP를 사용하기 위한 문법이다. 클래스에 대한 구체적인 구현 방법은 [4. 객체지향 프로그래밍]을 참고하도록 한다.

example10.cpp

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

// student class의 정의
class student {
        private:
                int id;
                char name[10];
                char grade;
        public:
                void setStudent(int stId, char *stName, char stGrade);
                int getId(void);
                char* getName(void);
                char getGrade(void);
                void show(void) {
                        cout<<"ID : "<<id<<", Name : "<<name<<", Grade : "<<grade<<endl;
                };
};

// student 멤버 함수의 정의

void student::setStudent(int stId, char *stName, char stGrade){
        id = stId;
        strcpy(name, stName);
        grade = stGrade;
}

int student::getId(void){
        return id;
}

char* student::getName(void){
        return name;
}

char student::getGrade(void){
        return grade;
}

// main 함수의 정의
int main()
{
        student s;
        s.setStudent(123,"홍길동", 'A');
        s.show();
        return 0;
}
```

[Code 10] 클래스의 간단한 사용 예

4. 객체지향 프로그래밍(Object Oriented Programming: OOP)

4-1. OOP의 기본 개념

OOP는 문법적으로 정해진 **프로그래밍 양식**을 의미하는 것이 아니라 **프로그램을 설계하는 방법**을 의미하며 다음과 같은 특성을 지닌다.

- ☛ 데이터 추상화(Abstraction)
- ☛ 캡슐화(Encapsulation)와 데이터 은닉(Data Hiding)
- ☛ 상속성(Inheritance)
- ☛ 다형성(Polymorphism)
- ☛ 재활용 코드(Reusable Code)

C++에서는 클래스를 이용하여 이러한 기능을 구현한다. 우선 객체가 무엇인지를 살펴보고 이것을 C++로 구현하는 과정을 살펴보도록 한다.

4-1-1. 객체(Object)의 정의

객체는 다음과 같은 두 가지 구성 요소를 지니고 있으며 이러한 것을 모델링하는 것이 객체 지향이다.

- ☛ 상태(State): 객체가 가지고 있는 속성 또는 특성
- ☛ 행동(Behavior): 객체가 가지고 있는 기능 또는 할 수 있는 반응 양식

실제 프로그래밍에 있어 실세계 객체가 가지는 상태를 나타내기 위해 **멤버 변수(Member Variable)**를 이용한다. 또한 이러한 상태를 변경시키는 행동을 표현하기 위해 변수의 값을 변경하거나 다른 객체로부터의 요청을 서비스하기 위한 함수(Function or Method)를 사용하게 된다. 이것이 **멤버 함수(Member Function)**이다.

하나의 예로 자동차 객체는 바퀴, 핸들, 배기량, 속도, 기어 위치 등의 상태를 가지고 있으며, 달린다, 멈춘다, 기어를 변경한다, 속도를 줄인다 등의 행동 양식을 포함한다. 여기서 데이터 추상화(Abstraction)를 통해 실세계의 복잡한 객체의 형태를 몇몇의 상태와 행동을 가지는 단순화된 객체를 생성한다. C++에서는 추상화된 정보를 정의하고 이를 제어하기 위한 인터페이스(Interface)를 구현하는 형태로 객체를 표현한다. 이 인터페이스를 멤버 함수로 이해하면 된다.

또한 프로그래밍의 개념에서 **객체는 클래스를 사용하여 생성한 인스턴스**라고 할 수 있다(4-2 참조).

4-1-2. 절차식 프로그래밍과 객체 지향 프로그래밍

절차식 프로그래밍과 객체 지향 프로그래밍은 코딩 방식의 차이가 아닌 프로그램 설계 방식의 차이를 가진다. 즉, 단순히 문법적인 차이로 절차식 프로그램인지 객체지향 프로그램인지를 나누는 것이 아니라, 전체적인 **프로그램 구조 설계 방식이 전혀 다르다는 것을 의미**한다. 그러므로 C++에서 객체 지향 프로그래밍을 한다는 것은 곧 클래스를 작성하는 것이 중심이 된다. 프로그램의 대부분의 기능이 클래스로 정의되고, 여기에서 생성된 인스턴스(추후 설명)의 멤버 함수를 실행하는 것으로 처리를 진행한다. 따라서 C언어에서처럼 독

립된 함수를 만드는 상황은 일단 없다(C++에서는 C언어처럼 전역 함수(Global Function) 작성을 허용하지만, Java나 C#에는 아예 전역 함수라는 개념이 없다).

클래스를 사용한 객체지향 프로그래밍은 클래스를 데이터와 멤버 함수를 모두 가질 수 있는 일종의 데이터 타입으로 간주하여, 클래스 내부에서는 객체가 가질 수 있는 상태 데이터를 변수로 선언하고, 멤버 함수에서는 C에서 사용하던 방식과 같이 논리적인 기능을 처리하도록 작성한다. 중요한 것은 **멤버 함수의 외부적 요소, 즉 클래스의 작성 방법, 멤버 함수의 분할 방법이 객체 지향 프로그래밍의 포인트**이다.

[표 2] 절차식 프로그래밍과 객체 지향 프로그램 작성의 예

[Problem] 한 프로그래머에게 소프트 볼 팀 선수들의 통계를 내라는 업무 지시가 내려짐

절차식 프로그래밍	객체 지향 프로그래밍
▪ 이름과 타석수, 안타수와 타율, 통계가 필요함 ▪ 이를 위해 각각을 계산할 수 있는 함수를 고려 (1) 선수의 데이터를 넣을 수 있는 함수 (2) 타율 및 통계치를 계산해주는 함수 (3) 얻어진 결과를 출력해주는 함수 (4) 통계를 갱신할 수 있는 함수 ▪ 전체 구조를 작성 ▪ 데이터의 입력, 계산, 갱신, 출력 메뉴 고려 ▪ 이를 위해 선수의 데이터 표현을 위한 구조체 배열 설계 결정 **즉, 함수 작성에 중점을 두며 이를 위한 데이터 표현 방법을 고려하게 됨**	▪ 가장 먼저 고려해야 할 대상으로 선수의 데이터를 구조화 ▪ 선수를 표현할 수 있는 데이터를 결정한 뒤 각각을 접근 할 수 있는 함수들을 작성 ▪ 선수들의 데이터를 관리하는 객체를 선언 ▪ 이 객체 내에서 선수 객체로부터 데이터를 얻어올 수 있는 멤버 함수를 작성 ▪ 이 객체 내에서 얻어진 데이터를 이용하여 통계치를 얻어내는 멤버 함수를 작성 ▪ 이 객체 내에서 사용자가 선수들의 데이터를 제어할 수 있는 함수들, 즉 인터페이스들을 작성 **즉, 객체를 묘사하는데 요구되는 데이터와 그 데이터에 대해 사용자가 가지는 인터페이스를 묘사하는 방법을 고려**

4-2. 클래스(Class)와 인스턴스(Instance)

그러면 클래스는 무엇을 의미하는가? 클래스에 대해서는 다양한 정의가 있으나 일단은 앞에서 설명한 **객체에 대해, 이를 기술하는 틀(Template)**로 이해할 수 있다. 즉 객체는 클래스를 통하여 생성된다. 예를 들어, 클래스를 설계도라고 한다면, 이 설계도를 사용하여 만든 집을 객체라고 할 수 있다. 이 때, 집을 짓는 과정과 같이, **클래스를 실제로 사용할 수 있도록 메모리에 할당하는 것을 인스턴스 생성(Instantiation)**이라고 하며, **생성된 객체를 인스턴스(Instance)**라 한다. 클래스와 인스턴스에 대한 다른 예들은 다음과 같다.

- ☞ 한 예로 자동차 클래스는 철수가 가진 "소나타", 영희가 가진 "포니" 등으로 구체화.
- ☞ 즉, 클래스에 대한 실제 변수를 선언하는 것을 "인스턴스 생성" 이라고 한다.
- ☞ 각각의 인스턴스는 그 구조가 같더라도 담고 있는 내용은 서로 다를 수 있다.
- ☞ 즉, 같은 붕어빵 틀에서 나온 붕어빵일지라도 각각은 서로 다른 붕어빵이다.

하나의 클래스를 선언하면 여러 개의 인스턴스를 생성 가능하다. 이는 하나의 데이터 타입을 사용하여 여러 개의 변수를 선언 가능하고, 이 변수들의 값을 제각기 다르게 설정 가능한 것과 유사하다.

4-3. C++에서 객체 표현하기

4-3-1. 클래스(class)의 구현

클래스의 구현은 우선 클래스를 정의하는 것으로부터 시작된다. 클래스 내부에는 멤버 변수와 멤버 함수가 존재한다. 이미 알고 있는 C언어에서, 가장 유사한 것으로 구조체를 떠올릴 수 있다. 그러나 **구조체는 멤버로 오직 데이터만을 가질 수 있고, 클래스는 멤버로 함수를 가질 수 있다**는 것이 다른 점이다.

멤버 함수는 클래스의 데이터 멤버 변수의 처리를 전문으로 하는 클래스 전속 함수이다. C언어의 함수에 익숙해져 있는 시각으로 볼 경우에 이는 특별하게 보일 수 있다. 왜냐하면 C언어에서는 데이터와 함수가 별개의 것으로 취급되기 때문이다.

예를 들어, C언어에서 puts()라는 함수가 있다. 이 함수는 오직 문자열을 표시하기 위해서만 사용된다. 이와 같이, 많은 함수는 극히 제한된 상황에서만 사용된다. 그러나 C언어에서는 문자열이 아닌 char*형의 어떤 변수도 puts()의 인자로 넘겨 줄 수 있다. 이는 심각한 오류를 야기할 수 있다. 이렇게 함수 작성자가 함수에 대해 특정 데이터 종류를 결정해 놓은 경우, C언어와는 다르게 **데이터 속에 함수를 심어 넣음으로써 데이터와 함수를 한 덩어리로 보이게끔 프로그래밍 하는 것**이 클래스의 작성 방법이다.

[Code 11]은 클래스를 사용한 간단한 예제이다. 클래스를 선언하는 방법과 멤버 함수의 정의 방법을 코드를 통해서 살펴보자.

example11.cpp

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

// MyString 클래스 선언(객체의 표현)
class MyString
{
      private:    // 접근 권한 설정: 은닉
            char str[256];
            int length;
      public:     // 접근 권한 설정: 공개
            void setString(const char* pstr);
            const char* getString();
            int getLength();
            void prnString();
};

// 멤버 함수의 정의
void MyString::setString(const char* pstr){
      length=strlen(pstr);
      if(length>255)
            cout<<"Too long string"<<endl;
      else
            strcpy(str, pstr);
}

const char* MyString::getString(){
      return str;
}

int MyString::getLength(){
      return length;
}

void MyString::prnString(){
      cout<<str;
}

int main()
{
      // 인스턴스 생성
      MyString s;

      s.setString("Hello World!\n");
      int s_length=s.getLength();
      cout<<"Length: "<<s_length<<endl;
      s.prnString();
      return 0;
}
```

[Code 11] MyString 객체의 구현

- ☞ MyString 객체를 표현하기 위해 str, length변수와 이를 조작하는 멤버 함수를 정의
- ☞ 객체의 변수는 외부로부터 은닉하고, 함수는 공개(다음에 설명)
- ☞ 멤버 함수를 정의할 때는 C언어에서의 함수와 같이 하되, 리턴 값 타입 바로 다음의 멤버 함수명에 **"클래스이름::"**을 붙여서 멤버 함수의 소속을 표시함
- ☞ main()함수에서 MyString 객체의 인스턴스를 생성하여 사용함. 이 인스턴스는 MyString

클래스가 정의한 멤버변수와 멤버 함수를 가짐.

☞ 클래스의 멤버에 접근하기 위해서는 구조체와 같이 **클래스이름.멤버이름**의 형식을 사용(포인터로 접근할 때는 **클래스이름->멤버이름**)

4-3-2. 접근 권한

C++에서는 멤버 변수와 멤버 함수를 은닉할 것인지 아닌지를 결정할 수 있다. 이는 프로그램 작성시에 불필요한 부분의 접근을 막도록 하여 프로그램을 좀 더 견고하게 작성하기 위함이다. 이것을 **정보 은폐(Information Hiding)**라고 한다. 일반적으로 멤버 변수는 private로 선언하여 숨기고, 나머지 멤버 함수는 public으로 외부에 공개한다. 접근 지정자를 사용하지 않았을 때에는 기본적으로 private로 설정된다.

[표 3] 접근 지정자에 따른 멤버 사용 범위

접근 지정자	내 용
public	public으로 선언된 이후의 멤버 변수, 멤버 함수들은 외부의 객체가 마음대로 읽거나 쓸 수 있다.
private	private로 선언된 이후의 멤버 변수, 멤버 함수들은 그 클래스 내의 멤버 함수에서만 접근 가능하다.
protected	private의 속성과 비슷하나 차이점은 상속시 하위클래스가 멤버변수와 멤버 함수에 접근 가능하다.

[표 4] 접근 지정자에 따른 각 상황별 멤버 사용 가능 여부

접근 지정자	객체 내 멤버 함수	상속 받은 클래스의 객체 내 멤버 함수	외부 함수 (특히 main()함수 등)
public	O	O	O
private	O	X	X
protected	O	O	X

앞에서의 [Code 11]에서 str 변수를 public으로 선언하면, setString()함수를 사용하는 대신, MyString의 인스턴스인 s에 멤버 접근 연산자(.)를 사용하여 손쉽게 데이터 영역의 접근이 가능하다. 그러나 [Code 11]에서처럼 데이터 영역을 모두 은폐하여 클래스를 구현하였을 경우 얻게 되는 몇 가지 장점이 있다.

먼저, 이 클래스를 실제로 사용하는 프로그래머는 main()함수에서 사용되는 예를 살펴보면 알 수 있듯이, **클래스의 데이터 멤버를 신경 쓰거나 의식할 필요가 전혀 없다.** 단지 함수의 사용 방법만 숙지하면 된다. 또한, length변수를 외부에서 직접적으로 변경하지 못하게 함으로써(만약 str의 내용이 일치되게 변경되지 않는다면 문제가 생김) 프로그램의 오류를 방지하고, str을 setString()함수를 통해서만 변경되게 함으로써 **문자열 설정에 의한 메모리 영역 파괴의 가능성을 원천적으로 봉쇄**하였다. 즉, 클래스와 접근 권한을 이용함으로써 위의 예에서는 메모리 오류에서 안전한 문자열을 사용자가 직접 만들 수 있었다. 객체 지향 프로그래밍의 장점은 여러 가지가 있지만, 이처럼 클래스 정의를 통해 새로운 수준의 데이터 타입 정의와 재사용성이 편리한 모듈의 작성을 가능하게 한다.

위 설명에서처럼, **데이터와 데이터에 대한 조작을 하나로 묶는 것**을 **캡슐화(Encapsulation)**라고 하며, 캡슐화의 기본 원칙은 프로그래머에게 상세한 내부 구현을 숨기는 정보 은폐이다. 이를 통해 높은 모듈성(Modularity)을 제공한다.

4-3-3. 생성자(Constructor)

객체가 생성될 때 멤버 변수를 초기화해주는 작업을 수행하며 **객체가 처음 생성될 때 무조건 호출이 되는 함수로서, 함수 리턴값이 존재하지 않는다**(심지어 void type 조차도 가져서는 안 됨). 생성자 역시 함수의 일종이므로 함수 다중정의(Function Overloading, 2-4-2절 참고)가 가능하다. 즉, 다수의 생성자가 존재 가능하다.

생성자 이름은 클래스의 이름과 반드시 동일해야 하며, 정의하지 않으면 디폴트 생성자가 사용된다. 디폴트 생성자는 클래스와 이름이 같고 전달 인자 리스트가 없는 생성자를 의미하며 만약 사용자가 다른 생성자를 정의하게 되면 디폴트 생성자는 제공되지 않는다.

☞ 객체를 생성하는 방법에 따라 복사 생성자라는 것을 사용하는 경우도 있음
복사 생성자에 관한 내용은 *C++ Primer Plus (SAMS, stephen prata)* 등을 참고

example12.cpp

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

// circle 클래스 선언(객체의 표현)
class circle {
        private:
                double radius;
        public:
                circle();
                circle(int r);
                void prnRadius(void);
};

// 디폴트 생성자의 정의
circle::circle() {
         radius = 0;
         cout<<"생성자 circle()호출"<<endl;
}
```

```
// 또 다른 생성자의 정의
circle::circle(int r) {
        radius = r;
        cout<<"생성자 circle(int r)호출"<<endl;
}

// 멤버 함수의 정의
void circle::prnRadius(void){
        cout<<"Radius : "<<radius<<endl;
}

int main() {
    // 전달인자 값이 존재하지 않는 생성자 호출
        circle c;
    // 전달인자 값이 존재하는 생성자를 호출
        circle d(3);
        c.prnRadius();
        d.prnRadius();
        return 0;
}
```

[Code 12] 다중 생성자의 사용.

생성자는 인스턴스를 생성할 때 자동으로 호출되는 함수이므로 이곳에서 초기화뿐만 아니라 여러 가지 처리를 작성하고 싶어질 것이다. 그러나 생성자는 리턴값을 가질 수 없기 때문에 에러를 발생시킬 가능성이 있는 처리는 에러 발생 시 그것을 외부에 자세히 알려줄 수 있는 방법이 없으므로 위험하다. 또한, 생성자를 실행할 때, 인스턴스는 내부적으로 완전한 상태가 아니기 때문에(생성 중, 즉 메모리를 얻어오려는 작업을 실행 중) 코드만 봐서는 해결되지 않는 버그를 발생시킬 수 있다. 따라서 **생성자에서는 변수 초기화만 처리**하는 것이 좋다.

4-3-4. 소멸자(Destructor)

객체가 **소멸될 때 호출하는 멤버 함수**이다. 생성자와 같이 소멸자는 매개 변수와 리턴값이 없고 객체와 같은 이름을 갖는다. 생성자와 구별하기 위해 소멸자의 이름 앞에는 틸다(~)를 붙인다. 대개 생성자에서 동적 메모리 할당을 받고 소멸자에서 이를 해제한다. 소멸자는 다중정의될 수 없으며, 리턴값이 존재하지 않는다.

소멸자 또한 앞의 생성자의 설명과 같이 리턴값이 없으므로 오류에 취약하며, 완전한 상

태가 아닐 수 있기 때문에(메모리 해제 중), 소멸자에서는 new로 생성한 인스턴스를 delete로 해제하는 종료 작업 이외에는 다른 작업은 하지 않는 것이 좋다.

그리고 **소멸자를 선언할 때는 키워드로 virtual을 선언하는 것이 일반적**이다. 붙이지 않아도 컴파일 에러가 나타나지는 않지만 붙이지 않는 경우에는 잠재적으로 버그를 야기할 수 있으므로 반드시 붙여 주는 것이 좋다(뒤에서 다룰 4-5-2에서와 같이 기반 클래스로 파생 클래스를 접근하는 방법을 적용한다면, 인스턴스가 소멸할 때 기반 클래스의 포인터가 소멸의 주체가 되므로 파생 클래스의 소멸자가 호출되지 않는 문제가 발생함).

example13.cpp

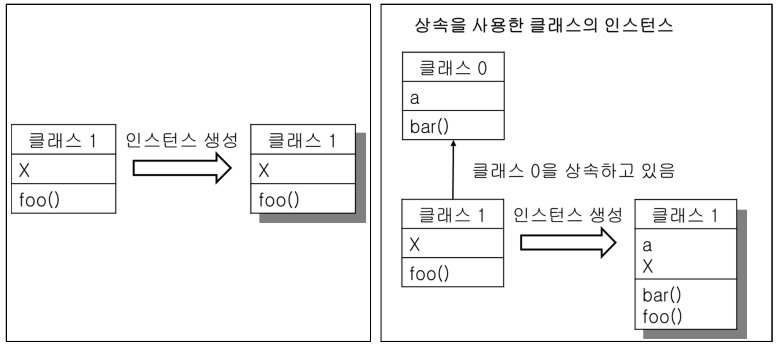
```
#include <iostream>
using namespace std;

class student{
        private:
                int id;
                char *name;
        public:
                student(int stId, char *stName);
                virtual ~student();
};

// 생성자에서 메모리 할당
student::student(int stId, char *stName){
        id = stId;
        int tmp = strlen(stName);
        name = new char[tmp+1];
        strcpy(name, stName);
        cout<<"Student 생성"<<endl;
}

// 소멸자에서 메모리 해제
student::~student(){
        delete(name);
        cout<<"Student 소멸"<<endl;
}

// main 함수
int main()
{
        // 동적으로 객체 생성
        student *s = new student(10,"홍길동");

        delete s;
        return 0;
}
```

[Code 13] 생성자와 소멸자의 이해

4-4. 상속성(Inheritance)

4-4-1. 상속의 개념

상속(Inheritance)은 객체 지향에서 가장 혁신적인 발명이라고 일컬어지는 개념으로, 언듯 매우 추상적으로 보일 수 있다. 우선 상속의 메커니즘을 정의하자면, **클래스가 인스턴스를 생성할 때, 다른 클래스의 속성/오퍼레이션을 빌려와서 자신이 갖고 있는 것과 합친 후 하나의 인스턴스를 생성하는 것**이라고 말할 수 있다. 개념도는 다음 그림과 같다.



[그림 4] 상속의 메커니즘

기존의 절차적 프로그래밍에서, 프로그래머들은 동일한 처리가 반복적으로 프로그램에 나타나는 것을 방지하기 위해, 서브루틴(Subroutine, 리턴값이 없는 함수)을 통한 처리의 공통화를 꾀하기도 한다. 왜냐하면 동일한 처리가 코드의 여러 부분에 산재할 경우 나중에 그 처리 부분을 고치려고 할 때 모든 부분을 정확히 고쳐야 하는 어려움이 있고, 또한 코드의 길이도 길어지기 때문이다.

마찬가지로, 객체 지향 프로그래밍에서 클래스의 일부를 공통화 하는 방법이 상속이다. 그러나 절차적 프로그래밍에서 코드의 일부를 잘라내어 그 부분을 복수의 위치에서 사용하는 것처럼, 개별 클래스에서 공통된 부분을 잘라내어 이를 공통 클래스로 만드는 것이 상속인가? 그러나 상속은 이와 같은 간단한 테크닉이 아닌 좀 더 다른 차원의 개념으로 접근하여야 한다. 그러면 상속은 어떤 경우에 사용하는가? 첫 번째는 **클래스의 일반화(Generalization)**, 두 번째는 **클래스의 특수화(Specialization)**이다.

4-4-2. 일반화로서의 상속

일반화라는 것은 **여러 클래스의 공통 개념을 추출하여 독립적인 클래스로 표현**하는 것이다. 다음 그림은 학교 관리 시스템의 분석 내용이다.

[그림 5] 학교 관리 시스템의 분석 내용

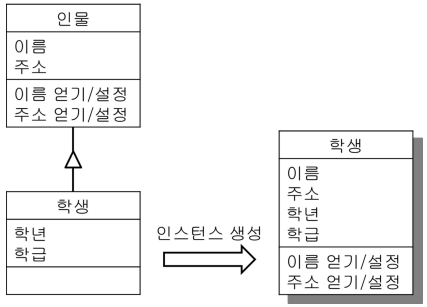
그림에서, 학생과 교사는 많은 부분에서 같은 이름의 속성과 오퍼레이션을 갖고 있다. 이것은 간단하게 이름 정도가 같은 것이 아니라, 실제 교사의 이름을 얻는 경우와 학생의 이름을 얻는 경우에 그 처리 내용이 같다는 것을 알 수 있다. 이것을 각 클래스마다 별도로 구현하는 것은 무의미하며, 낭비이다.

C언어적 발상에서 보면, 이러한 공통적 처리를 공통 함수로 나타내는 등의 방법으로 해결하는 방법이 일반적이나 객체 지향에서는 그런 방식을 사용하지 않는다. 이는 정보 은폐의 원칙에도 부합하지 않는다(함수가 사용되는 모든 클래스들의 정보가 함수 측에 공개되어야 하기 때문이다).

그렇다면 이를 어떻게 해결할 것인가를 생각하는 것보다, 객체 지향적인 사고방식에 따라, 왜 이와 같은 일이 일어나는지를 생각해 보자. 왜 교사와 학생은 양쪽 모두에 이름과 주소가 있고, 같은 오퍼레이션이 중복되는가? 그 이유는 학생과 교사 모두 "인물"이기 때문이다. 즉, 학생과 교사의 추상적 개념인 "인물"을 알아내는 것은, 두 클래스에서 공통된 부분을 찾아내는 것과 같은 일이다. 이렇게 일반화로서의 상속은 **어떤 개념 내의 추상적인 부분을 별개의 개념으로 추출하여 구현하는 것**을 가능하게 한다.

이와 같이, 학생과 교사 안에 은밀히 존재하고 있는 추상적인 개념인 "인물"을 상속을 사용하여 클래스화하면서, 학생 클래스를 정의해 본 것이 [그림 6]이다. 이 구조는 "학생"이라는 개념은 "인물"이라고 하는 개념에 포함된다(상속하고 있다)는 의미가 된다. 이와 같은 관계를 **is-a 관계**라고 한다. 결국 "학생 is a 인물"이 된다. **상속은 이와 같은 is-a관계를 구현하는 경우에 사용**된다.

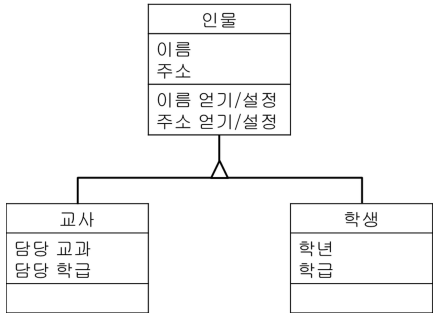
이 경우, 인물과 같이 상속을 제공하는 클래스를 **기반 클래스(Base Class)** 혹은 상위 클래스(Super Class)라 하고, 학생과 같이 상속을 받는 클래스를 **파생 클래스(Derived Class)** 혹은 하위 클래스(Sub Class)라고 한다. 그리고 "학생 클래스의 기반은 인물 클래스이다", "인물 클래스로부터 학생 클래스를 파생했다"라는 식으로 사용한다.



[그림 6] 상속의 기법 및 인스턴스의 생성

그러면 인물 클래스를 상속하고 있는 학생 클래스의 인스턴스는 어떻게 생성될까? 학생 클래스를 보면, 이는 자신만의 속성과 오퍼레이션이 그다지 없는 클래스이다. 그러나 **인물 클래스를 상속하는 것으로 인물 클래스의 속성/오퍼레이션을 학생 클래스 자신의 속성/오퍼레이션과 합쳐, 마치 원래부터 자신의 속성/오퍼레이션으로 존재했던 것처럼 인스턴스를 생성한다.** [그림 6]의 학생 인스턴스를 [그림 5]의 학생 클래스와 비교해 보자. 이 두 가지 경우에서 생성되는 인스턴스는 완전히 같음을 알 수 있다. 이런 경우에 학생 인스턴스를 사용하는 입장에서는 "이름 설정"의 오퍼레이션이 학생 클래스에 구현되어 있는지 또는 기반이 되는 별도의 클래스에 구현되어 있는지의 여부는 인식할 필요가 전혀 없다.

이와 같은 상속의 장점은 어떤 것이 있는가?



[그림 7] 유사한 클래스가 공통의 기반 클래스를 갖는 예

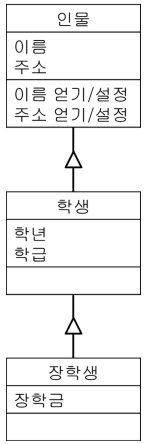
일반화로서의 상속은 다음과 같은 장점이 있다.

☞ 공통의 기반 클래스를 미리 만들어 두면(위 그림에서는 인물) 파생 클래스들의 공통되는 부분에 대한 분석, 설계, 구현, 테스트, 디버그, 유지관리 등의 모든 작업이 필요 없어진다.

☞ 확장성 및 유지관리성의 측면에서의 장점이 있다. (예를 들어, 위에서 "나이"라는 속성을 추가하고자 할 경우, 공통 클래스인 인물 클래스에 이를 추가함으로써 각각의 파생 클래스 모두에 자동적으로 나이라는 속성이 추가된 것과 같이 만들 수 있다.)

4-4-3. 특수화로서의 상속

특수화라는 것은 **어떤 클래스를 기반으로 그 클래스를 특수화한 클래스를 파생 클래스로 작성**하는 것이다. 이 경우에도 상속을 사용한다. 예를 들어, 학교에 "장학생"이라는 제도가 있다고 가정하자. 이 경우 장학생 클래스는 앞에서의 학생 클래스에 장학금을 추가 속성으로 가지게 될 것이다. 그러므로 "장학생 is a 학생"이라는 상속 관계를 이끌어낼 수 있다.



[그림 8] "장학생" 클래스의 파생

이처럼 상속을 사용하면 기반 클래스를 사용하여 여러 가지 특별한 클래스를 파생하는 것이 가능하다. 물론, 아무리 클래스를 파생시키더라도 원래의 기반 클래스에는 아무 것도 추가할 필요가 없다. 위에서 기반 클래스인 학생 클래스는 장학생 클래스의 영향을 받지 않고, 기존과 같이 독립적으로 사용할 수 있다. **기반 클래스는 파생 클래스로부터 완전히 독립된 존재이다.**

이러한 상속의 특성은 프로그램의 유지 보수 및 기능 업그레이드 등에 매우 요긴하게 사용될 수 있다. 예를 들어, 이미 만들어진 문자열 클래스가 있을 때 프로그래머가 문자열의 끝 부분의 공백을 자동으로 없애는 문자열 클래스를 작성하고 싶다고 하자. 그러면 기존의 문자열 클래스를 특수화하여 "문자열의 마지막 공백 문자들을 제거"하는 오퍼레이션(또는 함수)을 하나 추가하면 된다. 이처럼 문자열 클래스를 확장할 때 기반이 되는 문자열 클래스에는 어떤 추가 작업도 필요 없다. 따라서 특수화된 문자열 클래스를 작성하더라도 원래의 문자열 클래스의 재사용성은 완벽하게 보장된다.

4-4-4. 커스터마이즈(Customize)로서의 상속

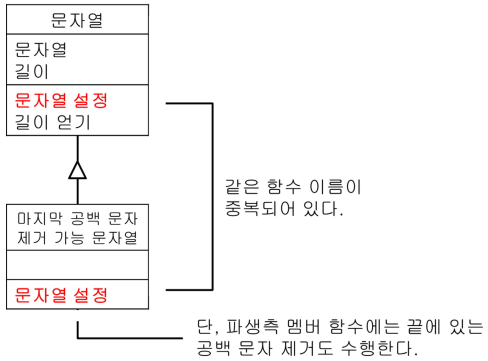
지금까지 살펴본 내용에 따르면, 상속을 사용하면 각각의 클래스의 독립성을 보존하고 속성과 오퍼레이션을 추가하여 클래스를 확장하고, 새로운 클래스를 생성하는 것이 가능하다.

그런데 클래스를 확장한다는 것은 위와 같은 작업들뿐만 아니라, 경우에 따라 오퍼레이션

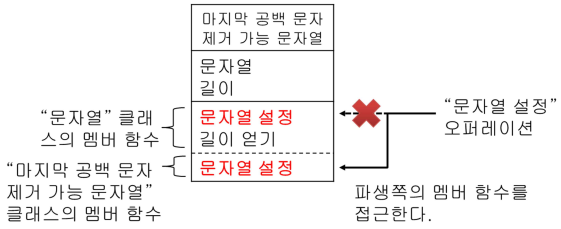
자체의 처리 내용을 변경하는 소위 커스터마이즈(Customize)와 같은 확장도 있을 것이다. 기존의 프로그래밍의 방법으로는, 이와 같은 처리를 위하여 기존의 코드를 복사하여 붙이기 한 새로운 클래스를 만들고 오퍼레이션의 처리 내용 자체를 변경하는 방법이 있다. 그러나 이는 비슷한 클래스를 여러 개 생성한다는 점에서 비효율적이다.

상속에서 이와 같은 문제를 해결하는 방법이 **재정의(Overriding)**이다. 이는 **기반 클래스의 멤버 함수와 완전히 같은 이름의 멤버 함수를 파생 클래스에 만드는 것**이다. 따라서 파생 클래스의 인스턴스에는 기반 클래스와 파생 클래스에 같은 이름을 갖는 2개의 멤버 함수가 존재하게 된다. 이는 에러가 아니며, 실행 후에 그 이름의 멤버 함수를 호출하면 파생 클래스 쪽의 멤버 함수가 호출된다. 즉, 재정의는 파생 클래스의 오퍼레이션이 기반 클래스의 오퍼레이션을 가로채는 방법이다.

구체적인 예를 들어 보자. 이전의 예에서 언급한 끝 부분의 공백을 자동으로 없애는 문자열 클래스에서, 단순히 공백 제거를 가능하게 하는 함수를 추가하는 것으로는 충분하지 않은 경우도 있다. 즉, 프로그램의 명세에서 문자열의 끝에 공백 문자가 포함되면 버그가 되는 경우에는 공백 제거 함수를 추가하기보다는 삭제하는 것을 잊어버리는 것을 막기 위해, 문자열이 설정되는 시점에서 문자열의 인스턴스가 자신의 끝에 붙어 있는 공백을 자동적으로 삭제하게 하는 방식을 사용하는 것이 더 낫다. 이를 위해, 파생 클래스에도 "문자열 설정"이라고 하는 오퍼레이션을 준비하여 기반 문자열 클래스에 있는 "문자열 설정"이라고 하는 오퍼레이션을 가로채버리면 된다. [그림 9]는 클래스를 상속하는 방법이고, [그림 10]은 실제로 이것이 어떻게 작동하는지를 보여 준다.



[그림 9] 오퍼레이션 재정의의 예



[그림 10] "문자열 설정"의 실제 동작

재정의를 사용한 커스터마이즈로서의 상속은 다음과 같은 장점이 있다.

☛ 기반 클래스의 모듈의 소스가 변경되지 않기 때문에 과거 버전을 신경 쓰지 않고 커스터마이즈할 수 있다. 즉 기존 모듈을 변경함으로써 이를 사용하고 있던 다른 부분의 소스까지 다 바꾸어야 하는 불편함이 사라짐

☛ 공개되어 있거나 시판되는 라이브러리를 커스터마이즈하여 적은 노력으로 자신만의 클래스를 만들 수 있으므로 효율적

☛ 다형성(Polymorphism)의 구현을 가능하게 한다(이후 설명).

4-4-5. 상속의 구현

C++에서 상속을 구현하는 것은 간단하다. 파생 클래스를 선언할 때에 기반 클래스를 명시하여 준다.

example14.cpp

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

// 기반 클래스
class Shape{
        protected:
        // 파생 클래스에서는 접근 가능
        // 외부에서는 접근 불가능
                int mX;
                int mY;
        public:
                void MoveTo(int x, int y);
};

// 기반 클래스의 멤버 함수
void Shape::MoveTo(int x, int y){
        mX = x;
        mY = y;
}
```

```
// 파생 클래스
class Circle:public Shape{
/* 기반 클래스 상속시, 상속 멤버 권한
그대로 */
        public:
                void Draw(void);
};

// 파생 클래스의 멤버 함수
void Circle::Draw(void){
        cout<<"Draw Circle"<<endl;
}

int main()
{
        Shape s;
        Circle c;

        s.MoveTo(10,10);
        c.MoveTo(10,20);
        c.Draw();
        return 0;
}
```

[Code 14] 간단한 상속의 예

☛ Circle 객체는 Shape 객체가 가진 멤버 변수와 멤버 함수의 기능을 모두 포함

☛ Circle 객체에서 추가된 멤버 함수는 Circle 객체에서만 사용 가능함

예제 코드에서, Circle 객체가 Shape 객체를 상속할 때 public이라는 상속 접근 지정자(Access Modifier)를 사용하는 것을 볼 수 있다. 이 상속 접근 지정자는 기반 클래스의 각 멤버가 파생 클래스로 상속될 때 원래의 접근 지정을 어떻게 전달할 것인가를 지정하는 구문으로 4-3-2에서 설명한 접근 지정자와 마찬가지로 public, private, protected가 사용된다. 아래와 같이 사용함으로써 상속을 수행할 수 있다.

```
class [파생클래스] : public (or private or protected) [기반클래스] {
        멤버 정의;
}
```

상속 접근 지정자를 명시하지 않을 경우에는 C++에서는 기본적으로 private가 사용된다. 각 상속 접근 지정자에 대하여, 기반 클래스에서 접근 지정자를 사용하여 지정한 멤버들이 파생 클래스에서는 어떻게 속성이 정해지는지를 다음 [표 5]에 나타내었다. 또한 [Code 15]는, 가장 많이 사용되는 상속인 public 상속의 경우, 기반 클래스의 접근 속성이

파생 클래스의 접근 속성에 어떻게 영향을 미치는지를 볼 수 있는 예제이다.

[표 5] 상속 접근 지정자의 기본적인 성질

기반 클래스의 접근 속성	상속 접근 지정자	파생 클래스의 접근 속성
public	public	public
private		접근 불가능
protected		protected
public	private	private
private		접근 불가능
protected		private
public	protected	protected
private		접근 불가능
protected		protected

example15.cpp

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

class Base{
        private : int mPrivate;              // private로 선언, 클래스 내 멤버 함수만이 접근 가능
        protected : int mProtected;          // protected로 선언, 클래스 내 멤버함수와 상속된 클래스들만 접근 가능
        public : int mPublic;                // public으로 선언, 어느 곳에서든지 상관없이 접근 가능
        void f();                            // public으로 선언
};

class Derived : public Base{                 // Base 클래스를 상속받은 Derived 클래스
        public : void g();                   // Base 클래스를 상속받은 상태에서 g()함수가 추가되었다.
};

void Base::f(){
        mPrivate=10; mProtected=20; mPublic=30;
        cout<<mPrivate<<", "<<mProtected<<", "<<mPublic<<endl;
}

void Derived::g(){
        mPrivate = 10;       // error: private로 선언된 기반 클래스의 멤버 변수엔 접근 불가능
        mProtected = 20;     // ok: protected로 선언된 경우 상속받은 클래스에서는 접근 가능함
        mPublic = 30;        // ok
        cout<<mPrivate<<", "<<mProtected<<", "<<mPubic<<endl;   // mPrivate error
}

int main(){
        Base b;
        Derived d;
        b.mPrivate = 10;     // error: private로 선언된 클래스 내 멤버 변수는 접근 불가능
        b.mProtected = 20;   // error: protected로 선언된 멤버 변수는 외부 함수에서 접근 불가능
        b.mPublic = 30;      // ok
        d.f();
        return 0;
}
```

[Code 15] 상속 접근 지정자의 사용

주의할 점은, **생성자와 소멸자는 상속이 되지 않는다**는 것이다. 대신, 파생 클래스에서는 기반 클래스의 생성자와 소멸자 또한 호출한다. 즉, 파생 클래스의 인스턴스가 생성될 때, 생성자는 기반 클래스의 것이 먼저 실행되고 그 이후 파생 클래스의 생성자가 실행된다. 또한, 파생 클래스의 인스턴스가 소멸될 때는 파생 클래스의 소멸자가 먼저 실행되고 그 이후 기반 클래스의 것이 실행된다.

[Code 14]와 [Code 15]에서는 기반 클래스와 및 파생 클래스 모두에서 생성자를 따로 정의하지 않았으므로 4-3-3에서 살펴본 바와 같이 디폴트 생성자가 생성된다. **파생 클래스의 생성자는 자동적으로 기반 클래스의 디폴트 생성자를 호출**하는 것이 원칙이다.

그러나 만약 사용자가 **기반 클래스에서 인자가 있는 생성자를 따로 정의하였을 경우에는 주의가 필요**하다. 예를 들어, 기반 클래스에서 인자가 있는 생성자를 정의하고 디폴트 생성자(인자가 없는 생성자)를 구현하지 않은 경우, 파생 클래스에서는 자동적으로 기반 클래스의 디폴트 생성자를 자동 호출하려고 하지만 이것이 존재하지 않기 때문에 에러가 발생한다. 따라서 기반 클래스에 디폴트 생성자를 추가하여 주거나, 또는 **파생 클래스에서 기반 클래스의 인자가 있는 생성자를 명시적으로 호출**하여야 한다.

생성자를 명시적으로 호출하는 방법은 다음과 같다. 생성자의 구현부에서 중괄호({)가 시작되기 직전에 기반 클래스를 호출해 준다.

```
파생클래스명::파생클래스명(인수1, 인수2, ...) : 기반클래스명(인수1, 인수2. ...)
{
    파생클래스 생성자 본체;
}
```

위에서, 파생 클래스와 기반 클래스의 인수 부분의 형태는 달라도 상관이 없다(단순 호출이기 때문). 다음은 실험 CPP-1에서 사용될 생성자로, 기반 클래스의 생성자를 명시적으로 호출하는 예이다(Array는 기반 클래스이고, RangeArray는 파생 클래스이다).

```
RangeArray::RangeArray(int l, int h) : Array(h-l+1)
{
    ...
}
```

4-5. 다형성(Polymorphism)

다형성이라는 말은 그리스어에서 유래된 것으로, “많은, 여러”의 뜻을 가진 “Poly”와 “형태”의 뜻을 가진 “Morph”의 합성어이다. 즉, “여러 가지의 형태”를 뜻하는 단어로, 이는 단지 객체 지향 프로그래밍에서만 사용되는 용어는 아니다. 생물학 등의 여러 분야에서 이 용어가 사용되는데, 프로그래밍 언어에서의 다형성은 **하나의 인터페이스를 사용하여 여러 형태의 데이터 타입(또는 함수)들을 사용할 수 있게 하는 특징**을 뜻한다. 여기서는 C++의 프로그래밍적 관점에서 다형성이 어떻게 발휘되는지 살펴보도록 한다.

4-5-1. 파라미터적 다형성(Parametric Polymorphism)

어떤 소프트웨어 시스템을 개발하든지 간에 자료구조는 필수적으로 사용되는 것들이다. 스택(Stack), 큐(Queue), 트리(Tree), 테이블(Table) 등이 바로 그러한 것들인데, 스택의 예를 고려해 보자. 먼저 정수를 담을 수 있는 스택이 필요하다고 했을 때 다음과 같이 선언하고 작성할 수 있다.

```
class Stack_Int{
public:
        void push(int item);
        int pop();
        ...
};
```

그런데 만약 실수를 담을 수 있는 스택이 필요한 경우가 생겼을 때는 이미 작성한 정수 스택과 별로 다를 것이 없음에도 모든 코드를 다시 작성해야 한다. 물론 코드를 그대로 복

사-붙여넣기 한 후, 바꾸어야 할 부분만 바꾸어서 코드를 빠르게 작성할 수 있지만 이는 더욱 복잡하고 큰 프로그램에 대해서는 비효율적이다. 또한 비슷한 이름의 클래스의 수가 많아져서 혼란을 야기할 수 있다. 이처럼, **순수하게 자료형(Type)만이 다르기 때문에 모든 코드를 다시 작성해야 하는 문제를 해결해 주는 것이 바로 파라미터적 다형성이다.** 이것의 예로, 다음과 같은 코드로 스택을 다시 작성할 수 있다.

```
template <class T>
class Stack{
public:
        void push(T item);
        T pop();
        ...
};
```

이는 C++의 **템플릿(틀, Template)**을 사용하여 선언부를 다시 작성한 것이다. 맨 윗줄의 template <class T> 부분은, T라는 클래스를 임의의 클래스(또는 자료형)로 쓰겠다는 의미이다. 이를 클래스의 선언부 앞에 명시하였기 때문에, 이 구문의 의미는 앞으로 클래스 Stack에서 사용되는 모든 T라는 키워드는 임의의 클래스로 간주하겠다는 의미가 된다. 사용할 때는 기존 클래스명 Stack대신 Stack<T>를 사용하면 된다. 만약, 함수 push()를 호출할 때 파라미터로 변수를 넘겨주면 그 변수의 자료형에 따라서 T가 결정되게 된다. 즉, T로 지칭되지만 이 T는 그것이 구체적으로 결정되는 시점에서 여러 가지 다양한 형태의 자료형이 될 수 있기 때문에 도입부에서 설명한 다형성의 의미에 부합한다.

템플릿은 위 예에서 살펴본 것처럼, 템플릿 클래스를 만드는 데 사용할 수도 있고, 함수 템플릿을 만들 때도 사용할 수 있다. 함수의 경우에도 함수 선언 앞에 template <class T> 와 같이 써 주고 사용하면 된다. 그런데 주의해야 할 점은, 보통 C++로 프로그래밍을 할 때 클래스나 함수의 정의, 선언은 헤더 파일(.h)에, 구현은 소스 파일(.cpp)에 기술하지만 템플릿 클래스나 함수는 헤더 파일에 모두 기술하여야 한다는 것이다. 그 이유는 컴파일러가 코드를 변환할 때의 처리 방법이 일반 코드와 다르기 때문이다.

참고로, 이러한 파라미터적 다형성의 편리함 때문에, C++에서는 **표준 템플릿 라이브러리(Standard Template Library, 줄여서 STL)**를 통해 일반적으로 많이 사용하는 클래스와 함수들을 모아서 제공하고 있다. 예를 들어, 연결 리스트(Linked List), 동적 배열 클래스, 정렬 함수, 검색 함수 등이 있다. 일단 표준이기 때문에, 표준 템플릿 라이브러리를 사용하여 프로그램을 작성하면 표준을 준수하는 어떤 컴파일러에서도 컴파일이 가능하며, 누구든지 동일한 형태로 프로그램을 작성할 수 있다. 또한, 전문가들이 만들어 놓은 것이기 때문에 효율적이고, 안전하다. 따라서 표준 템플릿 라이브러리를 이용하면 온갖 자료구조의 구현 등을 고민하고 프로그래밍하고 검증하는 시간과 노력을 절약하여 자신이 만들고자 하는 프로그램의 구현에 더욱 더 시간과 노력을 집중할 수 있다.

4-5-2. 서브타입 다형성(Subtype Polymorphism)

보통 객체 지향 프로그래밍에서 흔히 이야기하는 다형성이 바로 서브타입 다형성이다. 이것의 핵심은 바로 대체가능성(Substitutability)이라고 하는 것인데, 즉, **어떤 자료형 T를 요구하는 상황에서, 자료형 T를 가지는 객체뿐만 아니라 그것의 서브타입을 가지는 객체도 대신 사용할 수 있다**는 것이다. 예를 들어, 다음과 같이 T라는 자료형이 존재하고 T1과 T2는 이의 서브타입이라고 가정하자. 4-4에서 살펴본 상속의 개념으로 보자면, 기본 클래스

에 대해 파생된 클래스를 기본 클래스의 서브타입이라고 볼 수 있다.

```
class T { ... };
class T1 : public T { ... };
class T2 : public T { ... };
```

그리고 T, T1, T2 각각의 인스턴스가 t, t1, t2 라고 하자.

```
T t(...);
T1 t1(...);
T2 t2(...);
```

여기서, 자료형 T의 값을 인자로 요구하는 함수 f가 있다고 한다면, 이 인자에는 다음과 같이 t뿐만 아니라 t1, t2도 대신 사용할 수 있다. 왜냐하면, t1, t2는 T의 서브타입의 인스턴스이기 때문이다.

```
void f(T x); // 함수의 선언부
...
f(t);  // OK
f(t1); // OK
f(t2); // OK
...
```

위에서 함수 f의 입력 x는 그것의 서브타입이 어떤 형태인지 구체적으로 결정되는 시점에서 다양한 형태의 자료형이 될 수 있기 때문에 도입부에서 설명한 다형성의 의미에 부합한다. 이러한 서브타입 다형성의 특성은, **특히 C++에서는 주로 상속 및 재정의와 같이 사용되어 기반 클래스에서 파생 클래스의 멤버 함수를 호출하는 방법을 구현하는 것에 널리 사용**되고 있다. 이는 어려운 개념이지만 충분히 습득할 가치가 있다.

예를 들어, 스타크래프트 게임의 예를 생각해 보자. 게임의 각 유닛은 몇몇 유닛을 제외하고는 모두 공격 커맨드를 가지고 있다. 그러나 각각의 유닛은 공격력, 공격 속도, 공격 방법 등이 모두 다르기 때문에 공격 커맨드의 내부 구성도 모두 다를 것이다. 공격 커맨드를 각 유닛 클래스의 attack()이라는 함수가 담당한다고 하자. 그렇다면 여러 가지 유닛이 한꺼번에 공격을 수행하도록 명령을 내리는 코드를 작성한다면 다음과 같이 작성할 수 있을 것이다.

```
// marine과 firebat과 vulture는 각 유닛 클래스(Marine, Firebat, Vulture)의 인스턴스
marine.attack(...);
firebat.attack(...);
vulture.attack(...);
```

만약 유닛의 수가 매우 많다면 어떻게 될까? 위의 예에서 marine, firebat, vulture가 다수일 경우 각각의 클래스마다 따로 attack()함수를 호출하여야 한다. 그러나 1)만약 각 유닛의 클래스가 어떤 기반 클래스를 공통으로 가지고 있는 파생 클래스라면, 2)그리고 앞에서의 4-4-4에서 다룬 재정의를 통해 기반 클래스의 attack()함수를 커스터마이즈한다면, 3)또한 기반 클래스의 attack()함수는 가상(Virtual) 함수라면, 다음과 같이 간단하게 공격 명령 코드를 작성할 수 있다.

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

// 기반 클래스 Unit의 선언부
class Unit{
public:
	virtual void attack(){cout<<"Which unit attacks?"<<endl;};
};

// 파생 클래스들
class Marine : public Unit{
public:
	virtual void attack(){cout<<"A marine attacks."<<endl;};
};
class Firebat : public Unit{
public:
	virtual void attack(){cout<<"A firebat attacks."<<endl;};
};
class Vulture : public Unit{
public:
	virtual void attack(){cout<<"A vulture attacks."<<endl;};
};

int main(){
	Unit unit, *units[3];
	Marine marine;
	Firebat firebat;
	Vulture vulture;

	// 포인터를 사용하여 기반 클래스가 서브타입(파생 클래스)을 대표하도록 형변환
	units[0] = &marine;
	units[1] = &firebat;
	units[2] = &vulture;

	unit.attack();
	// 세 유닛 동시에 공격 명령
	for(int i = 0; i < 3; i++) units[i]->attack();
}
```

앞에서 설명한 것처럼, 기반 클래스 Unit이 있고, 여기서 attack()함수는 파생 클래스에서 재정의 될 것이기 때문에 virtual을 붙여서 가상 함수임을 나타내었다. 그리고 파생 클래스들에서는 attack()함수를 각각의 유닛의 특성에 맞추어 재정의하였다. 그 다음에 서브타입이 기반 클래스 대신 사용되는 서브타입 다형성을 살펴볼 수 있는데, 원래대로라면 units변수에는 클래스 Unit의 인스턴스의 주소가 할당되는 게 맞지만 서브타입인 Marine, Firebat, Vulture의 인스턴스의 주소가 사용되었다. 이렇게 함으로써, 맨 아래 부분과 같이 간단히 반복문을 사용하여 여러 유닛의 공격 명령을 가능하게 할 수 있었다. 같은 이름을 사용하여

이루어지는, 다른 작동의 공격의 다형성이 이를 통해 가능하게 된 것이다. 예제 코드에서는 각 클래스 인스턴스의 포인터를 사용하였지만, 3-3의 참조(Reference)를 사용해서도 같은 작업을 구현할 수 있다.

가상 함수(virtual function)란, 포인터의 정적 타입(포인터의 자료형)이 아닌 동적 타입(포인터가 가리키는 객체의 자료형)을 따르는 함수이다. virtual 키워드는 클래스 선언문 내에서만 쓸 수 있으며 함수 정의부에서는 쓸 수 없다. 정의부에 virtual을 쓰면 에러 처리되므로 함수를 외부 정의할 때는 virtual 키워드 없이 함수의 본체만 기술해야 한다. 또한 기반 클래스에서 virtual 키워드를 명시하면 이후 파생클래스에선 해당 함수는 계속 가상 함수로 사용되지만, 일반적으로 전부 virtual 키워드를 명시하여 가상 함수임을 명확히 밝히는 것이 좋다. 아래 예는 가상 함수인지 아닌지에 따른 차이를 보여준다.

```
// nonvirtual
#include <iostream>
using namespace std;
class Base{
public:
	void func();
};
class Derived : public Base{
public:
	void func();
};
void Base::func(){
		cout<<"Base"<<endl;
}
void Derived::func(){
		cout<<"Derived"<<endl;
}
int main(){
		Base *p = new Base();
		p->func();
		p = new Derived();
		p->func();
		return 0;
}
```

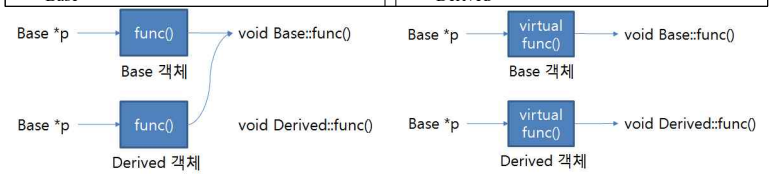
실행결과

```
Base
Base
```

```
// virtual
#include <iostream>
using namespace std;
class Base{
public:
	virtual void func();
};
class Derived : public Base{
public:
	virtual void func();
};
void Base::func(){
		cout<<"Base"<<endl;
}
void Derived::func(){
		cout<<"Derived"<<endl;
}
int main(){
		Base *p = new Base();
		p->func();
		p = new Derived();
		p->func();
		return 0;
}
```

실행결과

```
Base
Derived
```



이러한 서브타입 다형성을 이용한 프로그래밍은 개발자에게 많은 이득을 가져다준다. 코드의 양을 대폭 줄여 주며, 프로그래밍에 유연성을 가져다준다. 이는 개발과 유지보수의

측면에서 많은 장점이 있다.

4-5-3. 애드혹 다형성(Ad-hoc Polymorphism)

애드혹 다형성은 엄밀히 말해 맨 처음에 제시했던 다형성의 의미와 일치하는 개념은 아니다. 앞의 두 다형성(파라미터적 다형성, 서브타입 다형성)이 동일한 개체가 여러 가지 특성을 가질 수 있음을 나타낸다면, 2-4-2에서 설명한 **다중정의(Overloading)로 대표되는 애드혹 다형성**은 다중정의의 특성상 미리 여러 함수를 작성하여 놓고, 단지 인수만 다르게 사용하여 호출할 때 그에 맞는 함수만 골라서 호출해 주는 방식으로 작동하기 때문이다. 즉, 하나의 개체의 다형성이 아니라 여러 개체가 하나의 개체인 척 행동하는 것이다. 그러나 프로그래밍 언어학에서 다형성을 말할 때 앞의 두 다형성과 함께 애드혹 다형성도 함께 언급하기도 한다.

C++에서는 함수의 다중정의도 제공하지만 일반적으로 프로그래밍에서 사용되는 연산자들(+, -, * 등)에 대해 다른 작업을 할당함으로써, 연산자가 복수의 의미를 가질 수 있게 하는 **연산자 다중정의(Operator Overloading)** 또한 제공한다. 예를 들어, 벡터(Vector)를 정의하는 클래스를 작성했을 경우, 벡터에 사용할 수 있는 연산들(합, 곱 등)이 존재하며 이러한 연산을 표현하는 방법이 필요하다. 이 때 멤버 함수로서 벡터 연산을 구현할 수 있지만 해당 연산자(+, *)들을 사용하는 것이 직관적으로 이해하기 쉽고 사용자에게 더 편리하다.

C++에서 다중정의가 가능한 연산자는 다음과 같으며,

```
+   -   *   /   =   < >   +=   -=   *=   /=   << >> <<=   >>=   ==   !=   <=   >=
++   --   %   &   ^   !   |   ~   &=   ^=   |=   &&   ||   %=   [ ]   ( )   new   delete
```

연산자의 정의 형식 및 규칙은 다음과 같다.

```
리턴값 operator 연산자(인수1, 인수2, ...) {
    함수의 본체;
}
```

- ☞ 기존에 있는 연산자만을 재정의 할 수 있다.
- ☞ 연산자 함수는 클래스 내부에서 멤버 함수여야 한다.
- ☞ 다항연산자는 다항연산자로, 단항 연산자는 단항 연산자로 재정의해야 한다.
- ☞ 연산자 함수가 멤버함수 일 때는 자신이 속한 객체 그 자체를 인수로 전달받게 됨.
- ☞ 연산자 함수는 디폴트(default) 인수를 가질 수 없다.
- ☞ 재정의된 연산자일지라도 연산자 우선순위는 변하지 않는다.

연산자 다중정의를 사용하여 벡터 클래스에 특정 연산자들을 사용하는 예제를 다음 코드에 나타내었다. 연산자에 내포된 의미를 잘 활용한다면 이와 같이 코드의 가독성을 향상시킬 수 있음을 볼 수 있다.

example16.cpp

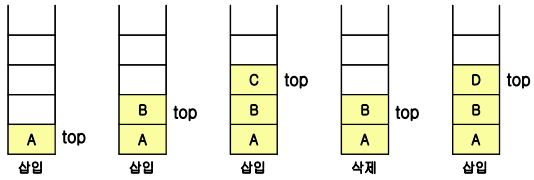
```
#include <iostream>
using namespace std;

class Complex {
    private:
        double real;      //실수부
        double image;     //허수부
    public:
        Complex(){real=0; image=0;};
        Complex(double r, double i){
            real=r;   image=i;
        };
        Complex add(Complex c);
        Complex operator+(Complex c);
        Complex operator-(Complex c);

        void prnComplex(void){
          cout<<real<<" + "<<image<<"i"<<endl;
        };
};

// 만약 operator 다중정의를 사용하지 않으면
// 이와 같은 형태로 멤버함수를 사용해야 함
// 이 함수를 사용하는 경우 가독성이 떨어짐
Complex Complex::add(Complex c){
    Complex T;
    T.real = real+c.real;
    T.image = image+c.image;
    return T;
}

// + operator 연산자 다중정의
Complex Complex::operator+(Complex c){
    Complex T;
    T.real = real+c.real;
    T.image = image+c.image;
    return T;
}

// - operator 연산자 다중정의
Complex Complex::operator-(Complex c){
    Complex T;
    T.real = real-c.real;
    T.image = image-c.image;
    return T;
}

int main(){
    Complex Com1(1.1,1.2);
    Complex Com2(2.2,3.3);
    Complex Com3, Com4, Com5;
    Com3 = Com1+Com2;
    Com4 = Com1-Com2;
    Com3.prnComplex(); Com4.prnComplex();
    // 멤버함수를 사용하는 경우
    Com5 = Com1.add(Com2);
    // 가독성이 떨어짐
    return 0;
}
```

[Code 16] 연산자 다중정의

5. 클래스를 이용한 스택(Stack)의 간단한 예

5-1. 스택

스택이란 순서 리스트의 특별한 경우로 톱(Top)이라 불리는 한 쪽 끝에서 모든 삽입과 삭제가 발생하는 순서 리스트이다. 스택에서는 제일 나중에 들어온 원소가 제일 먼저 삭제되므로 후입선출(Last-In-First-Out, 줄여서 LIFO) 리스트라고도 한다.

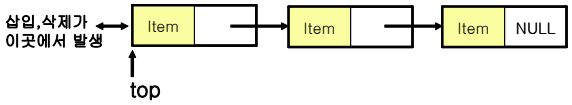


[그림 11] 스택에서의 삽입과 삭제.

5-2. 구현할 스택의 정의

스택은 데이터의 삽입과 삭제가 한 방향에서만 이루어지는 자료구조로 삽입 연산이 발생하는 경우 top이 증가, 삭제 연산이 발생하는 경우 top이 감소한다. 그리고 빈 스택을 생성할 수 있어야 하고 스택의 크기는 제한점이 존재한다. 스택이 얼마나 사용되는지 확인 가능해야 하고 스택에 저장되는 객체는 Item 객체로 사용자가 정한 객체 또는 일반 자료형을 사용하기로 한다. 스택의 내부 구현은 리스트로 구현하기로 결정하였다.

☞ 이를 위해 리스트 구조체를 표현하는 방법이 필요함.



struct Node

```
struct Node{
        Item item;    // 리스트에 담을 임의의 자료형
        struct Node* next;
}
```

Stack Abstact Data Type

```
structure : Stack
object    : 0개 이상의 원소를 가진 유한 순서 리스트
variable : MaxSize positive integer
function :
        Stack() ::= 스택의 생성자. 최대 크기를 사용자가 정하지 않았을 경우 default 크기 제공
        Stack(int maxStackSize) ::= 스택의 최대 크기가 maxStackSize인 스택을 생성
        ~Stack() ::= 스택의 소멸자
        bool isEmpty(void) ::= 스택이 현재 비어있는지를 확인하는 함수
        bool isFull(void) ::= 스택이 현재 최대 크기를 사용하고 있는지를 확인하는 함수
        bool push(const Item &item) ::= Item 원소를 삽입
        bool pop(Item &item) ::= Item 원소를 삭제
```


스택의 구현

스택 클래스

```
typedef int Item;  // 구조를 간단히 하기 위해 int형을 Item으로 변경

class Stack{

        struct Node {
                Item item;
                struct Node *next;
        };

        enum {StackSize=10}; // Stack 클래스 내에서 사용할 상수(StackSize) 선언
        private:
                Node *top;
                int MaxSize;
                int currentSize;
        public:
                Stack(){
                        currentSize = 0;
                        MaxSize=StackSize;
                        top = NULL;
                };
                Stack(int maxStackSize){
                        currentSize = 0;
                        MaxSize = maxStackSize;
                        top = NULL;
                };
                ~Stack(){};

                bool isEmpty(void) const; // 함수 뒤의 const는 상수 멤버 함수임을 나타냄
                bool isFull(void) const;   // 상수 멤버 함수는 멤버 변수 값을 변경할 수 없음
                int stackCount(void) const;
                bool push(const Item &item);
                bool pop(Item &item);
};
```

전체 코드

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

typedef int Item;  // 구조를 간단히 하기 위해 int형을 Item으로 변경

class Stack{
	struct Node {Item item; struct Node *next;};
	enum {StackSize=10}; // Stack 클래스 내에서 사용할 상수(StackSize) 선언
	private:
		Node *top;
		int MaxSize;
		int currentSize;
	public:
		Stack(){
			currentSize = 0;
			MaxSize=StackSize;
			top = NULL;
		};
		Stack(int maxStackSize){
			currentSize = 0;
			MaxSize = maxStackSize;
			top = NULL;
		};
		~Stack(){};

		bool isEmpty(void) const; // 함수 뒤의 const는 상수 멤버 함수임을 나타냄
		bool isFull(void) const;   // 상수 멤버 함수는 멤버 변수 값을 변경할 수 없음
		int stackCount(void) const;
		bool push(const Item &item);
		bool pop(Item &item);
};

bool Stack::push(const Item &item)
{
	if(isFull())
		return false;

	Node *add = new Node;
	if(!add)
		return false;
	add->item = item;
	add->next = NULL;
	currentSize++;
	if(!top)
		top = add;
	else{
		add->next = top;
		top = add;
	}
	return true;
}

bool Stack::pop(Item &item){
	if(isEmpty())
		return false;
	Node *tmp;
	currentSize--;
	item = top->item;
	tmp = top;
	top = top->next;
	tmp->next = NULL;
	delete(tmp);
	return true;
}
```


전체 코드 (계속)

```
bool Stack::isEmpty(void) const{
	return currentSize==0;
}
bool Stack::isFull(void) const{
	return currentSize==MaxSize;
}
int Stack::stackCount(void) const{
	return currentSize;
}
// 함수의 선언
void prnMenu(void);

int main()
{
	int selectNumber, size;
	Item tmpItem;
	Stack *s;
	bool flag = false;

	cout<<"원하시는 stack size를 입력해주세요 : ";
	cin>>size;
	s = new Stack(size);
	while(1){
		prnMenu();
		cin>>selectNumber;
		switch(selectNumber){
		case 1:
			cout<<"원하시는 값을 입력해주세요 : ";
			cin>>tmpItem;
			if(s->push(tmpItem)==true)
				cout<<tmpItem<<"가 삽입되었습니다."<<endl;
			else
				cout<<"용량 초과, 삽입 실패"<<endl;
			break;
		case 2:
			if(s->pop(tmpItem)==true)
				cout<<tmpItem<<"가 삭제되었습니다."<<endl;
			else
				cout<<"스택이 비어있습니다. 삭제 실패"<<endl;
			break;
		case 3:
			cout<<"스택의 크기 : "<<s->stackCount()<<endl;
			break;
		case 4:
			flag = true;
			break;
		default:
			cout<<"잘못 입력하셨습니다."<<endl;
			break;
		}
		if(flag)
			break;
	}
	return 0;
}
void prnMenu()
{
	cout<<"***********************************************"<<endl;
	cout<<"* 1. 삽입     2. 삭제     3. 출력     4. 종료 *"<<endl;
	cout<<"***********************************************"<<endl;
	cout<<endl;
	cout<<"원하시는 메뉴를 골라주세요 : ";
}
```