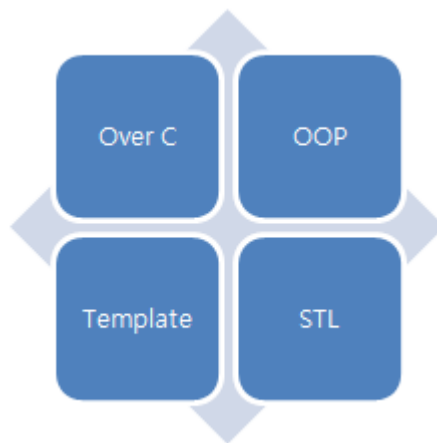


1. Over C

1. 1 C++소개

C++ 언어는 C언어를 기반으로 만들어진 개체 지향 프로그래밍 언어입니다. C++언어에서는 OOP 특징을 지원하기 위하여 클래스 문법이 추가되었습니다. 이 책에서는 개체 지향 프로그래밍 언어)의 주요 특징인 캡슐화, 상속, 다형성을 중심으로 클래스 문법을 기술하였습니다. 그리고 C++언어에서는 멤버의 형식에는 차이가 있지만 알고리즘에는 차이가 없는 행위나 형식을 정의할 때 가상의 코드를 정의하는 템플릿 문법이 있습니다. 이 외에도 C++언어를 구성하는 것으로 ANSI 기구에서 제공하는 표준 템플릿 라이브러리를 들 수 있습니다.

이번 장에서는 클래스, 템플릿, STL을 제외한 C++문법 중에서 C언어와 차이가 있는 부분을 다루겠습니다. C++언어는 C언어 문법에 비해 데이터의 높은 신뢰성을 추구하고 있습니다. C++ 언어에서는 신뢰성에 관한 많은 문법적인 제약을 두어 개발자의 실수로 버그를 만들 수 있는 부분을 컴파일러가 오류를 발생하여 개발 단계에서 수정할 수 있게 도움을 줍니다.



[그림 1.1] C++ 구성 요소

1. 2 신뢰성 강화

C++언어는 C언어보다 높은 데이터 신뢰성을 추구하여 개발자의 실수로 버그가 될 수 있는 부분을 컴파일 단계에서 수정할 수 있게 도와줍니다.

1.2.1 열거형

C언어에서 열거형은 정수형과 묵시적 형 변환이 됩니다. 이러한 특징때문에 C언어에서 열거형은 별도의 형식을 정의한다기보다 매크로 상수를 그룹화하는 목적으로 사용이 되었습니다. 하지만 C++언어에서 열거형은 정수형과 다른 형식으로 인식됩니다.

C언어에서는 정수형이 오기를 기대하는 곳에 열거형이 오거나 열거형이 오기를 기대하는 곳에 정수형이 온다고 하더라도 아무런 오류가 발생하지 않습니다. 하지만 C++언어에서는 열거형을 기대하는 곳에 정수형이 오면 컴파일 오류가 발생합니다.

C++ 언어에서는 열거형이 오기를 기대하는 곳에 정수형이 오면 컴파일 오류가 발생합니다. 이러한 이유는 열거형에 나열하지 않은 정수 값이 사용했을 때 데이터 신뢰성이 떨어지기 때문입니다. 반면 C++에서도 정수형이 오기를 기대하는 곳에 열거형이 오는 것은 아무런 문제가 되지 않습니다. C++언어에서 열거형에 나열할 수 있는 값은 정수 값이므로 정수형이 올 곳에 열거형이 와도 데이터 신뢰성에 문제가 되지 않기 때문입니다.

```
1 enum Gender
2 {
3     FEMALE,
4     MALE
5 };
6 void main( )
7 {
8     Gender g;
9     int i = 0;
10
11     g = i;
12     i = g;
13 }
```

❌ 1 error C2440: '=' : 'int'에서 'Gender'(으)로 변환할 수 없습니다. stub.cpp 11

[그림 1.2] 열거형이 올 곳에 정수가 왔을 때 오류 화면

1.2.2 const 포인터

C언어에서는 const 포인터 변수를 const가 아닌 포인터 변수에 대입하면 컴파일 오류가 발생하지 않고 경고에 그칩니다. 이러한 경고를 무시하고 const가 아닌 포인터 변수를 이용하여 메모리의 값을 변경할 수 있어서 데이터 신뢰성이 떨어질 수 있습니다.

예를 들자면 입력 인자로 const 포인터로 전달받은 것을 const가 아닌 포인터 변수에 대입하여 메모리의 값을 변경하는 것입니다. C언어에서는 const 포인터 변수를 const가 아닌 포인터 변수에 대입하면 컴파일 경고에 그칩니다.

하지만 C++에서는 const 포인터 변수를 const가 아닌 포인터 변수에 대입할 때 컴파일 오류를 발생하여 개발 단계에서 이를 수정할 수 있게 도와주어 신뢰성을 높여줍니다.

```
1 void Foo(const char *str);
2
3 void main( )
4 {
5     char arr[10]="hello";
6
7     Foo(arr);
8 }
9
10 void Foo(const char *str)
11 {
12     char *p = str;
13     *p = 'a';
14 }
```

✖ 1	error C2440: '초기화 중' : 'const char *'에서 'char *(으)로 변환할 수 없습니다.	program.	12
-----	---	----------	----

[그림 1.3] const 포인터를 const가 아닌 포인터에 대입할 때 오류 화면

1.2.3 void 포인터

void 포인터 형식의 변수는 선언 시에 원소 형식을 명시하지 않고 코드 상에서 어떠한 형식의 포인터도 받을 수 있는 형식입니다. 하지만 반대로 void 포인터 변수의 값을 일반(void 포인터형이 아닌) 포인터 변수에 대입할 경우 그 이후에 간접 연산자에 의해 모순된 작업을 할 개연성이 있습니다. C++에서는 경고가 아닌 오류를 발생시켜 사용자로 하여금 코드를 수정할 수 있게 해 줍니다.

```
1  #include <stdio.h>
2  void main( )
3  {
4      char arr[10]={1,2,3,4,5,6,7,8,9,10};
5      void *pv = 0;
6      int *pi = 0;
7      int i = 0;
8
9      pv = arr;
10     pi = pv;
11
12     *pi = 10;
13
14     for(i=0; i<10; i++)
15     {
16         printf("arr[%d]: %d\n",i, arr[i]);
17     }
18 }
```

❌ 1 error C2440: '=' : 'void *'에서 'int *'(으)로 변환할 수 없습니다. program. 10

[그림 1.4] void 포인터를 다른 포인터에 대입할 때 오류 화면

-1.2.4 bool 형식의 제공

프로그래밍하다 보면 특정 연산 결과가 참인지 거짓인지를 판별해야 하는 경우가 많이 있습니다. C언어에서는 참과 거짓을 값으로 표현하는 형식을 제공하지 않았지만 C++에서는 bool 형식을 제공합니다. 물론, C언어에서 어떠한 변수의 값이 0이면 거짓, 그 이외의 값일 경우에 참으로 인식하기 때문에 프로그래밍할 때 큰 지장이 생기지는 않습니다. 하지만 참과 거짓만을 값으로 갖는 별도의 형식을 제공하는 것보다는 가독성이 떨어질 수 있습니다.

C++언어의 bool 형식은 값으로 true나 false를 가질 수 있으며 이를 통해 좀 더 가독성이 높고 신뢰성 있는 코드를 작성할 수 있습니다.

```
#include <iostream>
using namespace std;
void main()
{
    bool check = false;
    int num = 0;
    cout<<"아무 수나 입력하세요."<<endl;
    cin>>num;
    check = (num % 2) == 0;
    if(check)
    {
        cout<<num<<"은 짝수입니다."<<endl;
    }
    else
    {
        cout<<num<<"은 홀수입니다."<<endl;
    }
}
```

[예제 1.1] bool 형식 사용 예

1.3 편의성 제공

C++언어에서는 신뢰성에 문제가 되지 않는 범위에서 사용자에게 많은 편의성을 제공하고 있습니다. 이번에는 C언어에서는 없었던 문법 사항 중에 사용자 편의성에 관한 부분을 다루어 보시다.

1.3.1 태그 명이 형식 명으로 사용

C언어에서는 사용자 정의 형식을 만들 때 명명하는 태그 명(struct, union, enum 뒤에 오는 명칭)을 바로 형식 명으로 사용할 수 없습니다.

```
#include <string>
using std::string;
enum Gender{  FEMALE,  MALE  };

struct StuInfo
{
    int num;
    string name;
};

int main()
{
    Gender g = MALE; //태그 명인 Gender를 형식 명으로 사용
    StuInfo si = {2,"홍길동"}; //태그 명인 StuInfo를 형식 명으로 사용
    return 0;
}
```

[예제 1.2] 태그 명을 형식 명으로 사용한 예

1.3.2 원하는 위치에 변수 선언

C언어에서 변수 선언은 블록 시작 위치에서만 가능합니다. C++언어에서는 변수 선언에 대한 위치가 블록 중간에 오는 것을 허용합니다.

1.3.3 레퍼런스 변수의 등장

C++언어에서는 변수 선언 시에 다른 변수에 의해 할당된 메모리를 참조하는 레퍼런스 변수를 선언할 수 있습니다. 이 경우에 레퍼런스 변수는 별도의 메모리가 할당되지 않습니다. 이러한 이유로 레퍼런스 변수는 선언 시에 우항에 대입 연산자의 좌항에 올 수 있는 표현(l-value라고 부름)이 와야 합니다.

주의할 것은 레퍼런스 변수는 선언 부 이외에는 기존 변수처럼 값 기반으로 동작합니다.

```
#include <iostream>
using std::cout;
using std::endl;

void main()
{
    int a = 0;
    int &ra = a;
    ra=3;
    cout<<"a:"<<a<<endl;
    cout<<"ra:"<<ra<<endl;
}
```

[예제 1.3] 레퍼런스 변수의 사용 예

1.3.4 함수 중복 정의(function overloading)

C언어에서는 같은 이름을 갖는 함수를 정의할 수가 없었습니다. C++에서는 특정 조건을 만족하게 하는 경우 같은 이름을 갖는 함수를 중복해서 정의할 수 있습니다. C++에서는 컴파일 과정에서 사용자가 정의한 코드를 전개하는 과정에서 사용자가 정의한 함수명을 매개 변수 리스트에 따라 유일한 이름의 함수명으로 결정하는 함수 부호화(코드화) 과정이 진행됩니다. 그리고 함수를 호출하는 부분은 가장 적절한 매개 변수를 갖는 함수가 호출될 수 있게 연결(함수 이름 Mangling)해 줍니다. 이러한 이유로 C++에서는 사용자가 정의한 함수를 호출할 때 사용하는 이름을 함수명이라고 부르는 것 보다 메서드 명이라 부르고 있습니다. 즉, 정의하는 것은 함수이고 이를 호출할 때 사용하는 이름은 메서드 명이라 할 수 있습니다.

```
#include <iostream>
using namespace std;
int GetMax(int a,int b)
{
    if(a>b){ return a; }
    return b;
}
char GetMax(char a,char b)
{
    if(a>b){ return a; }
    return b;
}
int main()
{
    cout<<GetMax(2,3)<<endl; //int GetMax(int a, int b);로 연결
    cout<<GetMax('a','b')<<endl; //char GetMax(char a,char b);로 연결
    return 0;
}
```

[예제 1.4] 함수 중복 정의 예

함수 중복 정의는 결국 C++ 컴파일러의 전개과정에서 유일한 이름의 함수명으로 변경되고 호출부의 코드도 인자가 가장 적절한 함수로 연결되는 것입니다. 그렇다고 모든 경우에 함수 중복 정의가 가능하지는 않습니다. 컴파일러에서 호출하는 메서드명을 어떠한 함수를 호출하는 것인지 결정할 수 있어야 함수 중복 정의가 가능합니다.

만약 리턴 형식만 다르다면 어떠한 함수를 호출하는 것인지 컴파일러가 판단하지 못하기 때문에 함수 중복 정의할 수 없습니다.

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 void Foo( )
6 {
7     cout<<"void Foo( )" <<endl;
8 }
9 int Foo( )
10 {
11     cout<<"int Foo( )" <<endl;
12     return 0;
13 }
14 void main( )
15 {
16     Foo( );
17 }
```

✖ 1	error C2556: 'int Foo(void)' : 오버로드된 함수가 'void Foo (void)'과(와) 반환 형식만 다릅니다.	program.cpp	10
✖ 2	error C2371: 'Foo' : 재정의. 기본 형식이 다릅니다.	program.cpp	10
✖ 3	error C3861: 'Foo': 식별자를 찾을 수 없습니다.	program.cpp	16

[그림 1.5] 반환 형식만 다른 함수 중복 정의할 때 오류 화면

이 외에도 함수 호출부에 있는 코드를 어느 함수로 연결(함수 name mangling)할지 판단하기 모호한 경우에도 에러가 발생합니다.

1.3.5 디폴트 매개 변수

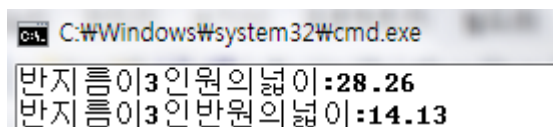
C++언어에서는 특정 함수를 호출할 때 사용하는 입력 매개 변수의 값이 대부분 같은 값을 전달하는 경우 디폴트 매개 변수를 사용할 수 있습니다.

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

double CalculateArea(double radius, double radian=3.14)
{
    return radius*radius*radian;
}

int main()
{
    cout<<"반지름이 3인 원의 넓이:";
    cout<<CalculateArea(3)<<endl;
    cout<<"반지름이 3인 반원의 넓이:";
    cout<<CalculateArea(3,3.14/2)<<endl;
    return 0;
}
```

[예제 1.5] 디폴트 매개 변수의 사용 예



[그림 1.6] 예제 1.5 실행 화면

위의 CalculateArea함수는 반지름과 부채꼴의 중심 각에 대한 radian을 인자로 받아 넓이를 구하는 함수입니다. 두 번째 입력 매개 변수의 디폴트 값을 3.14로 지정하였는데 이 경우 두 번째 입력 매개 변수를 전달하지 않으면 디폴트 값을 사용하게 됩니다. 물론, 특별한 인자 값을 전달하면 해당 값을 사용하게 됩니다.

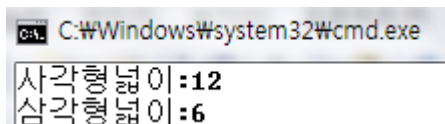
1.3.6 매개 변수명이 없는 입력 매개 변수

C++에서 함수 중복이 가능하다는 것은 위에서 이미 언급한 바가 있습니다. 그런데 메서드 명을 같게 부여하고 싶은데 입력 인자가 같다면 어떻게 해야 할까요? C++언어에서는 이 경우에 두 개의 함수를 구분하기 위해 매개 변수명이 없는 입력 매개 변수를 사용할 수 있습니다. 물론 호출하는 곳에서는 피 호출함수에 값이 전달되어 사용되지는 않지만, 호출 시 이에 대한 값도 반드시 넣어야 합니다.

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

int CalculateArea(int width,int height)
{
    return width*height;
}
int CalculateArea(int width,int height,bool)
{
    return width*height/2;
}
int main()
{
    cout<<"사각형 넓이:";
    cout<<CalculateArea(3,4)<<endl;
    cout<<"삼각형 넓이:";
    cout<<CalculateArea(3,4,false)<<endl;
    return 0;
}
```

[예제 1.6] 매개 변수 명이 없는 입력 매개 변수 사용 예



```
C:\Windows\system32\cmd.exe
사각형 넓이 : 12
삼각형 넓이 : 6
```

[그림 1.7] 예제 1.6 실행 화면

1.3.7 namespace

C++ 언어는 1988년에 만들어진 이후에 계속해서 새로운 문법이 추가되고 있습니다. 이렇게 추가된 문법 중의 하나가 namespace인데 이를 이용하면 같은 이름의 형식이나 개체 등이 정의된 여러 라이브러리 중에 원하는 부분을 선별적으로 사용할 수 있습니다. 가령, ALib와 BLib에 Stack과 Queue라는 사용자 형식을 제공하고 있는데 ALib에 있는 Stack과 Queue를 사용한다고 가정해 봅시다. 만약 namespace로 구분되어 있지 않다면 ALib를 추가하고 BLib를 추가를 하면 같은 이름이 사용자 형식이 정의되어 있어 컴파일 오류가 발생합니다. 이러한 문제점을 위해 C++에서는 namespace문법이 추가되었습니다.

이에 대해 살펴보기 위해 다음의 예를 들어보기로 하겠습니다.

```
namespace ALib
{
    class Stack{    //...중략...    };
    class Queue{    //...중략...    };
};

namespace BLib
{
    class Stack{    //...중략...    };
    class Queue{    //...중략...    };
};

using namespace ALib; //using 문 이용
int main()
{
    Stack stack1;
    BLib::Stack stack2; //namespace 이름 BLib와 스코프 연산자 사용
    return 0;
}
```

[예제 1.7] namespace를 이용하는 예

예제 코드를 보면 namespace ALib와 BLib 내부에 Stack과 Queue가 정의하였습니다. 만약 namespace로 묶지 않았다면 이름 충돌이 나서 컴파일 오류가 발생합니다. 이 때 사용하는 곳에서 특정 namespace 내에 있는 이름을 사용할 때는 namespace 이름과 스코프 연산자(::)를 사용하거나 using 문을 이용하면 됩니다.

using 문을 이용할 때는 특정 namespace 내의 모든 명칭을 사용하게 지정할 수도 있지만 특정 이름만 사용하게 지정할 수도 있습니다. 다음은 ALib의 Stack과 BLib에 Queue를 사용하기 위해 using 문을 사용한 예입니다.

namespace를 이용하는 예2

```
namespace ALib
{
    class Stack {    //...중략...  };
    class Queue{    //...중략...  };
};
namespace BLib
{
    class Stack{    //...중략...  };
    class Queue{    //...중략...  };
};
using ALib::Stack; //using 문 이용
using BLib::Queue; //using 문 이용
int main()
{
    Stack stack;
    Queue queue;
    return 0;
}
```

2. 캡슐화

2.1 캡슐화란?

이번 장부터 C++의 클래스에 관한 얘기가 시작됩니다. 클래스에 대해 문법을 효과적으로 이해하고 사용하기 위해서는 OOP(Object Oriented Programming, 개체 지향 프로그래밍)의 특징을 잘 인지하여야 합니다. 개체 지향 프로그래밍(많은 곳에서 개체를 객체라 부르고 있습니다. 여러분이 MSDN에서 번역한 것처럼 개체라고 부르겠습니다).

OOP의 특징을 얘기할 때 많은 이들이 OOP의 세 가지 기둥을 얘기합니다. OOP의 세 가지 기둥에는 캡슐화와 상속, 다형성이 있습니다. 이 중에 캡슐화는 여러 개의 멤버를 하나의 형식으로 묶어서 정의하는 것을 말합니다.

2.2 접근 지정자

C언어에서는 구조체와 공용체를 이용하여 사용자 형식을 정의합니다. 그리고 C언의 구조체와 공용체에서는 멤버 변수만 캡슐화가 가능하였습니다. C++언어에서는 캡슐화할 수 있는 대상이 멤버 필드(멤버 변수라고도 부름)외에도 멤버 메서드(멤버 함수라고도 부름)를 캡슐화 할 수 있습니다.

또한 C언어에서는 모든 멤버들이 어디에서나 접근이 가능하였지만 C++언어에서는 멤버들에 대한 접근 지정자를 통해 접근할 수 있는 가시성을 다르게 지정할 수 있습니다. 접근 지정자로 가시성을 차별화함으로써 내부에 중요한 멤버의 접근을 차단하여 정보 은닉화가 가능하여 데이터 신뢰성을 높일 수 있습니다.

접근 지정자는 모든 곳에서 접근이 가능한 public, 내부에서만 접근할 수 있는 private, 해당 형식과 파생된 형식에서 가시성이 있는 protected가 있습니다. 그리고 구조체는 접근 수준을 명시하지 않으면 모든 곳에서 접근 가능한 public 수준이 되고 클래스는 형식 내부에서만 접근 가능한 private 수준이 됩니다. protected에 대한 문법은 기반 클래스에서 파생 클래스로 상속을 다루는 일반화 관계에서 다루도록 하겠습니다.

다음의 예를 보면 Stu클래스 내에 iq의 접근성은 디폴트로 하고 나머지 멤버는 public으로 지정하였습니다. 클래스는 디폴트 접근성이 private이므로 iq의 접근성은 private이 되어 Stu 클래스내에서만 접근이 가능하고 다른 곳에서는 접근할 수 없습니다.

```
Stu.h

#pragma once
#define MAX_IQ 200
class Stu
{
    int num;
    int iq;
public:
    Stu(int _num);
    void Study(int tcnt);
};
```

```
Stu.cpp

#include "Stu.h"
Stu::Stu(int _num)
{
    num = _num;
    iq = 100;
}
void Stu::Study(int tcnt)
{
    iq += tcnt;
    if(iq>MAX_IQ)
    {
        iq = MAX_IQ;
    }
}
```

```

Program.cpp

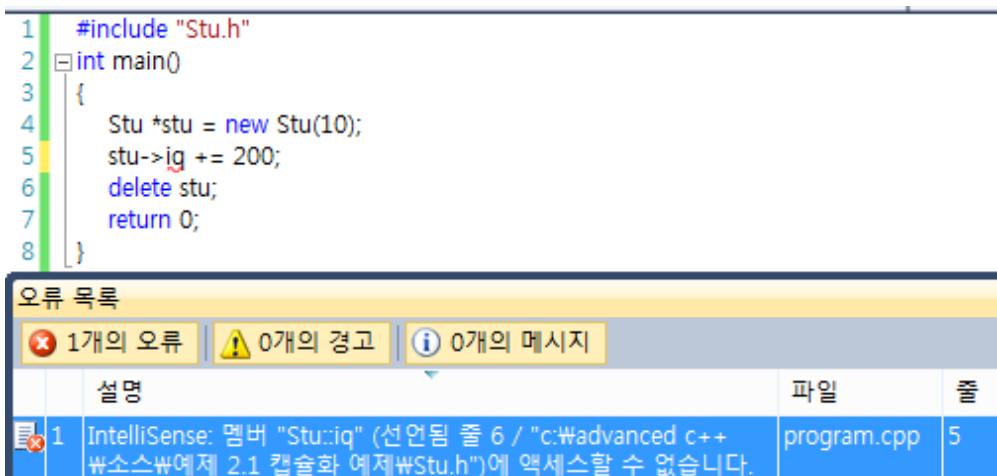
#include "Stu.h"

int main()
{
    Stu *stu = new Stu(10);
    stu-> Study(200);
    delete stu;
    return 0;
}

```

[예제 2.1] 캡슐화 예제

만약 Stu 형식 외부에서 private으로 접근 지정한 멤버인 iq에 접근하면 컴파일 오류를 발생하여 정보 은닉할 수 있습니다.



[그림 2.1] private 접근 지정한 멤버에 접근할 때 오류 화면

C++에서 지정할 수 있는 접근 지정자는 public, protected, private이 있습니다. 이 중에서 protected는 해당 형식과 파생된 형식에서 접근이 가능합니다. 이에 대한 부분은 일반화 관계를 다루면서 언급하고 여기서는 설명을 생략하도록 하겠습니다.

2.3 캡슐화의 대상

C언어에서는 구조체를 정의할 때 멤버 변수를 캡슐화할 수 있었습니다. C++에서는 멤버 변수(멤버 필드라고 부름)와 멤버 메서드를 캡슐화할 수 있습니다. 그리고 개체를 생성할 때 수행하는 생성자와 소멸할 때 수행하는 소멸자를 캡슐화할 수 있습니다.

그리고 C++에서 캡슐화하는 멤버는 개체마다 부여하는 개체의 멤버와 클래스에 유일하게 부여하는 클래스의 멤버(정적 멤버라고 부름)가 있습니다. 또한 멤버 필드 중에 값을 변경할 수 없는 상수화 멤버 필드가 있고 메서드 내에서 개체의 상태를 변경할 수 없는 상수화 멤버 메서드를 지정할 수 있습니다.

2.3.1 멤버 메서드

이번에는 멤버 메서드에 대해서 알아보기로 합니다. C++언어에서 사용자가 형식을 정의할 경우 멤버 필드와 멤버 메서드를 캡슐화할 수 있다고 하였습니다. 멤버 메서드를 캡슐화를 할 경우 메서드에서 수행할 코드를 정의하는 것은 클래스 정의문 내에서 할 수도 있고 클래스 정의문 외부에서도 할 수 있습니다.

```
class Stu
{
public:
    void Study();
};

void Stu::Study()
{
    cout<<"공부하다."<<endl;
}
```

[예제 2.2] 멤버 메서드 캡슐화

2.3.2 생성자

멤버 메서드에는 이미 이름이 정해진 메서드들도 있는데 대표적인 것이 생성자와 소멸자입니다. 생성자는 형식 이름과 같은 메서드를 말하며 소멸자는 ~와 형식 이름으로 정의한 메서드를 말합니다.

```
class Stu
{
    ...중략...
public:
    Stu(); //생성자
    ~Stu(void); //소멸자
};
```

C++에서 특정 클래스 형식의 개체 인스턴스를 생성할 때 new 연산자를 사용합니다. 그리고 생성자를 호출하면 개체를 위해 메모리를 할당하는 등의 초기 작업을 수행한 후에 생성자 메서드를 수행합니다.

개발자가 생성자를 정의하지 않으면 매개 변수가 없는 기본 생성자인 디폴트 생성자가 컴파일러에 의해 만들어집니다. 하지만 개발자가 생성자를 정의하면 디폴트 생성자는 만들어지지 않습니다.

Stu.h

```
#pragma once
#include <iostream>
using namespace std;
class Stu
{
public:
    void Study();
};
```

Stu.cpp

```
#include "Stu.h"
void Stu::Study()
{
    cout<<"Stu::Study()"<<endl;
}
```

Program.cpp

```
#include "Stu.h"
int main()
{
    Stu *stu = new Stu();
    stu->Study();
    delete stu;
    return 0;
}
```

[예제 2.3] 생성자를 개발자가 정의하지 않은 예

```

Stu.h

#pragma once
#include <string>
using std::string;
class Stu
{
    string name;
public:
    Stu(string _name);
};

```

```

Stu.cpp

#include "Stu.h"
Stu::Stu(string _name)
{
    name = _name;
}

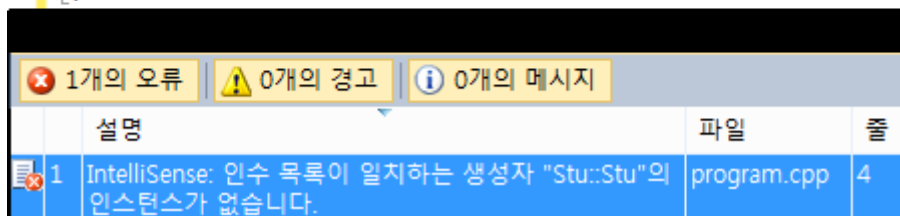
```

[예제 2.4] 입력 인자가 있는 생성자만 정의

```

1  #include "Stu.h"
2  int main()
3  {
4      Stu *s = new Stu();
5      delete s;
6      return 0;
7  }

```



[그림 2.2] 입력 인자가 있는 생성자만 정의하고 기본 생성 요청시 오류 화면

그리고 생성자는 중복 정의가 가능한 메서드입니다. 다음의 예를 살펴보세요.

Stu.h

```
#pragma once
#include <iostream>
#include <string>
using namespace std;
class Stu
{
    int num;
    string name;
    string addr;
public:
    Stu(int _num, string _name);
    Stu(int _num, string _name, string _addr);
    void View();
};
```

Stu.cpp

```
#include "Stu.h"
Stu::Stu(int _num, string _name)
{
    num = _num;
    name = _name;
    addr = "";
}
Stu::Stu(int _num, string _name, string _addr)
{
    num = _num;
    name = _name;
    addr = _addr;
```

```

}
void Stu::View()
{
    cout<<"번호:"<<num<<" 이름:"<<name<<endl;
    if(addr != "")
    {
        cout<<"주소:"<<addr<<endl;
    }
}

```

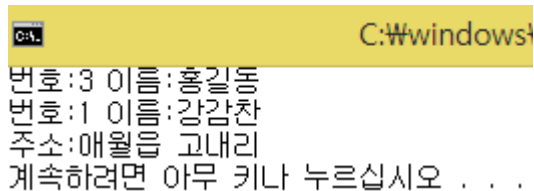
Program.cpp

```

#include "Stu.h"
int main()
{
    Stu *s1 = new Stu(3,"홍길동");
    Stu *s2 = new Stu(1,"강감찬","애월읍 고내리");
    s1->View();
    s2->View();
    delete s2;
    delete s1;
    return 0;
}

```

[예제 2.5] 생성자 중복 정의



```

C:\windows
번호:3 이름:홍길동
번호:1 이름:강감찬
주소:애월읍 고내리
계속하려면 아무 키나 누르십시오 . . .

```

[그림 2.3] 예제 2.5 실행 화면

[그림 2.3]을 보시면 생성자를 중복 정의하였을 때 사용자가 전달한 인수에 적절한 생성자 메서드가 호출되는 것을 확인할 수 있습니다.

매개 변수가 있는 생성자 중에서 다른 개체를 인자로 전달받아 복사된 개체를 생성하는 복사 생성자가 있습니다. 개발자가 복사 생성자를 정의하지 않으면 컴파일러는 디폴트 복사 생성자를 만들어줍니다. 디폴트 복사 생성자에서는 입력 인자로 전달된 개체의 메모리를 덤프하여 생성하는 개체의 메모리에 복사하는 작업을 수행합니다. 즉 새로 생성되는 개체는 입력 개체와 같은 값들을 갖는 개체가 생성되는 것입니다.

개발자가 복사 생성자를 정의하려면 입력 인자의 형식은 `const` 클래스명 `&`로 합니다. 입력 인자로 전달된 개체정보를 변경하지 말아야 하므로 `const`로 지정한 것입니다. 그리고 입력 인자로 전달된 개체 자체가 전달되어야 하므로 레퍼런스 변수로 받습니다.

디폴트 복사 생성자를 제공하기 때문에 개발자가 디폴트 생성자를 정의해야 하는 것은 아닙니다. 하지만 개체 내부에 동적으로 다른 개체를 가지고 있을 때는 복사 생성자를 정의하는 것이 안전할 수 있습니다. 디폴트 복사 생성자는 단순히 메모리 덤프하기 때문에 내부에 동적으로 다른 개체를 갖고 있을 경우에 복사 생성자에서는 단순히 해당 개체의 메모리 주소만 복사하기 때문에 두 개의 개체가 내부에 같은 개체를 참조하게 됩니다. 만약 하나의 개체에서 내부의 개체를 소멸하면 다른 개체에서는 이미 소멸한 개체를 참조하는 것이 되어 심각한 버그를 갖게 됩니다.

Stu.h

```
#pragma once
#include <iostream>
#include <string>
using namespace std;
class Stu
{
    string name;
public:
    Stu(string _name);
    Stu(const Stu &stu);
    void View();
};
```

Stu.cpp

```
#include "Stu.h"

Stu::Stu(string _name)
{
    name = _name;
}

Stu::Stu(const Stu &stu)
{
    cout<<"복사 생성자 수행됨"<<endl;
    name = stu.name;
}

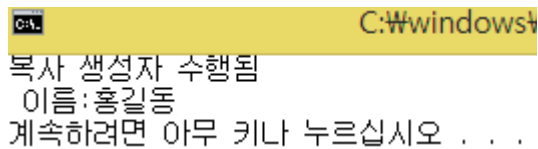
void Stu::View()
{
    cout<<" 이름:"<<name<<endl;
}
```

Program.cpp

```
#include "Stu.h"

int main()
{
    Stu s1("홍길동");
    Stu s2(s1);
    s2.View();
    return 0;
}
```

[예제 2.6] 복사 생성자



```
C:\windows\st
복사 생성자 수행됨
이름:홍길동
계속하려면 아무 키나 누르십시오 . . .
```

[그림 2.4] 예제 2.6 실행 화면

2.3.3 소멸자

C++은 Java나 C#과 달리 플랫폼에서 개체들을 관리(Managed)하지 않습니다. 물론, 여기서 얘기하는 C++은 Native 기반의 C++을 얘기하는 것입니다. 이러한 이유로 C++에서는 생성한 개체의 소멸에 관한 책임을 개발자가 져야 합니다.

소멸자도 생성자처럼 직접 호출하지 않고 delete 연산자를 사용합니다. 또한 소멸자는 생성자와 다르게 중복 정의할 수 없습니다. 소멸자도 개발자가 정의하지 않으면 컴파일러에 의해 디폴트 소멸자가 만들어집니다. 하지만 개체 내부에서 동적으로 다른 개체를 생성하였다면 소멸자를 정의하여 동적으로 만든 개체를 소멸하시기 바랍니다.

Head.h

```
#pragma once
#include <iostream>
using namespace std;
class Head
{
public:
    Head();
    ~Head();
};
```

Head.cpp

```
#include "Head.h"
Head::Head(void)
{
    cout<<"Head 생성자"<<endl;
}
Head::~Head(void)
{
    cout<<"Head 소멸자"<<endl;
}
```

Animal.h

```
#pragma once
#include "Head.h"
class Animal
{
    Head *head;
public:
    Animal(void);
    ~Animal(void);
};
```

Animal.cpp

```
#include "Animal.h"
Animal::Animal(void)
{
    cout<<"Anmail 생성자"<<endl;
    head = new Head();
}
Animal::~~Animal(void)
{
    delete head;
    cout<<"Animal 소멸자"<<endl;
}
```

[예제 2.7] 내부에 동적으로 개체를 생성하는 형식의 소멸자 정의

2.3.4 클래스의 멤버(정적 멤버)

캡슐화된 멤버의 종류를 나누는 기준은 여러 기준이 있을 수 있습니다. 그중에 하나가 해당 멤버가 개체의 멤버인지 혹은 형식의 멤버인지로 구분을 할 수가 있습니다. 이러한 기준으로 구분할 때 형식의 멤버를 정적(static) 멤버라 부르고 개체의 멤버를 비 정적 멤버라 부릅니다.

C++에서 정적 멤버는 형식 정의 내에서 해당 멤버를 static 키워드를 붙여 명시해야 합니다. static 키워드가 붙여 명시된 정적 멤버들은 개체에 종속적인 멤버가 아닌 형식에 종속적인 멤버가 됩니다. 예를 들어, 학생을 생성할 때 학생의 일련번호를 차례대로 부여한다고 할 때 학생의 일련번호는 각각의 학생마다 별도로 유지가 되어야 할 것입니다. 하지만 이번에 생성할 학생에게 어떠한 일련번호를 부여할 것인지는 각각의 학생 개체에 보관하는 것이 아니라 학생 형식 스코프 내에 유일해야 할 것입니다.

정적 멤버는 형식 정의문에 멤버를 선언할 때 static 키워드를 명시합니다. 특히 정적 멤버 필드는 반드시 소스 파일에 멤버 선언을 해 주어야 합니다. 그리고 형식 내부에 선언한 멤버 필드를 소스 코드에 작성할 때는 static 키워드를 생략합니다.

그리고 정적 멤버 메서드에서는 개체와 상관없기 때문에 비 정적 멤버를 사용할 수 없습니다.

```
Stu.h

#pragma once

class Stu
{
    int num;
    static int last_num; //정적 멤버 필드
public:
    Stu(void);
    int GetNum();
    static int GetLastNum(); //정적 멤버 메서드
};
```

Stu.cpp

```
#include "Stu.h"
int Stu::last_num; //static 멤버 필드는 멤버 필드 선언을 해야 함
Stu::Stu(void)
{
    last_num++;
    num = last_num;
}
int Stu::GetNum()
{
    return num;
}
int Stu::GetLastNum() //static 멤버 메서드 구현 정의에서는 static 키워드 사용 안 함
{
    return last_num;
}
```

Program.cpp - 사용 예

```
#include <iostream>
using namespace std;
#include "Stu.h"
int main()
{
    cout<<"현재 학생 수:"<<Stu::GetLastNum()<<endl;
    Stu *stu = new Stu();
    cout<<"학생번호:"<<stu->GetNum()<<endl;
    cout<<"현재 학생 수:"<<Stu::GetLastNum()<<endl;
    Stu *stu2 = new Stu();
    cout<<"학생번호:"<<stu->GetNum()<<endl;

    delete stu;
    delete stu2;
    return 0;
}
```

[예제 2.8] 정적 멤버

2.3.5 상수화 멤버

C++에서는 멤버 필드 중에 값을 변경하지 못하는 상수화 멤버 필드와 메서드 내에서 개체의 상태를 변경하지 못하는 상수화 메서드를 지정할 수 있습니다.

상수화 멤버 필드는 선언문 앞에 `const` 키워드를 붙여주고 상수화 멤버 메서드는 입력 인자를 명시하는 괄호 뒤에 `const` 키워드를 붙여줍니다.

예를 들어, 학생 생성 시에 학번을 부여하고 이후에는 학번을 변경하지 못하게 하고자 한다면 상수화 멤버 필드로 사용할 수 있습니다. 상수화 멤버 필드는 생성자 정의문에서 초기화 기법을 사용하여 값을 지정해야 합니다. 초기화 기법은 생성자 메서드의 입력 인자를 명시하는 함수 뒤에 콜론(:)을 붙인 후에 초기화할 멤버 이름을 명시하고 괄호에 초기화할 구문을 작성하면 됩니다.

```
Stu::Stu(int num):num(num)
{
    ...중략...
}
```

그리고 학생의 번호를 반환하는 함수처럼 개체의 상태를 확인하는 메서드들은 개체의 상태를 바꾸지 않기 때문에 상수화 메서드로 정의하는 것이 높은 데이터 신뢰성을 추구할 수 있습니다.

참고로 상수화 멤버 메서드에서 다른 멤버 메서드(상수화로 지정하지 않은 메서드)를 호출하는 것은 데이터 신뢰성에 문제가 되어 컴파일 오류가 발생합니다.

Stu.h

```
#pragma once
class Stu
{
    const int num;        //상수화 멤버 필드
    int iq;
public:
    Stu(int num);
    void Study(int tcnt);
    int GetIq()const;      //상수화 멤버 메서드
    int GetNum()const;     //상수화 멤버 메서드
};
```

Stu.cpp

```
#include "Stu.h"
Stu::Stu(int num):num(num) //초기화
{
    iq = 100;
}
void Stu::Study(int tcnt)
{
    iq += tcnt;
}
int Stu::GetIq()const //상수화 멤버 메서드
{
    return iq;
}
int Stu::GetNum()const //상수화 멤버 메서드
{
    return num;
}
```

[예제 2.9] 상수화 멤버

2.2.6 특별한 정적 멤버 this

모든 클래스에는 컴파일러에 의해 자동으로 캡슐화하는 정적 멤버 this가 있습니다. 개발자는 멤버 this를 선언할 수 없지만 해당 클래스 스코프 내에서 사용할 수 있습니다. 개체의 메서드를 호출하면 컴파일러는 개체의 주소를 this에 설정하는 코드를 작성해 주어 메서드 내에서 어떠한 개체의 멤버를 사용해야 하는지 알 수 있는 것입니다.

또한 지역 변수명과 멤버 필드명이 같을 때도 this 키워드를 명시하면 개체의 멤버를 사용할 수 있습니다. 물론, 지역 변수명과 다른 멤버 필드를 사용할 때는 this 키워드를 붙이지 않아도 컴파일러가 자동으로 this 키워드를 사용하는 코드로 전개해 주기 때문에 개발자가 this 키워드를 붙이지 않아도 됩니다. 참고로 전역 변수명과 멤버 필드명, 지역 변수명이 같을 때 전역 변수를 사용할 때는 스코프 연산자(::)를 명시하여 사용하면 전역 변수를 사용할 수 있습니다.

```
int a;
void Stu::Stub(int a)
{
    a = 3; //지역 변수 a 사용
    this->a = 4; //멤버 필드 a 사용
    ::a = 5; //전역 변수 a 사용
}
bool Stu::IsEqual(int num)const
{
    return this->num == num;
}
void Stu::View()const
{
    cout<<"번호:"<<num<<" 이름:"<<name<<endl; //멤버 필드 num, name 사용
}
```

2.3 신뢰성을 높이는 캡슐화

C++언어에서는 접근 지정자를 통해 형식 외부에서 접근하지 못하는 멤버를 지정할 수 있는 문법을 제공하고 있습니다. 여러분이 클래스를 정의할 때 다음의 규칙을 준수하면 비교적 데이터 신뢰성이 높은 형식을 구현할 수 있을 것입니다.

- 멤버 필드는 `private`으로 접근 지정하라.
- 멤버 필드의 값을 외부에서 확인해야 한다면 상수화 멤버 메서드를 제공하라.
`int Stu::GetNum()const;`
- 멤버 필드의 값을 외부에서 설정하는 것을 제공해야 한다면 메서드를 제공하라.
`void Stu::SetIq(int iq)`

여러분이 생각할 때 형식 외부에서 멤버 필드의 값을 확인할 수 있어야 하고 값을 설정할 수 있어야 한다고 생각할 때 멤버 필드의 접근 지정을 `public`으로 설정하는 것이 맞다고 생각할 수 있습니다. 하지만 멤버 필드의 값의 유효 범위가 있다면 `Set`메서드를 통해 전달받은 값이 유효 범위를 벗어나는지 점검하여 설정함으로써 신뢰성을 높일 수 있습니다. 그리고 현재 시나리오에서 특정 멤버 필드의 값의 유효 범위가 지정하지 않았다고 하더라도 미래에 바뀔 수도 있기 때문에 멤버 필드의 가시성은 `private`으로 지정하세요. 필요하다면 `Get` 메서드와 `Set` 메서드를 제공하시기 바랍니다.

3. 일반화 관계(상속)

3.1 일반화 관계

일반화 관계는 음악가와 피아니스트처럼 “피아니스트는 음악가이다.”라는 논리적 관계를 형성하는 관계를 말합니다. C++에서는 이와 같은 관계를 효과적으로 사용할 수 있도록 파생에 관한 문법을 제공하고 있으며 이러한 특징은 OOP의 상속에 속합니다.



[그림 3.1] 일반화 관계

3.1.1 일반화 관계와 파생

C++에서 일반화 관계를 표현할 때 파생에 관련된 문법을 이용합니다. 파생을 표현할 때는 파생 클래스에서 어느 클래스를 기반 클래스로 할 것인지를 다음과 같이 명시합니다.

```
class Derived : public Base
{
};
```

파생을 표현할 때 기반 클래스의 접근 지정된 것을 파생 형식에서 그대로 계승하고자 할 때 public 키워드를 명시합니다. 물론, 기반 클래스의 private으로 지정된 멤버들도 파생 형식에 포함되지만 파생 형식에서 접근할 수 없습니다.

Musician.h - base 클래스 헤더

```
#pragma once
#include <iostream>
using namespace std;
class Musician
{
public:
    void Play();
};
```

Musician.cpp

```
#include "Musician.h"
void Musician::Play()
{
    cout<<" 연주하다."<<endl;
}
```

Pianist.h - 파생 클래스 헤더

```
#pragma once
#include "Musician.h"
class Pianist : public Musician
{
public:
    void Tuning();
};
```

Pianist.cpp

```
#include "Pianist.h"
void Pianist::Tuning()
{
    cout<<" 조율하다."<<endl;
}
```

```
#include "Pianist.h"
int main()
{
    Pianist *pianist = new Pianist();
    pianist->Play();
    pianist->Tuning();
    delete pianist;
    return 0;
}
```

[예제 3.1] 상속

3.1.2 파생 개체 생성 과정 및 초기화

파생 개체가 생성될 때는 먼저 기반 클래스 부분이 형성된 후에 파생 클래스 부분이 형성됩니다. 즉, 기반 클래스의 생성자 메서드가 수행된 후에 파생 클래스의 생성자 메서드가 수행됩니다. 그리고 파생 개체가 소멸될 때는 역으로 파생 클래스의 소멸자 메서드가 수행되고 나서 기반 클래스의 소멸자 메서드가 수행됩니다.

만약, 기반 클래스에 입력 매개 변수가 없는 생성자(기본 생성자)가 없고 매개 변수가 있는 생성자만 정의하면 파생 개체가 생성될 때 어떻게 될까요?

파생 개체를 생성하기 위해선 기반 클래스의 생성자 메서드 부분이 수행되어야 하는데, 인자를 어떤 값으로 전달할 것인지 컴파일러가 결정하지 못합니다. 이 경우에 파생된 클래스의 소스 코드에서 멤버 필드 초기화하는 방식처럼 초기화 작업을 해야 합니다. 그렇지 않으면 오류가 발생합니다.

```
Pianist::Pianist(const char *_name):Musician(_name)
{
}
```

그리고 기반 형식에서 멤버를 캡슐화할 때 접근 지정을 protected로 명시하면 기반 형식 내부뿐만 아니라 파생 형식에서도 접근할 수 있습니다. 물론, 다른 형식에서는 접근하지 못하므로 정보 은닉이 필요한 수준에 맞게 접근 지정할 수 있습니다.

Musician.h

```
#pragma once
#include <iostream>
#include <string>
using namespace std;
class Musician
{
    const string name;
public:
    Musician(string _name);
    virtual ~Musician(void);
protected:
    string GetName()const;//파생 클래스에서 가시성이 있음
};
```

Musicain.cpp

```
#include "Musician.h"
Musician::Musician(string _name):name(_name)
{
    cout<<name<<"Musician 생성자 메서드"<<endl;
}
Musician::~~Musician(void)
{
    cout<<name<<"Musician 소멸자 메서드"<<endl;
}
string Musician::GetName()const
{
    return name;
}
```

Pianist.h

```
#pragma once
#include "Musician.h"
class Pianist : public Musician
{
public:
    Pianist(string _name);
    ~Pianist();
    void Tuning();
};
```

Pianist.cpp

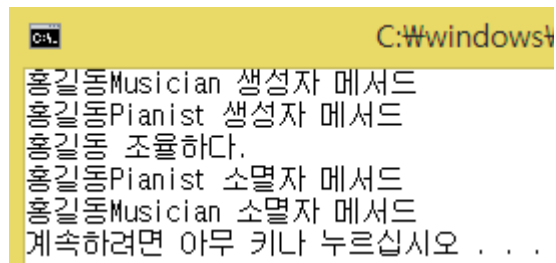
```
#include "Pianist.h"
Pianist::Pianist(string _name):Musician(_name)
{
    cout<<GetName()<<"Pianist 생성자 메서드"<<endl;
}
Pianist::~Pianist()
{
    cout<<GetName()<<"Pianist 소멸자 메서드"<<endl;
}
void Pianist::Tuning()
{
    cout<<GetName()<<"조율하다."<<endl;
}
```

Program.cpp

```
#include "Pianist.h"

int main()
{
    Pianist *pianist = new Pianist("홍길동");
    pianist->Tuning();
    delete pianist;
    return 0;
}
```

[예제 3.2] 기반 클래스에 기본 생성자가 없을 때 파생 클래스 생성자에서 초기화



```
C:\Windows\system32\cmd.exe
홍길동Musician 생성자 메서드
홍길동Pianist 생성자 메서드
홍길동 조율하다.
홍길동Pianist 소멸자 메서드
홍길동Musician 소멸자 메서드
계속하려면 아무 키나 누르십시오 . . .
```

[그림 3.2] 예제 3.2 실행 화면

[그림 3.2] 실행 화면을 보시면 파생 개체의 생성은 기반 클래스 → 파생 클래스 순으로 생성되고 소멸은 역순으로 소멸하는 것을 확인하실 수 있습니다.

3.2 무효화

파생을 이용해서 일반화 관계를 형성했을 때 파생 클래스에서 기반 클래스에 정의한 이름과 같은 이름으로 메서드를 만들면 기반 클래스에 정의한 메서드는 무효화가 됩니다.

기반 클래스에 있는 멤버 메서드들 중에 구체적인 구현을 다르게 하고자 한다면 무효화를 이용하면 가능합니다. 만약 파생 형식에서 기반 형식의 메서드와 같은 이름의 메서드를 정의하면 기반 형식의 같은 이름의 모든 메서드는 무효화됩니다.

Programmer.h

```
#pragma once
#include <iostream>
using namespace std;
class Programmer
{
public:
    void Programming();
    void Programming(int tcnt);
};
```

Programmer.cpp

```
#include "Programmer.h"
void Programmer::Programming()
{
    cout<<"생각하면서 코딩을 한다."<<endl;
}
void Programmer::Programming(int tcnt)
{
    cout<<tcnt<<"시간 생각하면서 코딩을 한다."<<endl;
}
```

EHProgrammer.h

```
#pragma once
#include "Programmer.h"
#include <string>
using std::string;
class EHProgrammer:
    public Programmer
{
public:
    void Programming(string title);
};
```

EHProgrammer.cpp

```
#include "EHProgrammer.h"
void EHProgrammer::Programming(string title)
{
    cout<<"프로젝트명:"<<title<<endl;
    cout<<"생각한 것을 문서화하고 이를 보면서 코딩을 한다."<<endl;
}
```

[예제 3.3] 무효화

```
1  #include "EHProgrammer.h"
2  int main()
3  {
4      EHProgrammer *ep = new EHProgrammer();
5      ep->Programming(3);
6      delete ep;
7      return 0;
8  }
```

✖ 1 error C2664: 'EHProgrammer::Programming': 매개 변수 1을(를) 'int'에서 'std::string'(으)로 변환할 수 없습니다.

[그림 3.3] 예제 3.3 컴파일 오류 화면

3.2.1 무효화 된 멤버 사용하기

파생 형식에서 기반 형식의 무효화 된 멤버를 사용할 때는 기반 클래스 명과 스코프 연산자를 붙여 메서드를 호출하면 무효화 된 멤버를 사용할 수 있습니다.

EHProgrammer.h

```
#pragma once
#include "Programmer.h"
class EHProgrammer:
    public Programmer
{
public:
    void Programming(int tcnt);
};
```

EHProgrammer.cpp

```
#include "EHProgrammer.h"
void EHProgrammer::Programming(int tcnt)
{
    cout<<"생각을 하고 이를 문서화한다."<<endl;
    Programmer::Programming(tcnt); //무효화 된 멤버 사용
}
```

Program.cpp

```
#include "EHProgrammer.h"
void main()
{
    EHProgrammer *ehclub = new EHProgrammer();
    ehclub->Programming(3);
    delete ehclub;
}
```

[예제 3.4] 무효화 된 멤버 사용하기

3.3 파생 시에 액세스 지정

지금까지 파생을 표현할 때 기반 클래스의 접근 수준을 변경없이 파생하였습니다. 이 외에도 `protected` 상속과 `private` 상속이 있는데 어떠한 차이가 있고 어떠한 경우에 사용될 수 있는지 알아보시다.

파생할 때 기반 클래스 명 앞에 붙는 접근 지정자는 기반 클래스의 멤버들의 가시성이 허용되는 수준 중에 가장 넓은 가시성 수준을 지정합니다.

	public 상속	protected 상속	private 상속
public	public	protected	private
protected	protected	protected	private
private	가시성 없음	가시성 없음	가시성 없음

`protected` 상속이나 `private` 상속은 기존에 제공되던 라이브러리에 제공되는 클래스를 기반으로 파생 클래스를 만들 때 많이 사용됩니다. `protected` 상속으로 지정하면 파생 클래스에서는 기반 클래스에 `public`과 `protected`로 지정한 멤버를 사용할 수 있습니다. 하지만 파생 클래스를 사용하는 곳에서는 기반 클래스의 `public` 멤버에 접근할 수 없고 파생 클래스에서 `public`으로 지정한 멤버만 접근할 수 있게 됩니다. 그리고 `private`으로 상속을 지정하면 파생 클래스를 기반으로 다시 파생 클래스를 만들 때 라이브러리에 있는 기반 클래스의 `public` 멤버를 사용하지 못합니다.

다음은 IntArr 클래스를 private 상속으로 파생 클래스 EhStack을 구현한 예제입니다.

IntArr.h

```
#pragma once
#define MAX_ELEMENTS    100
class IntArr
{
    int base[MAX_ELEMENTS];
    int cnt;
public:
    IntArr(void);
    bool PushBack(int val);
    int GetFront()const;
    bool PopFront();
    int GetBack()const;
    bool PopBack();
    int GetAt(int index)const;
    bool InsertAt(int index,int val);
protected:
    bool IsFull()const;
    bool IsEmpty()const;
    bool AvailIndex(int index)const;
};
```

IntArr.cpp

```
#include "IntArr.h"
#include <memory.h>

IntArr::IntArr(void)
{
    cnt = 0;
}

bool IntArr::PushBack(int val)
{
    if(IsFull()){ return false; }
    base[cnt] = val;
    cnt++;
    return true;
}

int IntArr::GetFront()const
{
    return base[0];
}

bool IntArr::PopFront()
{
    if(IsEmpty()){ return false; }
    cnt--;
    memcpy(base,base+1,sizeof(int)*cnt);
    return true;
}

int IntArr::GetBack()const
{
    return base[cnt-1];
}
```

```

bool IntArr::PopBack()
{
    if(IsEmpty()){ return false; }
    cnt--;
    return true;
}

int IntArr::GetAt(int index)const
{
    return base[index];
}

bool IntArr::InsertAt(int index,int val)
{
    if(AvailIndex(index)==false){ return false; }
    base[index] = val;
    return true;
}

bool IntArr::IsFull()const
{
    return cnt==MAX_ELEMENTS;
}

bool IntArr::IsEmpty()const
{
    return cnt==0;
}

bool IntArr::AvailIndex(int index)const
{
    return (index>=0) && (index < MAX_ELEMENTS);
}

```

EhStack.h

```
#pragma once
#include "IntArr.h"
class EhStack : private IntArr
{
public:
    bool Push(int val);
    int Pop();
    bool IsFull()const;
    bool IsEmpty()const;
};
```

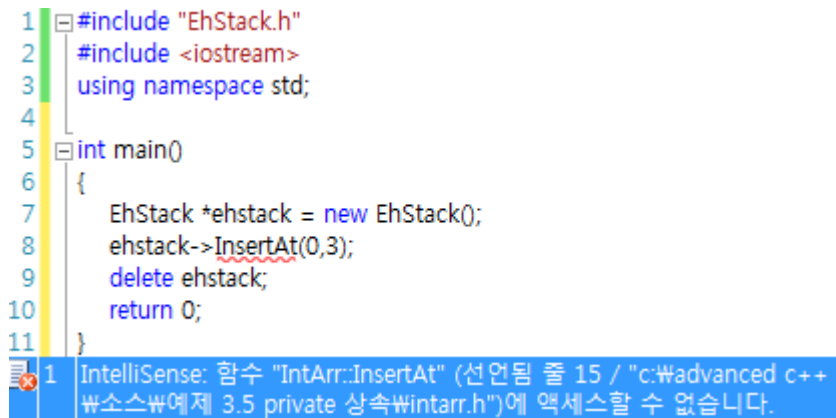
EhStack.cpp

```
#include "EhStack.h"
bool EhStack::Push(int val)
{
    return PushBack(val);
}
int EhStack::Pop()
{
    int val = GetBack();
    PopBack();
    return val;
}
bool EhStack::IsFull()const
{
    return IntArr::IsFull();
}
bool EhStack::IsEmpty()const{ return IntArr::IsEmpty(); }
```

Program.cpp

```
#include "EhStack.h"
#include <iostream>
using namespace std;
int main()
{
    EhStack *ehstack = new EhStack();
    ehstack->Push(3);
    ehstack->Push(7);
    cout<<ehstack->Pop()<<endl;
    cout<<ehstack->Pop()<<endl;
    delete ehstack;
    return 0;
}
```

[예제 3.5] private 상속



```
1 #include "EhStack.h"
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     EhStack *ehstack = new EhStack();
8     ehstack->InsertAt(0,3);
9     delete ehstack;
10    return 0;
11 }
```

1 IntelliSense: 함수 "IntArr::InsertAt" (선언됨 줄 15 / "c:\wadvanced c++\소스\예제 3.5 private 상속\wintarr.h")에 액세스할 수 없습니다.

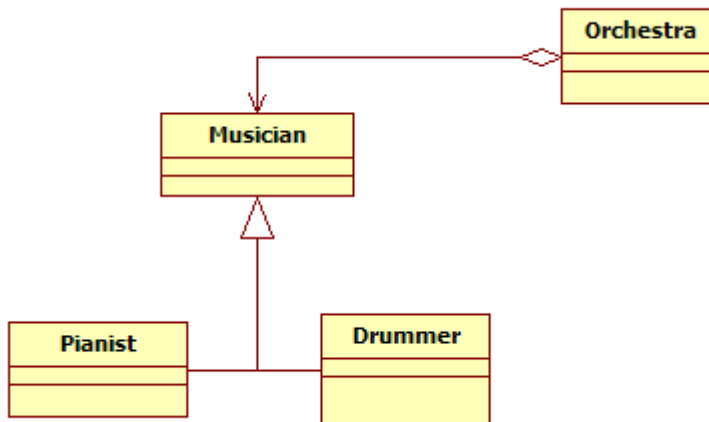
[그림 3.4] private 상속 예제에서 기반 형식의 public 멤버 접근 시 오류 화면

4. 다형성

4.1 개체의 다형성

파생을 통해 얻을 수 있는 이점 중의 하나는 기반 클래스 형식의 포인터 변수로 파생된 개체를 관리할 수 있다는 것입니다. 오케스트라를 형성하는 여러 종류의 음악가를 기반 클래스인 음악가에서 파생시켜 피아니스트, 드러머 등을 정의함으로써 하나의 컬렉션에서 관리하는 것은 참으로 매력적인 일이 아닐 수 없습니다.

이처럼 기반 클래스 형식의 포인터 변수로 파생된 개체를 관리할 수 있기 때문에 특정 변수가 관리하는 개체는 다양한 형태의 개체를 관리할 수 있게 됩니다. 이때 기반 클래스 형식의 포인터 변수로 파생된 개체 인스턴스를 대입할 때 묵시적으로 상향 캐스팅이 됩니다. 그리고, 이러한 특징은 다형성의 일부가 됩니다.



[그림 4.1] 예제 4.1의 클래스 다이어그램

[예제 4.1]에서는 음악가들로 구성된 오케스트라를 표현하였습니다. 오케스트라 형식은 음악가를 단원으로 합류하는 Join 메서드를 구현하였는데 입력 인자 형식을 Musician 포인터 형식으로 지정하였습니다. 그리고 오케스트라를 사용하는 곳에서 피아니스트 개체와 드러머 개체를 생성하여 오케스트라 개체의 Join 메서드를 호출하여 각 개체를 전달합니다. 이는 C++에서 상향 캐스팅을 묵시적으로 제공하기 때문에 가능한 것입니다.

Musician.h

```
#pragma once
#include <iostream>
using namespace std;
class Musician
{
    const int mnum;
public:
    Musician(int _mnum);
    void Greeting()const;
};
```

Musician.cpp

```
#include "Musician.h"
Musician::Musician(int _mnum):mnum(_mnum)
{
}
void Musician::Greeting()const
{
    cout<<mnum<<"번 음악가가 인사합니다."<<endl;
}
```

Pianist.h

```
#pragma once
#include "Musician.h"
class Pianist :
    public Musician
{
public:
    Pianist(int mnum);
};
```

Pianist.cpp

```
#include "Pianist.h"
Pianist::Pianist(int mnum):Musician(mnum)
{
}
```

Drummer.h

```
#pragma once
#include "musician.h"
class Drummer :
    public Musician
{
public:
    Drummer(int mnum);
};
```

Drummer.cpp

```
#include "Drummer.h"
Drummer::Drummer(int mnum):Musician(mnum)
{
}
```

Orchestra.h

```
#pragma once
#include "Musician.h"

class Orchestra
{
    Musician **base;
    const int max_members;
    int mcnt;
public:
    Orchestra(int max_members);
    ~Orchestra(void);
    bool Join(Musician *musician);
    void Greeting()const;
};
```

Orchestra.cpp

```
#include "Orchestra.h"

Orchestra::Orchestra(int max_members):max_members(max_members)
{
    base = new Musician *[max_members];
    mcnt = 0;
}

Orchestra::~Orchestra(void)
{
    delete[] base;
}
```

```

bool Orchestra::Join(Musician *musician)
{
    if(mcnt < max_members)
    {
        base[mcnt] = musician;
        mcnt++;
        return true;
    }
    return false;
}

void Orchestra::Greeting()const
{
    cout<<"최대 인원수:"<<max_members<<" 현재 인원수:"<<mcnt<<endl;
    for(int i = 0; i<mcnt; i++) {    base[i]->Greeting();    }
}

```

Program.cpp

```

#include "Orchestra.h"

int main()
{
    Orchestra *orche = new Orchestra(50);
    Pianist *pianist = new Pianist(1);
    Drummer *drummer = new Drummer(2);
    orche->Join(pianist); //Musician * 형식이 기대하는 곳에 Pianist *형식 전달
    orche->Join(drummer); //Musician * 형식이 기대하는 곳에 Drummer *형식 전달
    orche->Greeting();
    delete pianist;
    delete drummer;
    delete orche;
    return 0;
}

```

[예제 4.1] 개체의 다형성

4.2 메서드의 다형성

기반 클래스 형식 포인터 변수로 파생된 개체를 관리를 할 수 있다는 것은 매우 매력적입니다. 하지만 파생 형식의 구체적 행위가 다르다면 어떻게 해야 할까요? C++에서는 메서드의 다형성을 제공하여 기반 형식에서 제공한 알고리즘과 다르게 동작하는 메서드를 파생 형식에 재정의(override)할 수 있는 문법을 제공하고 있습니다.

4.2.1 가상 메서드

만약, 오케스트라의 모든 음악가가 같은 연주를 한다면 어떤 느낌이 들까요? 아마도 각 음악가가 연주하는 모습이 다르지만 각각의 연주가 조화를 이루기 때문에 더욱 더 장엄하고 아름답게 들리는 것으로 생각됩니다. 프로그램에서도 마찬가지입니다.

예를 들어, 기반 클래스 Musician에서 Play 메서드를 정의하고 파생된 Pianist에서 Play 메서드를 재정의하면 Pianist 개체는 어떻게 연주를 할까요? 아무런 명시도 하지 않으면 Pianist 개체를 관리하는 변수의 형식에 따라 연주하게 됩니다.

하지만 기반 형식에서 메서드를 선언할 때 virtual 키워드를 명시하면 개체를 참조하는 변수의 형식이 아닌 실제 개체 형식에 따라 동작하게 됩니다.

이처럼 기반 형식에서 virtual로 명시한 멤버 메서드를 가상 메서드라 얘기합니다. 그리고 가상 메서드를 파생 형식에서 재정의하면 기반 형식 포인터 변수로 해당 메서드를 호출하면 실제 개체 형식에 따라 동작하기 때문에 메서드의 다형성이라고 부릅니다.

[예제 4.2]를 실행하면 가상 메서드가 아닌 Greeting은 호출할 때 사용하는 변수 형식에 따라 동작합니다. 하지만 가상 메서드를 호출할 때는 참조하는 변수에 관계없이 실제 개체의 메서드가 호출되는 것을 알 수 있습니다. 즉, 가상 메서드를 호출할 때는 개체가 참조하는 형식에 따라 실제 호출하는 메서드가 달라지므로 메서드의 다형성이라 부르는 것입니다.

Program.cpp

```
#include <iostream>
using namespace std;
class Musician
{
public:
    virtual void Play(){cout<<"칼라라"<<endl;}
    void Greeting(){ cout<<"안녕하세요."<<endl;}
};
class Pianist:public Musician
{
public:
    void Play(){cout<<"딩동댕"<<endl;}
    void Greeting(){cout<<"안녕"<<endl;}
};
int main()
{
    Musician *musician = new Pianist();
    Pianist *pianist = new Pianist();
    musician->Play(); //가상 메서드이므로 실제 개체인 Pianist의 Play 호출
    pianist->Play()
    musician->Greeting(); //변수 형식에 따라 Musician의 Greeting 호출
    pianist->Greeting();
    delete pianist;
    delete musician;
    return 0;
}
```

[예제 4.2] 가상 메서드

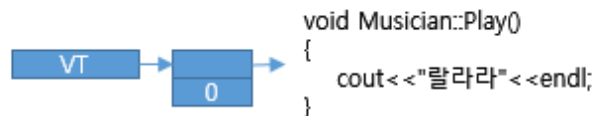
위와 같이 기반 클래스에서 특정 메서드에 virtual 키워드를 명시한 메서드를 가상 메서드라 얘기합니다. 이와 같은 가상 메서드를 하나라도 존재하는 개체가 생성될 때에는 멤버필드 외에 가상 함수들의 코드 메모리 주소를 보관하는 테이블이 동적으로 생성되고 이 테이블의 위치 정보를 개체는 갖게 됩니다. 그리고 파생된 클래스에서 파생 개체의 생성자를 수행하면서 해당 함수가 정의된 코드 주소로 변경하는 작업을 수행하게 됩니다. 그리고, 가상 메서드를 호출하는 부분은 컴파일러 전개에서 가상 함수 테이블에 있는 코드 주소를 호출하는 구문으로 전개합니다. 이러한 이유로 인해 virtual 키워드가 명시된 메서드는 관리하는 형식이 아닌 실존하는 개체의 메서드가 호출하게 되는 것입니다. (여기에서 메서드는 호출할 때 사용되는 도구를 의미하고 함수는 수행할 코드가 정의된 부분을 의미합니다.)

상속에서 설명했듯이 파생 개체를 생성할 때는 먼저 기반 형식 부분이 만들어 진다고 하였습니다. 컴파일러는 기반 형식에 가상 메서드의 주소를 가상 함수 테이블의 요소에 대입합니다. 만약 파생 형식에서 재정의한다면 파생 형식이 만들어질 때 재정의한 메서드 주소로 변경합니다.

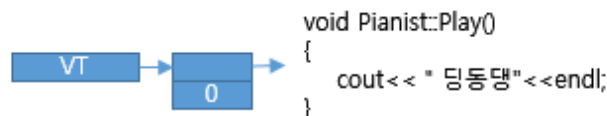
그리고 가상 메서드를 호출할 때는 가상 함수 테이블에 있는 코드 주소를 호출하기 때문에 참조하는 변수 형식에 관계없이 실제 개체의 메서드가 호출되는 것입니다.

```
Musician *musician = new Pianist(1);
```

1. Pianist 생성 과정 중 기반 형식 형성



2. Pianist 생성 과정 중 Pianist 형식 형성 (가상 메서드 Play를 재정의하였기 때문에 변경)



```
musician->Play(); //Pianist::Play 메서드 호출
delete musician;
```

[그림 4.2] 가상 메서드를 재정의한 파생 개체 생성 과정

4.2.2 순수 가상 메서드(추상 메서드)

C++에서는 기반 형식을 정의할 때 특정 행위가 있다고 선언만 하고 정의를 하지 않을 수 있습니다. 이러한 메서드를 순수 가상 메서드라 얘기하며 일반적인 OOP 언어에서 얘기하는 추상 메서드에 해당합니다.

순수 가상 메서드는 형식 정의문에서 가상 메서드 선언문 뒤에 `=0` 를 명시합니다. 이렇게 캡슐화한 순수 가상 메서드는 기반 형식에서 메서드 정의문을 만들 수 없습니다. 그리고 순수 가상 메서드가 있는 형식은 개체를 생성할 수도 없기 때문에 추상 클래스라고 부릅니다.

```
class Musician
{
public:
    virtual void Play()=0;
};
```

```
1  class Musician
2  {
3  public:
4      virtual void Play()=0;
5  };
6  int main()
7  {
8      Musician *musician = new Musician();
9      delete musician;
10     return 0;
11 }
```

IntelliSense: 추상 클래스 형식 "Musician"의 개체를 사용할 수 없습니다.

그림 4.3 추상 클래스 형식 개체를 생성 요청시 오류 화면

추상 클래스를 기반으로 파생한 형식에서 기반 형식의 순수 가상 메서드를 재정의하지 않으면 해당 형식도 추상 클래스가 되어 개체를 생성할 수 없습니다. 따라서 개체를 생성할 수 있는 클래스를 만들기 위해서는 기반 형식의 순수 가상 메서드를 재정의해야 합니다.

특히 모든 멤버가 순수 가상 메서드이면서 접근 지정이 public으로 지정한 형식을 인터페이스라 얘기할 수 있습니다. Java나 C#에서는 인터페이스를 형식으로 제공하지만 C++에서는 제공하지 않기 때문에 위와 같은 방법을 사용합니다.

```
Program.cpp
#include <iostream>
using namespace std;
struct IStudy
{
    virtual void Study()=0;
};
class Stu:public IStudy
{
public:
    void Study(){ cout<<"공부하다."<<endl; }
};
int main()
{
    IStudy *istudy = new Stu();
    istudy->Study();
    delete istudy;
    return 0;
}
```

[예제 4.3] 인터페이스

4.3 하향 캐스팅

다형성은 캡슐화와 상속을 보다 효과적이고 현실 세계에 근접하게 표현할 수 있게 해주는 매력적인 특징입니다. 하지만 기반 클래스 형식 포인터 변수로 파생된 개체를 관리하는 것은 치명적인 단점을 가져오게 합니다. 만약, 기반 클래스 형식에서는 약속할 필요가 없는 메서드가 파생 클래스 형식에 있을 경우 해당 메서드의 접근 수준을 public으로 제공하여도 접근하지 못하게 됩니다. 이러한 약점을 보완하기 위해 많은 OOP언어에서는 런타임에 파생 개체 형식으로 캐스팅하는 방법을 제공하고 있으며 이를 하향 캐스팅이라 합니다.

C++언어에서는 dynamic_cast를 통해 하향 캐스팅을 제공하고 있습니다. dynamic_cast를 사용하면 관리되는 실제 개체의 형식이 캐스팅이 가능하지 않다면 0을 반환하고 가능하면 변환해 줍니다. 예를 들어, 피아니스트에만 튜닝할 수 있는 기능이 있고 기반 형식 포인터로 관리한다면 하향 캐스팅을 통해 피아니스트에 있는 튜닝 기능을 사용할 수 있습니다.

```
Musician *players[2];
players[0] = new Pianist();
players[1] = new Drummer();

for(int i = 0; i<2; i++)
{
    players[i]->Play();
}

Pianist *pianist=0;
for(int i = 0; i<2; i++)
{
    pianist = dynamic_cast<Pianist *>(players[i]); //하향 캐스팅
    if(pianist != 0)
    {
        pianist->Tuning();
    }
}
```

5. 기타 문법

5.1 연산자 중복 정의

연산자 중복 정의란 "피연산자 중에 최소 하나 이상이 사용자 정의 형식일 경우에 해당 연산에 대한 기능을 정의하는 것"을 말합니다. (참고로, 포인터 형식은 사용자 정의 형식이 아닙니다.)

C++에서 연산자 중복 정의를 지원하는 이유는 사용자로서 ==와 같은 연산자를 사용하는 것이 isEqual이라는 메서드를 사용하는 것보다 더 직관적일 수 있기 때문입니다. 하지만 사용자가 생각하는 것과 제공자의 의도가 서로 다르다면 오히려 이는 신뢰성이 떨어지고 유지 보수 비용이 늘어나게 되는 요인이 될 수가 있습니다. 이러한 이유로 모든 OOP언어에서 연산자 중복 정의를 문법적으로 지원하는 것은 아닙니다. 그리고 이를 지원하는 언어들도 사용자가 연산자 중복 정의를 할 때 지켜야 하는 수준이 조금씩 다릅니다.

여러분은 연산자 중복 정의 문법을 이용할 때 언어에서 요구하는 문법 수준이 까다롭지 않으면 언어에서 제공하는 유연성을 효과적으로 활용할 필요가 있습니다. 하지만 이에 따르는 모든 신뢰성에 관한 책임을 개발자가 가져야 한다는 점에 유의하시기 바랍니다. 이를 위해 언제나 연산자 중복 정의하는 것이 해당 연산이 상징하는 의미와 객관적으로 보았을 때 일치하는지 고민을 할 필요가 있습니다.

C++언어에서 연산자 중복 정의에서는 피연산자 중에 최소 하나 이상이 사용자 정의 형식이어야 한다는 것과 피연산자의 개수를 지키는 범위에서 대부분 허용하고 있습니다. 기본적인 사항은 다음과 같습니다.

- 피연산자 중에 최소 하나는 사용자 정의 형식이어야 한다.
- 기본적으로 함수 중복 정의의 규칙을 따른다.
- 피연산자의 개수를 바꿀 수 없다.
- 모든 연산에 대해 중복 정의할 수 있는 것은 아니다.
- 연산자 우선 순위를 변경할 수 없다.
- 연산자 중복 정의는 전역 스코프와 클래스 스코프에서 할 수 있다.
- 논리적으로 연산 행위에 맞는 지에 대한 여부에 대한 판단을 컴파일러가 하지 않는다.

5.1.1 전역 연산자 중복 정의

전역에서 연산자 중복 정의를 할 때에는 다음과 같은 포맷을 갖게 됩니다.

[리턴 형식] operator [연산기호] (피 연산자 리스트)

```
{  
    [기능 구현]  
}
```

전역 연산자 중복 정의를 하는 간단한 예를 들어 보기로 할게요.

학생 클래스에 기본 키 역할을 하는 번호 필드를 반환하는 메서드를 이용을 하는 것을 = 연산자를 사용할 수 있도록 만들어 보겠습니다.

```
Stu.h  
  
#pragma once  
#include <iostream>  
#include <string>  
using namespace std;  
  
class Stu  
{  
    const int num;  
    string name;  
public:  
    Stu(int _num,string _name);  
    int GetNum()const;  
};  
  
bool operator == (int num, const Stu &stu);
```

Stu.cpp

```
#include "Stu.h"

Stu::Stu(int _num,string _name):num(_num)
{
    name = _name;
}

int Stu::GetNum()const
{
    return num;
}

bool operator == (int num, const Stu &stu)
{
    return stu.GetNum() == num;
}
```

[예제 5.1] 전역 연산자 중복 정의

이처럼 구현하면 사용하는 곳에서 int형식과 Stu형식을 피 연산자로 하는 == 연산자를 사용하면 전역에 정의한 operator== 메서드가 호출이 됩니다. 이처럼 연산자 중복 정의를 하면 사용자로서는 좀 더 직관적으로 사용할 수 있게 됩니다. 하지만 컴파일러에서는 연산자 중복 정의를 구현한 코드의 논리가 해당 연산에 적절한지에 대한 논리적 부분은 검증하지 않기 때문에 이에 관한 책임은 개발자의 몫입니다.

5.1.2 클래스에 연산자 중복 정의

연산자 중복 정의는 전역과 클래스에서 할 수 있다고 하였습니다. 이번에는 클래스에서 연산자 중복 정의를 하는 방법에 관해서 얘기해 보도록 합니다.

클래스에서 연산자 중복 정의를 할 때에는 클래스 내에 어떠한 연산에 대해서 중복 정의할 것인지 캡슐화하고 구현해야 합니다. 이때, 전역에서 정의하는 것과 다른 점은 피연산자 중에 좌항은 개체 자신으로 약속된다는 점입니다. 따라서 캡슐화할 때 자신의 형식에 대한 피연산자는 입력 매개변수 리스트에서 생략합니다.

```
Stu.h

#pragma once
#include <iostream>
#include <string>
using namespace std;

class Stu
{
    const int num;
    string name;
public:
    Stu(int _num,string _name);
    int GetNum()const;
    bool operator == (int num)const;
};

bool operator == (int num, const Stu &stu);
```

Stu.cpp

```
#include "Stu.h"

Stu::Stu(int _num,string _name):num(_num)
{
    name = _name;
}

int Stu::GetNum()const
{
    return num;
}

bool Stu::operator == (int num)const
{
    return this->num == num;
}

bool operator == (int num, const Stu &stu)
{
    return stu == num;
}
```

[예제 5.2] 클래스 내에 연산자 중복 정의

Stu.cpp 소스 파일에는 클래스 정의에서 캡슐화를 약속한 연산자 중복 정의 함수를 구현해야 할 것입니다. 이 경우에 같은 연산자에 대한 중복 정의를 전역에서도 구현할 경우 위처럼 클래스에 중복 정의한 연산을 사용하여 구현을 하십시오. 그러면 논리적 버그가 있을 때 클래스 내에 중복 정의한 코드만 수정하더라도 자연스럽게 전역에 정의한 것에도 반영됩니다.

위와 같이 교환 법칙이 성립하는 연산을 중복 정의할 때는 클래스와 전역에 모두 정의하여 사용하는 개발자가 유연하게 사용할 수 있게 하세요. 또한, 지금과 같이 == 연산에 대해 중복 정의했다면 사용하는 개발자가 이와 연관된 연산(!=)도 사용 가능할 것이라 유추할 수 있으므로 이에 대해서도 구현해 주는 것이 효과적일 것입니다. 참고로 C#언어에서는 이 같은 경우에 컴파일러가 연관되는 연산에 대해 중복 정의를 같이 하지 않으면 오류를 발생시킵니다. C++언어에서는 컴파일러에서 오류를 발생하지 않으므로 개발자가 신중하게 구현하여야 합니다.

5.2 특수한 연산자 중복 정의 예

C++ 언어에서 연산자 중복 정의를 할 때 연산 종류에 따라 좀 더 세심한 주의를 해야 하는 연산들이 있습니다. 여기에서는 이러한 연산 중에 [] 연산, 묵시적 형 변환 연산을 살펴 볼게요.

5.2.1 [] 연산자 중복 정의

[]연산자는 배열처럼 요소들을 보관하는 컬렉션에서 요소에 접근할 때 사용합니다. 그리고 []연산자는 대입 연산자 좌항에 올 수 있어야 하기 때문에 []연산의 결과는 요소가 갖는 값을 복사한 것이 아닌 요소 자체여야 할 것입니다. 따라서 연산 결과는 요소 레퍼런스를 반환하게 정의해야 할 것입니다.

```
IntArr.h
#pragma once
class IntArr
{
    int *base;
    const int capa;
public:
    IntArr(int _capa);
    ~IntArr(void);
    int &operator[](int index); // 요소(int) 레퍼런스 반환
private:
    void Initailize();
    bool AvailIndex(int index);
};
```


IntArr.cpp

```
#include "IntArr.h"

IntArr::IntArr(int _capa):capa(_capa)
{
    base = new int[capa];
    Initailize();
}

IntArr::~IntArr(void)
{
    delete[] base;
}

int &IntArr::operator[](int index)
{
    if(AvailIndex(index))
    {
        return base[index];
    }
    throw "잘못된 인덱스를 사용하였습니다.";
}

void IntArr::Initailize()
{
    for(int i = 0;i<capa;i++)
    {
        base[i] = 0;
    }
}

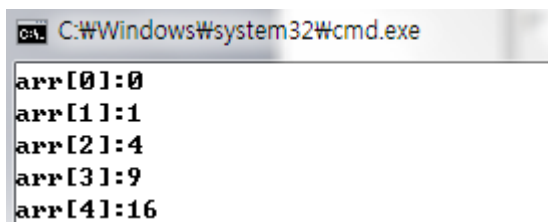
bool IntArr::AvailIndex(int index)
{
    return (index>=0)&&(index<capa);
}
```

Example.cpp

```
#include "IntArr.h"
#include <iostream>
using namespace std;
int main()
{
    IntArr arr(5);

    for(int i = 0;i<5;i++)
    {
        arr[i] = i * i;
    }
    for(int i = 0;i<5;i++)
    {
        cout<<"arr["<<i<<"]:"<<arr[i]<<endl;
    }
    return 0;
}
```

[예제 5.3] []연산자 중복 정의



```
C:\Windows\system32\cmd.exe
arr[0]:0
arr[1]:1
arr[2]:4
arr[3]:9
arr[4]:16
```

[그림 5.1] 예제 5.3 실행 화면

5.2.2 묵시적 형 변환 연산자 중복 정의

C++언어에서 int 형식과 char 형식은 상호 묵시적 형 변환이 가능합니다. 이는 int 형식 변수에 char 형식의 값을 대입한다고 하더라도 컴파일 내부에서 묵시적으로 char 형식의 값을 int 형식의 값으로 묵시적 형 변환하여 대입하기 때문입니다. C++언어에서는 사용자 정의 형식에 대해서도 묵시적 형 변환 연산자를 중복 정의할 수 있게 문법을 제공합니다.

묵시적 형 변환 연산자 중복 정의할 때의 리턴 형식은 형변환하고자 하는 형식임이 자명하므로 리턴 형식을 명시할 수 없습니다. 그리고 묵시적 형 변환 연산자는 단항 연산자이므로 클래스에 정의할 때에는 입력 매개 변수 리스트가 비게 됩니다.

Stu.h

```
#pragma once
class Stu
{
    const int num;
public:
    Stu(int _num);
    operator int();
};
```

Stu.cpp

```
#include "Stu.h"
Stu::Stu(int _num):num(_num)
{
}
Stu::operator int()
{
    return num;
}
```

[예제 5.4] 묵시적 형 변환 연산자 중복 정의

5.3 개체 출력자

C++ 표준 기구에서는 `iostream`은 프로그램의 데이터를 출력 스트림에 보내거나 입력 스트림으로부터 데이터를 얻어오기 위한 목적으로 제공하고 있습니다. 여기에서는 우리가 정의하는 형식 개체도 출력 스트림인 `ostream`을 통해 내보내는 방법에 대해 먼저 얘기해 보도록 합시다.

개체의 정보를 다른 매체로 내보내는 도구를 개체 출력자라 합니다. 여기서는 `ostream`을 통해 개체의 정보를 다른 매체로 내보내는 방법을 살펴볼 것입니다. 우리는 이미 `cout`이라는 `ostream` 기반의 개체와 `<<` 연산자를 통해 여러 기본 형식들을 화면에 출력하고 있습니다. 이는 `ostream` 클래스 내부에서 다양한 기본 형식에 대해 `<<` 연산자 중복 정의가 되어 있기 때문입니다.

LikeAsOStream.h

```
#pragma once
#include <stdio.h>
extern const char *Endl;
class LikeAsOStream
{
public:
    LikeAsOStream &operator<<(int val);
    LikeAsOStream &operator<<(char val);
    LikeAsOStream &operator<<(const char *val);
    LikeAsOStream &operator<<(float val);
};
```

LikeAsOStream.cpp

```
#include "LikeAsOStream.h"
const char *Endl="\n";
LikeAsOStream &LikeAsOStream::operator<<(int val)
{
    printf("%d",val);
    return (*this);
}
LikeAsOStream &LikeAsOStream::operator<<(char val)
{
    printf("%c",val);
    return (*this);
}
LikeAsOStream &LikeAsOStream::operator<<(const char *val)
{
    printf("%s",val);
    return (*this);
}
LikeAsOStream &LikeAsOStream::operator<<(float val)
{
    printf("%f",val);
    return (*this);
}
```

[예제 5.5] ostream 클래스 흉내내기

이와 비슷한 형태로 ostream 클래스에는 << 연산자 중복 정의가 되어 있어 printf 함수와 다르게 어떠한 형식을 출력하기를 원하는지 명시하지 않아도 컴파일러가 적절하게 호출해 줍니다.

또한, ostream 클래스는 ofstream의 기반 클래스이며 파일 스트림에 출력할 때도 cout을 통해 화면에 출력하는 것과 같은 방식으로 가능합니다.

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

void main()
{
    ofstream of("data.txt");
    of<<"번호:"<<3<<" 이름:"<<"홍길동"<<endl;
    of.close();
}
```

[예제 5.6] ofstream 사용

우리는 여기에서 출력 스트림이 화면에 출력하기 위한 개체이든 파일에 출력하기 위한 개체이든 상관없이 사용자가 정의한 클래스 형식도 출력할 수 있게 해 봅시다.

이를 위해서는 ostream 형식과 우리가 출력하고자 하는 형식을 입력 인자로 받는 << 연산자를 중복 정의해야 할 것입니다. 그런데 ostream 클래스는 우리가 정의한 형식이 아니기 때문에 전역 연산자 중복 정의를 해야 합니다. 그리고 cout<<num<<name<<endl; 처럼 연쇄 작업이 가능하게 하기 위해 반환 형식은 ostream &로 정의합니다.

그리고 특정 형식의 개체 출력자를 정의할 때 전역 연산자 중복 정의를 하기 때문에 정보 은닉한 개체의 멤버에 접근하지 못할 수 있습니다. 이 때 friend 키워드를 등록하여 전역 연산자 중복 정의하는 곳에서 개체의 멤버에 접근을 허용하세요. 아무 곳이나 friend 키워드를 사용하면 OOP의 캡슐화를 통해 얻을 수 있는 정보 은닉의 장점을 깨질 수 있지만 개체 출력자는 해당 형식을 정의하는 곳에서 불가피하게 전역으로 정의하는 것이기 때문에 정보 은닉의 장점이 깨지지 않는다고 볼 수 있을 것입니다. 예제처럼 friend로 지정된 함수가 클래스 내부에 정의되어 있어도 이는 클래스의 멤버가 아니라 전역 함수입니다.

```

#pragma once
#include <iostream>
#include <string>
using namespace std;
class Stu
{
    const int num;
    string name;
public:
    Stu(int _num,string _name):num(_num)
    {
        name = _name;
    }
    friend ostream &operator << (ostream &os,const Stu *stu)
    {
        os<<(*stu);
        return os;
    }
};
int main()
{
    Stu *stu = new Stu(3,"홍길동");
    cout<<stu;
    delete stu;
    return 0;
}

```

[예제 5.7] 개체 출력자

5.4 함수 개체

함수 개체란 함수 호출 연산자가 중복 정의되어 해당 개체를 함수처럼 사용할 수 있는 개체를 말합니다. 함수 개체를 사용하면 함수 호출 시에 일부 알고리즘을 호출부에서 전달하여 피호출 함수에서 전달받은 알고리즘을 이용할 수 있습니다.

먼저, 함수 호출 연산자를 중복 정의하는 방법을 살펴볼게요. 함수 호출 연산자는 () 연산자를 중복 정의하면 되고 입력 매개 변수 리스트는 함수 개체를 정의하는 개발자가 정하게 됩니다.

다음은 함수 개체를 정의한 간략한 예제입니다. 함수 개체를 입력 인자로 전달하여 사용하는 예를 살펴보실 분은 "IT 전문가로 가는 길 Escort C++"을 참고하시기 바랍니다.

```
#include <iostream>
using namespace std;
class Comparer
{
public:
    int operator()(int a,int b)
    {
        return a-b;
    }
};
int main()
{
    Comparer comparer;
    cout<<comparer(3,4)<<endl;
    return 0;
}
```

[예제 5.8] 함수 개체

5.5 구조화된 예외 처리

C++ 언어는 탄생 후에 시대 흐름에 맞게 생존하기 위해 새로운 문법들이 계속 추가되었습니다. 이 중의 하나가 namespace이고 또 다른 하나로 구조화된 예외처리가 있습니다. 이 외에도 #pragma 를 비롯하여 여러 종류의 확장자를 가능하게 하는 등의 많은 사항이 추가되었는데 여기에서는 구조화된 예외처리에 대해 살펴보기로 하겠습니다.

예외나 버그, 오류 등의 용어는 서로 구분하는 것이 모호할 수 있습니다. 일반적으로 버그는 개발자의 논리가 잘못되어 잘못 작성한 코드를 말하며 오류는 사용자가 잘못 사용하여 발생하는 것을 말합니다. 그리고 예외는 버그나 오류는 아니지만 프로그램이 더 이상 진행할 수 없는 상황을 말합니다. 예를 들어 다른 서버와 연결하여 동작하는 프로그램에서 서버에 연결하려고 할 때 서버가 정상적이지 않아서 프로그램을 더 이상 진행하지 못하는 상황을 들 수 있을 것입니다.

이러한 예외를 처리하는 방법은 if 문을 사용하거나 assert 함수를 이용하는 방법이 있습니다. 하지만 개발 단계에서 빠르게 확인하고 개발자의 의도에 맞게 해당 상황을 처리하기에는 무리가 있습니다. 이에 C++에서는 구조화된 예외처리를 제공하고 있는데 Java나 C#에서도 비슷한 문법을 제공하고 있습니다.

구조화된 예외처리 구문은 크게 try 블록과 catch블록, throw 문으로 구성합니다.

try 블록에서는 예외가 발생할 수 있는 구문을 시도하는 코드를 작성하는 부분입니다. 그리고 catch 블록은 try 블록 다음에 작성하여 발생한 예외를 잡아 처리합니다. throw문은 예외 상황이 발생할 때 예외를 catch 블록으로 던지는 역할을 수행합니다.

[] 연산자 중복 정의에서 사용한 IntArr 클래스를 보시면 입력 인자로 잘못된 인덱스를 전달받았을 때 더 이상 진행하지 못하여 throw 문을 이용하였습니다.

여기서는 IntArr 개체를 생성하여 인덱스 연산을 사용할 때 잘못된 인덱스를 사용하는 시도를 하고 예외를 잡아 처리하는 예를 보여드릴게요.

```

#include "IntArr.h"
#include <iostream>
using namespace std;
void main()
{
    try
    {
        IntArr arr(10);
        int i = 20;
        arr[i] = 80;
    }
    catch(const char *msg)
    {
        cout<<msg<<endl;
    }
}

```

[예제 5.9] 예외 처리

[예제 5.9]에서는 용량이 10인 IntArr 개체를 생성하여 인덱스 20에 80을 보관하려 하고 있습니다. IntArr의 인덱스 연산 중복 정의한 메서드에서는 잘못된 인덱스를 사용할 때 예외를 던지는 throw문을 사용하였는데 throw 문을 만나면 throw 문이 있는 함수까지 스택을 되감아 catch 블록에 예외를 전달합니다.

그리고 catch 블록은 메서드처럼 보이지만 메서드가 아닙니다. catch 블록은 예외 종류에 따라 여러 개를 둘 수 있는데 앞쪽에서 예외를 받으면 뒤에 있는 catch 블록들은 무시됩니다. 따라서 throw할 예외를 클래스로 정의할 때 파생 클래스 개체를 받는 catch 블록을 기반 클래스 개체를 받는 catch 블록보다 앞에 두어야 합니다. 만약 기반 클래스 개체를 받는 catch블록을 앞에 두면 모든 예외를 받아 처리하므로 적절한 처리를 하기 힘들 수 있습니다.

```

#include <iostream>
#include <string>
using namespace std;
class BaseException
{
public:
    string ToString(){ return "Base Exception"; }
};
class DerivedException:
    public BaseException
{
public:
    string ToString(){ return "Derived Exception"; }
};
void TestA(){ throw new DerivedException(); }
int main()
{
    try
    {
        TestA();
    }
    catch(BaseException *e) //DerivedException 개체도 받아 처리함
    { cout<<e->ToString()<<endl; }
    catch(DerivedException *e) //앞쪽 catch블록 때문에 아무런 예외도 받지 못함
    { cout<<e->ToString()<<endl; }
    return 0;
}

```

[예제 5.10] 파생 형식 예외 처리하는 catch문이 동작하지 않는 예외처리

6. 템플릿

6.1 템플릿이란?

템플릿은 '틀'이라는 사전적 의미를 지니고 있습니다. C++언어의 템플릿 문법은 가상의 코드를 정의하면 컴파일러가 이를 사용하는 부분을 컴파일하면서 구체화한 코드를 생성하는 틀을 말합니다. 즉, 템플릿으로 정의한 코드는 가상의 코드이며 실제 구체화한 코드는 컴파일 시에 컴파일 전개로 생성합니다. 이러한 이유로 템플릿 코드는 헤더에 작성하는 것이 일반적입니다.

6.2 전역 템플릿 함수

템플릿 문법을 이용하여 template 전역 함수를 만드는 방법에 대해 살펴보십시오.

```
template <typename [가상타입명],...>
[리턴형식] 템플릿 함수명(입력인자리스트)
{
    [코드]
}
```

이와 같은 형식으로 함수를 작성하면 이를 사용하는 부분을 만났을 때 컴파일러가 구체화 된 함수를 작성하고 이를 호출하는 구문으로 변경해 줍니다.

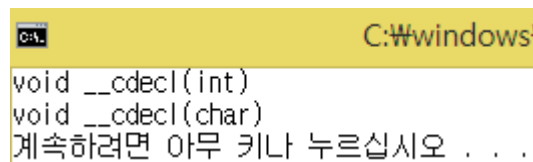
```
TemplateMethod.h
#pragma once
#include <iostream>
using namespace std;
template <typename T>
void Foo(T t)
{
    cout<<typeid(Foo<T>).name()<<endl;
}
```

```

Program.cpp
#include "TemplateMethod.h"
int main()
{
    Foo(1);
    Foo('a');
    return 0;
}

```

[예제 6.1] 전역 템플릿 함수



```

C:\windows
void __cdecl(int)
void __cdecl(char)
계속하려면 아무 키나 누르십시오 . . .

```

[그림 6.1] 예제 6.1 실행 화면

[예제 6.1]에서는 정수를 인자로 사용하는 경우와 문자형을 인자로 사용하고 있습니다. 컴파일러는 사용하는 인자에 맞게 템플릿 함수를 아래처럼 전개해 줍니다.

```

void Foo<>(int t)
{
    cout<<typeid(Foo<int>).name()<<endl;
}
void Foo<>(char t)
{
    cout<<typeid(Foo<int>).name()<<endl;
}
int main()
{
    Foo(1);
    Foo('a');
    return 0;
}

```

6.2.1 명시적 템플릿 인수 사용하여 함수 구현

전역 템플릿 함수를 제공하려고 하는데 특정 형식의 인수일 경우에는 템플릿 함수를 기반으로 구체화 된 코드가 만들어지는 것을 피하고 미리 정의된 함수를 사용하게 할 때에는 명시적으로 템플릿 인수를 사용하여 미리 구체화 된 함수를 구현할 수 있습니다. 컴파일러는 명시적으로 구현된 함수와 인수가 일치하면 템플릿 함수를 기반으로 구체화 된 함수를 작성하는 과정이 생략되고 이미 구현된 함수를 호출하도록 컴파일이 됩니다.

MyCompare.h

```
template <typename T>
int Compare(T t1,T t2){ return t1-t2; }
```

Example.cpp

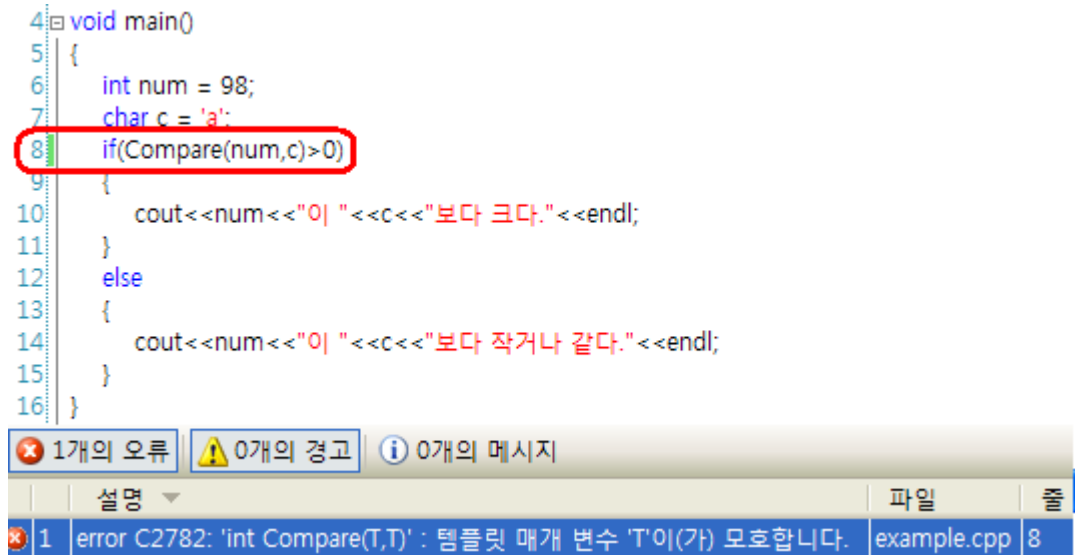
```
int Compare<>(const char *str1,const char *str2)
{
    return strcmp(str1,str2);
}
void main()
{
    if(Compare(10,5)>0){ cout<<"첫번째 인수가 크다."<<endl; }
    else{ cout<<"첫번째 인수가 크지 않다."<<endl; }
    if(Compare("hello","yahoo")>0){ cout<<"첫번째 인수가 크다."<<endl; }
    else{ cout<<"첫번째 인수가 크지 않다."<<endl; }
}
```

[예제 6.2] 명시적 템플릿 인수 사용하여 함수 구현

[예제 6.2]에서 템플릿 인자 형식이 int인 Compare<> 함수는 템플릿 코드를 기반으로 컴파일러가 작성하여 이를 호출합니다. 하지만 템플릿 인자 형식이 const char *를 사용할 때는 명시적으로 구현하였기 때문에 컴파일 전개가 이루어지지 않고 구체적으로 구현한 Compare<>(const char *str1,const char *str2) 함수를 호출합니다.

6.2.2 템플릿 인자 형식을 명시하여 호출하기

템플릿 인자 형식이 같은 여러 개의 입력 인자를 전달받는 템플릿 함수를 호출할 때 사용하는 입력 인자의 형식이 다르면 컴파일러는 전개하지 못합니다.



[그림 6.2] 모호한 템플릿 매개 변수 사용 화면

이와 같은 경우 사용자는 다음과 같이 템플릿 인자 형식을 명시하여 호출을 하여 이를 해결할 수 있습니다.

```
if(Compare<int>(num,c)>0)
{
    cout<<num<<"이"<<c<<"보다 크다."<<endl;
}
else
{
    cout<<num<<"이"<<c<<"보다 작거나 같다."<<endl;
}
```

6.3 템플릿 클래스

템플릿 클래스는 멤버 필드의 형식과 일부 멤버 메서드의 인수의 형식만 다르고 메서드 내부의 논리 전개가 같은 경우에 사용합니다. 템플릿 클래스도 가상의 클래스이며 사용하는 코드의 템플릿 형식 인자에 따라 구체화 된 클래스를 컴파일러에 의해 만들어지게 됩니다.

[예제 6.3]은 멤버 메서드를 템플릿 클래스 내부에 구현한 것입니다.

```
#pragma once
template <typename T>
class MyTemplate
{
    T data;
public:
    MyTemplate(T _data)
    {
        data = _data;
    }
    int Compare(T in)
    {
        return data - in;
    }
    operator T()
    {
        return data;
    }
};
```

[예제 6.3] 멤버 메서드를 템플릿 클래스 내부에 정의

[예제 6.4]는 멤버 메서드를 템플릿 클래스 외부에 구현하는 경우입니다. 이 경우에는 해당 멤버 메서드가 템플릿 클래스의 멤버라는 것을 각 멤버 메서드 정의문에 명시하여야 합니다.

```
#pragma once
template <typename T>
class MyTemplate
{
    T data;
public:
    MyTemplate(T _data);
    int Compare(T in);
    operator T();
};
template <typename T>
MyTemplate<T>::MyTemplate(T _data)
{
    data = _data;
}
template <typename T>
int MyTemplate<T>::Compare(T in)
{
    return data - in;
}
template <typename T>
MyTemplate<T>::operator T()
{
    return data;
}
```

[예제 6.4] 멤버 메서드를 템플릿 클래스 외부에 정의

[예제 6.4]를 보시면 아시겠지만 템플릿 클래스명은 실제 클래스명이 될 수 없으며 사용할 템플릿 형식 인자를 포함해야 클래스명이 됩니다. 그리고 멤버 메서드를 템플릿 클래스 외부에 정의할 때에는 각각의 메서드가 템플릿 메서드임을 명시하여야 합니다.

이처럼 템플릿 클래스(위의 두 가지 방법 중 어떠한 방법을 사용하더라도 차이는 없습니다.)를 정의하였을 때 사용하는 코드에서 어떠한 템플릿 형식 인자에 해당하는 개체를 만들 것인지 선언부에서 명확하게 명시하여야 합니다. 컴파일러는 명시된 템플릿 형식 인자에 맞게 템플릿 클래스를 기반으로 실제 클래스 코드를 작성하고 이를 사용하는 코드로 전개해 줍니다.

```
#include "MyTemplate.h"
#include <iostream>
using namespace std;
int main()
{
    MyTemplate<int> mt(3);
    int diff = mt.Compare(8);
    cout<<"차이:"<<diff<<endl;
    if(mt == 3){ cout<<"같다."<<endl; }
    else{ cout<<"다르다."<<endl; }
    return 0;
}
```

[예제 6.5] 템플릿 클래스를 사용

사용하는 곳에서 템플릿 인자를 명시하여 변수를 선언하고 new 연산을 통해 동적으로 개체를 생성할 때에도 이를 명시하여야 함에 주의합니다. 미리 typedef을 이용하면 간략하게 표현할 수 있습니다.

```
typedef MyTemplate<int> MyInt;
...중략...
MyInt mt(3);
MyInt *mt2 = new MyInt(4);
```

7. 들어가기에 앞서

7.1 STL 소개

STL(Standard Template Library, 표준 템플릿 라이브러리)은 개체들을 보관하기 위한 다양한 컨테이너와 이들 컨테이너에 보관된 개체들을 반복적으로 순회할 수 있게 해 주는 반복자, 사용자에서 정의한 코드를 입력 인자로 전달받아 처리할 수 있게 추상화한 함수 개체, 다양한 문제 해결 방법이 구현된 함수들로 구성된 알고리즘 등으로 구성되어 있습니다.

이 책에서는 STL에 제공되는 일부 컨테이너와 반복자, 함수 개체 및 알고리즘을 소개합니다. vector(배열)와 map(이진 탐색 트리)는 사용법을 소개하고 list는 STL에서 제공하는 것과 비슷한 구조로 만들어 볼게요. 보다 자세하게 STL의 내부를 살펴보실 분은 [IT 전문가로 가는 길 Escort 자료구조와 STL]를 참고하세요.

개정된 STL에서는 namespace std 에 정의되어 있기 때문에 해당 namespace를 사용하겠다는 표시해야 합니다.

```
using namespace std;  
using std::[사용할이름];
```

STL에서 제공하는 컨테이너 종류에는 선형 자료구조인 vector와 list, 선형 자료구조를 특정 목적에 맞게 변형한 stack, queue, priority_queue가 있으며 자료를 비선형으로 보관하는 set, multiset, map, multimap, hash_map, 기타 컨테이너로 bitset, valarray 등을 제공하고 있습니다. 이 외에도 map이나 multimap에서 사용되는 단순히 키와 값의 쌍으로 구성된 pair를 제공합니다.

STL에서 제공되는 각 컨테이너는 별도의 헤더 파일을 포함하여 사용할 수 있게 되어 있으며 일반적으로 사용하려는 컨테이너 이름과 헤더 파일명은 일치합니다.

```
#include <vector>  
using std::vector;
```

이번 장에서 설명하는 예제 코드들은 이 책에서 다루는 내용을 미리 보여주는 것일 뿐 어떠한 문법 사항이나 코드에 대한 설명을 위한 것이 아닙니다.

STL에서 제공하는 반복자는 컨테이너의 종류에 상관없이 컨테이너의 특정 구간에 보관된 개체들에 대해 차례대로 같은 방법으로 작업할 때 사용합니다. 실제 각 컨테이너의 자료를 보관하는 구조에 따라 각 반복자의 구현은 다르게 정의되어 있지만 같은 방법으로 사용할 수 있습니다. 그리고 반복자의 간접 연산을 하면 컨테이너에 보관한 원소 형식으로 캐스팅됩니다.

```
vector<int> vi;
... 중략...
vector<int>::iterator seek = vi.begin();
vector<int>::iterator end = vi.end();
for( ; seek != end ; ++seek)
{
    cout<<*seek<<endl;
}
```

STL에서는 다양한 알고리즘을 라이브러리화하여 제공하고 있습니다. 그리고 일부 알고리즘을 사용자가 결정해야 할 때 함수 개체를 이용할 수 있습니다. 예를 들어, 학생 관리 프로그램에서 학생 이름순으로 정렬할 것인지 번호 순으로 정렬할 것인지에 따라 비교하는 구문은 달라질 수 있는데 이 경우에 비교하는 부분에 대한 알고리즘을 구현하여 이를 정렬을 수행하는 함수에 입력 인자로 전달하면 됩니다. 알고리즘을 전달할 때는 함수 이름을 사용할 수도 있지만 함수 개체를 전달할 수도 있습니다.

```
bool Compare(Stu *s,Stu *b)
{
    return s->GetNum()<b->GetNum();
}
vector<Stu *> stues(100);
sort(stues.begin(), stues.end(),Compare);
```

STL에서 제공되는 알고리즘은 다양한 정렬 알고리즘, 검색 알고리즘뿐만 아니라 수치 해석, 통계와 같은 특수한 목적을 위한 것들도 많습니다. 이 책에서는 컨테이너를 중심으로 반복자와 함수 개체 및 알고리즘을 소개하고 설계 및 구현 방법을 전달할 것입니다.

7.2 이 책에서 공통적으로 사용하는 것들

앞으로 이 책에서 사용할 클래스 ehglobal을 소개를 하겠습니다. ehglobal 클래스에는 이 책에 소개되는 전반적인 예제 프로그램에서 공통으로 사용할 만한 함수들을 정적 멤버 메서드로 캡슐화되어 있습니다. 이 책에 공통으로 사용 가능한 것들에 대한 정의에서는 형식 명과 메서드 명 모두 소문자만을 사용하고 있습니다.

먼저, 콘솔 화면을 지우는 메서드로 clrscr을 제공할 것입니다. 여기에서는 자주 사용하는 함수들을 ehglobal 클래스의 정적 멤버 메서드로 캡슐화하여 사용자 편의를 제공하는 것 외에는 특별한 의미가 없습니다. 여러분께서 의미 없다고 생각하시면 무시하셔도 무관합니다.

```
void ehglobal::clrscr()
{
    system("cls");
}
```

정수를 입력받는 메서드로 getnum을 제공할 것입니다. istream 개체인 cin을 사용하여 cin>>num; 과 같은 구문으로 정수를 입력을 받게 되면 사용자가 잘못 입력하거나 정수 이외에 다른 문자열을 포함하여 입력한다면 기본 입력 버퍼에 계속 남아있게 되어 원하는 바대로 수행되지 않습니다. 여기에서는 cin개체의 getline 메서드로 사용자가 입력한 stream(아스키코드의 연속된 나열로써 연속의 끝은 '\n'입니다. 참고로, 문자열은 '\0'가 오기 전까지의 아스키코드의 연속된 나열입니다.)을 지역 변수에 입력받습니다. 그리고 입력 버퍼를 지운 다음에 입력된 stream을 정수로 치환하여 반환하게 하였습니다.

```
int ehglobal::getnum()
{
    int num;
    char buf[255+1];
    cin.getline(buf,255);
    cin.clear();
    sscanf(buf,"%d",&num);
    return num;
}
```

문자열을 입력받는 메서드로 `getstr`을 제공할 것입니다. 마찬가지로 `cin>>name;` 과 같은 형태로 문자열을 입력받으면 공백이나 탭이 중간에 오게 입력을 하면 그 이전까지만 입력을 받게 됩니다. 그리고 입력 버퍼에는 여전히 그 이후의 문자들이 남아있게 되어 다음에 입력 요청을 하면 새롭게 입력받지 않고 남아있는 데이터를 이용하게 됩니다. 이를 방지하기 위해 `getstr`을 제공하고 있습니다.

```
string ehglobal::getstr()
{
    char buf[255+1];
    cin.getline(buf,255);
    cin.clear();
    return buf;
}
```

그리고 메뉴를 입력받는 작업과 같이 기능 키를 입력받기 위한 메서드 `getkey`를 제공할 것입니다. 여기에서는 F1~F7까지와 ESC를 제공하기로 하겠습니다. 먼저, 이와 같은 키를 열거형 `keydata`로 정의하겠습니다.

```
enum keydata
{
    NO_DEFINED,F1,F2,F3,F4,F5,F6,F7,ESC
};
```

기능 키를 입력받는 메서드에서는 사용자로부터 입력받은 키를 `keydata` 형식에 열거된 값으로 변환하여 반환하도록 하겠습니다.

`getch` 함수를 호출하였을 때 ESC 키를 누르면 27을 반환합니다. 즉, `getch` 함수가 반환하는 값이 27인 경우에는 `keydata` 형식에 열거된 ESC를 반환하게 하였습니다.

그리고 기능 키를 누르면 0이 반환되며 다시 `getch`를 호출하면 사용자로부터 다시 입력을 받지 않고 F1일 경우에는 59, F2일 경우에는 60을 반환합니다. 즉, `getch` 함수를 호출하여 반환 값이 0이면 다시 `getch` 함수를 호출하여 반환 값에 따라 F1~F7을 반환하도록 하였습니다. 이 외에는 `NO_DEFINED` 값을 반환합니다.

EHGlobal.h

```
#pragma once
#pragma warning(disable:4996)

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

enum keydata
{
    NO_DEFINED,F1,F2,F3,F4,F5,F6,F7,ESC
};

//공통적으로 사용할 정적 메서드를 캡슐화한 클래스
class ehglobal
{
public:
    static void clrscr();//화면을 지우는 메서드
    static int getnum();//수를 입력받는 메서드
    static string getstr();//문자열을 입력받는 메서드
    static keydata getkey();//기능 키를 입력받는 메서드
private:
    ehglobal(void) //개체를 생성하지 못하게 하기 위해 private으로 접근 지정
    {
    }
    ~ehglobal(void)
    {
    }
};
```

EHGlobal.cpp

```
#include "ehglobal.h"
#include <conio.h>
#include <windows.h>

void ehglobal::clrscr()//화면을 지우는 메서드
{
    system("cls");
}

void ehglobal::timeflow(int millisecond) //원하는 시간동안 지연시키는 메서드
{
    Sleep(millisecond);
}

int ehglobal::getnum()//정수를 입력받는 메서드
{
    int num;
    char buf[255+1];
    cin.getline(buf,255); //버퍼에 입력받음
    cin.clear();//cin 내부 버퍼를 지움
    sscanf(buf,"%d",&num); //포맷에 맞게 버퍼에 내용을 정수로 변환
    return num;
}

string ehglobal::getstr()//문자열을 입력받는 메서드
{
    char buf[255+1];
    cin.getline(buf,255);
    cin.clear();
    return buf;
}
```



```

keydata ehglobal::getkey()//기능 키를 입력받는 메서드
{
    int key = getch();

    if(key == 27) //ESC를 누를 때의 key 값이 27임
    {
        return ESC;
    }
    if(key == 0) //기능 키를 눌렀을 때는 getch의 반환값이 0임
    {
        //어떤 기능 키를 눌렀는지 확인하려면 getch를 다시 호출해야 함
        //사용자에게 다시 키를 입력받는 것은 아님
        key = getch();
        switch(key) //입력한 키에 따라 약속된 값 반환
        {
            case 59: return F1;
            case 60: return F2;
            case 61: return F3;
            case 62: return F4;
            case 63: return F5;
            case 64: return F6;
            case 65: return F7;
        }
    }
    return NO_DEFINED; //열거되지 않은 키를 눌렀을 때
}

```

[예제 7.1] 공통적으로 사용할 ehglobal 클래스

8. vector (배열)

STL에서 제공하는 컨테이너 중에서 C++언어에서 제공하는 배열과 가장 흡사한 컨테이너는 vector입니다. vector 내부에는 원소 형식들을 연속적인 프로그램 메모리에 보관할 수 있는 물리적 공간을 가지고 있기 때문에 변수명과 인덱스 연산자를 통해 원하는 원소를 찾을 수 있습니다.

```
vector<int> arr(5);  
for(int index = 0; index < 5; ++index)  
{  
    arr[index] = index+1;  
}
```

C++언어에서 제공되는 배열은 유효하지 않은 인덱스를 통해 접근하였을 때 프로그램이 터지지 않는 경우도 발생합니다. 이러면 개발 단계에서 빠르게 논리적 버그를 찾지 못하여 비용이 커지게 됩니다. 하지만 STL에서 제공하는 vector에서는 보관된 원소의 개수를 기억하는 멤버가 있어 유효한 범위를 벗어난 요소에 접근하면 예외가 발생하여 버그가 존재하는 것을 알 수 있습니다.

vector에서는 인덱스 연산 이외에도 차례대로 자료를 보관하거나 특정 키순으로 보관할 수 있게 다양한 메서드들을 제공하고 있습니다. 실제 vector를 사용하는 개발자는 프로그램에서 관리할 데이터와 기능에 따라 사용방법을 결정하게 될 것입니다. 이 책에서는 특정 키를 갖는 요소가 vector의 약속된 인덱스에 보관하는 경우와 차례대로 보관, 특정 키순으로 보관하는 경우를 다룰 것입니다.

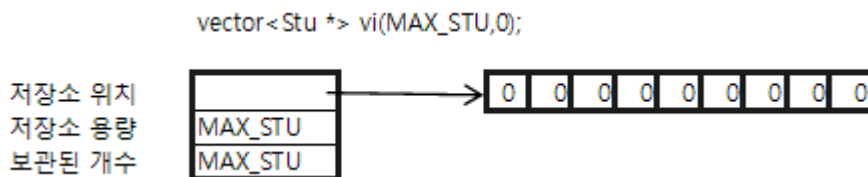
8.1 인덱스 연산을 통해 사용하기

vector에서 인덱스 연산을 사용하면 보관된 요소의 개수에 상관없이 인덱스 연산 한 번으로 원하는 요소에 접근할 수 있습니다. 이러한 장점을 사용하기 위해서는 보관할 요소의 특정 키를 특정 알고리즘을 통해 보관하거나 보관된 위치를 구할 수 있어야 할 것입니다.

보관할 요소의 특정 키를 특정 알고리즘을 통해 보관하거나 보관된 인덱스를 구할 수 있으면 vector를 인덱스 연산을 통해 사용하면 효과적입니다. 그리고 특정 키를 입력 인자로 받았을 때 알고리즘을 통해 얻어온 인덱스의 최대값이 결정될 수 있으면 보다 효과적입니다.

인덱스 연산을 통해 vector를 사용하면 vector 생성 시에 저장소의 크기를 최대값으로 하고 모든 원소를 초기값으로 설정해야 합니다. 사실 vector의 인덱스 연산은 보관된 요소를 참조하기 때문입니다. 그리고 보관하거나 삭제 혹은 검색 등의 작업에서 원하는 인덱스에 있는 값이 초기값인지 여부를 확인하여 사용해야 합니다. vector 개체를 생성할 때 최대 보관할 요소의 개수(사용할 최대 인덱스+1)와 초기값을 입력 인자로 전달하여 생성합니다.

```
vector<Stu *> base(MAX_STU,0);
```



[그림 8.1] vector 개체의 생성 시의 논리적인 메모리 구조

만약, vector 개체가 특정 클래스의 멤버 필드로 캡슐화되어 있다면 초기화 과정을 통해 vector의 저장소 크기를 조절하고 초기값으로 모든 요소의 값을 설정하는 작업을 할 필요가 있습니다.

```
typedef vector<Stu *> StuCollection;
class StuManager
{
    StuCollection base;
};
StuManager::StuManager(void)
{
    base.resize(max_stu,0);
}
```

보관할 때는 보관할 인덱스의 요소를 참조하여 초기값인지 확인할 필요가 있습니다.

```
if(base[num-1])
{
    cout<<"이미 보관된 학생이 있습니다."<<endl;
    return;
}
base[num-1] = new Stu(num,name);
```

삭제할 때도 해당 인덱스의 요소를 참조하여 초기값이 아닌 경우에만 삭제해야 할 것입니다. 삭제할 때에는 보관된 요소를 지우는 것이 아니고 초기값으로 설정하면 됩니다.

```
if(base[num-1]==0)
{
    cout<<"보관된 학생이 없습니다."<<endl;
    return;
}
base[num-1] = 0;
```

보관된 전체 요소에 대해 어떠한 작업을 수행할 때도 실제 보관된 것인지 초기값으로 설정된 것인지 확인하는 과정이 필요합니다.

```
for(int i = 0; i<max_stu; i++)
{
    if(base[i])
    {
        cout<<base[i]<<endl;
    }
}
```

이제 vector의 인덱스 연산을 사용하는 예제 프로그램을 작성해 보기로 합니다.

소재: 학생 관리 프로그램

요구 사항

프로그램 시작 시에 관리할 최대 학생 수를 설정할 수 있어야 합니다.

사용자에 의해 메뉴를 선택하여 선택한 기능을 수행하는 것을 반복합니다.

학생 정보 추가 (학생의 정보는 번호와 이름이 있습니다.)

학생 정보 삭제

번호로 학생 정보 검색

이름으로 학생 정보 검색

전체 보기

vector의 인덱스 연산을 사용하는 학생 관리 프로그램은 학생 정보에 대응되는 Stu 클래스와 학생 정보들을 관리하는 StuManager로 구성하겠습니다.



[그림 8.2] 클래스 다이어그램

먼저, 학생 정보에 해당하는 형식을 정의해 보기로 합시다. 학생 멤버 필드에는 번호와 이름을 추가하고 이들의 값을 확인하는 메서드와 개체 출력자를 제공합니다.

```
Stu.h

#pragma once
#include "EhGlobal.h"
class Stu
{
    const int num; //상수 멤버 변수
    string name;
public:
    Stu(int num,string name): num(num),name(name) //멤버 초기화 구문
    {
    }
    int GetNum()const
    {
        return num;
    }
    string GetName()const
    {
        return name;
    }
    friend ostream &operator<<(ostream &os,Stu *stu)
    {
        os<<"번호:"<<stu->num<<"이름"<<stu->name<<endl;
        return os;
    }
};
```

[예제 8.1] Stu 클래스

이번에는 학생 개체를 관리하는 StuManager를 만들어 봅시다. 먼저, Stu 형식이 정의된 Stu.h를 포함하고 학생 개체를 vector를 이용하여 보관할 것이므로 vector 파일도 포함하세요. vector는 std 이름 공간에 정의되어 있으므로 vector를 사용하겠다는 것을 표현합니다. 그리고 vector는 template 클래스로 제공하고 있어 매번 템플릿 인자를 명시하는 것이 불편하므로 typedef를 이용하여 Stu *를 보관하는 vector를 StuCollection으로 정의하겠습니다.

```
#include "Stu.h"
#include <vector>
using std::vector;
typedef vector<Stu *> StuCollection;
```

StuManager에는 학생 개체들을 보관하기 위한 멤버 필드가 필요하겠죠. 그리고 요구 사항에 명시된 최대 관리 학생 수를 보관할 멤버 필드도 선언하겠습니다.

```
class StuManager
{
    StuCollection base;
    const int max_stu;
};
```

StuManager를 사용하는 곳은 진입점인 main 함수 외에는 없습니다. 이를 위해 생성자와 소멸자, 사용자와 상호작용하는 Run 메서드를 제공할게요.

```
StuManager(void);
~StuManager(void);
void Run();
```

생성자 메서드에서는 요구 사항에 따라 최대 관리할 학생 수를 설정해야 합니다. 최대 관리할 학생 수를 관리하는 멤버 필드인 max_stu가 상수화 멤버 필드로 지정되어 있으므로 초기화 기법을 사용해야겠지요. 그리고 최대 관리할 학생 수는 사용자에게 입력받아 설정하면 됩니다. 최대 학생 수가 지정되면 vector의 버퍼공간을 늘려주기 위해 resize 메서드를 이용하여 저장소의 크기를 늘려주고 모든 요소를 0으로 설정합니다.

```
StuManager::StuManager(void): max_stu(SetMaxStu())//초기화 구문
{
    base.resize(max_stu,0); //보관한 개수를 max_stu로 조절(늘어난 곳은 0으로 보관)
}
int StuManager::SetMaxStu()
{
    cout<<"최대 관리할 학생 수를 입력하세요"<<endl;
    return ehglobal::getnum();
}
```

Run 메서드에서는 사용자에게 메뉴를 선택하게 하고 선택한 메뉴에 따라 해당 기능을 호출하는 것을 반복하면 되겠죠.

```
int key=0;
while((key = SelectMenu())!= ESC)
{
    switch(key)
    {
        case F1: AddStu(); break; //학생 추가
        case F2: RemoveStu(); break; //학생 삭제
        case F3: SearchStuByNum(); break; //번호로 학생 검색
        case F4: SearchStuByName(); break; //이름으로 학생 검색
        case F5: ListAll(); break; //전체 보기
        default: cout<<"잘못된 메뉴를 선택하였습니다."<<endl;
    }
    cout<<"아무키나 누르세요"<<endl;
    ehglobal::getkey();
}
```


Run 메서드에서 호출하는 각 메서드들은 StuManager 내부에서만 필요한 멤버이므로 다른 곳에서 접근할 수 없게 private으로 접근 수준을 지정하면 됩니다.

```
class StuManager
{
private:
    keydata SelectMenu();//메뉴를 보여주고 메뉴를 입력 받는 메서드
    void AddStu();//학생 추가 메서드
    void RemoveStu();//학생 제거 메서드
    void SearchStuByNum();//번호로 학생 검색 메서드
    void SearchStuByName();//이름으로 학생 검색 메서드
    void ListAll();//전체 보기 메서드
    int SetMaxStu();//관리할 수 있는 최대 학생수를 입력받아 설정하는 메서드
};
```

SelectMenu 메서드에서는 메뉴를 보여주고 난 후에 사용자가 입력한 키를 반환하세요.

```
keydata StuManager::SelectMenu()
{
    ehglobal::clrscr();
    cout<<"메뉴 [ESC]:종료"<<endl;
    cout<<"[F1]:학생 추가 [F2]:학생 삭제 [F3]:번호로 검색 ";
    cout<<"[F4]:이름으로 검색 [F5]:전체 보기"<<endl;
    cout<<"메뉴를 선택하세요"<<endl;
    return ehglobal::getkey();
}
```

AddStu 메서드에서는 추가할 학생의 번호를 입력받은 후에 존재 여부를 확인합니다. 만약, 보관된 학생 정보가 없으면 추가할 학생의 이름 정보를 입력받아 Stu 개체를 생성하고 vector에 보관하면 될 것입니다. 여기에서는 vector의 인덱스 연산을 사용하는 예이므로 해당 학생 번호를 가지고 보관할 인덱스를 구합니다. 그리고 해당 인덱스에 보관된 값이 0인지 확인을 통해 이미 보관되었는지 확인하면 될 것입니다. 보관할 때도 인덱스 연산을 통하여 생성한 Stu 개체 정보를 대입하면 됩니다. StuManager에서는 생성한 Stu 개체의 정보를 보관하는 것이지만 vector 내부에서는 해당 인덱스에 있는 요소를 변경하는 것입니다.

```
void StuManager::AddStu()
{
    int num = 0;

    cout<<"추가할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();

    if((num<=0)|| (num>max_stu))
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }

    if(base[num-1]) //번호 -1 위치에 보관하기로 했음, 초기에 0으로 보관했음
    {
        cout<<"이미 보관된 학생이 있습니다."<<endl;
        return;
    }

    string name = "";
    cout<<"이름을 입력하세요"<<endl;
    name = ehglobal::getstr();
    base[num-1] = new Stu(num,name);
}
```

RemoveStu 메서드에서는 삭제할 학생의 번호를 입력받은 후에 존재 여부를 확인하여 있다면 해당 Stu 개체를 소멸하고 해당 인덱스 요소의 값을 0으로 지정하면 되겠죠. 앞서도 얘기했듯이 vector입장에서는 Stu 개체를 보관하거나 삭제하는 것이 아니고 이미 보관된 요소의 정보를 변경하는 것입니다. vector를 사용하는 StuManager에서는 특정 인덱스 요소의 값이 0이면 보관이 안 된 것으로 취급하고 그 이외의 값이면 보관된 것으로 취급하는 것입니다. 즉, vector를 인덱스 연산으로 사용할 때에는 약속된 초기값을 설정하여 해당 인덱스 요소의 값이 초기값인지 아닌지에 따라 보관되었는지를 판단하면 됩니다.

```
void StuManager::RemoveStu()
{
    int num = 0;

    cout<<"삭제할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();

    if((num<=0)|| (num>max_stu))
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }
    if(base[num-1]==0)
    {
        cout<<"보관된 학생이 없습니다."<<endl;
        return;
    }
    delete base[num-1]; //개체를 소멸
    base[num-1] = 0;    //초기값으로 다시 설정
}
```

SearchStuByNum 메서드에서는 검색할 학생의 번호를 입력받은 후에 인덱스 연산을 통해 존재 여부를 확인합니다. 있다면 해당 요소의 Stu 개체 정보를 보여주면 되겠죠.

```
void StuManager::SearchStuByNum()
{
    int num = 0;
    cout<<"검색할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();

    if((num<=0)|| (num>max_stu))
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }

    if(base[num-1]==0)
    {
        cout<<num<<"번 학생은 보관되지 않았습니다."<<endl;
        return;
    }

    cout<<base[num-1]<<endl; //개체 출력자를 구현하였기 때문에 사용할 수 있음
}
```

SearchStuByName 메서드에서는 검색할 학생의 이름을 입력받은 후에 같은 이름을 갖는 학생을 찾아 개체 정보를 보여줍니다.

```
void StuManager::SearchStuByName()
{
    cout<<"검색할 학생 이름을 입력하세요."<<endl;
    string name = ehglobal::getstr();
    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //학생이 보관되었는지 확인
        {
            if(base[i]->GetName() == name)
            {
                cout<<base[i]<<endl;
                return;
            }
        }
    }
    cout<<name<<" 학생은 보관되지 않았습니다."<<endl;
}
```

ListAll 메서드에서는 인덱스 연산을 이용하여 학생 정보를 보여주면 되겠죠.

```
void StuManager::ListAll()
{
    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //학생이 보관되었는지 확인
        {
            cout<<base[i]<<endl;
        }
    }
}
```

마지막으로 StuManager 소멸자에서 생성한 모든 Stu 개체를 소멸해야겠지요. C++에서는 동적으로 생성한 개체를 개발자가 소멸해야 합니다. 개발할 때 개발자가 소멸의 책임을 다하지 않는 경우가 많이 있습니다. 실제 이를 하지 않아도 컴파일 오류가 발생하지 않고 동작에도 큰 문제가 없어 보이거든요. 하지만 작성하는 프로그램이 서버 프로그램이거나 라이브러리 형태라고 한다면 필요없는 개체에 할당된 메모리를 소멸하지 않으면 메모리 누수가 발생하여 메모리 폴트가 발생하는 치명적인 버그가 될 수 있습니다. 습관적으로 개체를 생성하는 코드를 작성할 때 소멸하는 코드를 작성하시기 바랍니다.

```
StuManager::~StuManager(void)
{
    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //학생이 보관되었는지 확인
        {
            delete base[i];
        }
    }
}
```

StuManager.h

```
#pragma once
#include "Stu.h"
#include <vector>
using std::vector;
typedef vector<Stu *> StuCollection;

class StuManager
{
    StuCollection base; //학생을 보관할 컬렉션(vector)
    const int max_stu; //최대 보관할 수 있는 학생 수(최대 학생 번호이기도 함)
public:
    StuManager(void);
    ~StuManager(void);
    void Run();
}
```

```
private:
    keydata SelectMenu();
    void AddStu();
    void RemoveStu();
    void SearchStuByNum();
    void SearchStuByName();
    void ListAll();
    int SetMaxStu();
};
```

StuManager.cpp

```
#include "StuManager.h"
StuManager::StuManager(void): max_stu(SetMaxStu())//초기화 구문
{
    base.resize(max_stu,0);
}
int StuManager::SetMaxStu()
{
    cout<<"최대 관리할 학생 수를 입력하세요"<<endl;
    return ehglobal::getnum();
}
StuManager::~StuManager(void)
{
    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //실제 학생이 보관된 곳인지 확인
        {
            delete base[i];
        }
    }
}
```

```

void StuManager::Run()
{
    int key=0;

    while((key = SelectMenu())!= ESC)
    {
        switch(key)
        {
            case F1: AddStu(); break;
            case F2: RemoveStu(); break;
            case F3: SearchStuByNum(); break;
            case F4: SearchStuByName(); break;
            case F5: ListAll(); break;
            default: cout<<"잘못된 메뉴를 선택하였습니다."<<endl;
        }
        cout<<"아무키나 누르세요"<<endl;
        ehglobal::getkey();
    }
}

keydata StuManager::SelectMenu()
{
    ehglobal::clrscr();
    cout<<"메뉴 [ESC]:종료"<<endl;
    cout<<"[F1]:학생 추가 [F2]:학생 삭제 [F3]:번호로 검색"
    cout<<" [F4]:이름으로 검색 [F5]:전체 보기"<<endl;
    cout<<"메뉴를 선택하세요"<<endl;
    return ehglobal::getkey();
}

```



```

void StuManager::AddStu()
{
    int num = 0;
    cout<<"추가할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();
    if((num<=0)|| (num>max_stu))
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }
    if(base[num-1]) //실제 학생이 보관되었는지 확인
    {
        cout<<"이미 보관된 학생이 있습니다."<<endl;
        return;
    }
    string name = "";
    cout<<"이름을 입력하세요"<<endl;
    name = ehglobal::getstr();
    base[num-1] = new Stu(num,name);
}

void StuManager::RemoveStu()
{
    int num = 0;
    cout<<"삭제할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();
    if((num<=0)|| (num>max_stu))
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }
}

```

```

if(base[num-1]==0) //실제 학생이 보관되었는지 확인
{
    cout<<"보관된 학생이 없습니다."<<endl;
    return;
}

delete base[num-1];
base[num-1] = 0; //초기값 0으로 다시 설정
}
void StuManager::SearchStuByNum()
{
    int num = 0;
    cout<<"검색할 학생 번호를 입력하세요. 1~"<<max_stu<<endl;
    num = ehglobal::getnum();

    if((num<=0)||num>max_stu)
    {
        cout<<"범위를 벗어났습니다."<<endl;
        return;
    }

    if(base[num-1]==0) //실제 학생이 보관되었는지 확인
    {
        cout<<num<<"번 학생은 보관되지 않았습니다."<<endl;
        return;
    }
    cout<<base[num-1]<<endl;
}

```

```

void StuManager::SearchStuByName()
{
    string name="";

    cout<<"검색할 학생 이름을 입력하세요."<<endl;
    name = ehglobal::getstr();

    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //실제 학생이 보관되었는지 확인
        {
            if(base[i]->GetName() == name)
            {
                cout<<base[i]<<endl;
                return;
            }
        }
    }

    cout<<name<<" 학생은 보관되지 않았습니다."<<endl;
}

void StuManager::ListAll()
{
    for(int i = 0; i<max_stu; i++)
    {
        if(base[i]) //실제 학생이 보관되었는지 확인
        {
            cout<<base[i]<<endl;
        }
    }
}

```

Demo.cpp

```
#include "StuManager.h"
void main()
{
    StuManager *sm = new StuManager();
    sm->Run();
    delete sm;
}
```

[예제 8.2] 인덱스 연산으로 vector 사용하기

8. 2 vector에 자료를 차례대로 보관하기

이번에는 vector를 사용해서 차례대로 보관하는 프로그램을 작성해 보기로 합시다.

소재: 학생 관리 프로그램

요구 사항

사용자에 의해 메뉴를 선택하여 선택한 기능을 수행하는 것을 반복합니다.

학생 정보 추가 (학생의 정보는 번호와 이름이 있습니다.)

학생 정보 삭제

번호로 학생 정보 검색

이름으로 학생 정보 검색

전체 보기

Stu 클래스는 그대로 사용하고 vector를 사용하는 StuManager 부분만 변경해 봅시다.

인덱스 연산자를 이용할 때는 보관할 최대 요소 개수를 입력받아 vector에 resize 메서드를 이용하여 0으로 모든 요소를 보관하는 초기 작업을 필요했지만 여기서는 필요가 없습니다. Run 메서드는 vector를 사용하는 부분이 아니라 흐름을 제어하는 부분이기 때문에 똑같이 사용해도 됩니다. 메뉴 선택을 하는 SelectMenu도 변경할 필요가 없겠죠.

학생 정보를 추가하는 AddStu 메서드에서는 번호를 입력받아 이미 있는지 확인합니다. vector의 인덱스 연산을 통해 관리할 때는 해당 인덱스에 학생 정보를 참조하여 0인지 아닌지로 보관 여부를 판단하였습니다. 하지만 여기에서는 vector의 시작 위치에서 마지막 위치 중에 원하는 요소가 있는지를 확인해야 합니다. 이를 위해 특정 번호에 해당하는 학생 정보가 보관되었는지를 판단하는 Exist 메서드를 추가하겠습니다.

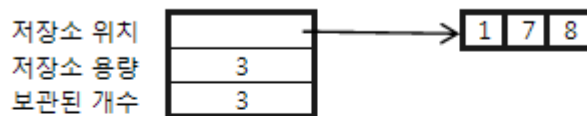
Exist 메서드에서는 반복자를 이용하여 보관된 모든 요소를 차례대로 비교해 나가야 합니다. 이를 위해 vector에는 보관된 첫 위치를 얻어올 수 있는 begin 메서드를 제공합니다. 그리고 마지막 보관된 다음 위치(이번에 보관할 위치라고 생각할 수 있겠죠.)를 얻어오는 end 메서드를 제공합니다. begin 메서드와 end 메서드는 iterator를 반환하며 증감 연산자로 다음 위치로 이동하고 간접 연산을 통해 보관된 요소를 참조할 수 있습니다.

```
bool StuManager::Exist(int num)
{
    Stulter seek = base.begin();
    Stulter end = base.end();
    Stu *stu=0;
    for( ; seek != end; ++seek) //반복자는 비교 연산, 증감 연산이 가능
    {
        stu = *seek; //iterator의 간접 연산의 결과는 보관한 요소(학생 위치 정보)
        if( stu->GetNum() == num)
        {
            return true;
        }
    }
    return false;
}
```

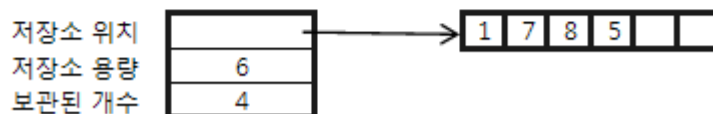
AddStu 메서드에서는 사용자가 입력한 번호의 학생 정보가 없을 때 Stu 개체를 생성하여 vector에 차례대로 보관합니다. vector에 차례대로 보관할 때는 push_back 메서드를 사용합니다. vector에서는 보관할 저장소의 용량이 부족하면 내부에서 자동으로 늘려줍니다. 이러한 이유때문에 vector에 차례대로 자료를 보관할 때 저장소의 용량에 대해 크게 신경 쓰지 않아도 됩니다.

```
int num = 0;
cout<<"추가할 학생 번호를 입력하세요."<<endl;
num = ehglobal::getnum();
if(Exist(num))
{
    cout<<"이미 보관된 학생이 있습니다."<<endl;
    return;
}
string name = "";
cout<<"이름을 입력하세요"<<endl;
name = ehglobal::getstr();
//차례대로 보관할 때 맨 뒤에 보관하면 되므로 push_bakc을 호출함
base.push_back(new Stu(num,name));
```

꼭 찬 상태에서 자료 5를 보관하기 전



vi.push_back(5); 를 호출하고 난 후의 모습



[그림 8.3] vector가 꼭 찼을 때 push_back 메서드를 호출하기 전 후 모습

RemoveStu 메서드에서는 사용자에게 삭제할 학생 번호를 입력받아 보관된 위치를 찾아야 할 것입니다. 이를 위해 특정 번호에 해당하는 학생이 보관된 위치를 찾는 작업이 필요한 데 함수 개체와 find_if 알고리즘을 사용해 볼게요.

find_if 함수는 STL algorithm에서 제공하고 있습니다. 입력 인자로는 검색할 구간의 시작 위치와 마지막 위치와 비교에 사용할 알고리즘을 전달 받습니다. 그리고 구간 내에서 처음으로 전달받은 알고리즘이 참이 되는 위치를 찾아 반환합니다. 요소를 입력 인자로 받습니다. 그리고 보관된 자료를 입력 인자로 받아 조건을 판단할 수 있는 코드를 세 번째 입력 인자로 받습니다. find_if에서 하는 일은 입력받은 구간 사이에 보관된 자료들을 세 번째 입력 인자로 적용했을 때 처음으로 참이 되는 위치를 반환합니다. 만약, 입력한 구간 내에 원하는 조건이 참인 요소가 없으면 구간의 끝이 반환되니 주의하세요.

```
CompareByNum sbn(num);
Stulter seek = find_if(base.begin(),base.end(),sbn);
if(seek== base.end())//참인 곳이 없을 때
{
    cout<<num<<"번 학생 자료는 보관되지 않았습니다."<<endl;
    return;
}

class CompareByNum
{
    int num;
public:
    CompareByNum(int num)
    {
        this->num = num;
    }
    //학생의 번호와 멤버 변수 num이 같으면 참을 반환
    bool operator()(Stu *stu) //함수 호출 연산자 중복 정의
    {
        return(stu && stu->GetNum() == num);
    }
};
```

그리고 조건이 맞으면 erase 메서드를 이용하여 vector에서 제거하세요. 여기에서는 보관된 학생 개체를 소멸해야 하므로 소멸한 후에 erase 메서드를 호출해야 합니다. vector의 erase 메서드를 호출하면 삭제할 위치 뒤에 있는 요소들은 모두 한 칸씩 앞으로 이동하게 됩니다.

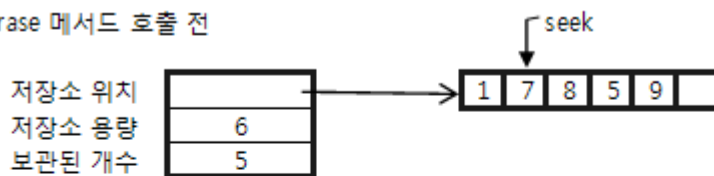
//iterator의 간접 연산을 통해 보관된 학생 정보를 얻어옴

Stu *stu = *seek;

delete stu;

base.erase(seek); //보관된 요소 삭제

erase 메서드 호출 전



erase 메서드 호출 후



[그림 8.4] vector에 erase 메서드 호출 전 후

AddStu에서 입력한 번호의 학생 데이터가 이미 있는지를 확인을 할 때도 RemoveStu 메서드처럼 함수 개체와 find_if를 사용해도 됩니다. 여기서는 두 가지 방법을 모두 보여 드리기 위해 다르게 사용하였습니다.

번호로 검색하는 SearchStuByNum 메서드는 RemoveStu처럼 사용자가 입력한 번호에 해당하는 학생 개체를 찾습니다. 단지, 찾은 위치의 학생 정보를 출력하는 것만 다릅니다.

```
CompareByNum sbn(num);
```

```
StuIter seek = find_if(base.begin(),base.end(),sbn);
```

```
if(seek== base.end())//참인 위치가 없을 때
```

```
{
    cout<<name<<"번 학생 자료는 보관되지 않았습니다."<<endl;
    return;
}
```

```
Stu *stu = *seek; //iterator의 간접 연산으로 보관된 요소를 얻어올 수 있음
```

```
cout<<stu<<endl;
```

이름으로 검색하는 SearchStuByName 메서드에서는 비교하는 함수 개체만 다릅니다.

```
class CompareByName
```

```
{
    string name;
public:
    CompareByName(string name)
    {
        this->name = name;
    }
    bool operator()(Stu *stu)
    {
        return (stu && stu->GetName() == name);
    }
};
```

전체 학생 정보 출력은 vector의 시작 위치에서 끝 위치를 만날 때까지 보관된 학생 정보를 출력하면 되겠죠.

```
Stulter seek = base.begin();
Stulter end = base.end();
Stu *stu = 0;
```

```
//iterator는 비교 연산과 증감 연산을 제공, ++연산을 통해 다음 위치로 이동
for ( ;seek != end; ++seek)
{
    //간접 연산자로 보관된 요소를 얻어올 수 있음
    stu = *seek;
    cout<<stu<<endl;
}
```

그리고 StuManager가 소멸할 때 자신이 생성한 모든 학생 개체를 소멸하는 것을 잊지 맙시다.

```
Stulter seek = base.begin();
Stulter end = base.end();
Stu *stu = 0;
for ( ;seek != end; ++seek)
{
    stu = *seek;
    delete stu;
}
```

StuManager.h

```
#pragma once
#include "Stu.h"
typedef vector<Stu *> StuCollection;
typedef vector<Stu *>::iterator Stulter;
class StuManager
{
    StuCollection base; //학생을 보관하는 컬렉션(vector)
public:
    ~StuManager(void);
    void Run();
private:
    keydata SelectMenu();
    void AddStu();
    void RemoveStu();
    void SearchStuByNum();
    void SearchStuByName();
    void ListAll();
};

class CompareByNum
{
    int num;
public:
    CompareByNum(int num){ this->num = num; }
    bool operator()(Stu *stu)
    {
        return (stu && stu->GetNum() == num);
    }
};
```

```

class CompareByName
{
    string name;
public:
    CompareByName(string name)
    {
        this->name = name;
    }
    bool operator()(Stu *stu) //함수 호출 연산자 중복 정의
    {
        return (stu && stu->GetName() == name);
    }
};

```

StuManager.cpp

```

#include "StuManager.h"
StuManager::~StuManager(void) //반복자를 통해 보관된 모든 학생 요소 소멸
{
    Stulter seek = base.begin();
    Stulter end = base.end();

    Stu *stu = 0;
    for ( ;seek != end; ++seek)
    {
        stu = *seek; //반복자의 간접 연산의 결과는 보관한 요소(학생 위치 정보)
        delete stu;
    }
}

```

```

void StuManager::Run()
{
    keydata key=0;
    while((key = SelectMenu())!=ESC)
    {
        switch(key)
        {
            case F1: AddStu(); break;
            case F2: RemoveStu(); break;
            case F3: SearchStuByNum(); break;
            case F4: SearchStuByName(); break;
            case F5: ListAll(); break;
            default: cout<<"잘못된 메뉴를 선택하였습니다."<<endl;
        }
        cout<<"아무키나 누르세요"<<endl;
        ehglobal::getkey();
    }
}

keydata StuManager::SelectMenu()
{
    ehglobal::clrscr();
    cout<<"메뉴 [ESC]:종료"<<endl;
    cout<<"[F1]:학생 추가 [F2]:학생 삭제 [F3]:번호로 검색";
    cout<<" [F4]:이름으로 검색 [F5]:전체 보기"<<endl;
    cout<<"메뉴를 선택하세요"<<endl;
    return ehglobal::getkey();
}

```

```

void StuManager::AddStu()
{
    int num = 0;
    cout<<"추가할 학생 번호를 입력하세요."<<endl;
    num = ehglobal::getnum();
    if(Exist(num))
    {
        cout<<"이미 보관된 학생이 있습니다."<<endl;
        return;
    }
    string name = "";
    cout<<"이름을 입력하세요"<<endl;
    name = ehglobal::getstr();
    base.push_back(new Stu(num,name));
}

bool StuManager::Exist(int num)
{
    Stulter seek = base.begin();
    Stulter end = base.end();
    Stu *stu=0;
    for( ; seek != end; ++seek)
    {
        stu = *seek; //반복자의 간접 연산의 결과는 보관된 요소(학생 위치 정보)
        if( stu->GetNum() == num)
        {
            return true;
        }
    }
    return false;
}

```

```

void StuManager::RemoveStu()
{
    int num = 0;
    cout<<"삭제할 학생 번호를 입력하세요."<<endl;
    num = ehglobal::getnum();
    CompareByNum sbn(num); //학생 번호가 num과 같으면 참 반환
    Stulter seek = find_if(base.begin(),base.end(),sbn);
    if(seek== base.end())//참인 곳이 없음
    {
        cout<<num<<"번 학생 자료는 보관되지 않았습니다."<<endl;
        return;
    }
    Stu *stu = *seek;
    delete stu;
    base.erase(seek); //특정 위치에 보관된 요소를 지우는 메서드
}

void StuManager::SearchStuByNum()
{
    int num = 0;
    cout<<"검색할 학생 번호를 입력하세요."<<endl;
    num = ehglobal::getnum();
    CompareByNum sbn(num); //함수 개체 선언
    Stulter seek = find_if(base.begin(),base.end(),sbn);
    if(seek== base.end())//참인 곳이 없음
    {
        cout<<num<<"번 학생 자료는 보관되지 않았습니다."<<endl;
        return;
    }
    Stu *stu = *seek; //간접 연산의 결과는 보관된 요소(학생 위치 정보)
    cout<<stu<<endl;
}

```

```

void StuManager::SearchStuByName()
{
    string name="";
    cout<<"검색할 학생 이름을 입력하세요."<<endl;
    name = ehglobal::getstr();
    CompareByName sbn(name); //함수 개체 선언
    //구간 내에 요소들을 함수 개체를 적용했을 때 처음 참인 위치 찾을
    Stulter seek = find_if(base.begin(),base.end(),sbn);
    if(seek== base.end())//참인 곳이 없음
    {
        cout<<name<<"번 학생 자료는 보관되지 않았습니다."<<endl;
        return;
    }
    Stu *stu = *seek;
    cout<<stu<<endl;
}

void StuManager::ListAll()
{
    //차례대로 보여주기 위해 전체 구간의 반복자를 얻어옴
    Stulter seek = base.begin();
    Stulter end = base.end();

    Stu *stu = 0;
    for ( ;seek != end; ++seek)
    {
        stu = *seek;
        cout<<stu<<endl;
    }
}

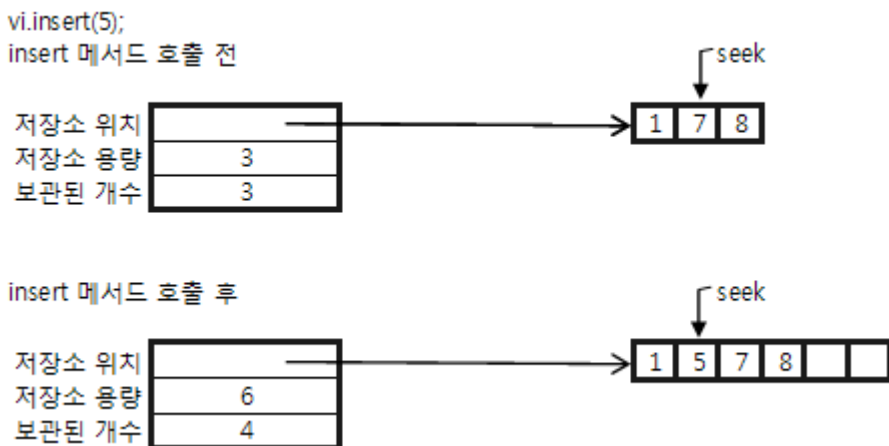
```

[예제 8.3] vector에 순차적으로 보관

2. 3 vector를 이용하여 특정 키 순으로 보관하기

이번에는 vector를 이용하여 특정 키순으로 보관하는 방법에 대해 살펴보기로 합시다. 삭제나 검색, 전체 보기 등은 차례대로 보관할 때와 같습니다. 단지 보관할 위치를 찾는 논리가 다를 뿐입니다.

특정 키순으로 보관하려면 보관된 시작 위치부터 사용자가 입력한 학생 번호가 더 크거나 같은 위치를 찾아서 해당 위치에 보관해야 할 것입니다. 물론, 같으면 필터링해야겠지요. 이 경우에도 함수 개체만 잘 정의하면 쉽게 사용할 수 있습니다. vector에서 원하는 위치에 보관할 때는 insert 메서드를 사용합니다. insert 메서드를 사용할 때에는 보관할 위치에 해당하는 iterator를 입력 인자로 전달해야 합니다. 그리고 vector 내부에서는 해당 위치에 보관된 원소부터 뒤에 보관된 모든 원소는 한 칸씩 뒤로 밀리게 됩니다. 또한, push_back 메서드처럼 저장소가 꽉 차게 되면 vector 내부에서 저장소의 크기를 갱신시킨 후에 보관하므로 사용하는 개발자는 저장소의 크기를 크게 신경 쓸 필요는 없지요.



[그림 8.4] vector에 insert 메서드 호출 전 후 모습

//학생의 번호가 멤버 변수 num보다 크거나 같으면 참을 반환하는 함수 개체 클래스

```
class MoreEqualByNum
```

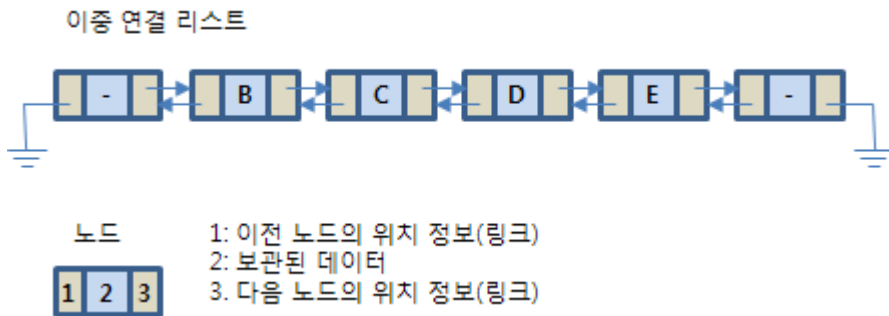
```
{
    int num;
public:
    MoreEqualByNum(int num)
    {
        this->num = num;
    }
    bool operator()(Stu *stu)
    {
        return (stu && stu->GetNum() >= num);
    }
};
```

```
void StuManager::AddStu()
```

```
{
    int num = 0;
    cout<<"추가할 학생 번호를 입력하세요."<<endl;
    num = ehglobal::getnum();
    MoreEqualByNum sbn(num);
    Stulter seek = find_if(base.begin(),base.end(),sbn); //보관할 위치 찾기
    if((seek== base.end())||((*seek)->GetNum() != num)) //없다면
    {
        string name = "";
        cout<<"이름을 입력하세요"<<endl;
        name = ehglobal::getstr();
        base.insert(seek,new Stu(num,name)); //seek 앞에 보관
    }
    else
    {
        cout<<"이미 존재하는 학생입니다."<<endl;
    }
}
```

9. list (연결리스트)

STL에서 제공하는 선형 컨테이너에는 vector 와 list가 있는데 연결 리스트를 표현한 것입니다. 연결 리스트는 노드들의 선형 집합이며 노드는 데이터와 링크의 조합입니다. 연결 리스트는 자료를 보관할 때마다 별도의 노드를 생성하여 하나의 노드에 하나의 자료를 보관하며 노드의 링크를 통해 노드들이 서로의 위치를 선형적으로 유지하는 자료구조입니다.



[그림 9.1] 연결 리스트 구조

연결 리스트는 링크의 개수에 따라 하나만 있는 단일(혹은 단순)연결 리스트, 두 개가 있는 이중 연결 리스트로 나눌 수 있습니다. 그리고 마지막 노드가 시작 노드의 위치 정보를 알고 있게 링크를 설정하는 연결 리스트를 원형 연결 리스트라 합니다. 이처럼 링크의 개수나 유형에 따라 연결 리스트를 구분하지만, 이는 연결 리스트를 만드는 개발자에 관련된 사항일 뿐 사용하는 개발자는 큰 의미를 있지는 않습니다.

STL에서 제공되는 list는 이중 연결 리스트로 되어 있습니다. 그리고 list는 vector와 제공되는 멤버들이 대부분 비슷합니다. 차이가 있는 부분은 인덱스 연산자([])를 사용할 수 없다는 것입니다. vector는 요소들을 보관하는 저장소가 연속된 메모리 하나로 되어 있으므로 저장소 시작 주소에서 상대적 거리를 계산하는 것이 효과적이지만 list에서는 각 요소가 별도의 노드에 보관되기 때문에 인덱스 연산자를 제공하지 않습니다. 또한, 저장소의 용량을 갱신하기 위해 제공했던 reserve 메서드와 용량을 얻어오기 위한 capacity 메서드를 제공하지 않습니다. 그렇지만 여전히 resize 메서드와 size 메서드는 제공하고 있습니다. list를 사용하는 방법은 vector를 사용하는 방법과 큰 차이가 없어서 이 책에서는 list를 사용하는 방법에 대해서는 별도로 논의하지 않겠습니다. 여기에서는 STL에서 제공되는 list와 비슷한 구조를 갖는 템플릿 클래스를 구현해 봅시다.

9. 1 list 만들기

먼저, 프로젝트를 만들어 앞에서 만든 파일들을 추가하는 것부터 하세요. 그리고 EHList.h를 추가합니다.

EHList.h에는 템플릿 클래스인 list를 정의할 것입니다. 우리가 만들 list도 EHLIB 이름 공간 내에 정의할게요.

```
#pragma once
namespace EHLIB
{
    template<typename T>
    class list
    {
    };
};
```

list 클래스 내부에는 node 형식이 정의해야 합니다. 그리고 node의 멤버로는 보관할 요소와 다른 노드의 위치 정보가 있어야 할 것입니다. 여기에서는 이중 연결리스트로 만들 것이기 때문에 두 개의 링크를 캡슐화할게요.

```
template<typename T>
class list
{
    struct node
    {
        node *prev; //이전 노드의 위치 정보
        node *next; //다음 노드의 위치 정보
        T data; //보관된 요소
        node(T data=0):data(data) //초기화
        {
            prev = next = 0;
        }
    };
};
```

list에는 첫 번째 노드 위치와 마지막 노드 위치 정보 및 보관된 요소 개수를 위한 멤버 필드를 캡슐화할게요.

```
template<typename T>
class list
{
    node *head; //맨 앞 노드의 위치 정보
    node *tail;  //맨 뒤 노드의 위치 정보
    size_t bsize; //보관된 요소 개수
};
```

vector와 마찬가지로 보관한 요소들을 순회하여 사용할 수 있게 iterator 형식을 정의합니다. iterator 형식은 list를 사용하는 개발자의 코드에서도 사용해야 하므로 노출 수준을 public으로 지정해야 합니다.

```
template<typename T>
class list
{
    public:
    class iterator
    {
    };
};
```

list에서는 요소가 보관된 노드를 알아야 하고 list를 사용하는 개발자의 코드에서는 보관된 요소를 알 수 있어야 합니다. 이 두 가지 목적을 달성하기 위해 최소한 노드의 위치 정보를 알고 있어야 합니다. 이러한 이유로 node의 위치 정보를 알고 있는 멤버 필드를 캡슐화할게요.

```
class iterator
{
    node *now; //현재 노드의 위치 정보
};
```

iterator 형식의 생성자 메서드는 특정 node의 위치 정보를 입력 인자로 받는 생성자가 필요할 것입니다.

```
class iterator
{
    public:
        iterator(node *now=0)
        {
            this->now = now;
        }
};
```

iterator는 list 내에서는 요소를 보관된 노드의 위치 정보를 알아야 하므로 node *와 묵시적 형변환 연산자를 중복 정의할게요. 그리고 list를 사용하는 개발자 코드에서는 보관된 요소를 알아야 하므로 간접 연산자를 중복 정의하여 보관된 요소 형식을 반환합니다.

```
class iterator
{
    public:
        T operator *() //간접 연산자 중복 정의, 보관된 요소를 반환
        {
            return now->data; //현재 노드에 보관된 요소 반환
        }
        operator node *() //list 내에서 node *와 반복자 간의 묵시적 형변환 가능하게 함
        {
            return now;
        }
};
```

list를 사용하는 개발자 코드에서 iterator를 이용하여 비교 연산을 할 수 있도록 같음(==) 연산자와 다름(!=) 연산자를 중복 정의할게요.

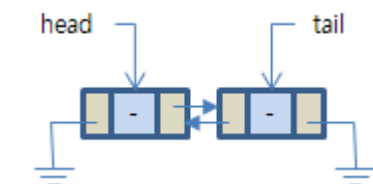
```
class iterator
{
    public:
    bool operator == (const iterator &iter) //now 노드의 위치 정보가 같은지 판별
    {
        return now == iter.now;
    }
    bool operator != (const iterator &iter) //now 노드위 위치 정보가 다른지 판별
    {
        return now != iter.now;
    }
};
```

그리고 list를 사용하는 개발자 코드에서 iterator를 다음으로 이동시킬 때 ++ 연산자를 사용할 수 있게 합시다.

```
class iterator
{
    public:
    iterator &operator++()//전위 ++연산자 중복 정의
    {
        now = now->next; //now를 다음 노드의 위치 정보로 변경
        return (*this); //변경된 자기 자신을 반환
    }
    const iterator operator++(int)
    {
        iterator re(*this); //변경되기 전 자신을 복사
        now = now->next; //now를 다음 노드의 위치 정보로 변경
        return re; //변경되기 전 자신을 복사한 반복자 반환
    }
};
```

list의 생성자 메서드에서는 두 개의 더미 노드를 생성하여 head와 tail 대입하고 size를 0으로 초기화하겠습니다. head와 tail에 더미 노드를 생성하여 대입하면 노드를 삽입하거나 제거하는 논리를 단순화시킬 수 있습니다. 더미 노드가 있으면 노드를 추가하는 논리는 언제나 특정한 노드와 노드 사이에 추가하는 논리로 작성하면 됩니다. 노드를 삭제할 때도 언제나 특정한 노드와 노드 사이에 있는 노드를 삭제하면 되겠죠. 더미 노드가 없다면 처음 추가되는 경우와 중간에 추가하는 경우, 맨 뒤에 추가하는 경우와 맨 앞에 추가하는 경우의 논리가 조금씩 달라집니다. 삭제의 경우도 마찬가지로 맨 앞의 노드를 제거하거나 맨 뒤에 노드를 제거, 중간에 있는 노드를 제거하는 경우의 논리가 달라집니다.

```
template<typename T>
class list
{
public:
    list()
    {
        head = new node();//더미 노드를 생성하여 head 에 대입
        tail = new node(); //더미 노드를 생성하여 tail에 대입
        head->next = tail; //head가 가리키는 노드의 next 정보를 tail로 변경
        tail->prev = head; //tail이 가리키는 노드의 prev 정보를 head로 변경
        bsize = 0; //보관된 요소 개수를 0으로 대입
    }
};
```



[그림 9.2] 연결 리스트 초기화 모습

list의 소멸자 메서드에서는 list 내에서 생성한 모든 노드를 소멸해야 합니다.

```
~list()
{
    while(head != tail) //맨 처음 노드와 맨 마지막 노드가 다르면
    {
        head = head->next; //head를 head가 가리키는 노드의 다음 노드로 변경

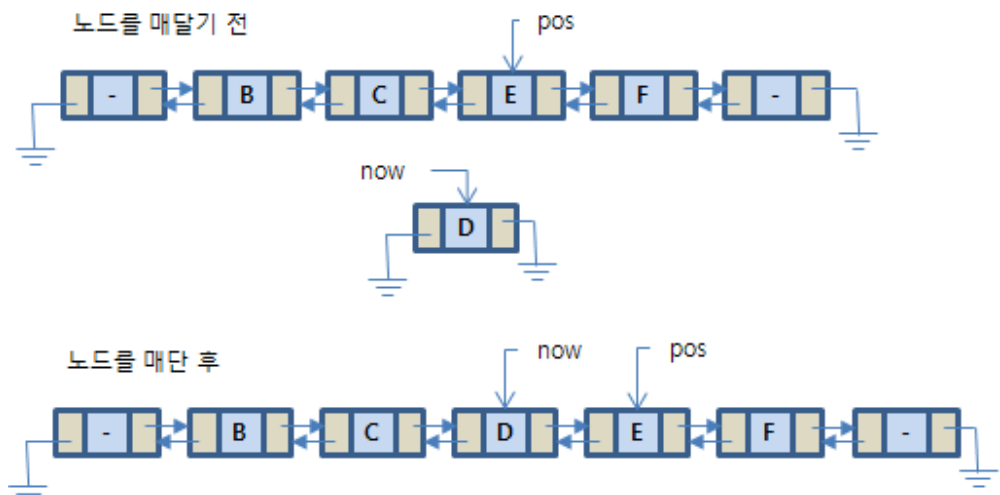
        //변경되기 전의 head를 삭제
        delete head->prev; //head가 가리키는 노드의 이전 노드 소멸(이전 head임)
    }
    delete head; //마지막 남은 노드 소멸
}
```

list에는 차례로 보관할 때 사용하는 push_back 메서드를 제공하고 있습니다. vector처럼 push_back 메서드는 end() 메서드가 반환하는 iterator 앞에 보관하면 됩니다. 즉, insert 메서드를 호출하면 되겠죠.

```
void push_back(T t)
{
    insert(end(),t); //맨 뒤에 보관, tail이 가리키는 노드 앞에 t를 보관
}
```

list에서도 특정한 iterator 앞에 요소를 보관할 때 사용하는 insert 메서드를 제공하고 있습니다. insert 메서드에서는 요소를 보관하는 노드를 생성한 후에 생성한 노드를 입력 인자로 전달받은 iterator 앞에 매달면 됩니다.

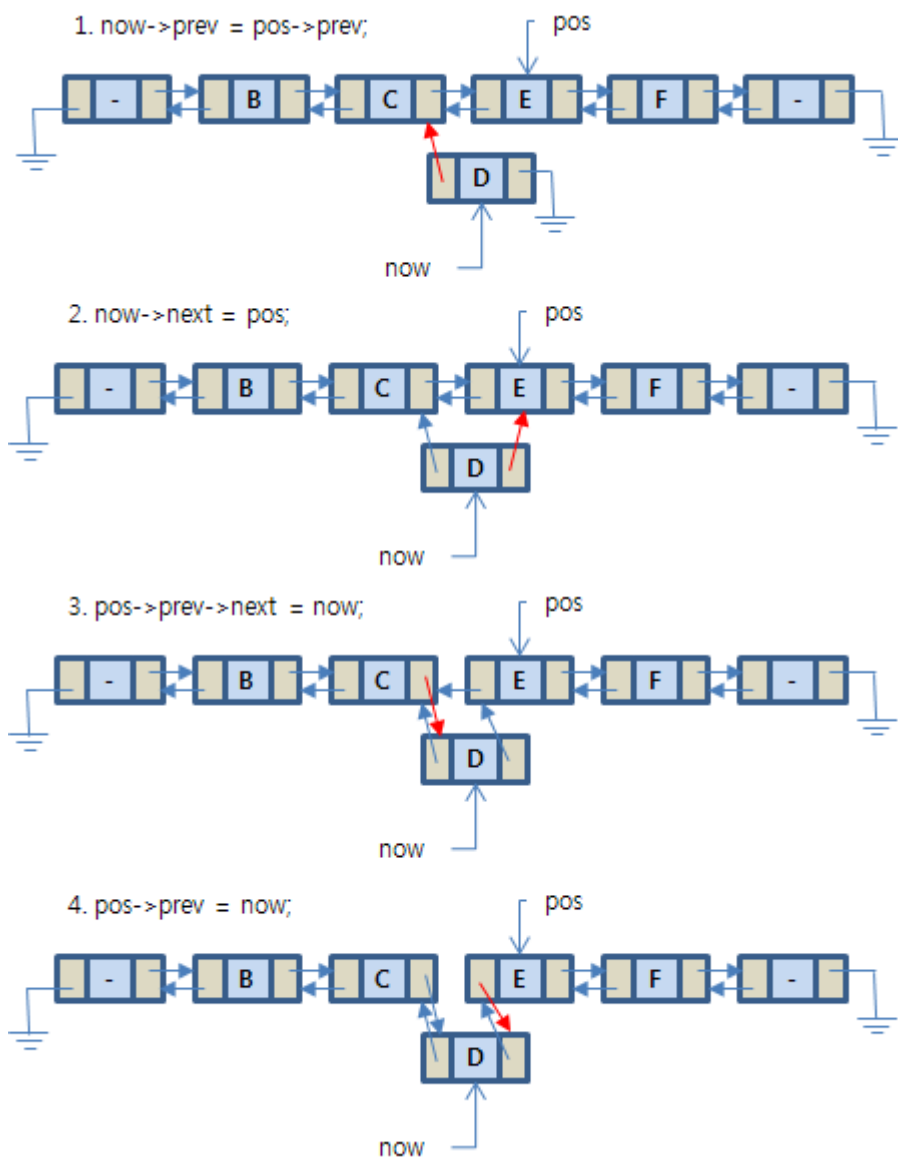
```
void insert(iterator at, T t)
{
    node *pos = at; //at에 있는 노드를 pos에 대입(반복자에 묵시적 형변환 연산자)
    node *now = new node(t); //t를 보관한 노드를 생성하여 now에 대입
    hang_node(now,pos); //생성한 now 노드를 pos 앞에 연결
    bsize++; //보관된 요소 개수 1 증가
}
```



[그림 9.3] 연결 리스트에 노드를 매달기 전후의 논리적 모습

특정한 노드 앞에 노드를 매다는 메서드인 hang_node는 list 내에서 사용하는 메서드로 노출 수준을 private으로 지정할게요. [그림 9]를 보면 연결 리스트에 노드를 매달기 전후의 논리적 모습을 알 수 있습니다. 여러분은 어느 노드의 어떤 링크들을 어떠한 순으로 조절해야 매달 수 있는지 파악해야 합니다. 이를 위해 여러분이 논리적인 그림을 도식하고 이를 코드로 옮기는 순서로 해결하신다면 좀 더 효과적으로 연결 리스트의 동작 원리를 이해할 수 있을 것입니다.

```
void hang_node(node *now,node *pos)
{
    //now가 가리키는 노드의 prev를 pos가 가리키는 prev로 변경
    now->prev = pos->prev;
    //now가 가리키는 노드의 next를 pos로 변경
    now->next = pos;
    //pos의 이전 노드의 next를 now로 변경
    pos->prev->next = now;
    //pos가 가리키는 노드의 prev를 now로 변경
    pos->prev = now;
}
```

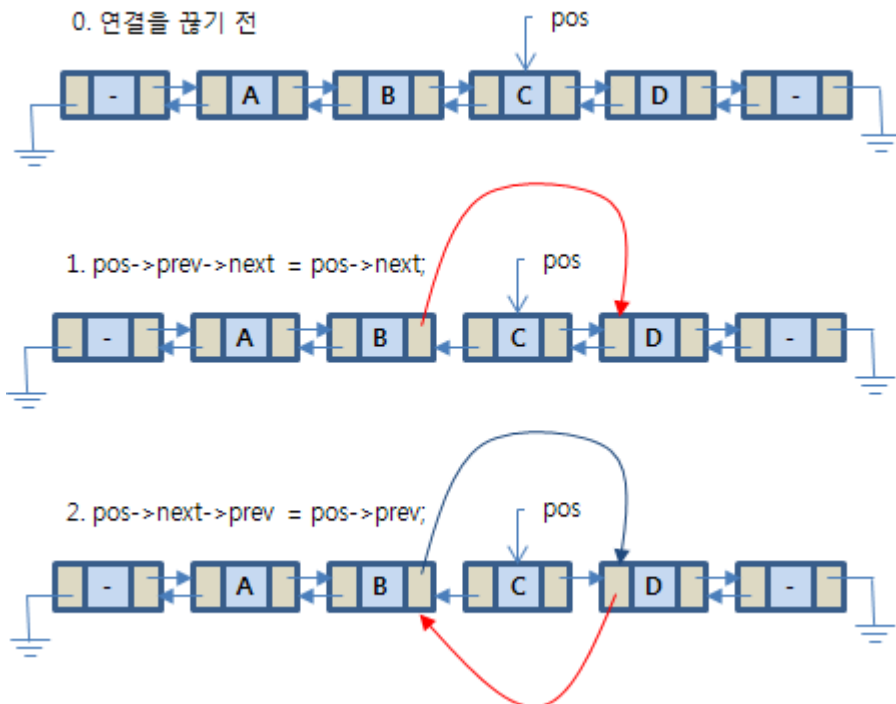


[그림 9.4] 연결 리스트에 노드를 삽입하는 과정

연결 리스트에서도 특정 위치에 보관된 요소를 지우는 erase 메서드를 제공하고 있습니다. erase 메서드에서는 리스트에서 해당 위치에 있는 노드의 연결을 끊는 작업이 필요합니다. 그리고 해당 요소를 보관하기 위해 동적으로 생성한 node 개체를 소멸해야겠지요.

```
void erase(iterator at)
{
    dehang_node(at); //at에 있는 노드를 리스트에서 연결을 끊는다.
    node *pos = at;
    delete pos; //at에 있는 노드를 소멸
    bsize--; //보관한 요소 개수 1 감소
}
```

de_hang 메서드에서는 입력 인자로 전달된 노드의 위치 정보를 통해 이전 노드와 이후 노드의 링크를 변경해 주어야 할 것입니다. 어떠한 노드의 어느 링크를 조절해야 하는지 논리적으로 그림을 그려보세요.



[그림 9.5] 연결리스트에서 노드를 끊는 과정

```

void dehang_node(node *pos)
{
    //pos의 이전 노드의 next를 pos의 next로 변경
    pos->prev->next = pos->next;
    //pos의 다음 노드의 prev를 pos의 prev로 변경
    pos->next->prev = pos->prev;
}

```

list 에서도 iterator를 사용할 수 있게 하려고 맨 앞에 보관된 요소가 있는 iterator를 반환하는 begin 메서드와 차례대로 보관할 때 보관할 위치(맨 마지막에 보관된 요소의 다음 위치)를 반환하는 end 메서드를 제공하고 있습니다. 연결 리스트를 생성할 때 head와 tail에 더미 노드를 생성하여 초기화를 하였기 때문에 맨 앞에 보관된 요소는 head가 가리키는 노드의 다음에 있게 되며 차례대로 보관할 때에는 tail이 가리키는 노드 앞에 보관하게 됩니다.

```

iterator begin()
{
    //head는 더미 노드이므로 head의 다음 노드가 첫 번째 보관된 요소가 있는 노드임
    return head->next; //head의 다음 노드를 반환(iterator와 node *는 묵시적 형변환)
}
iterator end()
{
    //end는 맨 마지막 보관된 요소의 다음 위치를 반환하므로 tail을 반환함
    return tail;
}

```

이 외에도 보관된 요소 개수를 반환하는 size 메서드와 보관된 요소의 개수를 갱신하는 resize 메서드를 제공할게요.

```

size_t size()
{
    return bsize;
}

```

```

void resize(size_t nsize)
{
    //0을 늘어난 개수 만큼 차례대로 보관
    for(size_t n=bsize; n<nsize ; n++)
    {
        insert(end(),0);
    }
}

```

EHList.h

```

#pragma once
namespace EHLIB
{
    template<typename T>
    class list
    {
        struct node
        {
            node *prev; //이전 노드의 위치 정보
            node *next; //다음 노드의 위치 정보
            T data; //보관된 요소
            node(T data=0):data(data) //초기화
            {
                prev = next = 0;
            }
        };

        node *head; //리스트 맨 앞에 있는 노드 위치 정보
        node *tail; //리스트 맨 뒤에 있는 노드 위치 정보
        size_t bsize; //리스트에 보관된 요소 개수

    public:

```

```

class iterator
{
    node *now; //현재 노드의 위치 정보
public:
    iterator(node *now=0)
    {
        this->now = now;
    }
    T operator *()//간접 연산자 중복 정의 , 보관된 요소 반환
    {
        return now->data; //현재 노드에 보관된 요소 반환
    }
    operator node *()//node *와 묵시적 형변환 연산자 중복 정의
    {
        return now;
    }
    bool operator == (const iterator &iter)
    {
        return now == iter.now;
    }
    bool operator != (const iterator &iter)
    {
        return now != iter.now;
    }
    iterator &operator++()//전위 ++ 연산자 중복 정의
    {
        now = now->next; //now를 다음 노드 위치로 변경
        return (*this); //변경된 자신을 반환
    }
}

```

```

const iterator operator++(int) //후위 ++ 연산자 중복 정의
{
    iterator re(*this); //자신을 복제
    now = now->next; //now를 다음 노드 위치로 변경
    return re; //변경 전 복제한 반복자 반환
}
};

list()
{
    head = new node(); //더미 노드 생성하여 head에 대입
    tail = new node(); //더미 노드 생성하여 tail에 대입
    head->next = tail; //head가 가리키는 노드의 next를 tail로 대입
    tail->prev = head; //tail이 가리키는 노드이 prev를 head로 대입
    bsize = 0; //보관된 요소 개수를 0으로 대입
}

~list()
{
    while(head != tail) //head와 tail이 다른 노드를 가리키면
    {
        head = head->next; //head를 head의 다음 노드 위치 정보로 변경

        //변경되기 전 head가 가리키는 노드를 소멸함
        delete head->prev; //head의 이전 노드를 소멸(변경 전 head)
    }
    delete head; //하나 남은 노드를 소멸
}

```



```

void resize(size_t nsize)
{
    //0으로 늘어난 요소 개수만큼 차례대로 보관
    for(size_t n=bsize; n<nsize ; n++)
    {
        insert(end(),0);
    }
}

void push_back(T t)
{
    //맨 마지막 보관된 요소의 노드 뒤에 보관
    //end는 맨 마지막 보관된 요소의 노드 뒤임
    insert(end(),t); //end위치 앞에 t를 보관
}

iterator begin()
{
    //맨 앞에 보관된 요소의 위치를 반환
    //head의 다음 노드가 요소가 보관된 맨 앞 노드임
    return head->next; //head의 next 반환(node *와 반복자는 묵시적 형변환)
}

iterator end()
{
    //맨 마지막에 보관된 요소의 다음 위치를 반환
    //tail은 맨 마지막에 보관된 요소의 다음 위치 노드
    return tail; //tail 반환(node *와 반복자는 묵시적 형변환)
}

size_t size()
{
    return bsize;
}

```

```

void insert(iterator at, T t)
{
    node *pos = at; //at에 있는 노드를 pos에 대입(묵시적 형변환)
    node *now = new node(t); //t를 보관한 노드 생성하여 now에 대입
    hang_node(now,pos); //생성한 노드를 pos앞에 연결
    bsize++; //보관한 요소 개수 1 증가
}

void erase(iterator at)
{
    dehang_node(at); //at에 있는 노드의 연결을 끊음
    node *pos = at; //at에 있는 노드 위치를 pos에 대입(묵시적 형변환)
    delete pos; //pos 위치의 노드를 소멸
    bsize--; //보관된 요소 개수 1 감소
}

private:
    //now 위치의 노드를 pos 앞에 연결하는 메서드
    void hang_node(node *now,node *pos)
    {
        //now가 가리키는 노드의 prev를 pos의 prev로 변경
        now->prev = pos->prev;
        //now가 가리키는 노드의 next를 pos로 변경
        now->next = pos;
        //pos의 이전 노드의 next를 now로 변경
        pos->prev->next = now;
        //pos가 가리키는 노드의 prev를 now로 변경
        pos->prev = now;
    }

```

```

//pos 위치의 노드를 리스트에서 연결 끊는 메서드
void dehang_node(node *pos)
{
    //pos의 이전 노드의 next를 pos의 next로 변경
    pos->prev->next = pos->next;
    //pos의 다음 노드의 prev를 pos의 prev로 변경
    pos->next->prev = pos->prev;
}
};
};

```

[예제 9.1] list 구현

작성한 list가 정상적으로 동작하는지 확인해 봅시다. 앞에서 얘기한 것처럼 list를 사용하는 방법은 vector와 매우 유사합니다. 단지, vector에서는 인덱스 연산을 이용할 수 있었지만 list에서는 사용하지 못하는 정도의 차이라고 보시면 됩니다. vector를 이용하여 차례대로 요소를 보관하는 프로그램이나 vecor를 이용하여 특정 키순으로 보관했던 응용 프로그램 코드에서 vector를 list로 교체하더라도 정상적으로 동작합니다.

이러한 이유로 우리가 만든 list가 정상적으로 동작하는지 확인하기 위해 vector에서 만든 프로젝트 조금 수정하여 테스트할 수 있습니다. 기존 프로젝트에 작성한 EHList.h 파일을 추가하시고 StuManager.h 에 이를 포함하세요. 그리고 EHLIB 이름 공간에 있는 list 형식을 사용할 수 있게 using 문을 작성하세요. 이와 같은 작업이 끝났으면 정상적으로 동작하는지 테스트해 보시기 바랍니다.

```

#pragma once
#include "Stu.h"
#include "EHAlgorithm.h"
#include "EHList.h"
using EHLIB::list;
typedef list<Stu *> StuCollection;
typedef list<Stu *>::iterator Stulter;

```

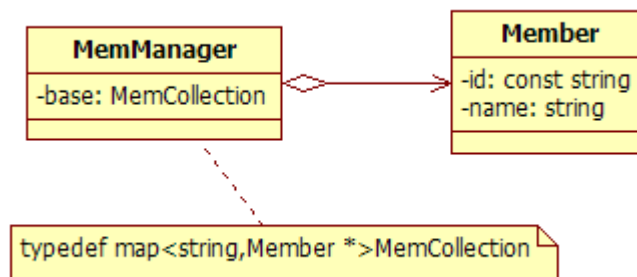
10. 맵 (이진 탐색 트리)

STL에서 제공되는 비선형 자료구조에는 set, multiset, map, multimap 등이 있습니다. 이들은 내부적으로 쓰레드 이진 탐색으로 구현되어 있습니다. set과 map은 같은 자료를 보관하지 못하며 multiset과 multimap은 같은 자료를 보관할 수 있습니다. 그리고 map과 multimap은 key와 value의 쌍을 보관하게 되어 있습니다. 이 책에서는 map을 사용하는 방법을 살펴볼게요.

이번에는 STL에서 제공하는 map을 사용하는 방법에 대해서 살펴보기로 합시다. map은 key와 value를 쌍으로 구성하는 pair를 보관하게 되어 있습니다. map에서는 key를 기준으로 자료를 보관하고 검색하게 됩니다. 만약, 회원 관리 프로그램에서 회원의 id를 기준으로 map에 보관하면 다음과 같이 string을 키로 하고 Member *를 value로 하는 pair를 보관하면 될 것입니다.

```
#include <map>
using std::map;
using std::pair;
using std::make_pair;
typedef map<string,Member *> MemCollection;
typedef map<string,Member *>::iterator MemIter;
```

여기에서는 map을 사용하는 방법을 두 가지 방법으로 나누어 설명할게요. 첫 번째 방법은 insert, find, erase, iterator를 사용하는 방법이고 두 번째 방법은 인덱스 연산자를 사용하는 방법입니다. map을 사용하는 방법을 설명하기 위한 프로그램은 두 가지 경우 모두 회원 관리 프로그램으로 하겠습니다.



[그림 10.1] 회원 관리 프로그램의 클래스 다이어그램

10.1 insert, find, erase, iterator 사용하기

이 책에서는 회원 관리 프로그램을 만드는 과정을 통해 map을 사용하는 방법을 설명하기로 할게요. 먼저, 두 가지 경우에 공통으로 사용하게 될 회원에 대한 정의를 하고 갑시다. 회원은 아이디와 이름을 멤버 필드로 갖는 클래스로 간단하게 정의를 할게요. 그리고 회원 관리 프로그램의 진입점도 두 가지 경우 모두 같게 구현할게요.

Member.h

```
#pragma once
#include "EHGlobal.h"
class Member
{
    const string id;
    string name;
public:
    Member(string id,string name):id(id),name(name) {    }
    string GetId()const{ return id;    }
    string GetName()const{ return name;    }
};
```

Demo.cpp

```
#include "MemManager.h"
void main()
{
    MemManager *mm = new MemManager();
    mm->Run();
    delete mm;
}
```

[예제 10.1] 공통으로 사용할 소스

이제, map을 사용하는 방법에 대해 구체적으로 살펴보기로 합시다. 여기에서는 map에 보관하는 방법과 삭제하는 방법, 검색하는 방법, 보관된 전체 요소를 확인하는 방법에 대해서 살펴볼게요.

회원 관리 프로그램의 main 함수를 보시면 회원 관리자(MemManager) 개체를 생성한 후에 Run을 수행한 후에 소멸하고 있습니다. 이에 MemManager 클래스에는 생성자와 소멸자, Run 메서드를 캡슐화하고 접근 수준을 public으로 지정해야 할 것입니다.

```
class MemManager
{
public:
    MemManager(void);
    ~MemManager(void);
    void Run();
};
```

회원 관리자에서는 회원의 아이디를 키로 하고 회원(Member) 개체의 위치 정보를 값으로 하는 pair를 보관하는 map으로 관리할 것입니다. 회원 관리자에는 MemCollection 개체를 멤버 필드로 캡슐화하고 있어야 할 것입니다. 이를 위해 다음과 같은 약속 할게요.

```
#include <map>
using std::map;
using std::pair;
using std::make_pair;
typedef map<string,Member *> MemCollection;
typedef map<string,Member *>::iterator MemIter;
class MemManager
{
    MemCollection base; //회원을 보관하는 컬렉션(map)
};
```

Run 메서드에서는 프로그램 사용자가 선택한 메뉴를 수행하는 것을 반복하는 형태로 구현할게요. 이에 대해 특별한 설명은 하지 않겠습니다.

```
void MemManager::Run()
{
    int key=0;
    while((key = SelectMenu())!=ESC)
    {
        switch(key)
        {
            case F1: AddMem(); break;
            case F2: RemoveMem(); break;
            case F3: SearchMem(); break;
            case F4: ListAll(); break;
            default: cout<<"잘못된 메뉴를 선택하였습니다."<<endl;
        }
        cout<<"아무 키나 누르세요"<<endl;
        ehglobal::getkey();
    }
}
```

Run 메서드에서 호출하여 사용할 멤버 메서드들은 접근 수준을 private으로 캡슐화하면 되겠죠.

```
class MemManager
{
private:
    keydata SelectMenu();
    void AddMem();
    void RemoveMem();
    void SearchMem();
    void ListAll();
    Member *FindByName(string name);
};
```

메뉴를 보여주고 선택하는 SelectMenu 메서드에 대한 설명은 특별한 것이 없으므로 생략하기로 할게요.

```
keydata MemManager::SelectMenu()
{
    ehglobal::clrscr();
    cout<<"메뉴 [ESC]:종료"<<endl;
    cout<<"[F1]:회원 추가 [F2]:회원 삭제 [F3]:회원 검색 [F4]:전체 보기"<<endl;
    cout<<"메뉴를 선택하세요"<<endl;

    return ehglobal::getkey();
}
```

회원을 추가하는 AddMem 메서드를 구현해 보기로 합시다. 회원 관리자에서는 회원의 아이디를 키로 하고 있습니다. 따라서 회원의 아이디를 입력받은 후에 이미 존재 여부를 확인해야겠지요. map에서는 키를 입력 인자로 받아 보관된 자료의 iterator를 반환해 주는 find 메서드를 제공하고 있습니다. vector와 list 등에서는 알고리즘으로 제공되는 find 함수나 find_if 함수를 사용하였는데 이는 순차적인 탐색이었습니다. 이에 반해 map에서 제공되는 find 메서드는 이진 탐색 트리에서 이진 논법에 따라 탐색을 하므로 검색 효율성을 높일 수 있습니다. 만약, 해당 키를 갖는 pair가 보관되지 않았다면 map의 end 메서드를 통해 반환되는 iterator와 같은 값이 반환됩니다.

```
cout<<"추가할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
```

```
MemIter seek = base.find(id); //map의 find 메서드에 키(회원 아이디)로 위치 찾기
if(seek != base.end())//seek가 base.end()와 같지 않으므로 seek위치에 보관되어 있음
{
    cout<<"이미 존재하는 아이디입니다."<<endl;
    return;
}
```


만약, 같은 아이디가 없다면 회원의 이름을 입력받아 회원 개체를 생성하여 map에 보관하면 되겠죠. map에서는 key와 value를 쌍으로 하는 pair 형식으로 보관합니다. 여기에서는 id를 키로 하고 회원 개체의 위치 정보를 value를 쌍으로 하는 pair를 보관합니다. STL에서는 make_pair 메서드를 사용하여 pair를 만들 수 있습니다. 그리고 map은 insert 메서드에 pair를 입력 인자로 전달하여 자료를 보관할 수 있습니다.

```
cout<<"회원 이름을 입력하세요."<<endl;
string name = ehglobal::getstr();
```

```
//map은 키와 값으로 구성된 pair를 보관
base.insert(make_pair(id,new Member(id,name)));
```

회원을 삭제하는 RemoveMem 메서드를 구현해 봅시다. 여기에서도 삭제할 id를 입력받아 map에서 찾는 작업이 선행되어야 하겠죠.

```
cout<<"삭제할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
```

```
MemIter seek = base.find(id); //map의 find 메서드에 키로 보관된 요소 찾기
```

```
if(seek == base.end())//seek가 base.end()와 같다면 검색 조건에 맞는 요소가 없음
{
    cout<<"존재하지 않는 아이디입니다."<<endl;
    return;
}
```

map에서 보관된 자료를 삭제할 때 사용하는 메서드는 vector와 list처럼 erase 메서드를 사용할 수 있습니다. map의 erase 메서드에도 삭제할 위치에 해당하는 iterator를 입력 인자로 전달하면 됩니다.

```
//반복자의 간접 연산의 결과는 보관한 요소 형식 pair<string,Member *>
delete (*seek).second; //pair의 second는 값인 보관된 요소의 회원 위치 정보
```

```
base.erase(seek);
cout<<"삭제하였습니다."<<endl;
```

검색하는 SearhMem에서도 보관된 자료를 찾는 것은 같습니다.

```
cout<<"검색할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
Memlter seek = base.find(id); //map의 find 메서드에 키로 보관된 요소 찾기
if(seek == base.end())//seek가 base.end()일 때는 검색 조건에 맞는 요소가 없을 때
{
    cout<<"존재하지 않는 아이디입니다."<<endl;
    return;
}
```

map 형식 내부의 iterator 형식은 간접 연산자를 통해 key와 value가 쌍인 pair로 변환됩니다. 여기에서는 value에 해당하는 회원 개체의 위치 정보를 얻어와야겠지요. pair 형식에서는 key를 얻어올 때 사용할 수 있는 멤버 first와 value를 얻어올 수 있는 second를 제공하고 있습니다. 즉, 회원 개체의 위치 정보를 얻어오기 위해서는 pair의 second를 이용하면 됩니다.

```
Member *mem = (*seek).second;
cout<<mem->GetId()<<"."<<mem->GetName()<<endl;
```

전체 회원 정보를 보여주는 ListAll 메서드에서는 map의 begin 메서드를 사용해 얻어온 반복자에서 end 메서드를 사용해 얻어온 반복자 사이에 있는 모든 회원 개체의 위치 정보를 얻어와 정보를 보여주면 될 것입니다. SearhMem을 구현하면서 설명했던 것처럼 반복자의 간접 연산을 통해 얻어온 pair의 second 멤버를 통해 회원 개체의 위치 정보를 얻어올 수 있습니다.

```
Memlter seek = base.begin();
Memlter end = base.end();
Member *mem = 0;
//반복자를 사용하여 차례대로 회원의 정보를 출력
for( ; seek != end ; ++seek)
{
    mem = (*seek).second; //pair의 second는 보관된 요소의 값(회원 위치 정보)
    cout<<mem->GetId()<<"."<<mem->GetName()<<endl;
}
```

마지막으로 회원 관리자 소멸자 메서드에서는 자신이 생성하여 map에 보관했던 모든 회원 개체를 소멸하는 책임을 지는 코드를 작성해야겠지요.

```
MemIter seek = base.begin();
MemIter end = base.end();
for( ; seek != end ; ++seek)
{
    delete (*seek).second;
}
```

10.2 인덱스 연산자 사용하기

이번에는 map에서 제공하는 인덱스 연산자를 사용하여 회원 관리 프로그램을 구현해 보기로 할게요. 여기에서도 앞에서 사용했던 회원에 대한 정의를 구현한 Member.h와 진입점이 있는 Demo.cpp는 같습니다. 여기에서는 차이가 있는 부분에 관해서만 얘기할게요.

먼저, 회원 정보를 추가하는 AddMem 메서드에 대해 살펴봅시다. map에서는 key 값을 인자로 인덱스 연산을 사용하면 value 값을 얻을 수 있습니다. 만약, 보관되지 않는 key값을 사용하면 해당 key값과 value가 0인 pair가 보관되며 연산 결과로 0이 반환됩니다. 즉, 인덱스 연산을 하였을 때 연산 결과가 0이라는 것은 개발자 의도에 의해 보관된 것이 없다는 것을 의미합니다.

```
cout<<"추가할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
```

```
Member *member = base[id]; //map의 인덱스 연산(키(id) 를 사용) 결과는 값(회원)
//만약, 없을 때 pair<id, 0>가 보관하고 0을 반환함
```

```
if(member) //키(id)에 해당하는 값(회원)을 찾았을 때
{
    cout<<"이미 존재하는 아이디입니다."<<endl;
    return;
}
```

추가하고자 할 경우에도 인덱스 연산을 이용할 수 있습니다.

```
cout<<"회원 이름을 입력하세요."<<endl;
string name = ehglobal::getstr();
```

```
//map의 인덱스 연산을 이용하여 pair<키,값>을 보관할 수 있음
base[id] = new Member(id,name);
```

보관된 회원 정보를 찾아 삭제하는 RemoveMem 메서드를 구현해 봅시다. 여기에서도 인덱스 연산을 통해 입력받은 id에 해당하는 회원 개체를 얻어오게 됩니다. 만약, 연산 결과가 0이라면 보관된 회원 정보가 없다는 것입니다.

```
cout<<"삭제할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
```

```
//map의 인덱스 연산을 이용하여 키(id)에 해당하는 값(회원 위치 정보)를 얻어올 수 있음
//키(id)에 해당하는 값이 없을 때 pair<id,0>을 보관한 후에 0을 반환
Member *member = base[id];
```

```
if(member == 0) //없을 때
{
    cout<<"존재하지 않는 아이디입니다."<<endl;
    return;
}
```

보관된 회원 정보가 있다면 회원 개체를 소멸하면 될 것입니다. 그리고 인덱스 연산을 통해 해당 id의 value값을 0으로 변경해 주어야겠지요.

```
delete member;
```

```
base[id] = 0; //map의 인덱스 연산으로 보관된 요소의 값을 변경도 가능
cout<<"삭제하였습니다."<<endl;
```

회원 정보를 찾아 보여주는 SearchMem 메서드를 구현해 봅시다. 여기에서도 입력한 id에 해당하는 보관된 회원 개체를 얻어오는 과정은 같습니다. 단지, 찾은 회원의 정보를 보여주는 부분만 다르겠죠.

```
cout<<"검색할 회원 아이디를 입력하세요."<<endl;
string id = ehglobal::getstr();
```

//map의 인덱스 연산을 이용하여 키(id)에 해당하는 값(회원 위치 정보)를 얻어올 수 있음
//키(id)에 해당하는 값이 없을 때 pair<id,0>을 보관한 후에 0을 반환

```
Member *member = base[id];
if(member == 0) //id에 해당하는 회원 정보가 없을 때
{
    cout<<"존재하지 않는 아이디입니다."<<endl;
    return;
}
cout<<member->GetId()<<"."<<member->GetName()<<endl;
```

전체 회원의 정보를 보여주는 ListAll에서는 map의 begin 메서드를 이용해 얻어온 반복자와 end 메서드를 호출해 얻어온 반복자 사이에 있는 회원 정보를 보여주면 될 것입니다. 주의할 사항은 회원 개체 값이 0인 것은 개발자 의도에 의해 보관된 것이 아니므로 필터링을 해야 합니다.

```
MemIter seek = base.begin();
MemIter end = base.end();
Member *member = 0;

for( ; seek != end ; ++seek)
{
    member = (*seek).second; //보관된 요소의 second는 값(회원 위치 정보)
    if(member)
    {
        cout<<member->GetId()<<"."<<member->GetName()<<endl;
    }
}
```

마지막으로 회원 관리자의 소멸자에서 생성했던 모든 회원 개체를 소멸해 주어야 합니다. ListAll처럼 필터링을 해야겠지요.

```
MemIter seek = base.begin();
MemIter end = base.end();
Member *member = 0;

for( ; seek != end ; ++seek)
{
    member = (*seek).second; //보관된 요소의 second는 값(회원 위치 정보)
    //인덱스 연산을 사용했으므로 값이 0인 요소가 있을 수 있으므로 필터링
    if(member) //실제 회원 위치 정보가 있을 때
    {
        delete member;
    }
}
```

다른 부분은 앞에서 구현했던 프로그램과 같아서 설명을 생략할게요.

이 책에서는 보다 자세한 문법 사항이나 구현 방법 및 설계 과정을 다루지 않았습니다. 자세한 내용을 원하시면 [IT 전문가로 가는 길 Escort C++], [IT 전문가로 가는 길 Escort 자료구조와 STL]을 참고하세요.