기초 C++ 프로그래밍 #1

C++ 프로그래밍 언어

- 1979년, Bjarne Stroustrup의 “C with Classes”의 작업으로부터 시작
  - C의 장점: 범용성, 빠른 속도, 높은 보급률
  - Simula의 특성: 최초의 객체 지향 프로그래밍 언어로서, 객체, 클래스, 상속 등의 요소 포함
  - C를 Simula의 특성을 통해 확장하고자 함
- 1983년 C++로 개명됨
- 1998년 국제 표준 제정(ISO/IEC 14882:1998, known as C++98)
- 2011년 8월 최신 국제 표준 개정(ISO/IEC 14882:2011, known as C++11, or C++0x)

객체 지향 프로그래밍

- Object-Oriented Programming(OOP)

- OOP는 컴퓨터 프로그램을 디자인하기 위해, 객체(objects) – 데이터 필드(data fields)와 메소드(methods) 및 이들 간의 상호 작용으로 구성되는 – 를 사용하는 프로그래밍 패러다임이다.

- OOP에서, 객체는 클래스(class)의 특정한 인스턴스(instance)

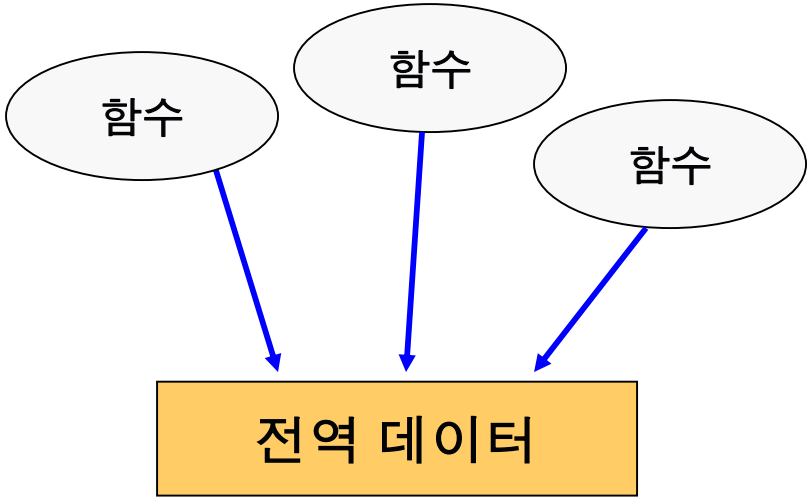
- 연구자들은 대부분의 객체 지향 언어에서, OOP 프로그래밍 스타일을 뒷받침하는 근본적인 특성들을 다음과 같이 확인함
  - 동적 결합(Dynamic Binding)
  - 캡슐화(Encapsulation)
  - 서브타입 다형성(Subtype polymorphism)
  - 상속(Inheritance)
  - …

절차적 프로그래밍 vs. 객체 지향 프로그래밍

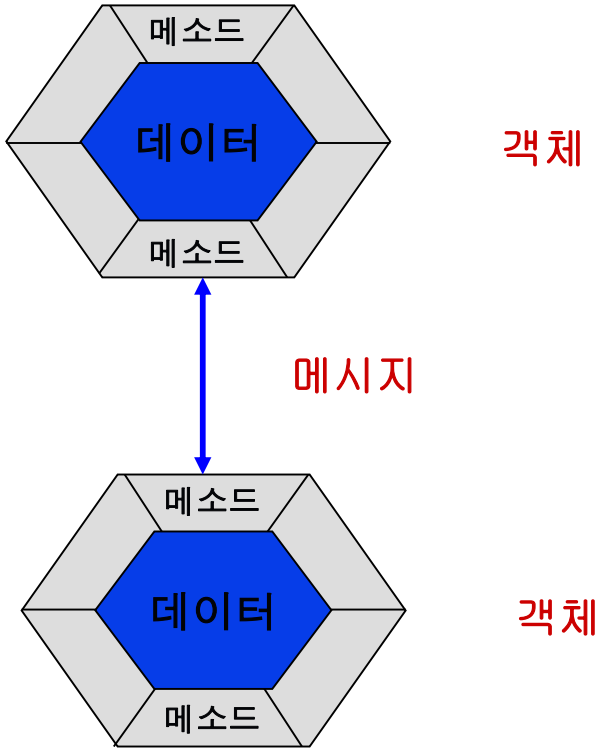
- 전통적인 절차적 프로그래밍(traditional procedural programming)
  - 알고리즘이 우선, 그리고 자료구조는 나중에 고려

- 객체 지향 프로그래밍(object-oriented programming)
  - 자료구조가 우선, 그 후 데이터를 활용하는 알고리즘에 대해 생각함
  - 각각의 객체가 일련의 연관된 작업들의 수행을 전담하도록 함
  - 재사용성을 최대화, 데이터 의존성을 축소, 디버깅 시간을 최소화

절차적 프로그래밍 vs. 객체 지향 프로그래밍



절차적 프로그래밍



객체 지향 프로그래밍

객체지향 프로그래밍의 예

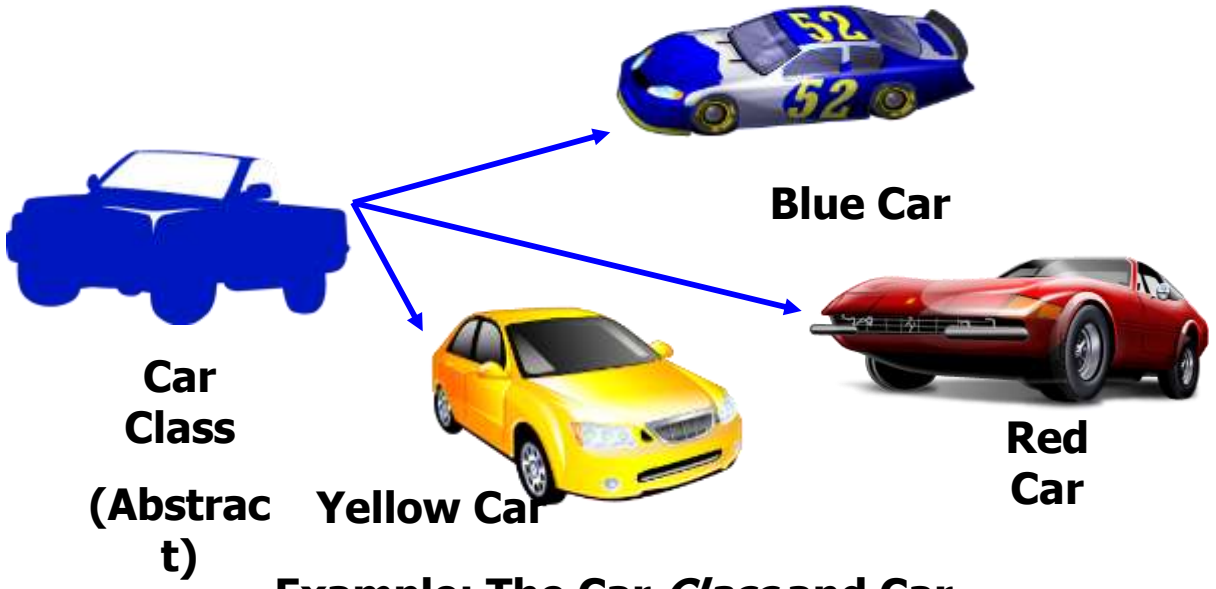
문제: 테트리스 게임을 만들어야 함

- **절차지향 프로그래밍**
  - 테트리스의 블록의 집합을 정의
  - 게임 데이터로 점수가 필요함
  - 블록이 쌓인 것을 표현하는 2차원 배열이 필요함
  - 테트리스 블록을 랜덤하게 생성하는 함수
  - 블록이 떨어지는 것을 계산하는 함수
  - 블록의 이동을 화면에 표현할 함수
  - 블록이 바닥에 도착했는지를 판단하는 함수
  - 특정 라인이 블록으로 가득 찼는지를 판단하여 지우는 함수
  - 특정 라인이 지워진 후 위의 내용을 한칸 내리는 함수
  - 사용자의 키보드 입력을 받아 블록을 움직이는 함수
- **프로그램 = 데이터 + 함수**

- **객체지향 프로그래밍**
  - 크게 블록 생성기와 게임 엔진, UI로 프로그램을 분해
  - 블록 생성기의 속성으로서 현재 블족의 종류를 정의
  - 블록 생성기의 행위로서 블록을 랜덤하게 생성하는 함수 정의
  - 게임 엔진의 속성으로서 점수, 블록의 상태를 정의
  - 게임 엔진의 행위로서 블록 생성기에 새로운 블록 요청, 블록의 떨어짐, 바닥에 도달, 라인의 꽉참, 지워진 칸 위를 내리는 등의 기능을 정의
  - UI의 속성으로서 현재 화면의 픽셀정도를 정의
  - UI의 행위로서 게임엔진에 현재 상태 요청, 화면에 새로 그리기 등의 기능을 정의
- **프로그램 = 객체 + 객체 + …**

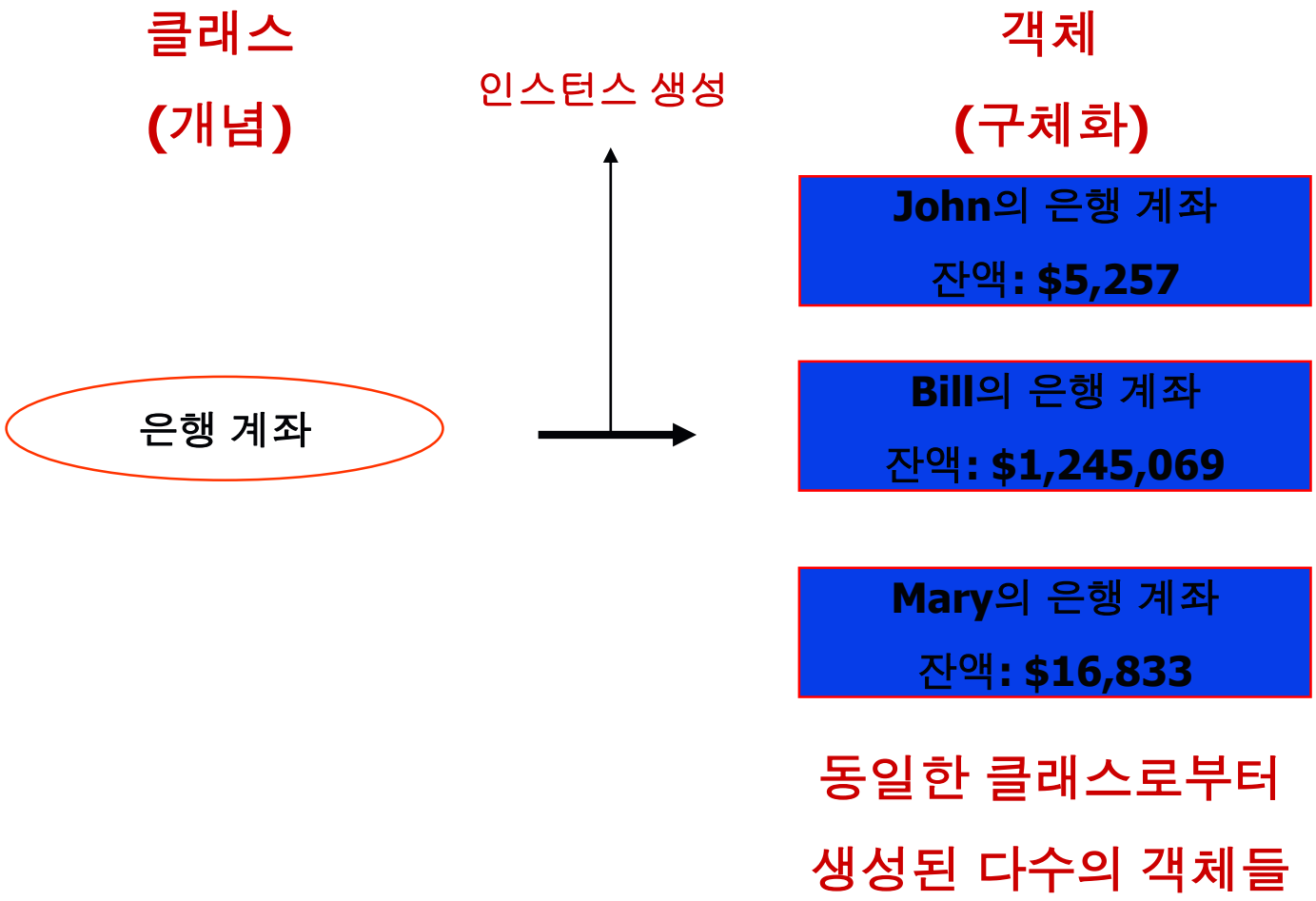
클래스와 객체

- 객체는 두 가지 구성요소를 지님
  - 상태(State): 객체가 가지고 있는 속성 또는 특성; 데이터, 속성
  - 행동(Behavior): 객체가 가지고 있는 행동 또는 할 수 있는 반응 양식; 메소드, 오퍼레이션
- OOP에서, 클래스는 그 자신의 인스턴스를 만들기 위하여 청사진(또는 형틀)으로 사용되는 구조물이라 할 수 있음



Example: The Car *Class* and Car *Instances*

클래스와 객체


클래스
(개념)
인스턴스 생성
객체
(구체화)
John의 은행 계좌
잔액: $5,257
은행 계좌
Bill의 은행 계좌
잔액: $1,245,069
Mary의 은행 계좌
잔액: $16,833
동일한 클래스로부터
생성된 다수의 객체들

클래스의 예

클래스: 자동차

데이터:

- 제조사명
- 모델명
- 제조 날짜
- 색상
- 문의 개수
- 엔진 크기
- **etc.**

메소드:

- 데이터 항목 정의(제자사명, 모델명, 제조 날짜 등등 명세)
- 데이터 항목 변경(색상, 엔진 등등)
- 데이터 항목 출력
- 비용 계산
- **etc.**

C++의 기초

- C++ 프로그래밍의 기초 개념을 공부하고, 그 내용을 바탕으로 유명한 자료구조 중 하나인 스택 자료구조가 C++로 어떻게 구현되는지 살펴본다.

- 학습할 내용
  - C++의 프로그래밍 패러다임
  - C++의 표준 입출력
  - C++에서의 동적 메모리 할당
  - C++의 참조 연산자
  - C++에서의 클래스
  - 접근 지정자, 생성자, 소멸자
  - C++로 구현한 스택

C++의 프로그래밍 패러다임

- 전역 함수는 main() 함수 하나만 필요함
- 모든 프로그램의 수행은 클래스에서 생성된 객체들 사이의 메시지 전달로 수행됨: 메시지의 세 가지 구성 요소는 1) 메시지가 전달될 객체, 2) 수행하고자 하는 멤버함수의 이름, 3) 그 멤버함수가 수행되는 데 필요한 인자
- Ex)

```
// C++ 표준 헤더는 확장자 .h를 붙이지 않는다.
#include <iostream>
#include <string>

// sample 클래스 선언
class sample{
private:
 int value;
public:
int getValue(){ return value; };
void setValue(int param)
{ value=param; };
};
```

```
int main(){
// sample 클래스의 인스턴스 생성
sample obj;

// C언어에서는 함수 호출을 통해 작업을 수행함
printf("함수 호출\n");

// C++에서는 메시지 전달로 작업을 수행함
/* 메시지를 통해 객체에게 메소드를 호출하여 줄
것을 요청하며, 객체가 알아서 스스로 동작함
(내부적인 활동을 외부에서 알 수 없음, Encapsulation)
*/
obj.setValue(10);
// 메시지가 전달될 객체: obj
// 수행하고자 하는 멤버함수: setValue()
// 멤버함수가 수행되는 데 필요한 인자: 10
}
```

C++에서의 표준 입출력

- C++에서의 표준 입출력은 I/O stream을 이용하여 수행됨
  - stream이란 데이터의 연속적인 흐름을 의미함
- 표준 입력은 cin 객체로 수행됨
  - Ex)
```
int row, col;
scanf("%d %d", &row, &clo);// C
cin >> row >> col;              // C++
```
  - 주소를 넘기지 않아도 ">>"가 참조(reference type)를 받도록 재정의(overloading)되어 있으므로 변수에 값이 저장됨
- 표준 출력은 cout 객체로 수행됨
  - Ex)
```
int var1, var2;
printf("%d %d", var1, var2); // C
cout << var1<< var2;          // C++
```
  - cout.setf()를 이용하여 printf()의 에서의 formatting은 cout.setf()명령어를 이용하여 수행됨
    - 자세한 내용은 reference manual 참고

C++에서의 동적 메모리 할당

- C++에서는 new와 delete 이용

```
int *var1;
// C++
var1 = new int; // 메모리 할당
delete var1;      // 메모리 해제
// C
var1 = (int*)malloc(sizeof(int));
free(var1);
```

- 1차원 배열 동적 메모리 할당

```
int *var1;
// 메모리 할당
var1 = new int[size];
// 메모리 해제
delete [] var1;
```

- 2차원 배열 동적 메모리 할당

```
int **var1;
// 메모리 할당
var1 = new int*[row];
for( i = 0; i < row; i++)
    var1[i] = new int[col];
...
// 메모리 해제
for( i = 0; i < row; i++)
    delete [] var1[i];
delete [] var1;
```

C++의 참조 연산자(Reference Operator)

- & 연산자가 C++에서 확장됨
- 참조 연산자는 포인터와 달리 별도의 메모리 공간을 차지하지 않으며, 객체를 지칭하는 또 다른 이름처럼 사용됨

```
// 포인터 이용
void swap(int *a, int *b){
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
swap(&a, &b);
```

```
// 참조 연산자 이용
void swap(int &a, int &b){
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
swap(a, b);
```

- 참조 연산자의 사용으로 함수간 인자 전달에서 포인터 연산이 사라짐
- 리턴 타입도 참조 연산으로 지정 가능

C++에서의 클래스

- C++에서의 클래스 선언과 구현은 분리되어 있다.
- 클래스의 선언

```
class <클래스이름>{
접근 지정자:
        멤버(멤버 변수 또는 멤버 함수)
        …
};
```

- 클래스의 구현
  - 클래스 선언의 멤버 함수들을 구현함

```
#include "클래스이름.h"
리턴 타입<클래스이름>::<멤버 함수 이름>(자료형 인자1, 자료형 인자2, …){
        code;
}
```

- 일반적으로 <클래스명>.h파일에서 클래스 선언, <클래스명>.cpp파일에서 클래스 구현
- #include "클래스명.h"으로 헤더를 포함하여 클래스 사용

접근 지정자, 생성자, 소멸자

- 접근 지정자는 객체의 멤버를 외부에서 접근(다른 객체나 main()함수 등)할 때 어떤 내용을 외부로 공개할지를 결정한다.

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

  - 일반적으로 멤버 변수는 private으로, 멤버 함수는 public으로 지정한다. (지정하지 않았을 경우에는 기본적으로 private로 설정)
- 생성자(constructor)
  - 클래스로부터 객체가 생성될 때 자동으로 호출되는 함수
  - 일반적으로 객체의 초기화를 수행한다.
  - 생성자는 반드시 클래스의 이름과 같아야 하며, 리턴 타입을 갖지 않는다.
- 소멸자(destructor)
  - 클래스로부터 생성된 객체가 소멸될 때 자동으로 호출되는 함수
  - 일반적으로 메모리의 해제 등에 사용된다.
  - 반드시 하나만 존재하며 ~<클래스이름>(); 의 형태로 선언된다.

접근 지정자, 생성자, 소멸자의 예

```
#include <iostream>
#include <string>
using namespace std;
/* cin, cout, endl 등과 같은 명령어들은 표준 라이브러리에 속하는데, 이들은 std라는 namespace안에 존재하므로 명령어 사용시 std안에 있는 것을 사용할 것임을 명시해 주는 구문이다. 이것이 없으면 "std::cout<<std::endl;" 과 같이 사용 */

// student 클래스 선언
class student
{
    private: // 객체 외부에서는 접근불가
        int id;
        char *name;
    public: // 객체 외부에서는 접근가능
        student(int stId, char *stName); // 생성자
        ~student(); // 소멸자
};
```

접근 지정자, 생성자, 소멸자의 예

```
// 생성자
student::student(int stId, char *stName){
    id = stId;
    int tmp = strlen(stName);
    name = new char[tmp+1]; /* 메모리 할당. 일반적으로 생성자에서는 변수 초기화만 처리하는
                               것이 좋다. 인스턴스가 내부적으로 완전한 상태가 아님. */
    strcpy(name, stName);
    cout<<"Student 생성"<<endl;
}

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

int main(){
    student *s = new student(10,"홍길동"); /* 인자가 int와 char[]인 생성자로 객체가 생성 된 후
                                              포인터 s에 연결 */
    delete s; // s에 할당된 객체가 제거되면서 소멸자가 자동으로 호출됨.
    return 0;
}
```

C++로 구현한 스택(stack.h)

```
#define MAX_SIZE 100

typedef int Item;                    // int를 스택의 자료 단위로 지정

Struct Node{
        Item item;                   // 스택의 각 노드에 들어갈 데이터
        struct Node *next;           // 다음 노드를 가리키는 포인터
}

class Stack{
private:  // 외부에서 직접 접근 불가
        Node *top;                   // 스택의 top을 지정하는 포인터
        int MaxSize;                 // 스택의 최대 크기
        int currentSize;  // 스택의 현재 크기
public:  // 외부에서 직접 접근 가능
        Stack();                     // 디폴트 생성자(스택의 크기가 최대 크기로 설정됨)
        Stack(int);                  // 스택의 최대 크기를 사용자가 정할 수 있는 생성자
        ~Stack();                    // 소멸자
        bool isEmpty(void) const;    // 스택이 비었는지를 판단하는 멤버 함수
        bool isFull(void) const;     // 스택이 가득 찼는지를 판단
        int stackCount(void) const;  // 스택에 노드가 몇 개 있는지를 판단
        bool push(const Item &item);  // 스택에 새로운 노드를 삽입
        bool pop(Item &item);        // 스택의 top에서 하나의 노드를 가져옴.
};
```

C++로 구현한 스택(stack.cpp)

```
#include "stack.h"


Stack::Stack(){
        currentSize=0;
        MaxSize=MAX_SIZE;
        top = NULL;
}
Stack::Stack(int maxStackSize){
        currentSize = 0;
        MaxSize=maxStackSize;
        top = NULL;
}
Stack::~Stack(){
        Item i;
        while(isEmpty()){
                pop(i);
        }
}
```

생성자에서 변수 초기화 수행

소멸자에서 객체가 소멸될 때 처리할 작업을 수행

(메모리에 할당된 아이템들을 모두 없애는 작업을 수행함)

```
bool Stack::push(const Item &item){
    if(isFull()) return false;
    Node *add = new Node;
    if(!add) return false;
    add->item = item;
    add->next = NULL;
    currentSize++;
    if(!top) top = add;
    else{
        add->next = top;
        top = add;
    }
    return true;
}

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

새로 노드를 할당한 후에 현재의 top 변수가 가리키는 노드를 그 뒤에 연결하고, 새로 할당한 노드를 top 변수가 가리키게 함

top의 바로 다음 노드를 새로운 top으로 변경하고 이전 top의 노드를 할당 해제

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

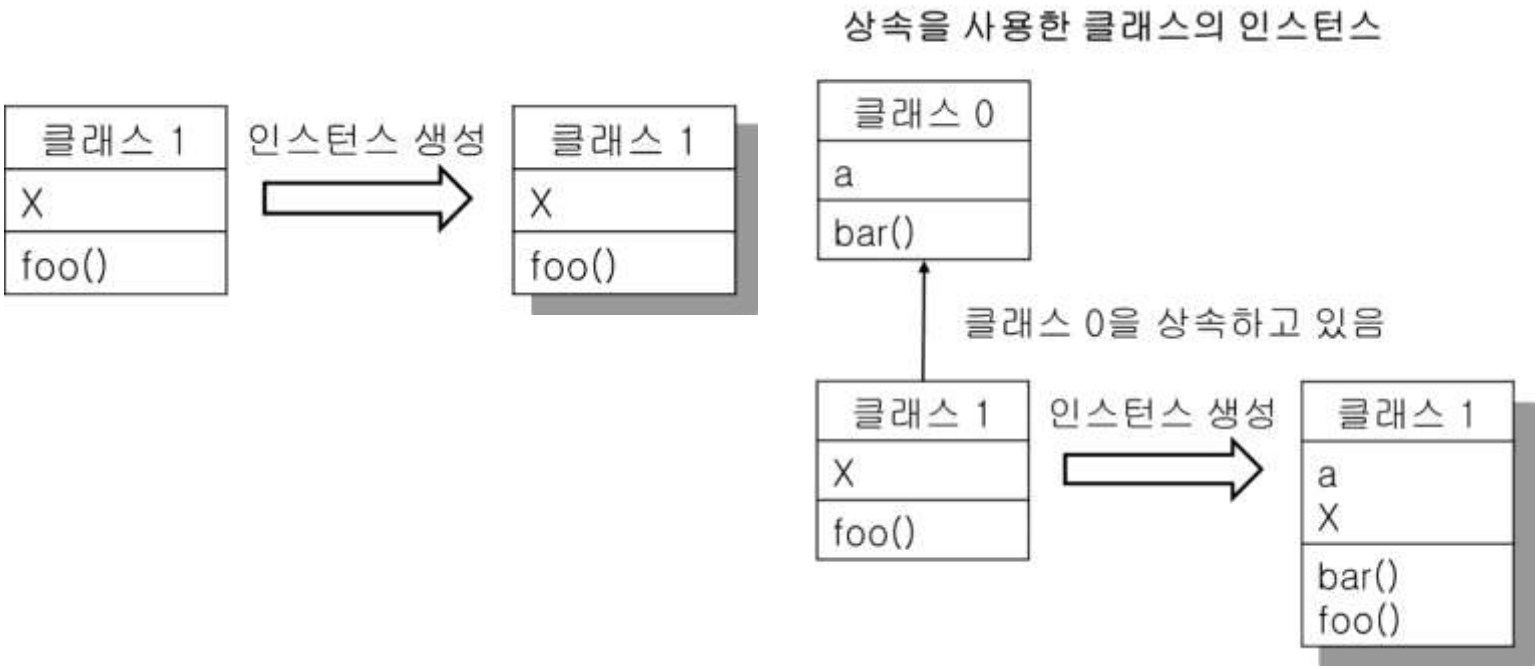
C++의 객체 지향적 특성

- (다시 3페이지의 OOP 설명으로 돌아가서,) 연구자들은 대부분의 객체 지향 언어에서, OOP 프로그래밍 스타일을 뒷받침하는 근본적인 특성들을 다음과 같이 확인함
  - 동적 결합(Dynamic binding)
  - 캡슐화(Encapsulation)
  - 서브타입 다형성(Subtype polymorphism)
  - 상속(Inheritance)
  - …
- 1주차에서는, C++에서 지원하는 객체 지향의 중요한 특성 중 하나인 상속(Inheritance)에 대해서 공부하고 실습을 통해 개념을 이해함
  - 캡슐화 및 서브타입 다형성은 다음 시간에 공부함

상속

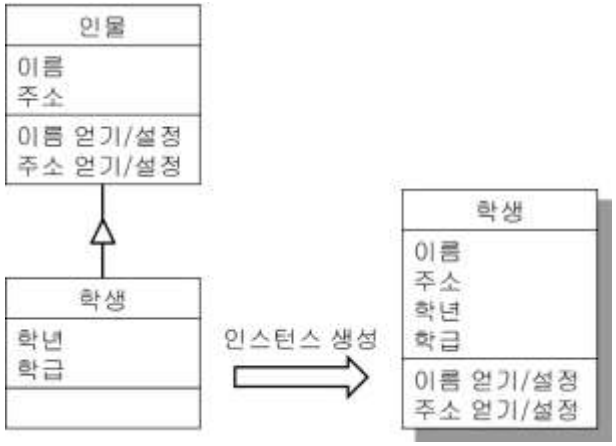
- 상속은 객체 지향에서 가장 혁신적인 발명이라고 일컬어지는 개념

- 클래스가 인스턴스를 생성할 때, 다른 클래스의 속성/오퍼레이션을 빌려와서 자신이 갖고 있는 것과 합친 후 하나의 인스턴스를 생성하는 것



상속

- 상속은 “is-a” 관계를 구현하는 경우에 사용
  - Ex> “학생 is a 인물”

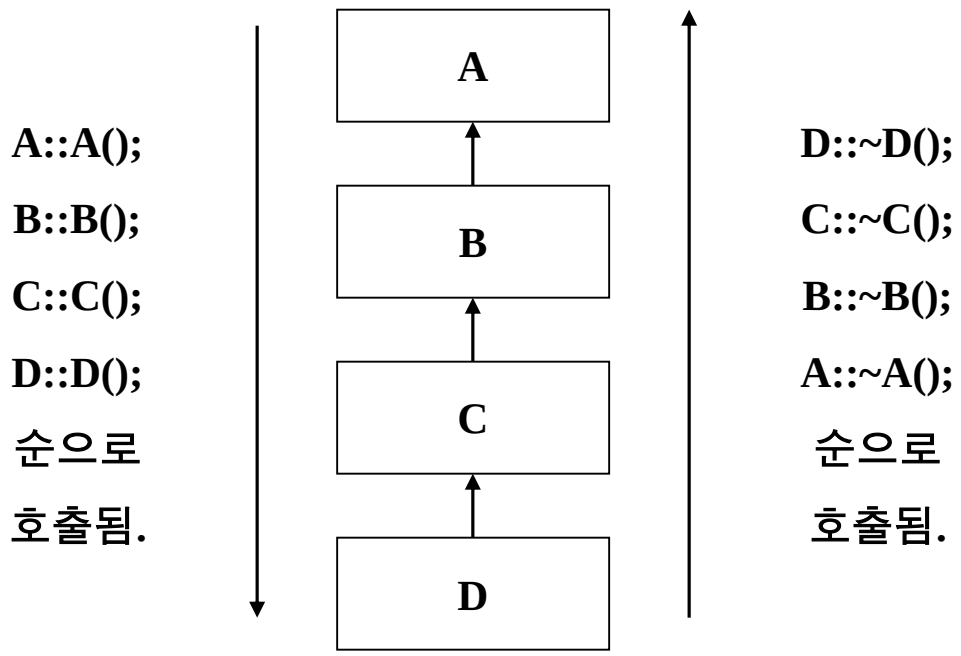



```
class Person { … };
class Student : public Person {…};
class Teacher : public Person {…};
```

  - 인물: 기반 클래스(Base Class), 학생: 파생 클래스(Derived Class)
  - 기반 클래스는 파생 클래스로부터 독립됨
    - 파생 클래스들에 대한 공통 기반 클래스를 미리 만들어 두면 파생 클래스의 공통된 부분에 대한 분석, 설계, 구현, 테스트, 디버그, 유지관리의 필요성이 사라짐
    - 아무리 클래스를 파생시키더라도(예: 인물의 파생 클래스 교사) 기반 클래스에는 영향이 없음

상속이 되었을때 생성자, 소멸자 호출 순서

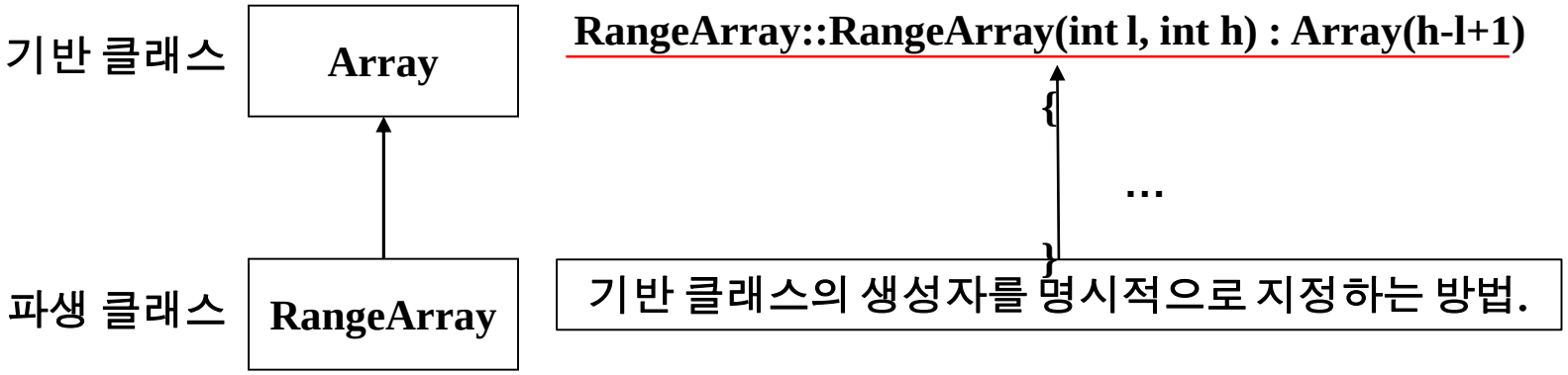
- 파생 클래스로부터 객체가 생성될 때는 상속 계층 구조 상의 모든 생성자가 호출되고, 소멸될 때는 모든 소멸자가 호출됨.
- Ex) D class로 객체를 생성한다면



상속이 되었을때 생성자, 소멸자 호출

- 파생 클래스의 생성자는 자동적으로 기반 클래스의 디폴트 생성자(default constructor, 인자가 없는 생성자)를 호출한다.

- 인자가 있는 생성자를 따로 정의하면 디폴트 생성자는 없어진다. 이 때 만약 기반 클래스에서 디폴트 생성자가 없는 체로 파생 클래스에서 생성자를 호출하면 존재하지도 않는 디폴트 생성자를 호출하게 되어 에러가 발생한다. 이를 해결하기 위해 기반 클래스에 디폴트 생성자를 추가해 주거나, 또는 기반 클래스의 생성자를 명시적으로 호출해 주어야 한다. (아래는 CPP-1에서의 예)



CPP-1: RangeArray

- RangeArray 클래스는 배열이 인덱스 0에서 시작하는 것이 아닌, 생성자에 전달된 임의의 범위를 갖는다.
  - EX> RangeArray A(-10, 10); // 인덱스 -10~10을 가지고 , 21개 원소 저장

- C++에서 제공하는 다른 기본 자료형(int, float, double…)과 마찬가지로, “[ ]”연산자를 사용하여 배열의 값을 얻거나 또는 배열에 값을 저장할 수 있도록 설계한다.
  - [ ] 연산자에 대한 연산자 다중정의가 필요
  - 배열에 원소를 삽입하는 경우(등호의 왼쪽에 연산자를 사용, left value), 원소가 저장되는 메모리 영역의 참조(Reference)를 반환하도록 해야 함
  - 배열의 값을 반환하는 경우(등호의 오른쪽에 연산자를 사용, right value), 내부 저장소에서 원소의 값을 찾아서 반환하도록 하여야 함

- 임의의 인덱스를 처리하게 하는 것이 어려우므로, 일단 인덱스 0부터 시작하는 Array 클래스를 완벽하게 작성한 후 이를 상속하는 RangeArray 클래스를 작성함

다중정의(Overloading)

- 본 실습에 필요한 연산자 다중정의를 위해 다중정의에 대해 알아보자.
- 다중정의는 크게 함수 다중정의(function overloading)와 연산자 다중정의(operator overloading)로 구분된다.
- 함수 다중정의는 함수의 이름은 같고, 인자의 개수나 자료형이 다른 함수들을 프로그램이 자동으로 구분하는 것이다.
  - Ex)
    - int func(int a, int b); int func(float a, float b);
      위의 함수는 이름은 같지만 서로 다른 함수다. 사용자가 입력하는 인자의 자료형에 따라 다른 함수가 호출된다.
    - int func(int a, int b); float func(int a, int b);
      위의 경우는 컴파일 에러를 유발한다. 함수 다중정의는 리턴 자료형만 다른 것으로는 구현 불가능
- 연산자 다중정의는 C++에서 사용되는 특정 객체에 적절한 연산을 수행하도록 연산자에 또 다른 의미를 부여하는 것이다.
  - Ex) cin, cout
    - cin>>a; 처럼 cin객체를 위해 '>>' 연산자를 추가로 구현하고 있다.
    - 이를 위해서는 다음과 같이 정의해야 한다.
      <리턴 자료형> <클래스이름>::operator <연산자>(인자1, 인자2...) { ... }

함수 다중정의의 예

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

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

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

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

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

두 함수의 이름은 같지만 인자의 자료형이 다르다.

int add(int, int)가 호출됨

double add(double, double)가 호출됨

연산자 다중정의의 예

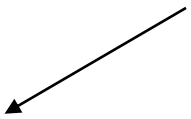
```
char string::operator[](int i)
{
        return s[i];
}
```

string 클래스의 멤버로 [] 연산자를 다중정의함으로써, '객체의 인스턴스명[int형값]'의 형태로 사용할 때 s의 인덱스의 값을 char형으로 리턴한다.

Ex> string ss("hello");

char ichar = ss[1]; // 'e'가 ichar에 저장

```
string& string::operator=(string& str)
  {
        strcpy(s,str);
        return *this;
  }
```



string 클래스의 멤버로 = 연산자를 다중정의함으로써, '객체의 인스턴스명=string객체의 레퍼런스'의 형태로 사용할 때 그 값을 내부 데이터로 복사하고 객체 자신의 포인터를 리턴한다.

Ex> string ss("hello"); string ss2("bye");

ss = ss2; // "bye"가 ss에 복사됨

this: 멤버 함수를 호출한 객체에 대한 포인터
즉, 객체 자기 자신에 대한 포인터

Array를 이용한 RangeArray의 구현

- Array를 구현한 후 RangeArray는 Array를 상속하여 필요한 부분만 코딩하고 나머지는 Array의 멤버 변수와 멤버 함수를 그대로 이용한다.
  - 예를 들어, RangeArray의 인스턴스를 인덱스 -10~10으로 생성하였다면, 이를 Array의 멤버 변수에 인덱스 0~20으로 저장하고, 대신에 값을 저장하거나 얻는 경우에 RangeArray의 인스턴스를 생성시 입력받은 인덱스 값을 이용하여 참조하면 된다. (-7 인덱스의 값을 얻고자 할 경우 Array의 변수 데이터에서 인덱스 7의 값을 사용하도록 함)
- '[ ]'의 경우 연산자 다중정의를 사용해야 한다. (29, 31쪽 참조)
- 생성자의 연결구조를 반드시 고려해야 한다.
  - 파생 클래스에서 생성자를 호출할 경우 기반 클래스의 생성자를 호출할 수 있도록 명시적으로 지정해 주어야 한다. (26쪽 참조)
- 다음 두 슬라이드에 있는 선언을 기초로 작성한다.

Array를 이용한 RangeArray의 구현 (샘플 프로그램과 출력)

```
int main(){
    int i,x,y;
    Array a(10), b(5);
    for (i=0; i<a.length(); i++) a[i] = i + 1;
    for (i=0; i<b.length(); i++) b[i] = i * 2;
    cout << "a(10)"; a.print();
    cout << "b(5)"; b.print();
    cout << "a[-1]"; a[-1] = 7;
    x = a[0]; y = b[0];
    cout << "a[0]=" << x << "b[0] = " << y << endl;

    RangeArray c(-1,3), d(3,7);
    for (i=c.baseValue(); i<=c.endValue(); i++) c[i] = i * 3;
    for (i=d.baseValue(); i<=d.endValue(); i++) d[i] = i * 4;
    cout << "c(-1,3)"; c.print();
    cout << "d(3,7)"; d.print();
    cout << "c[-2]"; c[-2] = 3;
    x = c[-1]; y = d[3];
    cout << "c[-1] = " << x << "d[3] = " << y << endl;
}
```

출력:

```
a(10) [1 2 3 4 5 6 7 8 9 10]
b(5) [0 2 4 6 8]
a[-1] Array bound error!
a[0] = 1  b[0] = 0
c(-1,3) [-3 0 3 6 9]
d(3,7) [12 16 20 24 28]
c[-2] Array bound error!
c[-1] = -3  d[3] = 12
```

Array.h

```
#ifndef __ARRAY__
#define __ARRAY__
class Array{
        protected:
                int *data;
                int len;

        public:
                Array(int size);
                ~Array();

                int length();

                int& operator[](int i);
                int operator[](int i) const;

                void print();
};
#endif
```

RangeArray.h

```
#include “Array.h"

class RangeArray : public Array{
        protected:
                int low;
                int high;

        public:
                RangeArray(int, int);
                ~RangeArray();

                int baseValue();
                int endValue();

                int& operator[](int);
                int operator[](int) const;
};
```

3주차 실습 안내
(C++ Programming)

3주차 실습

- CPP-1 Range Array 해결
- 세부 사항은 강의자료 참조.