

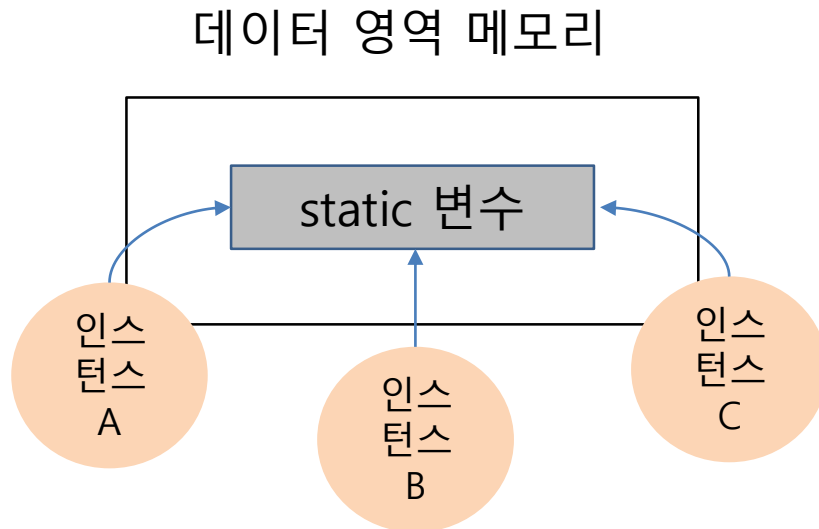
# C++\_클래스2, STL – vector(벡터)

*Visual Studio 2022*

# static 변수

## ▪ static 변수의 정의와 사용 방법

- 다른 멤버변수처럼 인스턴스가 생성될 때마다 새로 생성되는 변수가 아니다.
- 프로그램이 실행되어 메모리에 적재(load)될때 메모리 공간이 할당된다.
- 여러 개의 인스턴스가 같은 메모리의 값을 공유하기 위해 사용



Static 예약어

```
static int serialNum=1000;
```

# static 변수

## ■ 카드번호 자동 발급

```
class Card {  
private:  
    static int serialNum; //static 변수  
    string name;         //고객 이름  
    int cardNumber;      //카드 번호  
  
public:  
    /*Card(string name) {  
        serialNum++;  
        cardNumber = serialNum;  
        this->name = name;  
    }*/  
  
    Card(string name) : name(name), cardNumber(++serialNum) {}  
  
    string getName(){  
        return name;  
    }  
    int getCardNumber() {  
        return cardNumber;  
    }  
};
```

# static 변수

## ■ 카드번호 자동 발급

```
int Card::serialNum = 1000; //전역 변수

int main()
{
    Card card1("신유빈");
    cout << "고객 이름: " << card1.getName() << endl;
    cout << "카드 번호: " << card1.getCardNumber() << endl;

    Card card2("이정후");
    cout << "고객 이름: " << card2.getName() << endl;
    cout << "카드 번호: " << card2.getCardNumber() << endl;

    Card card3("한강");
    cout << "고객 이름: " << card3.getName() << endl;
    cout << "카드 번호: " << card3.getCardNumber() << endl;

    return 0;
}
```

```
고객 이름: 신유빈
카드 번호: 1001
고객 이름: 이정후
카드 번호: 1002
고객 이름: 한강
카드 번호: 1003
```

# static 멤버 함수

## ▪ Math 클래스

```
class Math {  
public:  
    static int abs(int x) { //절대값 함수  
        return (x < 0) ? -x : x;  
    }  
  
    static int max(int x, int y) { //큰 수 선택  
        return (x > y) ? x : y;  
    }  
  
    static int min(int x, int y) { //작은수 선택  
        return (x < y) ? x : y;  
    }  
};
```

# static 멤버 함수

## ▪ Math 클래스

```
int main()
{
    //객체(인스턴스)를 생성하지 않음
    /*Math math1;
    cout << math1.abs(-3) << endl;*/

    //클래스 이름으로 직접 접근(범위 연산자)
    cout << "-3의 절대값: " << Math::abs(-3) << endl;
    cout << "10과 20중 큰수: " << Math::max(10, 20) << endl;
    cout << "10과 20중 작은수: " << Math::min(10, 20) << endl;

    return 0;
}
```

```
-3의 절대값 : 3
10과 20중 큰수 : 20
10과 20중 작은수 : 10
```

# 객체의 동적 생성 및 반환

- 동적 메모리 할당

- 프로그램 실행 중에 필요한 메모리의 크기를 결정
- 시스템은 힙(heap)이라는 공간을 관리하고 있는데, 프로그램에서 요청하는 공간을 할당하여 시작 주소를 알려준다.
- 할당된 시작 주소는 반드시 어딘가에 저장되어야 하고 이때 포인터가 사용됨
- 할당시 **new** , 해제시 **delete** 사용

- 동적 객체 생성

```
Car* car1 = new Car()
```

- 동적 객체 반환

```
delete car1;
```

# 객체의 동적 생성 및 반환

## ■ 기본 생성자로 생성

```
//Car 클래스 정의
class Car{
private:
    string model;
    int year;

public:
    //기본 생성자
    Car() {}

    //멤버 함수
    void setModel(string model);
    void setYear(int year);
    void carInfo();
    void drive();
};
```



# 객체의 동적 생성 및 반환

## ■ 기본 생성자로 생성

```
void Car::setModel(string model) {  
    this->model = model;  
}  
  
void Car::setYear(int year) {  
    this->year = year;  
}  
  
void Car::carInfo() {  
    cout << "모델명: " << this->model << endl;  
    cout << "년식: " << this->year << endl;  
}  
  
void Car::drive() {  
    cout << "차가 달립니다.\n";  
}
```

# 객체의 동적 생성 및 반환

## ■ 기본 생성자로 생성

```
int main()
{
    //동적 객체 생성
    Car* car1 = new Car();
    car1->setModel("Sonata");
    car1->setYear(2017);
    car1->carInfo();
    car1->drive();

    cout << "-----\n";

    Car* car2 = new Car();
    car2->setModel("EV3");
    car2->setYear(2024);
    car2->carInfo();
    car2->drive();

    delete car1; //객체 반환
    delete car2;

    return 0;
}
```

# 객체의 동적 생성 및 반환

## ▪ 매개변수가 있는 생성자로 생성

```
class Car{
private:
    string model;
    int year;

public:
    //생성자 - this로 초기화
    Car(string model, int year) {
        this->model = model;
        this->year = year;
    }
    //멤버 함수
    void carInfo();
    void drive();
};

void Car::carInfo() {
    cout << "모델명: " << this->model << endl;
    cout << "년식: " << this->year << endl;
}

void Car::drive() {
    cout << "차가 달립니다.\n";
}
```

# 객체의 동적 생성 및 반환

## ▪ 매개변수가 있는 생성자로 생성

```
int main()
{
    //동적 객체 생성
    Car* car1 = new Car("Sonata", 2017);
    car1->carInfo();
    car1->drive();

    cout << "-----\n";

    Car* car2 = new Car("EV3", 2024);
    car2->carInfo();
    car2->drive();

    delete car1; //객체 반환
    delete car2;

    return 0;
}
```

# 객체 배열의 동적 생성 및 반환

- 동적 객체 배열 생성

```
Car* cars = new Car[3]
```

- 동적 객체 배열 반환

```
Delete [ ] cars;
```

# 객체 배열의 동적 생성 및 반환

## ■ 기본 생성자로 생성

```
int main()
{
    //동적 객체 배열 생성
    Car* cars = new Car[3];
    //인덱스로 저장
    cars[0].setModel("Sonata");
    cars[0].setYear(2017);
    cars[1].setModel("Ionic6");
    cars[1].setYear(2023);
    cars[2].setModel("EV3");
    cars[2].setYear(2024);
```

```
//전체 출력
for (int i = 0; i < 3; i++) {
    cars[i].carInfo();
    cars[i].drive();
    cout << "-----\n";
}

delete [] cars; //동적 객체 배열 반환

return 0;
}
```

# 객체 배열의 동적 생성 및 반환

- 매개 변수가 있는 생성자로 생성

```
int main()
{
    //객체 배열 생성
    Car* cars = new Car[3]{
        Car("Sonata", 2017),
        Car("Ionic6", 2023),
        Car("EV3", 2024)
    };

    //인덱싱 조회
    /*cars[0].carInfo();
    cars[1].carInfo();
    cars[2].carInfo();*/
}
```

```
//전체 출력
for (int i = 0; i < 3; i++) {
    cars[i].carInfo();
    cars[i].drive();
    cout << "-----\n";
}

delete [] cars; //객체 배열 반환

return 0;
}
```

# 템플릿- 함수 템플릿

## ■ 템플릿(template) 이란?

- 템플릿은 함수나 클래스 코드를 찍어내듯이 생산할 수 있도록 일반화 (generic) 시키는 도구이다.
- 함수 템플릿은 함수 내에서 사용하는 자료형을 일반화된 유형으로 정의하여 그 함수를 호출할때 적절한 자료형을 대입해서 사용

## • 템플릿 선언과 제네릭 타입

템플릿을 선언할 때는 **template** 키워드를 사용한다.

**template** <**typename** 일반화 유형 이름>

**template**<typename T>

T는 임의의 데이터  
형식(자료형)



# 템플릿- 함수 템플릿

## ■ 템플릿(template) 예제

```
template<typename T>
T maxVal(T data1, T data2) {
    if (data1 > data2)
        return data1;
    else
        return data2;
}
int main()
{
    cout << "정수 비교 결과: " << maxVal(10, 20) << endl;
    cout << "실수 비교 결과: " << maxVal(3.3, 2.2) << endl;
    cout << "문자 비교 결과: " << maxVal('a', 'b') << endl;

    return 0;
}
```

# 템플릿- 함수 템플릿

## ■ 템플릿(template) 예제

```
template<typename T>
T dataSum(T data1, T data2){
    return data1 + data2;
}
int main()
{
    int n1 = 3, n2 = 5;
    string str1 = "Hello, ", str2 = "World";
    char ch1[] = "apple, ", ch2[] = "banana";

    cout << "정수형 데이터 합: " << dataSum(n1, n2) << endl;
    cout << "문자형 데이터 합: " << dataSum(str1, str2) << endl;
    //명시적 형변환
    cout << "문자형 배열의 합: " << dataSum<string>(ch1, ch2) << endl;

    return 0;
}
```

배열은 더할 수 없  
으므로 <string>으  
로 형변환 해야함

정수형	데이터	합: 8
문자형	데이터	합: Hello, World
문자형	배열의	합: apple, banana

# STL – 표준 템플릿 라이브러리

C++의 **표준 템플릿 라이브러리(Standard Template Library, STL)**는 다양한 자료구조와 알고리즘들을 미리 만들어서 제공하는 라이브러리를 말한다.

- 컨테이너 : 자료를 저장하는 창고로 **벡터**, 리스트, 큐, 맵 등
- 알고리즘 : 탐색이나 정렬과 같은 다양한 알고리즘 제공
- 반복자(iterator) : 컨테이너에 저장된 자료들을 순회하는 객체이다. 포인터와 비슷한 동작을 한다.

□ **컨테이너**는 같은 타입의 여러 객체를 저장할 수 있는 묶음 단위의 데이터 구조이다.  
쉽게 말해서 화물을 싣는 컨테이너 또는 마트에서 물건을 담은 쇼핑카라고 할 수 있음

# 벡터(vector)

## ❖ 벡터(vector)

- vector는 내부에 배열을 가지고 원소를 저장, 삭제, 검색하는 가변 길이 배열을 구현한 클래스이다.
- 정적인 배열의 단점을 보완한 동적 배열로 배열의 크기 변경 및 데이터를 효율적으로 관리.

vector 객체 생성

**vector** <자료형> 객체 이름

삽입 : push\_back()

수정 : vi[0] = 3

# 벡터(vector) --> int형

## ❖ 벡터(vector)

```
#include <iostream>
#include <vector> //vector 컨테이너 사용
#include <string>
using namespace std;

int main()
{
    //여러 개의 정수를 저장할 벡터 생성
    vector<int> vec;

    //정수 추가
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
```

```
3
1
1 2 10
```

```
//리스트의 크기
cout << vec.size() << endl;

//요소 검색
cout << vec[0] << endl;

//2번 인덱스 값 수정
//vec[2] = 10;
vec.at(2) = 10;

//전체 조회
for (int i = 0; i < vec.size(); i++)
{
    cout << vec[i] << " ";
}
```

# 벡터(vector) --> string형

## ❖ 벡터(vector)

```
//여러 개의 문자열을 저장할 벡터 생성
vector<string> list;
string name;
```

```
//저장
list.push_back("jerry");
list.push_back("luna");
list.push_back("han");
list.push_back("elsa");
```

```
//리스트의 크기
cout << list.size() << endl;
```

```
for (int i = 0; i < list.size(); i++)
{
    cout << list[i] << " ";
}
```

```
//최대값 계산
name = list.at(0); //최대값으로 설정
for (int i = 0; i < list.size(); i++)
{
    if (list[i] > name)
        name = list[i];
}
cout << "사전에서 가장 뒤에 나오는 이름은 " << name << endl;
```

```
4
jerry luna hangang elsa
=====
사전에서 가장 뒤에 나오는 이름은 luna
```

# 벡터(vector) --> string형

## ❖ 이터레이터(Iterator) - 반복자

```
vector<int> vec; //정수형 벡터 생성

//요소 저장
vec.push_back(0);
vec.push_back(1);
vec.push_back(2);

cout << "***** iterator(반복자) 사용 *****\n";
vector<int>::iterator it = vec.begin();

/*cout << *it << endl; //vec에 저장된 원소 출력
cout << *(it + 1) << endl;
cout << *(it + 2) << endl; */

for (it = vec.begin(); it != vec.end(); it++)
cout << *it << " ";
cout << endl;
```

# array(배열)

## ❖ array

- array는 고정된 크기의 배열을 담는 컨테이너 이다.
- C 스타일 배열과 달리 크기 정보를 유지하며, STL 컨테이너 인터페이스를 제공합니다. 또한 범위 검사를 하는 at() 함수도 제공.

array 객체 생성

**array** <자료형, 크기> 객체 이름



# array(배열)

## ❖ array

```
//크기가 5인 정수형 array 생성
array<int, 5> myArray;

//배열 초기화
myArray = { 1, 2, 3, 4, 5 };

//선언과 동시에 초기화
//array<int, 5> myArray{ 1, 2, 3, 4, 5 };

//배열의 크기
cout << "배열 크기: " << myArray.size() << endl;

cout << "배열 출력: ";
for (int i = 0; i < myArray.size(); i++) {
    cout << myArray[i] << " ";
}
cout << endl;
```

# array(배열)

## ❖ array

```
//배열의 첫 번째 요소 출력
cout << "첫 번째 요소: " << myArray[0] << endl;

//배열의 두 번째 요소 변경
myArray[1] = 10;

cout << "변경된 배열: ";
for (int& element : myArray) {
    cout << element << " ";
}

return 0;
}
```

```
배열 크기: 5
배열 출력: 1 2 3 4 5
첫 번째 요소: 1
변경된 배열: 1 10 3 4 5
```

int(&) 참조를 사용하면 큰 객체를 복사하지 않고 직접 접근하므로 효율적이고, 배열 요소를 변경하고 싶을 때 유용하다.

# vector를 활용한 도서 관리

## ❖ 벡터(vector) – 객체 저장

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

//Book 클래스 정의
class Book {
private:
    int number;    //책 번호
    string title;  //책 제목
    string author; //저자
public:
    //생성자
    Book(int number, string title, string author);

    //멤버 함수
    int getNumber();
    string getTitle();
    string getAuthor();
    void bookInfo();
};
```

# vector를 활용한 도서 관리

## ❖ 벡터(vector) – 객체 저장

```
Book::Book(int number, string title, string author) {
    this->number = number;
    this->title = title;
    this->author = author;
}

int Book::getNumber() {
    return number;
}

string Book::getTitle() {
    return title;
}

string Book::getAuthor() {
    return author;
}

void Book::bookInfo() {
    cout << "책 번호 : " << getNumber() << endl;
    cout << "책 제목 : " << getTitle() << endl;
    cout << "책 저자 : " << getAuthor() << endl;
}
```

# vector를 활용한 도서 관리

## ❖ 벡터(vector) - 객체 저장

```
//객체 배열
/*Book book[3] = {
    Book(100, "채식주의자", "한강"),
    Book(101, "C++ 완전정복", "조규남"),
    Book(102, "모두의 C언어", "이형우"),
};*/

//vector 자료구조로 객체 생성
vector<Book> books;

books.push_back(Book(100, "채식주의자", "한강"));
books.push_back(Book(101, "C++ 완전정복", "조규남"));
books.push_back(Book(102, "모두의 C언어", "이형우"));

cout << "***** 책의 정보 *****" << endl;
for (int i = 0; i < 3; i++)
{
    books[i].bookInfo();
    cout << "=====\n";
}
```

```
***** 책의 정보 *****
책 번호 : 100
책 제목 : 채식주의자
책 저자 : 한강
=====
책 번호 : 101
책 제목 : C++ 완전정복
책 저자 : 조규남
=====
책 번호 : 102
책 제목 : 모두의 C언어
책 저자 : 이형우
=====
```

# vector를 활용한 은행계좌 관리

## ❖ 은행 계좌 관리

```
계좌 정보
계좌 번호 : 1000
계좌주 : 이우주
잔액 : 10000원
-----
계좌 정보
계좌 번호 : 1001
계좌주 : 정은하
잔액 : 40000원
-----
계좌 정보
계좌 번호 : 1002
계좌주 : 한강
잔액 : 25000원
-----
5000원이 입금되었습니다. 현재 잔액 15000원
계좌 정보
계좌 번호 : 1000
계좌주 : 이우주
잔액 : 15000원
10000원이 출금되었습니다. 현재 잔액 30000원
계좌 정보
계좌 번호 : 1001
계좌주 : 정은하
잔액 : 30000원
```

# vector를 활용한 은행계좌 관리

## ❖ BankAccount 클래스

```
class BankAccount {  
    int accountNumber; //계좌 번호  
    string owner;      //계좌주  
    int balance;       //잔액  
  
public:  
    BankAccount(int accountNumber, string owner, int balance) :  
        accountNumber(accountNumber), owner(owner), balance(balance) {}  
  
    void displayInfo(); //계좌 정보  
    void deposit(int amount); //입금  
    void withdraw(int amount); //출금  
};
```

# vector를 활용한 은행계좌 관리

## ❖ BankAccount 클래스

```
void BankAccount::displayInfo() {  
    cout << "계좌 정보\n";  
    cout << "   계좌 번호: " << accountNumber << endl;  
    cout << "   계좌주: " << owner << endl;  
    cout << "   잔액: " << balance << "원" << endl;  
}  
  
void BankAccount::deposit(int amount) {  
    if (amount < 0) {  
        cout << "유효한 금액을 입력하세요.\n";  
    }else {  
        balance += amount;  
        cout << amount << "원이 입금되었습니다. 현재 잔액: "  
            << balance << "원\n";  
    }  
}
```



# vector를 활용한 은행계좌 관리

## ❖ BankAccount 클래스

```
void BankAccount::withdraw(int amount) {  
    if (amount < 0 || amount >= balance) {  
        cout << "유효한 금액을 입력하세요.\n";  
    }  
    else {  
        balance -= amount;  
        cout << amount << "원이 출금되었습니다. 현재 잔액: "  
            << balance << "원\n";  
    }  
}
```

# vector를 활용한 은행계좌 관리

## ❖ 입금, 출금 테스트

```
int main()
{
    //은행 계좌를 저장할 벡터 컨테이너 생성
    vector<BankAccount> accounts;

    //계좌 생성
    accounts.push_back(BankAccount(1000, "이우주", 10000));
    accounts.push_back(BankAccount(1001, "정은하", 50000));
    accounts.push_back(BankAccount(1002, "한강", 20000));

    //accounts[0].displayInfo(); //첫번째 계좌 정보

    accounts[0].deposit(15000); //1000번 계좌에 입금
    accounts[1].withdraw(20000); //1001번 계좌에서 출금

    //전체 계좌 정보 - 출력1
    /*for (int i = 0; i < size(accounts); i++) {
        cout << "-----\n";
        accounts[i].displayInfo();
    }
}
```

# vector를 활용한 은행계좌 관리

## ❖ 입금, 출금 테스트

```
// 출력 2
for (BankAccount account : accounts) {
    cout << "-----\n";
    account.displayInfo();
}*/

// 출력 3
for (auto& account : accounts) { //auto는 각각의 BankAccount 객체 참조
    cout << "-----\n";
    account.displayInfo();
}
cout << "-----\n";

return 0;
}
```

# auto 자료형 키워드

## ● auto 자료형 키워드

- auto 키워드는 C++ 표준부터 의미가 수정되어, 변수 선언문으로부터 변수의 타입을 결정하도록 지시한다.
- auto는 복잡한 형식의 변수 선언을 간소하게 해주어 타입 선언의 오타나 번거로움을 줄여준다.

```
int square(int x) { return x * x; } //inline 함수
int main()
{
    auto ch = 'K';
    auto n = 2; //2가 정수이므로 n을 int로 선언
    auto* p = &n;

    cout << ch << endl;
    cout << "n=" << n << ", p=" << p << endl;

    //함수의 리턴 타입으로 추론
    auto value = square(8);
    cout << value << endl;
```

# auto 자료형 키워드

## ● auto 자료형 키워드

```
//vector에 사용
vector<int> v = { 1, 2, 3, 4 };

/*vector<int>::iterator it;
for (it = v.begin(); it != v.end(); it++) {
    cout << *it << endl;
}*/

for (auto it = v.begin(); it != v.end(); it++) {
    cout << *it << endl;
}

return 0;
}
```

```
K
n=2, p=000000C0141BF2F4
64
1 2 3 4
```