1) 배열 Arrays

3가지 속성 : 자료형(type 배열내부는 모두 같은 자료형 가져야함)
용적(capacity 보유할수있는 최대개수), 크기(size 그시점에 존재개수
제로 인덱스 : int a[4] 면 [0]부터 [capacity-1 = 3]까지 범위를 갖는 것
2차원 배열 : scc 열만 만들어 b[2];
행만 만들어 a[5]

```
char* myReverse(char* p) {
    char* temp = new char[strlen(p) + 1];
    for (int i = 0; i < strlen(p); i++)
        temp[i] = p[strlen(p) - i - 1];
    temp[strlen(p)] = NULL;
    return temp;
}

int main(){
    char orig[] = "ABCDEFG";
    char* copy = nullptr;
    cout << orig << '\n';
    copy = myReverse(orig);
    cout << copy << '\n';
    delete[] copy;
    copy = nullptr;
```

2) 참조 References

한 번 부여한 별명은 변경 불가 but 원본에 대입하는건 가능 !! Int& x = 3; // 별명에 직접 정수값 대입 불가 but const로는 가능
!! 원본에 const 붙으면 &에도 const 붙여야 오류 안남(&에만 붙이는건 가능)

```
int a = 9;
int& ra = a;
int& rra = a;
cout << "ra: " << ra << "\n"; //9
ra = 7; //원본 변경하기
cout << "rra: " << rra << "\n"; //7

const int& rb = 88; //정수대입가능
int c = 3;          // rb=7 불가
const int& rc = c; //c로 원본변경가능 !! 전달할때
```

값으로 : 인수,매개변수 독립적 존재 & 매개변수 변경이 인수에 영향ㄴ
but 복사비용... . 크기가 큰 자료형 복사할때 별로 안 좋음
참조로 : 인수,매개변수 동일한 객체 & 매개변수 변경하면 인수도 변경됨
so 크기 큰 자료형 복사 좋음 but 바로 값 지정은 불가 | void fun(int& rX)

!! 리턴할때 값으로 : 클래스 자료형이면 복사생성자 호출, 복사비용 ㅠ | fun(5) //에러

type& function(..) 값은 & 뺌 참조로 : 일반적 매개변수&지역변수값 리턴불가-함수종료시 다 파괴되니까 // but 참조 매개변수&정적지역변수는 리턴가능-파괴 안되니까

3) 포인터 Pointers

변수의 주소 출력 : 주소 연산자 '&' -> 16 진수 출력
간접참조 : 지정된 주소 활용해 참조함
2차원 배열과: int(*m)[4] -> m[i][j] 포인터로 선언 후 배열로 받기 가능

```
const int a = 1;    cout << *pa << a; //값
int b=2;            cout << pa << &a; //주소
const int* pa = &a;
//*pa = 3; //실행불가
pa = &b; //가능, 가르키는 대상 자체는 변경불가

int score=92
int* pScore = &score;
int** ppScore = &pScore;
--- for문 안에  numbers[i] = *(numbers + i) 동일표현 ----
- int* p1 = 0; -> '값:0 아님'
// 아무 값을 가지지 않음
- void* p; 는 아무 자료형 다 가르킴
// p = &x (x는int) and p = &y (y는double)

int arr[5]; // 5개의 int 자료형을 갖는 배열 선언
int* parr = &arr[0]; //int* parr = arr; 동일함
for(int i = 0; i < 5; i++){ // 다 같은 주소를 가짐
    cout << "포인터 연산을 사용한 주소: " << i;
    cout << arr + i << endl;
    cout << "& 연산자를 사용한 주소: " << i;
    cout << &arr[i] << endl;
    cout << "parr을 사용한 주소: " << i;
    cout << parr + i << endl << endl;}
```

4) 메모리 Memory Management (Code, Static, Stack, Heap Memory, Dynamic Memory Allocation)

- 스택해제(되감기)
    1) 코드 메모리
    2) 정적 메모리
    3) 스택 메모리(후입선출, return시 pop됨)
    4) 힙 메모리(무조건 포인터로 사용, 꼭 delete를 해줘야함)

```
    int size;
    int* parr = nullptr;
cout << "몇 분이세요?";   cin >> size;
parr = new int[size]; // 힙 메모리 동적할당(메인에 없음) //!!중요
    for(int i=0; i<size; i++){cout << "나이는? "; cin >> parr [i]; }
    for(int i=0; i<size; i++){cout << parr[i];}
delete[] parr; //!!중요 꼭 딜리트 해줘야함
parr = nullptr; // 해제 free heap memory
```

5) 클래스 관계 Relationships among Classes (Inheritance, Association, Dependency)

- inheritance 상속(is-a) : ex)나는 학생이다.
protected 접근으로 getter setter 없이 사용가능, 코드 중복방지, 가독성 좋음, 재사용성 but 강한 결속력으로 유연성 낮춤
but 보통 public만 쓰고 지정 안할경우 private로 설정됨
- association 연관(has-a & is related to) : ex) 조립된 컴퓨터, 여러 신체부위로 구성된 인간
    aggregation 소유(집합관계, 컴퓨터 조립 so 각 클래스의 수명이 다름) ex)moniter has a (is related to) computer
        외부에서 만들어 결합, composition보다 약함
    composition 구성(구성관계, 사람 죽으면 모든 세포 다 죽음) ex)사람 죽을때 눈 코 입 다 작동 안됨
        내부에서 만들어 결합, 강한 결합관계, 죽을때도 다 같이 죽음
- dependency 의존(uses-a) : ex) 보험기간
    쓰고 버림, 클래스 안에 다른 클래스를 매개변수로 줄 때, 수정이 용이함 가장 약한 결합관계

6) 다형성 Polymorphism and Other Issues

- virtual 키워드 맨 앞 & 값 0 주기 => 순수 가상 함수 만들기 ex) virtual void print() = 0;
- 추상클래스 : 하나 이상의 순수가상함수
- 인터페이스 : 전부 순수가상함수
- '동적할당을 하여라' -> new Cricle();
포인터로 해당힙에 생긴 메모리주소번지 가져옴

- Upcast
베이스 클래스 포인터에 파생 클래스 객체 넣음
ex) Person* ptr1 = new Student

- Downcast(dynamic_cast 씀)
파생 클래스 포인터에 베이스 클래스 객체 넣음
but 상속관계에서 오버헤드 많이 발생 so 이런 코드 별로 안 좋음
ex) Student* ptr2 = dynamic_cast<Student*>(ptr1)

7) 람다 Lambda Expression

익명함수, 함수객체를 생성
함수 포인터와 함수객체의 장점을 지닌다.
캡쳐 기능을 통해 함수 밖의 변수에 접근할 수 있고
& 기호를 통해 람다함수 안에서도 외부 변수의 값을 변경 가능
재귀 호출도 가능(but 재귀에 auto는 불가) #include <functional>
매개변수 타입이 다른 경우 : 리턴타입 명시-강제변환 / 명시ㄴ-더 큰 타입으로 변환
[=]로 모든 바깥변수 사용가능(수정은 불가)
[&]로 모든 바깥변수 사용 & 수정가능

/// 문법 [캡쳐블럭](매개변수리스트) -> 리턴타입{함수바디};
/// 아래처럼 생략 가능 / []는 생략불가
/// []()-> void {}; // 화살표 "->" 쓸거면 리턴타입 지정해야함
/// []() {};
/// [] {};

[ ]에 명시한 바깥변수들 람다 내부에 사용가능

```
// 재귀함수
function<int(int)> fact = [&fact](int n)->int{
    return n <= 1 ? 1 : n * fact(n-1); };  cout  << fact(5) << endl;
```

```
void flyBehavior(function<void(void)> f) const
    {cout << owner << "의 " << name << "가 ";
     f(); }
auto FlyRockets = [](){cout << "로켓~\n";};
auto FlyWings = [](){cout << "날개~\n";};
p1.flyBehavior(FlyRockets);
p1.flyBehavior(FlyWings);
```

8) 연산자 오버로드 Operator Overloading

- interface (특수 추상클래스, is-a, 모든 멤버함수:순수가상함수)
  자식 클래스에서 오버라이드 해야함
  구현 파일 x => 중괄호 x & cpp파일 x & h파일 o
- multiple inheritance 다중 상속
  장점 : 양쪽 부모의 자원을 다 쓸수있다
  단점 : 메서드 명이 겹치면 스코프 기호(::)로 부모를 지정필요

9) 예외처리 Exception Handling

1) traditional : if~ return
2) getter setter에서 throw를 쓰고 메인에 try~catch 구문
3) re throw : 생성자 안에서 try~catch, try, catch 둘다 throw 포함시켜 메인에서 받게함

try문에서 throw하면 해당 타입의 매개변수를 가진 catch 구문으로 넘어감
매개변수로 ...을 주면 모든 타입을 받을 수 있음(가장 마지막에 쓰는 것이 좋음)

try문에서 다른 함수를 호출할수도 있음

10) 제네릭 : 템플릿 Generic Programming: Templates

- 같은 목적의 함수 만들때 타입에 따른 중복코드 방지 가능
- 템플릿 자료형 1개 당 1개의 타입만 지정 가능
- 특수화 : 특정상황에 사용할수있도록 제네릭 함수 작성(c언어 사용 o)
- 클래스 템플릿 : 안의 로직(내용)은 같은데 변수들의 타입만 다른 경우 사용 가능

11) 재귀 Recursion

반복문
- 장점 : 빠름
- 단점 : 가독성 안좋음

재귀문
- 단점 : 성능 - 느림(스택을 사용하기 때문에) -> 메모이제이션으로 극복가능
- 장점 : 점화식에 활용할 수 있음, 가독성 좋음

12) 문자열 String

reseve 예약 걸어도 capacity는 컴파일러 마음대로 생김(강의 30명으로 시작해도 강의실은 40명으로 고정되는것처럼
- c는 문자 배열의 마지막이 '\0'이어야함
- <cstring>은 생성자, 소멸자 없으며 strlen 가짐
- c,c++는 문자열 입력받을때 공백(" " or ₩0)앞까지만 인식
- C++은 생성자, 소멸자 존재

!! char* str = "Hello" //오류, 문자열리터럴은 상수로 지;
!! 힙메모리에 생성
!! 문자열 리터럴 : const char* str = "Hello" // OK
1) char* str = new char[3] -> 파괴 : delete[] str;
!! 초기화1 : char str[] = "Hello" // OK
// 상수가아닌 문자열(수정가능)
!!초기화2 : const char str[] = "Hello" // OK
2) const 앞에 붙이면 상수문자열(수정 불가)
- c++는 문자 배열의 마지막 null로 안끝남

str.size() : str의 길이
str.size() : str의 길이

!!함수들
strcmp(s1, s2)
같으면 0
왼 작 -1 or 왼 큼
strlen(s1)길이
getline(문자열, 구분문자)//구분까지 읽어라
get(문자열, 구분문자)//구분까지 읽어라
char* sPtr = strstr(문자열,"문자")
sPtr-str -> 문자열에서 "문자"어디서 시작하는지