C++ 스터디

**1일차**

1. 동적 할당
   1. 동적 할당이란?  
      필요 이상으로 설정한 메모리로 인해 **낭비되는 메모리** 또는   
      더 많은 메모리가 필요할 때 나오는 **런타임 에러**를 방지하기 위해 사용  
        
      C언어에서는  
      malloc, calloc, realloc  
      를 사용  
      C++에서는  
      **new , delete**를 사용
   2. new 연산자
      1. malloc, calloc, realloc 역할을 수행
   3. delete 연산자
      1. free 역할을 수행 ( **메모리 해제** )
   4. 사용 방법
      1. 변수는 **포인터**로 선언

Ex) int \*num= new int(5);

* + 1. new (자료형) 다음에 초기화 값을 넣어주는데,   
       **초기화 해주지 않으면 기본 생성자가 호출된다**.
    2. **배열 동적 할당**도 똑같이 사용  
       Ex) int \*num = new int[50];
    3. 동적 할당된 배열의 delete는 **delete[] ( 중요! )**

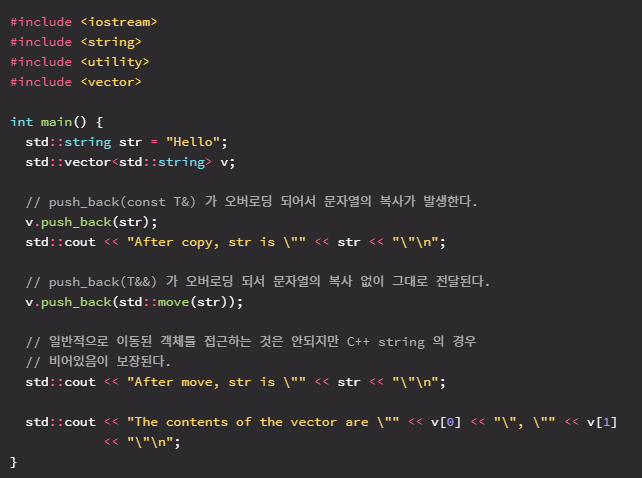
1. 참조자 형식
   1. **선언과 동시에 반드시 초기화**
   2. **자료형 &이름 = 선언된 변수;**
   3. 참조자는 변수의 주소를 저장하므로 변수가 없는 참조자는 존재 할 수 없음
   4. 큰 자료는 값을 전달하기 보단 주소를 전달하는 것이 효율적
   5. 상수에는 참조자를 선언할 수 없음
2. L-value 와 R-value  
   int num = 3  
   일 때,   
   L-value 는 num  
   R-value 는 3 이다.  
   int num = k  
   일 땐,  
   num 과 k는 둘 다 L-value이다.  
     
   이를 통해  
   **Lvalue 란  
   : 객체를 참조하는 표현식이다. ( 메모리 위치를 가지고 있다. )  
   Rvalue 란  
   : Lvalue가 아닌 모든 것. 즉 구분 가능한 메모리 영역을 가지는 객체를 나타낼 필요가 없는 표현식이다.**  
   메모리 위치를 가지지 않기 때문에, **성능 면에서 Rvalue는 중요하다**라고 볼 수 있다.  
     
   Rvalue를 사용하기 위해선 **&&** 를 사용  
   int a = 3;

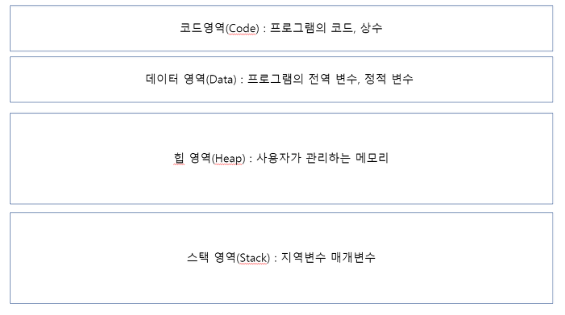
int& k = a; // 왼쪽 값일 때, 왼쪽 값은 가능

int& k = 3; // 에러 -> 왼쪽 값 이지만, 3은 오른쪽 값이기 때문에 에러

int&& k = 3; // 둘 다 오른쪽 값이라 가능

int&& k = a; // 에러 -> 오른쪽 값 = 왼쪽 값이라 에러

* 1. Std::move
     1. Std::move(변수) 로 사용
     2. 변수 자체는 Lvalue
     3. 변수 자체가 가지고 있는 자원을 다른 객체에게 효율적으로 전달하는 것을 의미
     4. **이동을 수행하지 않음. 이동 될 수 있음을 알려주는 것 뿐이다**
     5. Move를 사용하여 이동 시, **변수는 빈 껍데기가 된다**  
        빈 껍데기가 된 변수를 접근하는 것은 안됨 ( string 은 에외 )
     6. 참고 예시  
          
        <https://modoocode.com/301>

1. 함수의 다중 정의( = **함수 오버로딩** )
   1. 다른 매개변수를 가진 같은 이름의 여러 함수를 만들 수 있는 기능
   2. **매개변수를 통해서 구분한다**. 즉, **반환형에 대해서는 고려되지 않는다**.
   3. 오버라이딩과 햇갈리면 안된다.   
      ( 오버라이딩은 상속에서 사용. 매개변수, 이름 모두 동일해야 함. )
2. 오버라이딩  
   **상속**에서 사용하는 기능  
   부모에서 정의 된 것을 자식에서 재정의 하는 것.  
   함수의 이름, 매개변수 모두 동일해야 한다.  
     
   쉽게 설명하면,  
     
   캐릭터란 전체 관리해주는 부모 클래스가 있고,  
   안에 movespeed = 10 으로 설정  
     
     
   자식엔  
   도적, 전사, 마법사 이렇게 있는데  
   도적은 전사, 마법사보단 더 빨라야 한다.  
   그 때, 도적에 부모와 똑같은 함수를 만들어서 movespeed = 20으로 바꿔주면,  
   부모와 같은 이름 매개변수를 가진 함수지만, 도적 클래스에서 재정의된다.  
   이것이 오버라이딩
3. 메모리 구조  
     
   1. 코드 영역  
      실행할 프로그램의 **코드가 저장되는 영역**. 텍스트(code) 영역이라고 부름
   2. 데이터 영역  
      프로그램의 **전역 변수와 정적 변수가** 저장되는 영역  
      프로그램 시작과 함께 할당되며, 프로그램 종료 시 소멸
   3. 힙 영역  
      **동적 할당** 시, 사용 되는 영역  
      **사용자의 의해 메모리 공간이 동적으로 할당되고 해제 된다.**메모리의 낮은 주소에서 높은 주소 방향으로 할당
   4. 스택 영역  
      지역변수와 매개변수가 저장되는 영역  
      스택 영역에 저장되는 함수 호출 정보를 스택 프레임 이라 한다.  
        
      스택 영역은 한정된 공간이 있으므로, 공간을 전부 사용하면,  
      **스택오버플로우(Stackoverflow) 에러가 발생한다.**

**2일차**

1. 생성자
   1. 클래스 이름과 함수의 이름이 동일
   2. 반환형이 없음. ( 실제로 반환하지 않는다. )
   3. 오버로딩 가능
   4. 매개변수의 디폴트 값을 설정 가능
   5. 멤버 이니셜라이저 사용 가능하다  
      Ex)  
      class Dog{  
      private:  
       int num = 0;  
      public:  
       Dog(int \_num) : num(\_num){}  
      };
2. 복사 생성자
   1. 호출 될 때  
      참고 : <https://hwan-shell.tistory.com/43>
      1. 대입연산 할 때
      2. 매개 변수로 들어갈 때
      3. 반환 값이 있을 때
   2. 사용 법  
      className ( const className & ref){  
        
      }
      1. &로 받아야 하는 이유
         1. 참조를 해야 생성자 호출, 복사 생성자 호출을 하지 않음
         2. &를 사용하지 않으면, 무한 루프에 빠짐
            1. className ref는 className ref 에 대한 변수선언으로 들어가서  
               복사생성자가 계속 호출…  
               ( ref 라는 새로운 객체가 생성, ref의 복사 생성자가 호출 )
   3. 기본(디폴트) 복사 생성자는 얕은 복사
   4. 깊은 복사를 위해선 사용자가 새롭게 정의해줘야 함.
3. 소멸자
   1. 객체의 수명이 끝났을 때 객체를 제거하기 위한 목적
   2. 자동으로 컴파일러가 소멸자 함수를 호출.
   3. 사용법 : ~className(){}
   4. 멤버 변수에 new로 새롭게 메모리를 할당한 것이 있으면,   
      소멸에서 delete를 꼭 해줘야 함
   5. 상속 관계에 있을 시, 부모 클래스에서 virtual를 꼭 해줘야 한다  
      참고  
      <http://hyacinth.byus.net/moniwiki/wiki.php/C%2B%2B/%EC%86%8C%EB%A9%B8%EC%9E%90%EC%97%90%20virtual%EC%9D%84%20%EC%93%B0%EB%8A%94%20%EC%9D%B4%EC%9C%A0>
      1. Virtual를 해줌으로써, 소멸자도 자식 클래스에서 오버라이딩 된 함수임을 알림
      2. 자식 클래스의 소멸자가 호출 됨
      3. Vritual 해주지 않으면, 자식 클래스의 소멸자는 호출되지 않는다
4. 이동 시맨틱
   1. C++11에서 추가 인스턴스화된 객체의 메모리의 소유권을 이전하는 기능
   2. 메모리 소유권에 대한 이전 -> 얕은 복사 ( 복사 생성자와의 차이 점 )
   3. 이동 생성자, 이동 대입 연산자를 통해 지원
      1. 원본 객체 -> 새로운 객체로 복사 -> 원본 null로 초기화 과정을 뜻함
      2. 소유권을 이전 함으로 메모리 누수를 막아줌
   4. r-value 참조로 구현 ( &&을 사용 ) ( std::move( l-value ) 를 써야 함 )
   5. 불필요한 복사를 하지 않음. ( 메모리 할당 횟수를 줄인다 )
5. 정적(static) 멤버
   1. 클래스의 모든 멤버 함수와 멤버 변수는 static 멤버 가능
   2. 정적 멤버 변수
      1. 클래스에 하나만 생성 ( 다 같이 쓰기에 )
      2. 모든 인스턴스에서 공동으로 접근 ( 다 같이 쓴다 )
      3. 클래스 내부에서 선언 가능
      4. 클래스 내부에서 초기화 불가능 ( 클래스 외부에서 초기화 )
      5. 멤버 함수, 정적 멤버 함수 둘 다 정적 멤버 변수 사용 가능
      6. 접근 방법
         1. 클래스 이름으로 접근 Ex) className::변수이름
         2. 인스턴스를 통해 접근  
            ( 다른 인스턴스를 통해 접근해도, 공동으로 사용하기에 같은 변수 접근 )
   3. 정적 멤버 함수
      1. 클래스에 하나만 생성 ( 다 같이 쓰기에 )
      2. 접근 지정자(public, protected, private) 붙일 수 있음
      3. 멤버 함수 앞에 static 키워드 붙임. Ex) static void func();
      4. 정적 함수 내부에서는 정적 변수만 사용 가능

함수도 정적 함수만 호출 가능