개요 | 제출 | 편집 | 코딩 결과

(선택) 실습 066 - move semantics

제출 마감일: 2023-04-30 23:59
업로드 가능한 파일 수: 2
제출 방식: 개인

목적

이 실습은 class 의 move semantics 개념을 구현해 보는 연습을 합니다.

설명

C++03 표준까지는 copy semantics 만 존재 했습니다.

C++ 구루들이 copy semantics 의 비효율적인 부분을 많이 지적했습니다.

대표적인 복사가 읽어 나는 경우는 다음과 같습니다.

```
std::vector<std::string> vec1 { s1, s2, ..., s1000 };
std::vector<std::string> vec2;
vec2 = vec1;  // 문자열 1000개 복사
```

함수에 파라미터를 전달하거나 반환 받을 때도 복사가 일어 납니다.

https://en.wikipedia.org/wiki/Copy_elision#Return_value_optimization

```
#include <iostream>
struct C {
```

```
  C() = default;
  C(const C&) { std::cout << "A copy was made.\n"; }  // 복사 생성자
};

C f() {
  return C();
}

int main() {
  std::cout << "Hello World!\n";
  C obj = f();
}
```

컴파일러에 따라서 프로그램의 실행 결과는 달라집니다.

```
Hello World!
A copy was made.
A copy was made.
```

```
Hello World!
A copy was made.
```

```
Hello World!
```

C++11 에서는 move semantics 이 도입 되어 개발자가 직접 더 효율적으로 프로그램이 동작하도록 개발하는 방법을 제공합니다.

실습 6-4 에 이어서 String 클래스에 move constructor 와 move assignment operator 를 추가해 봅시다.

강의에서 설명 드린 대로 대표적인 이동 상황은 다음과 같습니다.

```
String s1 { "abc" } ;                 // constructor
String s2 = std::move(s1) ;           // move constructor

String s3;                            // constructor
s3 = createString();                  // move assignment operator   s3 = std::move(createString());

printString(createString());          // move constructor           printString(std::move(createString()));
```

참고로 저희가 수업에서 사용하고 있는 C++17 표준에서는

불필요한 객체의 복사를 방지하는 copy elision 을 컴파일러에게 준수하도록 합니다.

따라서, move constructor/assignment 를 확인하기 위해 std::move() 함수를 활용합시다.

이동 생성자의 형태는 클래스명(클래스명 rvalue 참조) 입니다.

String (String&& str) noexcept;

이동 생성자 내부에서는 파라미터로 전달 받은 객체의 내부 값을 자신의 멤버 변수로 이동시키는 것입니다.

- primitive types (int, float 등) 은 그냥 대입하면 됩니다. (기본 값 객체입니다)
- 포인터 류 데이터 역시 그냥 대입하면 됩니다.
- STL 객체들은 대부분 move semantics 을 지원하므로 std::move () 를 이용해서 대입합니다.

파라미터의 값을 이동하는 것은 크게 어렵지 않다는 것을 알게 됐습니다.

문제는 코어 가이드라인을 준수하려면 파라미터로 넘겨 받은 객체를 valid 상태로 만들어야 한다는 것입니다.

core guideline c.64 에 따르면 이동 후,

y = std::move(x) 이후로 x 객체의 상태는 valid state 이여야 하며,

이상적으로는 x 객체의 멤버 변수의 타입 초기값으로 초기화 되는 것이 좋다고 합니다.

확인은

1. valid state: x 변수에 다시 값을 대입할 수 있는지 확인해 봅시다.
2. x 변수의 소멸: 어떤 함수의 지역 변수 x 가 함수 종료 시 예외 발생하지 않고 소멸되는지 확인해 봅시다.

우리는 String 객체를 정의하면서

초기 상태 (생성자에서 만들어 주는 객체 상태, invariant) 를 다음과 같이 결정했습니다.

s = new char[1];

s = '\0';

파라미터로 넘겨 받은 객체의 상태를 valid 상태로 만들어 줍시다.

이동 대입 연산의 경우는 복사 대입 연산과 유사합니다.

자신의 자원을 clean-up 해주고, 멤버 변수들을 이동 시키고, 파라미터 객체의 상태를 valid 하게 만들면 됩니다.

또한, c.65과 c.66 을 준수하려면

3. 자기 이동에 안전해야 합니다. 즉, x = std::move(x); 문제가 없어야 합니다.

4. 이동 생성자, 대입 연산자는 noexcept 로 선언합시다.

참고로 복사/이동 대입 연산의 구현에 copy-swap idiom, move-swap idiom 등의 테크닉이 널리 사용됩니다.

문제

이동 시멘틱을 준수하는 String 클래스를 정의하시오.
단, 이동 생성자와 이동 대입 연산자를 구현하시오.
(프로그램 종료 코드는 exit code 0 으로 정상 종료 되어야 함)

<참조>

//StringMoveTest.cpp

```
void printString(String s){
   s.print("printCopyString");
}

String createName(){
   String name{"Kim"};
   return name;
}


int main() {
   String s1 = createName();
   s1.print("s1");
   printString(s1);

   printString("acb");
   printString(createName());
   printString(std::move(s1));

   String s2;
   s2 = s1;
   s1.print("s1");
   s2.print("s2");

   String s3 = "DEF";
   String s4 = std::move(s3);
   s3.print("s3");
   s4.print("s4");

   try {
      s3.at(0) = 'a';
      s2.print("s2");
      s3.print("s3");
   } catch(std::out_of_range& ex) {
      std::cout << ex.what() << std::endl;
   }
```

```
    s3 = "HIG";

    std::vector<String> vec1;
    vec1.push_back(std::move(s1));
    vec1.push_back(std::move(s2));
    vec1.push_back(std::move(s3));
    vec1.push_back(std::move(s4));

    std::vector<String> vec2(4);
    vec2 = std::move(vec1);

    for(auto& s: vec2)
        s.print();

    // compare std::string with our String

    std::string sa = "SSS";
    std::string sb;
    sb = std::move(sa);

    try {
        sa.at(0) = 'A';
        std::cout << sa << std::endl;
        std::cout << sb << std::endl;
    } catch(std::out_of_range& ex) {
        std::cout << "exception!" << std::endl;
    }

    String s9 = std::move(createName());
    printString(std::move(createName()));

    String s10;
    s10 = std::move(createName());
    s10 = std::move(s10);
}
```

입력

없음

출력

s1: Kim, size: 3

printCopyString: Kim, size: 3

printCopyString: acb, size: 3

printCopyString: Kim, size: 3

printCopyString: Kim, size: 3

s1: , size: 0

s2: , size: 0

s3: , size: 0

s4: DEF, size: 3

out of range at index: 0, but the length of String is 0

: , size: 0

: , size: 0

: HIG, size: 3

: DEF, size: 3

exception!

printCopyString: Kim, size: 3

s10: Kim, size: 3

제출파일

String.cpp
66.csv

<참고>

String.h ----------

```
#include <stddef.h>
class String {
public:
   String();
   String(const char* str);
   String(const String& str);
   String& operator=(const String& str);
   String(String&& str) noexcept;
   String& operator=(String&& str) noexcept;
   ~String();
   const char* data() const;
   char& at(size_t);
   size_t size() const;
   void print(const char* str="") const;
private:
   void swap(String& str);
   int len;
   char* s;
};
```

String.cpp---------- (일부함수)

```
void String::swap(String& str){
   using std::swap;
```

```
    swap(s, str.s);
    swap(len, str.len);
}

String::~String() {
    if(s) delete[] s;
}

const char* String::data() const {
    return s;
}

char& String::at(size_t pos){
    if(len!=0 && 0 <= pos && pos < len)
        return s[pos];
    throw std::out_of_range("out of range at index: " + std::to_string(pos) + ", but the length of String is " + std::to_string(len));
}

size_t String::size() const {
    return len;
}

void String::print(const char* str) const {
    // for debugging
    // std::cout << str << ": " << s << ", size: " << len << " address: " << (void *) s << std::endl;
    std::cout << str << ": " << s << ", size: " << len << std::endl;
}
```

VPL