Notes on Modern C++
(standard c++11, 14, 17, 20)

Gyubeom Edward Im*

July 6, 2024

Contents

1 Introduction

2 Intermediate

2.1 Temporary

2.2 Trivial constructor

2.2.1 Trivial default constructor

생성자가 trivial하다는 말은 컴파일러가 자동으로 생성해주면서 동시에 아무 일도 하지 않을 때를 말한다

2.2.2 Trivial copy constructor

복사 생성자가 trivial하다는 말은 멤버변수 값을 복사하는 것 이외에 아무 일도 하지 않을 때를 말한다. 복사 생성자가 trivial하다면 배열 전체를 memcpy와 memmove 등으로 복사하는 것이 빠르다! 복사 생성자가 trivial하지 않다면 배열의 모든 요소에 대해 하나씩 "복사 생성자"를 호출해서 생성자를 호출해야 한다

```
struct Point {
  int x=0;
  int y=0;
};

template<class T>
void constexpr copy_type(T* dst, T* src, std::size_t sz) {
  if(std::is_trivially_copy_contructible_v<T>) {
    std::cout << "using memcpy" << std::endl;
    memcpy(dst, src, sizeof(T)*sz);
  }
  else {
    std::cout << "using copy ctor" << std::endl;
```

*blog: alida.tistory.com, email: criterion.im@gmail.com

```
    while(sz--){
      new(dst) T(*src);
      --dst, --src;
    }
  }
}

int main)(){
  Point arr1[5];
  Point arr2[5];
  copy_type(arr1, arr2, 5);
}
```

위 코드에서 Point 클래스는 int x,y와 같이 간단한 멤버변수만 존재하므로 trivial copy constructor이다. 하지만 virvual void foo() 같이 가상함수를 사용하거나 string s;와 같이 복사하는 클래스를 사용하게 되면 trivial하지 않게 된다. 이런 경우에는 **placement new** 또는 **std::construct_at**을 사용해야 한다.

2.3 Type deduction

컴파일 타임에 타입이 결정되는 auto 키워드에 대해 살펴보자

```
int main(){
  int n=10;
  cosnt int c =10;

  auto a1 = n;  // int a1=n;
  auto a2 = c;  // (1) const int a2 = c; --> no.
                // (2) int a2 = c;     --> ok.
}
```

type deduction(타입 추론)이 발생하는 키워드는 다음과 같다: **template, auto, decltype**

```
#include <iostream>
template<class T> void foo(T arg){
  std::cout << typeid(T).name() << std::endl;
}
int main(){
  int n=10;
  foo(n);                    // T=int
  foo<const int&>(n); // T=const int&. But typeid(T).name() keep printing output 'int'
}
```

typeid(T).name()은 타입 이름만 추론할 뿐 const, volatile, reference 정보가 출력되지 않는다.

1. 이럴 때는 의도적으로 에러를 발생시켜서 정확한 타입을 에러 메시지를 통해 알 수 있다.
2. 또는 **boost::type_index** 라이브러리에 **type_id_with_cvr<T>().pretty_name()**을 사용하면 됨
3. 컴파일러가 제공하는 매크로를 사용하면 된다.
   - __FUNCTION__: 함수의 이름만 보여주므로 타입 추론에는 사용하지 않음
   - __PRETTY_FUNCTION__: g++,clang에서는 함수 이름과 타입이 나옴
   - __FUNCSIG__: cl.exe 버전

```
std::cout << __FUNCTION__ << std::endl;
std::cout << __PRETTY_FUNCTION__ << std::endl; // g++, clang
std::cout << __FUNCSIG__ << std::endl;              // cl.exe
```

T가 값인 경우를 살펴보자

```
  #include <iostream>
  template<class T> void foo(T arg){
    while(--arg>0) {}
  }

```

```
  int main(){
    int n=10;
    int& r = n;
    const int c = 10;
    const int& cr = c;
    foo(n);  // T=int
    foo(r);  // T=int& 일 것 같지만 T=int
    foo(c);  // T=const int 일 것 같지만 T=int
    foo(cr); // T=const int& 일 것 같지만 T=int
  }
```

T 인자를 값으로 받을 때는 복사본 객체가 만들어져서 **"const, volatile, reference" 속성을 제거하고 값만 받는다.** 헷갈리는 것 중 하나가 값으로 받을 때는 인자의 const 속성은 제거되고 "**인자가 가리키는 곳의 const 속성은 유지**"한다. 무슨 이야기인지 살펴보자

```
#include <iostream>
template<class T> void foo(T arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}
int main(){
  const char* const s = "hello";
  foo(s);  // 포인터의 const 속성은 제거되지만 가리키는 곳 "hello"의 const 속성은 유지됨!
              // const char* arg = "hello"가 됨!!
}
```

T 인자를 참조로 받을 때는 다음과 같다.

```
#include <iostream>
template<class T> void foo(T& arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}

int main(){
  int n = 10;         // T=int.         arg=int&
  int& r = n;         // T=const int.   arg=const int& (const 속성은 유지!)
  const int c = 10;   // T=int          arg=int&. (T에서 refernce 속성은 제거!)
  const int& cr = c; // T=const int    arg=const int& (const 속성은 유지!)
}
```

주의해야할 점은 T의 타입과 arg의 타입은 다르다는 것이다 (T& arg 이기 때문!). 함수 인자의 "reference를 제거하고 T의 타입을 결정한다". 인자가 가진 "const, volatile 속성은 유지한다.". 마지막으로 T에 배열이 전달된 경우를 살펴보자

```
template<class T> void foo(T arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}
template<class T> void goo(T& arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}

int main(){
  int x[3] = {1,2,3};
  foo(x); // T=int* 타입으로 받는다
  goo(x); // T=int[3] 타입으로 받는다. arg는 int()[3] 타입으로 받는다
}
```

T& arg로 배열을 받는 경우 int (&arg)[3] = x; 처럼 받는게 되어서 배열의 reference가 된다. 따라서 아래와 같이 goo()를 사용하면 에러가 발생한다.

```
template<class T> void foo(T arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}
template<class T> void goo(T& arg){
  std::cout << __PRETTY_FUNCTION__ << std::endl;
}
```

```
7
8  int main(){
9    foo("orange", "apple"); // ok
10   goo("orange", "apple"); // error
11 }
```

foo와 **goo** 둘 다 const char 형식으로 받지만 포인터는 개수에 제한이 없으므로 ok인 반면에 reference는 const char[7], const char[6]은 다른 reference이기 때문에 같은 T를 받는 상황에서 에러가 발생한다!
foo(const char[7], const char[6]), **goo**(const char[7], const char[6])

2.4 Auto type deduction

template은 함수 인자로 추론하는 반면 **auto**는 우변의 표현식으로 타입을 추론한다. template을 'T arg = 함수 인자' 처럼 추론한다고 볼 수 있으므로 사실 'auto a = 표현식'과 동일한 형태로 추론한다!

```
1  int main(){
2    int n=10;
3    int& r = n;
4    const int c = 10;
5    const int& cr = c;
6
7    auto a1 = n; // auto=int
8    auto a2 = r; // auto=int
9    auto a3 = c; // auto=int
10   auto a4 = cr; // auto=int
11
12   auto& a5 = n; // auto=int. a5=int&
13   auto& a6 = r; // auto=int. a6=int&
14   auto& a7 = c; // auto=const int. a7=const int&
15   auto& a8 = cr; // auto=const int. a8=const int&
16
17   int x[3] = {1,2,3};
18   auto a = x;  // auto=int*
19   auto& b = x; // auto=int[3]. b=int(&)[3]
20 }
```

3 References

[1] (lecture) CODENURI - C++ Master

4 Revision log

- 1st: 2024-07-16