

Core GuideLines

Chapter 6

최정호

- **Enum.7 열거형의 바탕 형식은 꼭 필요할 때만 명시하라**

Enum의 기본적으로 정의된 바탕형식은 int형 이므로 필요에 따라 바탕형식을 바꿀 수 있다.

```
#include<iostream>
#include<stdint.h>
enum class ItemType1
{
    Weapon,
    Armor,
    Accessories
};
enum class ItemType2 : uint8_t
{
    Weapon,
    Armor,
    Accessories
};
void main()
{
    ItemType1 eType = ItemType1::Weapon;
    std::cout << "ItemType1Size : " << sizeof(eType) << std::endl;
    ItemType2 eType2 = ItemType2::Weapon;
    std::cout << "ItemType2Size : " << sizeof(eType2) << std::endl;
}
```

Core GuideLines

Chapter 7

최정호

- **R.11 new 와 delete의 명시적인 호출을 피하라**

C++핵심 가이드라인 에서는 “메모리를 직접 할당/해제하지 말고 스마트 포인터를 사용하라”
에 이어 “R.22 shared_ptr 객체는 make_shared()로 생성하라”

“R.23 unique_ptr 객체는 make_unique()로 생성하라” 는 규칙이 있다.

다만 이를 적용시키기 어려운 경우 Custom Deleter를 지정해야 하는데 이를 활용할 수도 있다.

CustomDeleter.cpp 코드 참고

• 7장 전체 정리

- 메모리자원의 소유권을 확실히 하라
- 메모리의 생명주기를 명시적으로 표현할 땐 원시 포인터 보단 스마트 포인터위주로 사용하라
 - `std::unique_ptr` : 소유권을 독점하는 포인터
 - `std::shared_ptr` : 소유권을 공유하는 포인터
 - `std::weak_ptr` : 소유권을 주장하지 않는 포인터
- 스마트포인터 생성은 `make_xxx()`사용을 권장한다.

Core GuideLines

Chapter 8

최정호

- **항상 적용할 수 있다**

- 가장 당황스러운 구문 해석의 문제가 없다.

- ```

1 #include <iostream>
2
3 struct Test
4 {
5 Test(int d = 0) : i(d) {}
6 int i;
7 };
8
9 void main()
10 {
11 Test a(10);
12 Test b();
13 Test c;
14 Test d{};
15
16 std::cout << a.i << std::endl;
17 std::cout << b.i << std::endl;
18 std::cout << c.i << std::endl;
19 std::cout << d.i << std::endl;
20 }
21

```
- 163 % 1 0
- 오류 목록
- 전체 솔루션 2 오류 0/1 경고 0 메시지 빌드 + IntelliSense
- 코드 설명
- C2228 'i' 왼쪽에는 클래스/구조체/공용 구조체가 있어야 합니다.  
E0153 식에 클래스 형식이 있어야 하는데 "Test (\*)0" 형식이 있음

## • ES.23 {} 초기화 구문을 선호하라

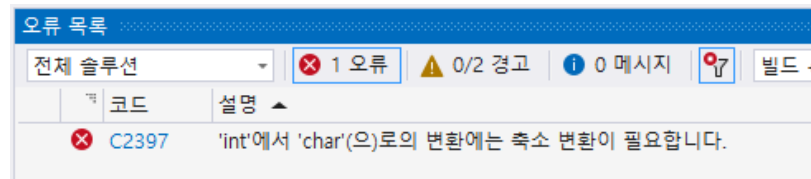
### - 좁아지는 변환이 일어나지 않는다.

- char 자료형은 -128 ~ 127까지 저장 가능 하므로 범위를 벗어나는 정보를 저장할 경우 오동작이 발생할 수 있다.

```
1 #include<iostream>
2
3 void main()
4 {
5 char ch = 300;
6 std::cout << ch << std::endl;
7 }
```

- 이때 {} 초기화를 하면 위와 같은 축소변환을 감지할 수 있다.

```
1 #include<iostream>
2
3 void main()
4 {
5 char ch{ 300 };
6 std::cout << ch << std::endl;
7 }
```





- ES.28 복잡한 초기화에는, 특히 `const` 변수의 복잡한 초기화에는 람다를 사용하라.

```
#include<iostream>
class Test
{
private:
 int Data{ 0 };
public:
 void operator += (int _iData)
 {
 Data += _iData;
 }
void main()
{
 const Test t = [&] {
 Test val;
 for (int i = 0; i < 10; i++)
 val += i;
 return val;
 }();
}
```

- ES.78 switch문에서 암묵적인 실행 지속에 의존하지 말라.

```
#include<iostream>
#include<conio.h>
void main()
{
 char ch = _getch();
 switch (ch)
 {
 case 'a':
 __fallthrough
 case 'A':
 std::cout << "왼쪽으로 이동";
 break;
 case 'd':
 __fallthrough
 case 'D':
 std::cout << "오른쪽으로 이동";
 break;
 default:
 break;
 }
}
```

**Thank you**

**최정호**