Chapter 11:
C++의 까다롭고 유별난 부분

Mijin An
meeeeejin@gmail.com

레퍼런스란?

- 일종의 변수에 대한 **별칭 (Alias)**
  - 레퍼런스를 이용해 수정한 내용은 레퍼런스가 가리키는 변수의 값에 그대로 반영
  - 함수나 메서드의 매개변수로 사용

- **레퍼런스 변수**
  - 생성하자마자 초기화

```
int x = 3;
int& xRef = x;
xRef = 10;
```

  - 정수 리터럴처럼 **이름 없는 값에 대해서는 생성 불가**
  - 단, const 값에 대해서는 생성 가능

```
int unnamedRef1 = 5;          // 컴파일 에러
const int& unnamedRef2 = 5; // 정상 작동
```

레퍼런스 대상 변경하기

- 레퍼런스는 **한 번 생성하면 가리키는 대상을 바꿀 수 없다**

```
int x = 3, y = 4, z = 5;
int& xRef = x;
int& zRef = z;

xRef = y;           // x = 4
xRef = &y;          // 컴파일 에러
zRef = xRef;        // z = 3
```

레퍼런스와 포인터

- 레퍼런스는 **모든 타입에 대해 생성 가능**

```
int* intP;
int*& ptrRef = intP;
ptrRef = new int;
*ptrRef = 5;
```

- **레퍼런스가 가져온 주소 = 레퍼런스가 가리키는 변수의 주소**

```
int x = 3;
int& xRef = x;
int* xPtr = &xRef; // 레퍼런스의 주소는 값에 대한 포인터와 같다
*xPtr = 100;
```

- 레퍼런스에 대한 레퍼런스는 선언 불가능

레퍼런스와 포인터의 선택 기준

- 레퍼런스가 포인터보다 훨씬 안전
  - 레퍼런스의 값은 널이 될 수 없음
  - 레퍼런스를 명시적으로 역참조 불가능

- **포인터를 사용한 코드는 거의 대부분 레퍼런스로 표현 가능**

- **반드시** 포인터를 사용해야 하는 경우 존재:
  - 가리키는 위치를 변경해야 할 때
  - 주솟값이 `nullptr`가 될 수 있는 `optional` 타입을 사용할 때
  - 컨테이너에 다형성 타입을 저장할 때

- 포인터 vs. 레퍼런스? 🤔 → **메모리의 소유권**을 따져봐라
  - 메모리의 소유권이 변수를 받는 코드에 **있으면, 포인터 사용**
  - 메모리의 소유권이 변수를 받는 코드에 **없으면, 레퍼런스로 전달**

키워드 혼동: const와 static (1)

- const 키워드
  - **변경되면 안 될 대상을 선언할 때 사용**

```
const double PI = 3.141592653589793238462;

void func (const int param) {
    // param 변경 불가능
}
```

  - const 포인터

1. 포인터로 가리키는 값이 수정되지 않게 하기

```
const int* ip;
ip = new int[10];
ip[4] = 5;          // 컴파일 에러

int const* ip;
ip = new int[10];
ip[4] = 5;          // 컴파일 에러
```

2. 포인터가 수정되지 않게 하기

```
int* const ip = nullptr;
ip = new int[10];   // 컴파일 에러
ip[4] = 5;          // 널 포인터 역참조 에러
```

키워드 혼동: const와 static (2)

- const 키워드
  - const 레퍼런스
    - 레퍼런스는 기본적으로 const 속성
    - 레퍼런스에 대한 레퍼런스를 만들 수 없기 때문에 참조가 한 단계뿐

```
int z;
const int& zRef = z;
zRef = 4; // 컴파일 에러
```

    - 주로 매개변수에 적용

```
void doSomething (const BigClass& arg) {
    // ...
}
```

키워드 혼동: const와 static (3)

- static 키워드
  - **특정한 스코프 안에서만 값을 유지하는 로컬 변수를 만들 때 사용**
  - extern 키워드
    - 외부 링크 지정할 때 사용

타입 앨리어스

- 기존에 선언된 타입에 다른 이름을 붙이는 것
  - 기존 타입과 호환

```
using IntPtr = int*;

int* p1;
IntPtr p2;

p1 = p2;
p2 = p1;
```

  - 복잡하게 선언된 타입을 간결하게 표현할 때 사용

```
using StringVector = std::vector<std::string>;

void processVector (const StringVector& vec) { /* ... */ }
```

캐스팅: #1 const_cast()

- 변수에 const 속성을 추가/제거

```
extern void ThirdPartyLibraryMethod(char* str);

void f (const char* str) {
    ThirdPartyLibraryMethod(const_cast<char*>(str));
}
```

- C++17부터 std::as_const() 헬퍼 메서드 추가

```
std::string str = "C++";
const std::string& constStr = std::as_const(str);
```

캐스팅: #2 static_cast()

- 명시적 변환 기능 수행

```
int i = 3;
int j = 4;
double result = static_cast<double>(i) / j;
```

- 상속 계층에서 하위 타입으로 다운캐스팅할 때 사용

캐스팅: #3 reinterpret_cast()

- C++ 타입 규칙에서 허용하지 않더라도 해당 객체의 **포인터/레퍼런스를 캐스팅**해야할 때 사용

```
class X{};
class Y{};

int main () {
    X x;
    Y y;
    X* xp = &x;
    Y* yp = &y;

    xp = reinterpret_cast<X*>(yp);  // 서로 관련 없는 클래스 타입의 포인터 변환

    void* p = xp;
    xp = reinterpret_cast<X*>(p);   // void*를 원래 포인터로 복원

    X& xr = x;
    Y& yr = reinterpret_cast<Y&>(x);// 서로 관련 없는 클래스 타입의 레퍼런스 변환

    return 0;
}
```

캐스팅: #4 dynamic_cast()

- 같은 상속 계층에 속한 타입끼리 캐스팅할 때 사용
  - **실행 시간에 타입 검사**

```
Base* b;
Derived* d = new Derived();

b = d;
d = dynamic_cast<Derived*>(b);
```

스코프

- 변수, 함수, 클래스를 비롯하여 프로그램에서 사용하는 **모든 이름은 일정한 스코프에 속함**
  - 이름으로 접근할 때 가장 가까운 스코프부터 시작하여 전역 스코프까지 검색

```
class Demo
{
    public:
        static int get() { return 5; }
};

int get() { return 10; }

namespace NS
{
    int get() { return 20; }
}
```

```
int main() {
    auto pd = std::make_unique<Demo>();
    Demo d;

    std::cout << pd->get() << std::endl;     // 5
    std::cout << d.get() << std::endl;       // 5
    std::cout << NS::get() << std::endl;     // 20
    std::cout << Demo::get() << std::endl;   // 5
    std::cout << ::get() << std::endl;       // 10
    std::cout << get() << std::endl;         // 10

    return 0;
}
```

어트리뷰트 (1)

- 특정 벤더에서만 제공하는 정보나 옵션을 소스 코드에 추가하는 메커니즘
  - [[어트리뷰트]] 문법으로 표기

- `[[noreturn]]`
  - 함수가 호출한 측으로 제어를 리턴하지 않는다는 것을 의미
  - Process/thread를 종료시키거나 exception을 던지는 함수에 지정

- `[[deprecated]]`
  - 더 이상 지원하지 않는 대상을 지정할 때 사용

- `[[fallthrough]]`
  - `switch` 문에서 의도적으로 fallthrough를 적용하고 싶을 때 사용

어트리뷰트 (2)

- `[[nodiscard]]`
  - 값을 리턴하도록 정의된 함수에 지정하면, 그 함수를 이용하는 코드에서 리턴 값을 사용하지 않을 때 경고 메시지 발생

- `[[maybe_unused]]`
  - 프로그램에서 사용하지 않는 코드를 발견해도 경고 메시지를 출력하지 말라고 컴파일러에 지시할 때 사용

사용자 정의 리터럴

- **반드시 _로 시작 + 첫 문자는 소문자**
  - e.g., `_i`, `_miles`

- **리터럴 연산자**를 정의하는 방식으로 구현
  - **미가공** 모드 (Raw Mode)
    - 123 ➜ 단순 문자
  - **가공** 모드 (Cooked Mode)
    - 123 ➜ 정숫값
    - 숫자값을 처리하려면 타입이 명시된 매개변수 필요
    - 스트링을 처리하려면 매개변수 두 개 (문자 배열, 배열의 길이) 필요

```
std::complex<long double> operator"" _i (long double d) {
    return std::complex<long double>(0, d);
}

auto c = 1.23_i;
```

헤더 파일

- 서브시스템이나 코드에 **추상 인터페이스를 제공**하는 메커니즘
  - 헤더 파일 중복 및 순환 참조 발생에 주의

- **인클루드 가드 (Include Guard)**를 사용해 중복 정의 방지

```
#ifndef LOGGER_H
#define LOGGER_H

class Logger
{
    // ...
};

#endif // LOGGER_H
```

- 최신 컴파일러는 대부분 `#pragma once` 디렉티브를 제공하여 인클루드 가드 작성 불필요

Reference

[1] Marc Gregoire, “Professional C++, 4th Edition”, Wiley, 2018