Virtual & Concepts

박동하

- Facebook Group
  - C++ Korea
  - C++, OpenSource Study
- github.com/luncliff
- Alchera Inc. 전문연구원

참고자료

- 문서
  - Concepts: The future of Generic Programming 🤔
    www.stroustrup.com/good_concepts.pdf, youtube.com/watch?v=HddFGPTAmtU
  - C++ Core Guidelines: Template
    github.com/CppKorea/CppCoreGuidelines
  - CppReference.com
    en.cppreference.com/w/cpp/language/constraints
- 동영상
  - CppCon 2018: Andrew Sutton - “Concepts in 60” 🤔
    www.youtube.com/watch?v=ZeU6OPaGxwM
  - CppCon 2018: Arthur O’Dwyer – “Concepts As She Is Spoke” 👍 👍
    www.youtube.com/watch?v=CXn02MPkn8Y

참고자료

- 책
  - Programming Languages: Principles and Paradigms 🤔
    - Allen B. Tucker, Robert E. Noonan. McGraw-Hill
    (국내 번역: 생능 출판사)
  - Programming: Principles and Practice Using C++, 2nd E. 한국어판
    - Bjarne Stroustrup 저, 최광민 옮김. 에이콘
      - 4부 22장. 프로그래밍의 이상과 역사
  - 객체지향 입문
    - Allen B. Tucker 저, 여인춘 옮김. 정보문화사

C++ 17, 20 에 대한 거부감...

Q. 친숙하게 느끼게 할 수 없을까?
A. 익숙한 기능과의 비교를 통해서, 좀 더 **주관적**으로 설명해보자!

(전문용어: 뇌피셜) 😅

사실 Modern C++ Next 프로젝트에 작성하려던 내용이었는데, 겸사겸사...
조만간 내용을 보강해서 GitHub 저장소에 업로드할 예정

오늘 다루지 않을 것들

- 뭔가 자세한 내용
  - Virtual 을 사용하면 Compiler가 어떤 일을 하는가?
  - Template 관련 트릭/우회법 등 (ex. SFINAE)

- C++ 책을 1권 이상 완독했다면 크게 문제 없을 수준

오늘 다룰 것들

- 패러다임 비교
  - Concepts, 어디에 초점을 두고 이해해야 하는가?

- 접근법 비교
- 코드 비교 (쉬운 코드!)
  - Parameter + Return

- 정리
  - Q&A: Concepts, Modern C++ …

패러다임 비교

쉬운 이야기부터...

갑자기 왠 패러다임?

- Virtual
  - OOPObject Oriented Programming 패러다임
  - **타입**의 상속과 확장: Class Hierarchy 에서 다형성 지원
- Concepts
  - GPGeneric Programming 패러다임
  - 코드 생성에 사용하는 **타입**을 단정(딱 잘라 판단하거나 결정하다)

- 결국 이 둘의 비교는 C++에서의 OOP와 GP를 비교한다는 것

타입을 강조했는데...

- 타입Type
  - 언어에서 값을 표현하고 의미를 전달하는 방법

```
struct tcp_pseudo_t final {
    uint32_t src;
    uint32_t dst;
    uint8_t zero;
    uint8_t protocol;
    uint16_t len;
};
```

- C++는 멀티 패러다임 언어
  - == 패러다임에 따라 타입을 다르게 사용한다

패러다임에 따른 타입 사용법

- Object Oriented
  데이터/연산의 조직화Organization
- Generic
  이미 정의된 알고리즘에 사용하는 인자Argument
- Functional
  공간Domain과 사상Mapping의 표현

관련 용어들을 잠깐 짚고 넘어가자면...

타입Type

- 값Value의 집합 + 연산Operation의 집합

- 기계에 저장된 값에는 오직 비트만 있을 뿐!
  - 값 --> 일련Sequence의 비트Bit
  - 연산 --> 일련의 명령Instruction

- 타입은 언어 레벨의 개념
  - 작성 + 번역
  - == 언어에서 값을 표현하고 의미를 전달하는 방법

타입 시스템Type System

- 타입에서 가질 수 있는 값 + 연산들의 관계를 정의
- 좋은 프로그래밍 언어의 기준이 되기도 함

- 주요한 목표들
  - 잘못된 프로그램 작성을 예방 (Machine can do that for you!)
  - 추상화 (코드의 목적을 보다 쉽게 이해)
  - 빠른 오류 발견 (정확성!)

정확하게 == 설계자가 의도한 대로

- 정확성 in C ?
  - Trust the programmer!
- 정확성 in C++ ?
  - 컴파일러 괴롭히기: 오류 탐지/예방의 수단으로 사용

- Q. 코드 보면... 아시죠?
  - C: 코드를 정독했는가? 이해했는가? 설명할 수 있는가?
  - C++: 컴파일 과정에서 검증 가능하도록 타입을 작성/사용하고 있는가?

  - 해석과 접근법이 근본적으로 다르다!
    == 두 언어의 사용자들이 C/C++ 라는 표현을 싫어하는 이유

타입 검사Type Checking

- 타입 오류의 검출 (컴파일 시간 + 실행시간)

타입 오류Type Error

- 타입: 값Value의 집합 + 연산Operation의 집합

- 타입 오류: 연산을 잘못된 값의 집합에 적용
  - 포인터에 곱셈
  - 부동 소수점에 Bit shift

- 모든 타입 오류를 검출해낼 수 있으면?
  - 강타입 언어 Strong Typed Language

윽, 머리가 ...

- 다시 패러다임 이야기로 돌아가봅시다

개체지향Object-Oriented 프로그래밍

- 핵심: **Data structure and operation for it**
  - 데이터를 조직화하고
  - 조직화된 데이터를 다루는 전용 처리방법을 둔다

- 조직화?
  - 조직화의 방향성과 처리의 방향성을 일치시키는 것
  - 추상화 == **문제를 표현하는 좋은 방법**을 찾는 것

- 표현력을 가지면 얻는 것?
  - 문제의 변화에 유연하게 대응
  - == 더 좋은 유지보수성
  - == 생산성!

Everything is an object

!= 모든 것을 Class로 만들어라 **(지나친 축약)**

== 추상화(조직화)하여 문제를 표현하고
그에 맞게 프로그램을 작성하라

- 개체지향적? 🤔
  - 분석 : 문제를 어떻게 표현할 것인가?
  - 설계 : 데이터와 연산을 어떻게 (분석에 맞게) 조직화할 것인가?
  - 구현 : Class는 언어에서 지원하는 수단Tool일 뿐

OOP in C++

- 상속Inheritance
  - 구조/연산을 확장하는 방법 == 추상화의 전파

- Virtual:
  - 상속 과정에서 Name Binding을 제어
  - == 상위 타입 메서드의 이름으로 하위 타입의 메서드 호출 (Override)
    (virtual member function)
  - == 상위 타입의 구조를 통해서 접근할 때 정확한 영역으로 유도
    (virtual base class)

```cpp
class IVoidDisposable {
  public:
    virtual ~IVoidDisposable() = default;
    virtual void dispose() = 0;
};

// In the class hierarchy
class MyDisposable : public IVoidDisposable {
    void dispose() override {
        puts(__PRETTY_FUNCTION__); // for MSVC, use __FUNCDNAME__
        return;
    }
};


// Out of the class hierarchy
class AnotherDisposable {
  public:
    errc dispose() { // <-- different return type
        puts(__PRETTY_FUNCTION__);
        return errc{};
    }
};
```

제네릭Generic 프로그래밍

- 타입 == 알고리즘의 매개변수
  - `stack<T>`
  - `max<T>(left, right)`
  - …

- 추상화 in GP
  - == **재사용 가능한 알고리즘**을 작성하는 것
  - == 생산성!

GP in C++

- 템플릿Template
  - 타입을 인자로 컴파일 시간에 코드 생성
  - 컴파일러가 추가하는 매크로(Macro)

- Lazy Evaluation
  - 사용하지 않으면? 생성된 코드가 없다
  - 코드가 없다? 프로그램이 작다
  - 프로그램이 작다? 기계 친화적! (Machine Friendly)

```cpp
// #include <algorithm>

template<typename It, typename Fn>
auto for_each(It begin, It end, Fn fn) {

    // it := range [begin, end)
    for (auto it = begin; it != end; ++it) {
        fn(*it);
    }

    return fn;
}
```

접근법 비교

패러다임의 차이가 어떤 차이를 만드는가?

코드 변화 패턴

OOP

- 상속을 통해 확장된다
- 확장을 **통제**할 방법이 필요
  - **`virtual`**
  - `override`
  - `final`

GP

- 필요에 의해 생성된다
- 적합한 타입이 사용되었는지 **확인**할 방법이 필요
  - `static_assert`
  - **`Concepts`**

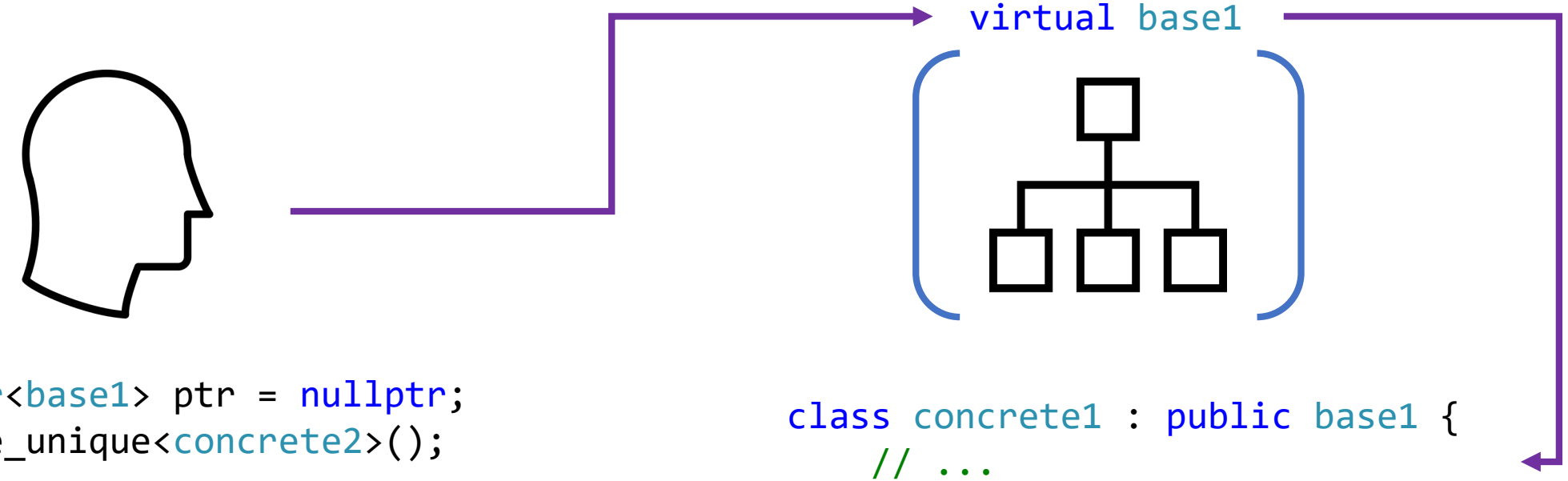
Virtual의 접근법

- 보증Guarantee
  - 타입 오류(잘못된 값/연산)가 발생하지 않도록 구현을 유도
  - Ex. Pure virtual function: 반드시 구현하도록 강제

  - **미래**의 코드 작성을 제약Constraint

- 엄격한 추상화를 통해 얻는 것?
  - 사려깊은 분석/설계를 유도
  - 좋은 구조를 가진 프로그램을 작성

Virtual을 그림으로 표현하면...




```
unique_ptr<base1> ptr = nullptr;
ptr = make_unique<concrete2>();
```

```
class concrete1 : public base1 {
    // ...
};
class concrete2 : public base1 {
    // ...
};
```

Icons are from www.flaticon.com
Face: authors/iconnice
Hierarchy: authors/smashicons

Concepts의 접근법

- 확정Assertion
  - 템플릿Template으로 코드를 생성하기 전에 타입인자들을 확인한다.
  - 기존의 템플릿 오류 내용이 복잡했던 이유
    == 먼저 코드를 생성한 이후에 문법/의미구조 검사를 적용

  - **현재**의 코드 생성을 제약

- 타입을 검사함으로써 얻는 것?
  - 알고리즘에 맞는 타입 만을 사용 (== Constraint)
  - 주어진 타입에 맞는 알고리즘을 적용 (== Overload Resolution)

Concepts를 그림으로 표현하면 …

```
struct concrete3 {
    void dispose();
    // ...
};

class concrete4 {
    void dispose();
    // ...
};
```

```
template <typename T>
concept disposable = requires(T res) {
    { res.dispose() } -> negligble;
};
```

Icons are from www.flaticon.com
Face: authors/iconnice
Filter: authors/freepik

코드 비교

어떻게 정의하고 적용할 수 있을까?

Warning!

- GCC-9 를 기준으로 작성된 코드입니다
  - 구체적인 버전: 9.2.0

- Human-Format이므로 가독성이 떨어져도 양해를...
  - Clang-Format이 아직 Concepts 를 처리 못합니다

- Concepts 헤더가 있는지 궁금하다면?

```
#if __has_include(<concepts>) // C++ 17 __has_include
#include <concepts>
#endif
```

```cpp
#include <catch2/catch.hpp>
#include <fmt/ostream.h>

TEST_CASE("print c++ info") {
    fmt::print("c++      : {}\n", __cplusplus);    // gcc: --std=c++1z
    fmt::print("concepts : {}\n", __cpp_concepts); // gcc: -fconcepts
}
```

Output

```
c++      : 201709
concepts : 201507
```

코드 워밍업

```cpp
// Concept definition
template <typename T>
concept integer_item = std::is_same_v<T, int>;

template <typename T>
concept string_item = (integer_item<T> == false) && requires(T a) {
    { fmt::format("{}\n", a) } -> std::string;
};

// Overloading
template <typename T>
    requires string_item<T>
void print_something(const T& ref) {
    return fmt::print("string item: {}\n", ref);
}

template <typename T>
    requires std::is_same_v<T, int> ||  // strange constraint,
             integer_item<T>            // but is possible
void print_something(const T& ref) {
    return fmt::print("integer item: {}\n", ref);
}
```

Concepts: Overloading

```cpp
// Just like normal template
TEST_CASE("overload resolution") {
    uint32_t v1 = 100;
    print_something(v1);
    int v2 = -100;
    print_something(v2);
}
```

Output

```
string item: 100
integer item: -100
```

```cpp
// Overloading
template <typename T>
    requires string_item<T>
void print_something(const T& ref) {
    return fmt::print("string item: {}\n", ref);
}

template <typename T>
    requires std::is_same_v<T, int> ||  // strange constraint,
             integer_item<T>            // but is possible
void print_something(const T& ref) {
    return fmt::print("integer item: {}\n", ref);
}
```

Concepts:
알고리즘 선택(Resolution)

```cpp
// Concept in the concept definition
template <typename T>
concept negligble = std::is_same_v<T, errc> ||
                    std::is_same_v<T, void>;

template <typename T>
concept disposable = requires(T res) {
    { res.dispose() } -> negligble;
};
```

Concepts:
Concept 정의에 Concept 사용

```
TEST_CASE("ref with virtual") {
    MyDisposable impl{};

    IVoidDisposable& item = impl;
    item.dispose();
}
```

Output

```
virtual MyDisposable::~MyDisposable()
```

```
TEST_CASE("ref with concept") {
    MyDisposable impl{};

    disposable& item = impl;
    item.dispose();
}
```

```
virtual MyDisposable::~MyDisposable()
```

Virtual & Concepts: Reference

```cpp
// Concept and Concept
void use_disposable(IVoidDisposable&) {
    puts(__PRETTY_FUNCTION__);
}

template <typename T>
    requires disposable<T>
void use_disposable(T&) {
    puts(__PRETTY_FUNCTION__);
}
```

```cpp
TEST_CASE("parameter with virtual") {
    MyDisposable impl{};
    IVoidDisposable& ref = impl;
    use_disposable(ref);
}
```

Output

```
void use_disposable(IVoidDisposable&)
```

Virtual & Concepts: Parameter

```cpp
// Concept and Concept
void use_disposable(IVoidDisposable&) {
    puts(__PRETTY_FUNCTION__);
}

template <typename T>
    requires disposable<T>
void use_disposable(T&) {
    puts(__PRETTY_FUNCTION__);
}
```

```cpp
TEST_CASE("paramter with concept") {
    // Generic!
    AnotherDisposable impl1{};
    use_disposable(impl1);

    // non-template
    MyDisposable impl2{};
    use_disposable(impl2);
}
```

Output

```
void use_disposable(T&) [with T = AnotherDisposable]
void use_disposable(T&) [with T = AnotherDisposable]
```

```cpp
// return for virtual
auto return_disposable_virtual() -> unique_ptr<IVoidDisposable> {
    puts(__PRETTY_FUNCTION__);
    return make_unique<MyDisposable>();
}

// return for concept
auto return_disposable_concrete() -> AnotherDisposable {
    puts(__PRETTY_FUNCTION__);
    return {};
}
```

Virtual & Concepts: Return

```
TEST_CASE("return with virtual") {
    unique_ptr<IVoidDisposable> ptr = return_disposable_virtual();
    ptr->dispose();
}
```

Output

```
std::unique_ptr<IVoidDisposable> return_disposable_virtual()
virtual MyDisposable::~MyDisposable()
```

```
TEST_CASE("return with concept") {
    disposable item = return_disposable_concrete();

    puts(typeid(decltype(item)).name());
    item.dispose();
}
```

Output

```
AnotherDisposable return_disposable_concrete()
17AnotherDisposable
std::errc AnotherDisposable::dispose()
```

정리를 해봅시다!

Virtual & Concepts

- Concepts
  - 타입을 검사할 뿐이므로 여러 타입에 적용 가능
  - 코드 증폭 문제의 소지는 여전히 존재
  - 일반적인 변수 처럼 사용할 수 있다 (`auto`-Like)

- Virtual
  - 엄격함. 한 타입이 명세를 구현했다는 것을 보증
  - 상속하지 않는 타입Out of hierarchy는 제약하지 않음
  - 항상 virtual이 강제되도록 사용 (Reference, Pointer ...)

감사합니다! Q & A ?

쉽게 이해할 수 있으셨나요?