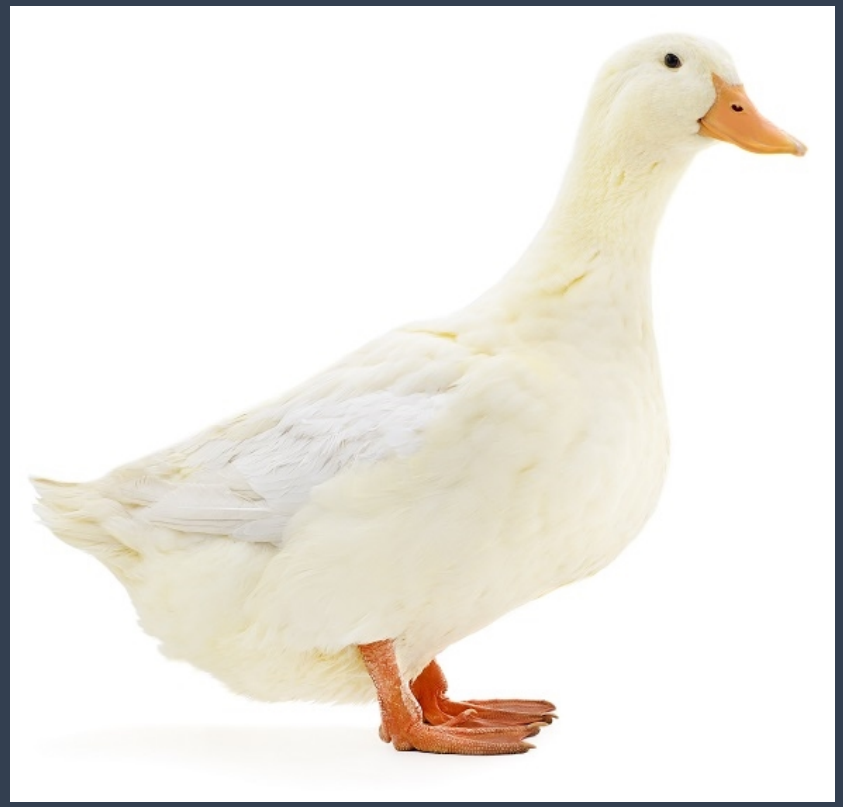
Strong Type

박재영

jaeyoungs.park@gmail.com

Intro



Intro

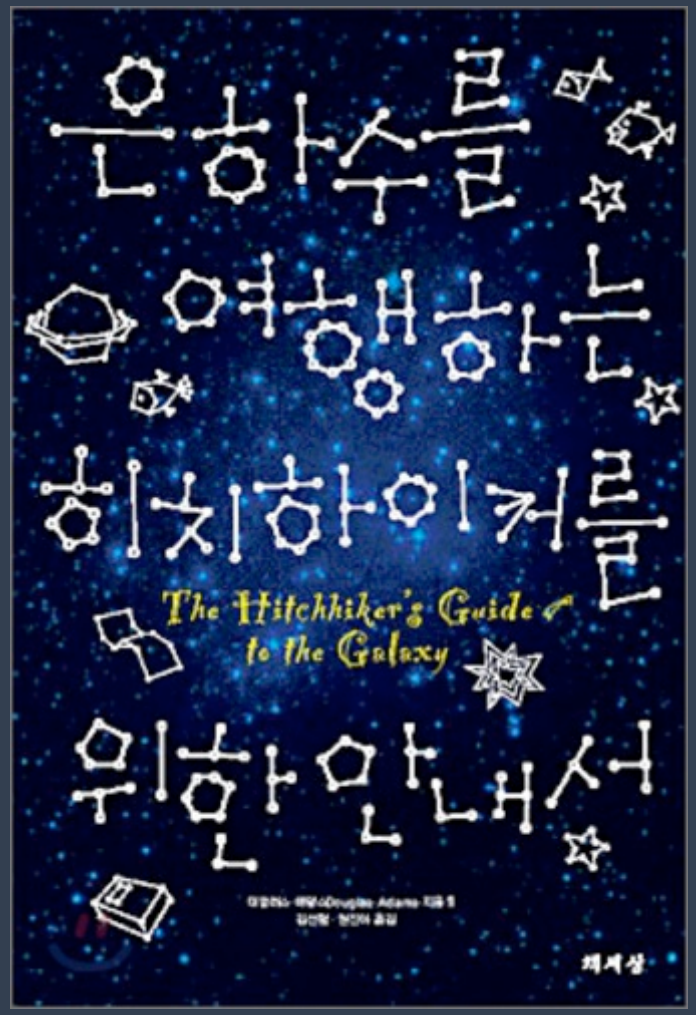
Intro

기억력 vs. 추상력

Intro

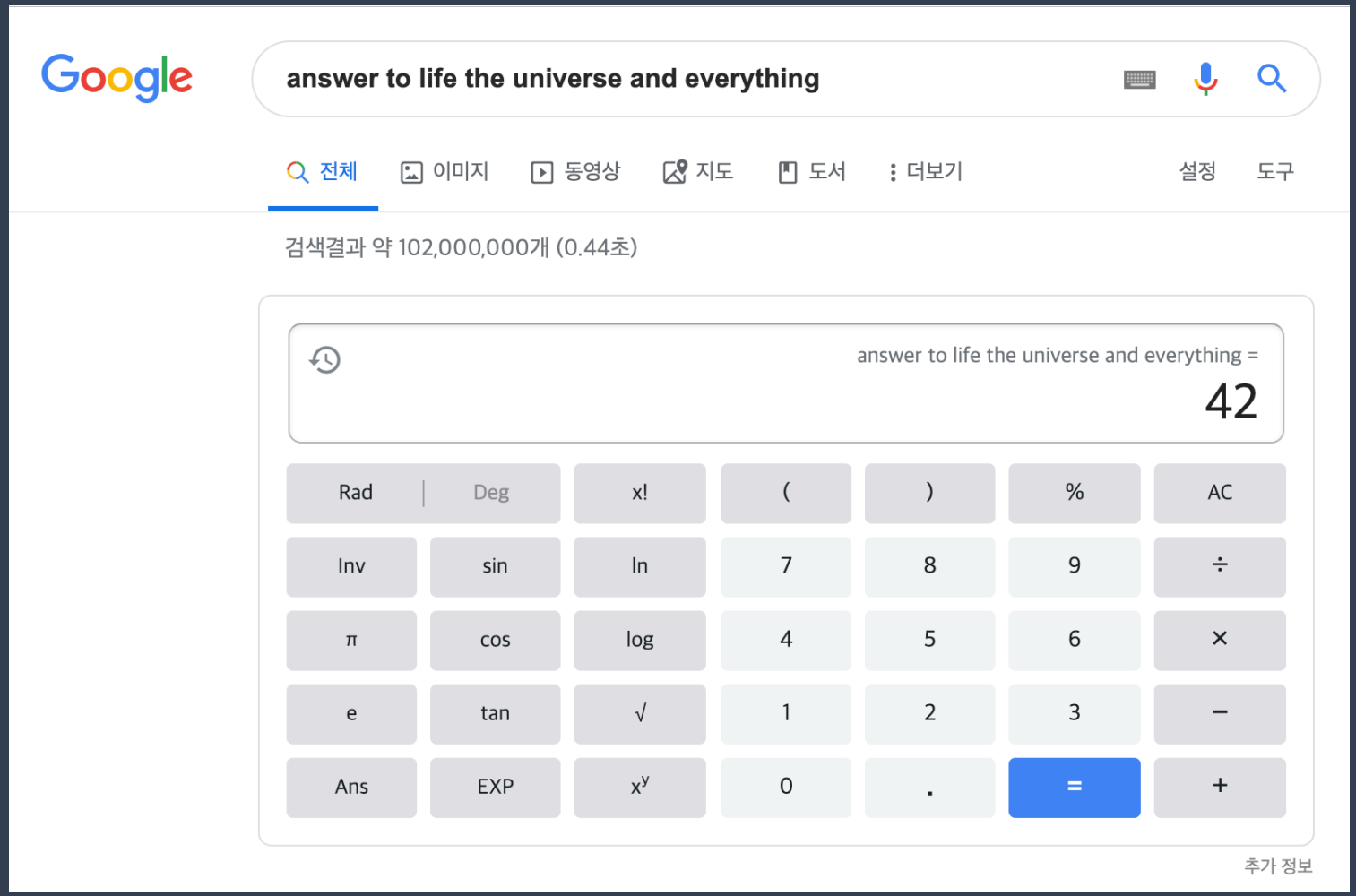
```
int validateImages(Image i1, Image i2) {

  int ducksInImage = countDucks(i1);

  int pianosInImage = countPianos(i2);

  return ducksInImage == pianosInImage;
}
```

Intro


은하수를
여행하는
히치하이커를
The Hitchhiker's Guide to the Galaxy
위한안내서
제우성


bruno murai

Intro


Google
answer to life the universe and everything
전체
이미지
동영상
지도
도서
더보기
설정
도구
검색결과 약 102,000,000개 (0.44초)
answer to life the universe and everything =
42
Rad
Deg
x!
(
)
%
AC
Inv
sin
ln
7
8
9
÷
π
cos
log
4
5
6
×
e
tan
√
1
2
3
−
Ans
EXP
x^y
0
.
=
+
추가 정보

Intro

```
int answerToLifeTheUniverseAndEverything(Image i1, Image i2){

  int ducksInImage = countDucks(i1);

  int pianosInImage = countPianos(i2);

  return (ducksInImage + pianosInImage) * 7;
}

// const int lifeTheUniverseAndEverything = 42;
```

Contents

- Strong Type
- Strong Type X Function
- Strong Type X Algebraic Data Type
- Limitation
- Conclusion

Strong Type

Strong Type : motivation

```
struct Duration {
  Duration(int second);
};

auto one_sec = Duration(1);
```

Strong Type : motivation

분해능이 millisecond 추가

Strong Type : motivation

```
struct Duration {
  Duration(int second);
  // OOPS
  Duration(int millisecond);
};

auto one_sec = Duration(1);
auto one_msec = Duration(???);
```

Strong Type : motivation

Solution 1. Tag Dispatch

Strong Type : motivation

```
// tag dispatch.
struct second_tag {};
struct millisecond_tag {};

struct Duration {
  Duration(int millisecond, millisecond_tag);
  Duration(int second, second_tag);
};

auto one_sec = Duration(1, second_tag());
auto one_msec = Duration(1, millisecond_tag());
```

Strong Type : motivation

Solution 2. Builder / Factory Pattern

Strong Type : motivation

```
// builder/factory pattern.
struct Duration {
  Duration(int millisecond);
};

DurationFactory factory;
auto one_sec = factory.millisecond(1);
auto one_msec = factory.second(1);
```

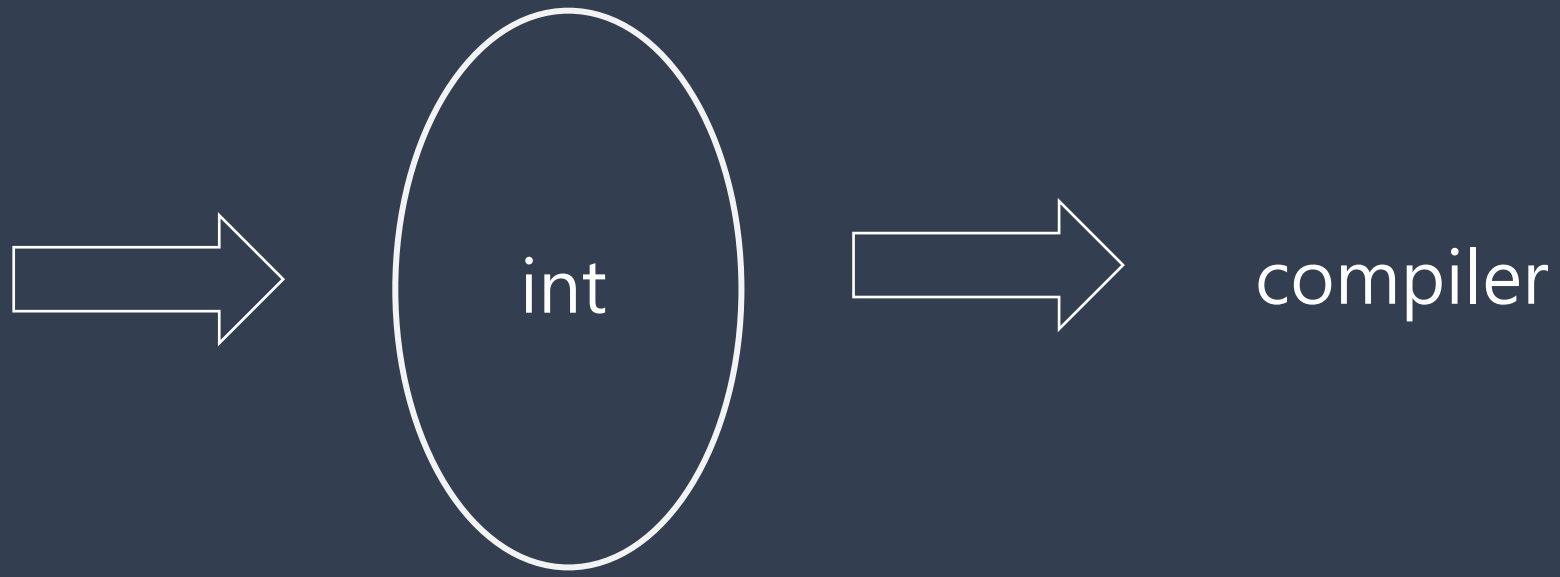
Strong Type : motivation

"Over Abstraction"

Strong Type : motivation

sec

msec



Strong Type : motivation

Solution 3. Strong Type

Strong Type : motivation

```
struct second;
struct millisecond;

auto one_sec = second(1); // std::chrono::second
auto one_msec = millisecond(1); // std::chrono::millisecond
```

Strong Type : motivation

Type	Definition
std::chrono::nanoseconds	duration</*signed integer type of at least 64 bits*/, std::nano>
std::chrono::microseconds	duration</*signed integer type of at least 55 bits*/, std::micro>
std::chrono::milliseconds	duration</*signed integer type of at least 45 bits*/, std::milli>
std::chrono::seconds	duration</*signed integer type of at least 35 bits*/>
std::chrono::minutes	duration</*signed integer type of at least 29 bits*/, std::ratio<60>>
std::chrono::hours	duration</*signed integer type of at least 23 bits*/, std::ratio<3600>>
std::chrono::days (since C++20)	duration</*signed integer type of at least 25 bits*/, std::ratio<86400>>
std::chrono::weeks (since C++20)	duration</*signed integer type of at least 22 bits*/, std::ratio<604800>>
std::chrono::months (since C++20)	duration</*signed integer type of at least 20 bits*/, std::ratio<2629746>>
std::chrono::years (since C++20)	duration</*signed integer type of at least 17 bits*/, std::ratio<31556952>>

Strong Type : motivation

"Right Abstraction"

Strong Type : motivation

Strong Type is

A type used in place of another type to carry specific meaning through its name.

Strong Type is

A type used in place of another type to carry specific meaning through its name.

"Right Abstraction".

Strong Type X Function

Strong Type X Function : example1

```
auto contains(std::string directory, std::string filepath) -> bool;

auto dir = "/a/b"s;
auto file = "/a/b/c.mp3"s;

// OOPS
contains(file, dir);
```

Strong Type X Function : example1

```
auto contains(Directory d, File f) -> bool;

auto dir = Directory("/a/b"s);
auto file = File("/a/b/c.mp3"s);

// COMPILE ERROR.
contains(file, dir);
```

Strong Type X Function : example2

```
auto exist(std::string file) -> bool;

auto exist(std::string directory) -> bool;
```

Strong Type X Function : example2

```
auto exist_directory(std::string directory) -> bool;

auto exist_file(std::string file) -> bool;
```

Strong Type X Function : example2

```
auto exist(File f) -> bool;

auto exist(Directory d) -> bool;
```

Strong Type X Function : sematic & syntax

"Type Safety"

Strong Type X Function : sematic & syntax

```
auto contains(std::string directory, std::string filepath) -> bool;

auto dir = "/a/b"s;
auto file = "/a/b/c.mp3"s;

// OOPS
contains(file, dir);
```

- Semantic & Syntax

“File이 Directory를 포함하고 있는가?”

Syntax : 문제 없음.
Sematic : 말이 안됨.

Strong Type X Function : sematic & syntax

```
auto contains(Directory d, File f) -> bool;

auto dir = Directory("/a/b"s);
auto file = File("/a/b/c.mp3"s);

// COMPILE ERROR.
contains(file, dir);
```

- Semantic & Syntax

"File이 Directory를 포함하고 있는가?"

Syntax : ~~문제 없음~~ -> 문제 있음.
Sematic : 말이 안됨.

Strong Type X Function : sematic & syntax

"Readability"

Strong Type X Function : sematic & syntax

```
auto exist(std::string file) -> bool;
auto exist(std::string directory) -> bool;
```

- Semantic & Syntax

"File이 존재하는가?"
"Directory가 존재하는가?"

Syntax : 문제 있음.
Sematic : 말이 됨.

Strong Type X Function : sematic & syntax

```
auto exist(File f) -> bool;
auto exist(Directory d) -> bool;
```

- Semantic & Syntax

"File이 존재하는가?"
"Directory가 존재하는가?"

Syntax : ~~문제 있음~~ -> 문제 없음
Sematic : 말이 됨.

Strong Type X Function : sematic & syntax

```
auto exist(File f) -> bool;
auto exist(Directory d) -> bool;

auto exist_directory(std::string directory) -> bool;
auto exist_file(std::string file) -> bool;
```

Strong Type X Function : sematic & syntax

"Generic Programming"

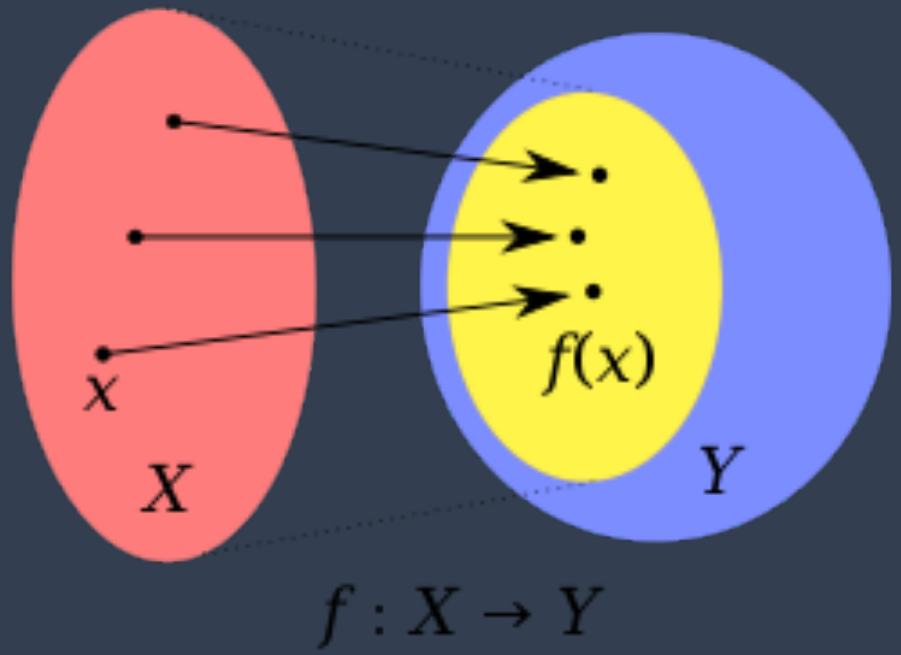
Strong Type X Function : sematic & syntax

```
auto exist(File f) -> bool;
auto exist(Directory d) -> bool;

template<typename T>
auto foo(T t) -> bool {
    //...
    exist(t);
    //...
}
```

Strong Type X Function : sematic & syntax

"Testability"

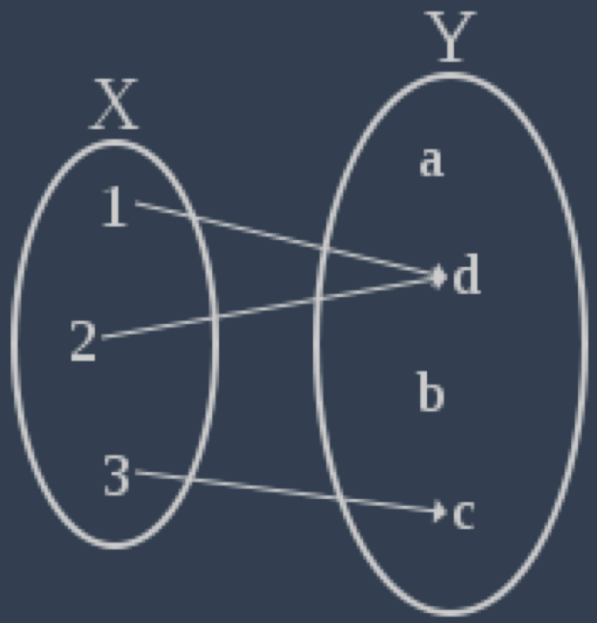
Strong Type X Function : total function



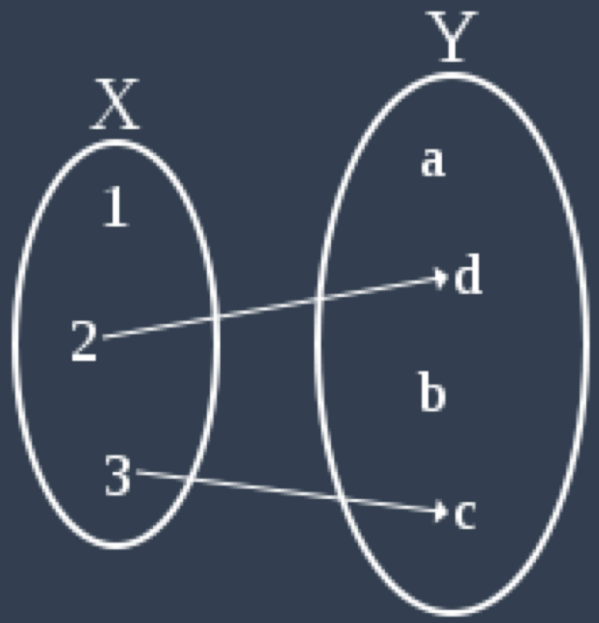
Domain: X
Codomain: Y
Function: f
Image(Range): f(X)

Strong Type X Function : total function

- Total Function vs. Partial Function



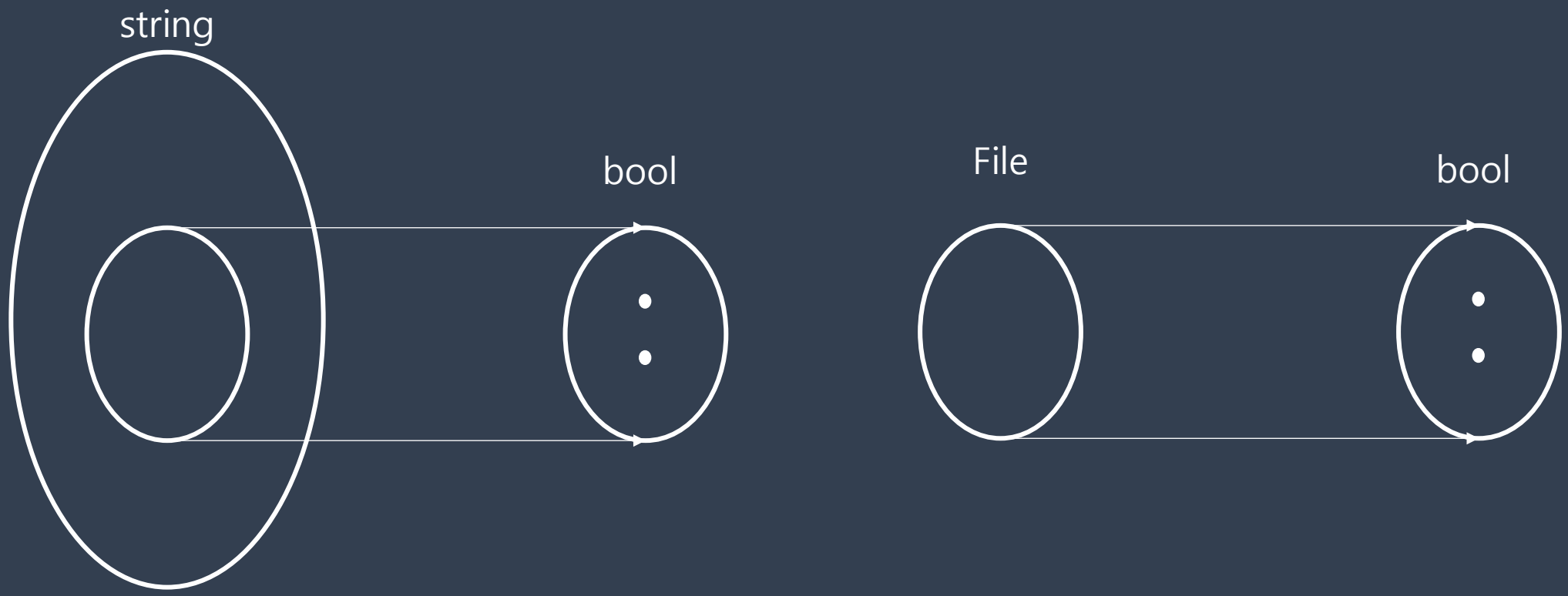
total



partial

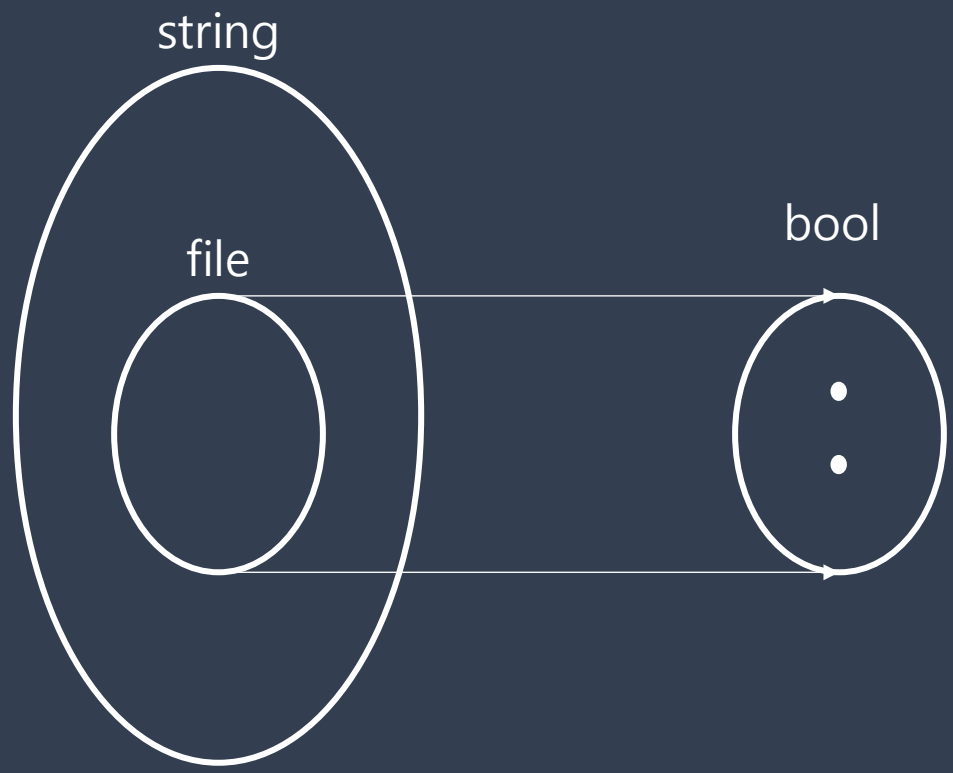
Strong Type X Function : total function

```
auto exist(std::string file) -> bool;

auto exist(File f) -> bool;
```



Strong Type X Function : total function

```
auto exist(std::string file) -> bool {
  if(!isFile(file)) {
    throw std::invalid_argument();
  }
  // ...
}

auto exist(File f) -> bool {
  // ...
}
```



Strong Type X Algebraic Data Type

Strong Type X Algebraic Data Type

Algebraic Data Type

*"an **algebraic data type** is a kind of composite type,*
i.e., a type formed by combining other types"

Algebra

"숫자 대신 문자를 사용하여 방정식을 풀거나, 구조를 연구하는 학문."

Strong Type X Algebraic Data Type

Algebra + Data Type

Sum Type : A+B

- optional<T>
- variant<A, B>

Product Type : A x B

- tuple<A, B>

List Type : 0 + A + (A x A) + (A x A x A) + ...

- vector<A>

Strong Type X Product Type

Strong Type X Product Type

```
/* requirement.
 * 1. artist, album , title로 구성된 Metadata
 * 2. Metadata를 화면에 출력하는 함수.
 */
```

Strong Type X Product Type

```
/* requirement.
 * 1. artist, album , title로 구성된 Metadata
 * 2. Metadata를 화면에 출력하는 함수.
 */

using Metadata = std::tuple<std::string, std::string, std::string>;

auto show(std::string artist, std::string album, std::string title) -> void;

// client
Metadata meta;
auto album  = std::get<0>(meta);
auto artist = std::get<1>(meta);
auto title  = std::get<2>(meta);

show(album, artist, title);
```

Strong Type X Product Type : with strong type

```
using Metadata = std::tuple<Artist, Album, Title>;

auto show(Artist artist, Album album, Title title) -> void;

// client
Metadata meta;

auto album = std::get<Album>(meta);
auto artist = std::get<Artist>(meta);
auto title = std::get<Title>(meta);

// show metadata
show(album, artist, title);
```

Strong Type X Product Type : structured binding

```
using Metadata = std::tuple<Artist, Album, Title>;

auto show(Artist artist, Album album, Title title) -> void;

// client
Metadata meta;

// structured binding : c++ language feature.
auto [artist, album, title] = meta;

// show metadata
show(artist, album, artist);
```

Strong Type X Product Type : higher order function

```
using Metadata = std::tuple<Aritst, Album, Title>;

auto show(Artist artist, Album album, Title title) -> void;

// client
Metadata meta;

// show metadata
std::apply(show, meta);
```

Strong Type X Sum Type

Strong Type X Sum Type

```
/* requirement
 * 1. file을 재생.
 * 2. audio file과 video file의 화면은 다르다.
 */

auto play(File f) -> void;
```

Strong Type X Sum Type

```
/* requirement
 * 1. file을 재생.
 * 2. audio file과 video file의 화면은 다르다.
 */

auto play(Audio a) -> void;

auto play(Video v) -> void;
```

Strong Type X Sum Type

```
/* limitation: 우리가 얻을 수 있는 정보는 File */

auto convert(File f) -> Audio;

auto convert(File f) -> Video;
```

Strong Type X Sum Type

```
/* limitation: 우리가 얻을 수 있는 정보는 File */

auto convertToAudio(File f) -> Audio;

auto convertToVideo(File f) -> Video;
```

Strong Type X Sum Type

```
/* limitation: 우리가 얻을 수 있는 정보는 File */

auto convertToAudio(File f) -> Audio;

auto convertToVideo(File f) -> Video;
/* client code */
{
  try {
    auto a = convertToAudio(f);
    play(a);
    return;
  } catch(...) { }
  try {
    auto v = convertToVideo(f);
    play(v);
    return;
  } catch(...) { }
}
```

Strong Type X Sum Type

```
/* limitation: 우리가 얻을 수 있는 정보는 File */

auto convertToAudio(File f) -> Audio;

auto convertToVideo(File f) -> Video;
/* client code */
{
  if(isAudio(f)) {
    auto a = convertToAduio(f);
    play(a);
  } else if(isVideo(f)) {
    auto v = convertToVideo(f);
    play(v);
  } else {
    // ...
  }
}
```

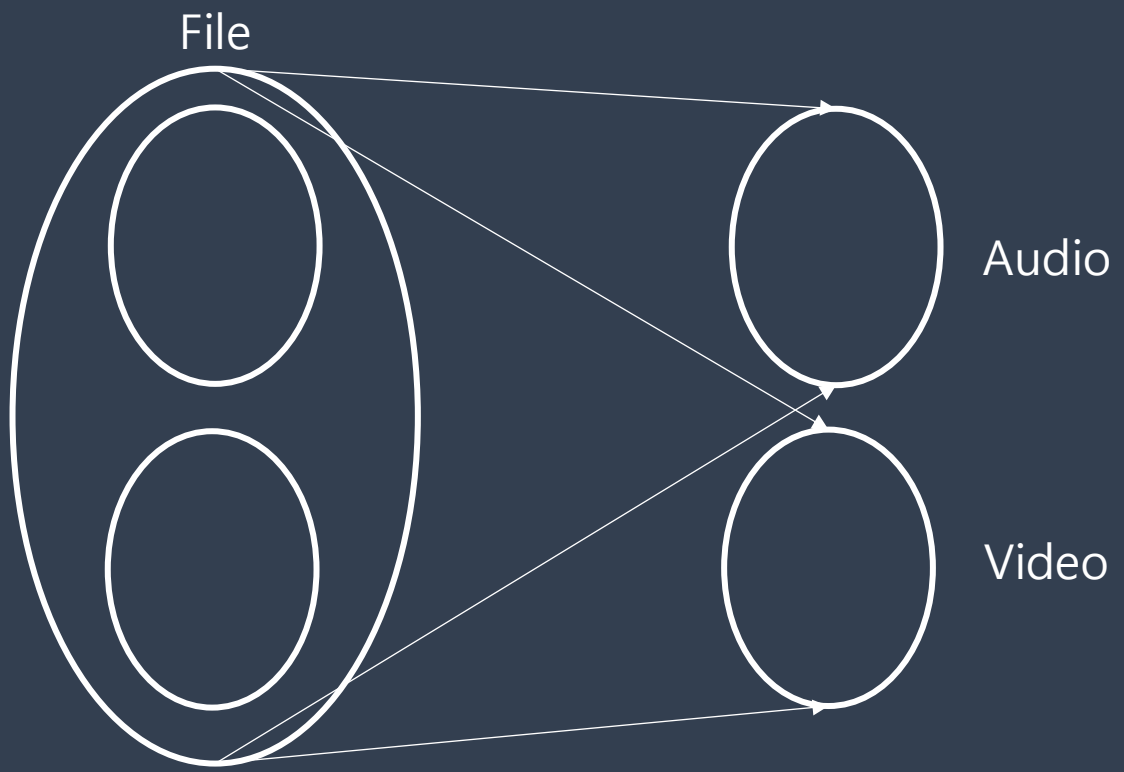
Strong Type X Sum Type

```
/* limitation: 우리가 얻을 수 있는 정보는 File */

auto convertToAudio(File f) -> std::optional<Audio>;

auto convertToVideo(File f) -> std::optional<Video>;
/* client code */
{
  auto audio = convertToAudio(File);
  if (audio) { play(audio.get()); return; }

  auto video = convertToVideo(File);
  if (video) { play(video.get()); return; }
}
```

Strong Type X Sum Type

"Under Abstraction".

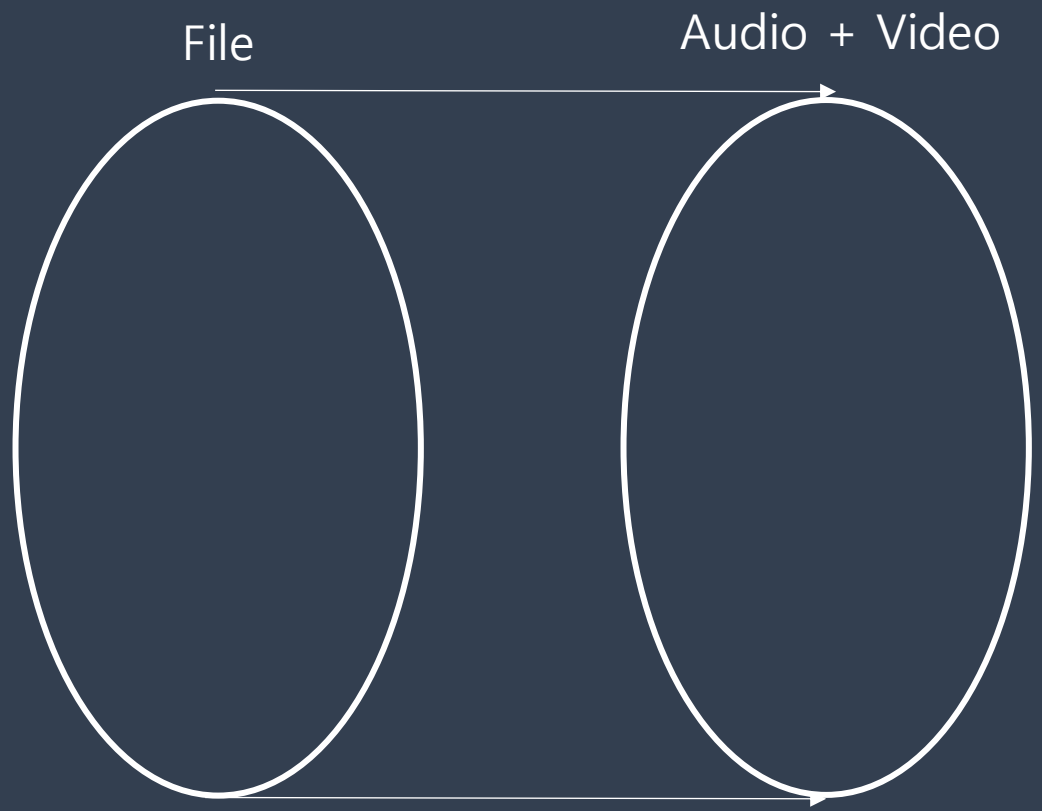
Strong Type X Sum Type



Strong Type X Sum Type

```
auto convert(File f) -> std::variant<Audio, Video>;

/* client code */
File f;
auto media = convert(f);
std::visit([](auto v){ play(v); }, media);
```

Strong Type X Sum Type

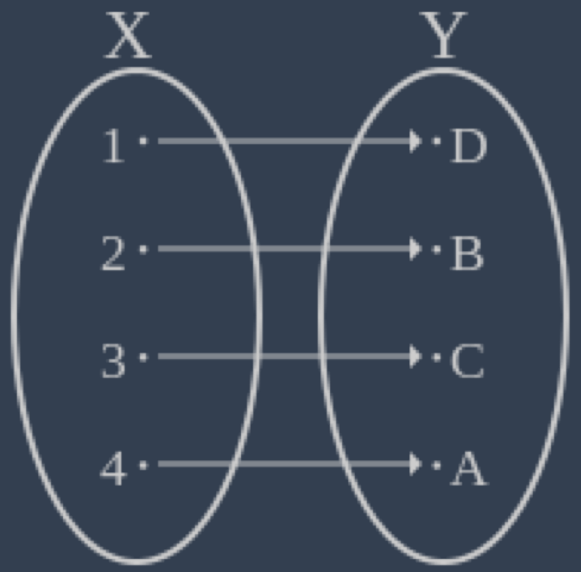


Strong Type X Sum Type

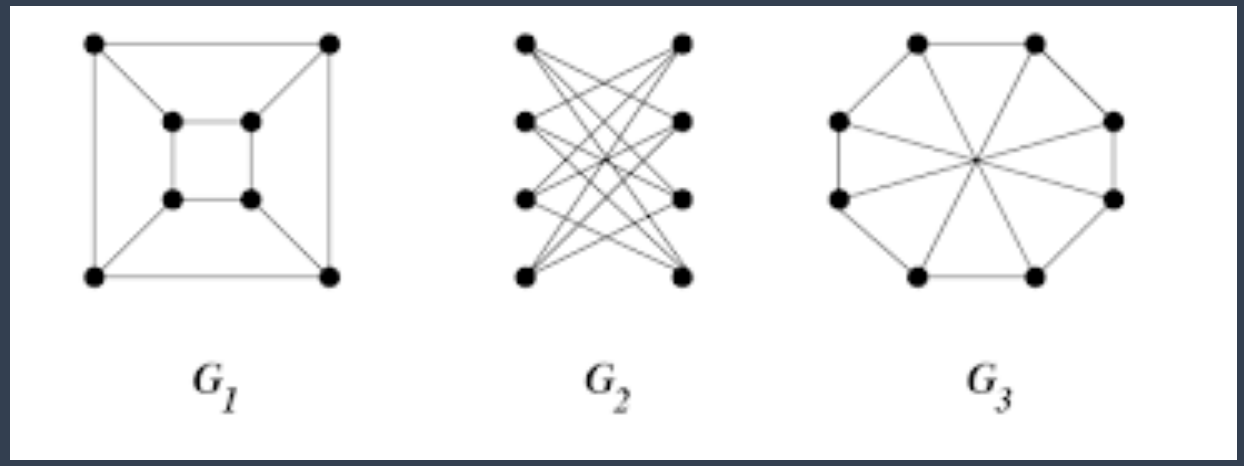
"Reusability"

Strong Type X Sum Type

- Isomorphism



bijective = injective + surjective



Strong Type X Sum Type

```
auto convert(File f) -> std::variant<Audio, Video>;

File f;
auto media = convert(f);
std::visit([](auto v){ play(v); }, media);


/* ************************ */
auto convert(File f) -> std::shared_ptr<IPlayable>;

File f;
auto media = convert(f);
media.play();
```

Strong Type X List Type

Strong Type X List Type

```
struct Duck {
  auto fly() -> void;
  auto swim() -> void;
};

struct Parrot {
  auto fly() -> void;
};
```

Strong Type X List Type

```
struct Bird {
  virtual auto fly() -> void = 0;
};

struct Duck : public Bird {
  auto fly() -> void override;
  auto swim() -> void;
};

struct Parrot : public Bird {
  auto fly() -> void override;
};

std::vector<std::shared_ptr<Bird>> birds;
```

Strong Type X List Type

```
// AT THE MOUNTAIN
for(auto bird : birds) {
  bird->fly();
}

// AT THE LAKE
for(auto bird : birds) {
  if(canSwim(bird)) {
    bird->swim();
  } else {
    bird->fly();
  }
}
```

Strong Type X Sum Type

"Static Polymorphism"

Strong Type X List Type

```
std::vector<std::variant<Duck, Parrot>> birds;

// AT THE LAKE
for(auto bird : birds) {
  std::visit(Ovld{[](Duck d)  { d.swim(); },
                  [](Parrot p){ p.fly(); }}, bird);
}
```

Strong Type X List Type

```
template<Swimmable T> void playAtLake(T); // #1
template<typename T> void playAtLake(T); // #1

std::vector<std::variant<Duck, Parrot>> birds;

// AT THE LAKE
for(auto bird : birds) {
  std::visit([](auto b){ playAtLake(b); }, bird);
}
```

Limitation

Limitation

- 수많은 타입들을 설계해야 한다.....

Limitation

- Current
  - Toward Opaque Typedefs for C++1Y (WG21/N3515)
  - Etc
- 3rd Party
  - Boost
  - named_type
  - type_safe

Limitation

- Why Standard didn't?

```
struct Widget {
  auto swap(Widget& w) -> void;
};

// assume 'newtype' keyword.
using Gadget = newtype Widget;

Gadget x;
Gadget y;
x.swap(y);
```

Limitation

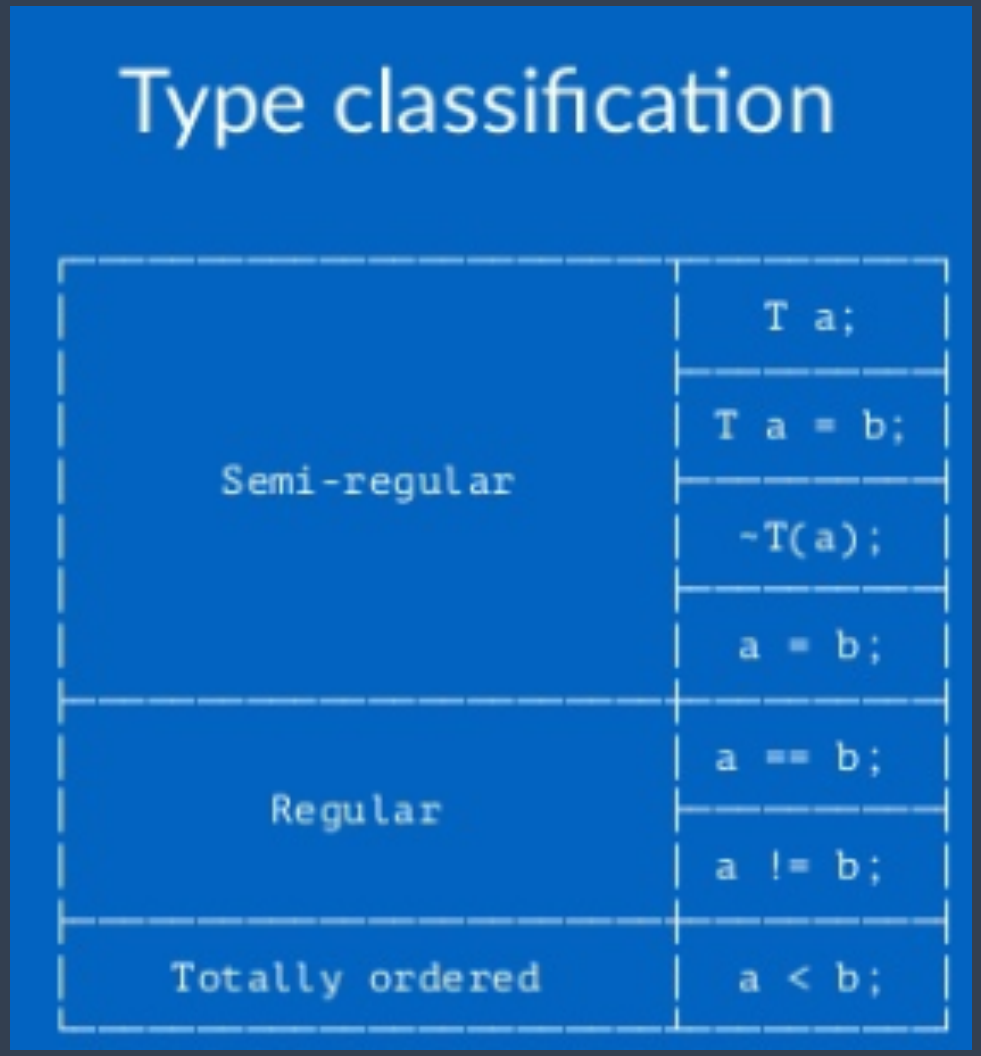
- Value Sematic & Move Sematic
  - Rule Of Five / Three

Limitation

"There is a set of procedures whose inclusion in the computational basis of a type lets us place objects in data structures and use algorithms to copy objects from one data structure to another. We call types having such a basis regular, since their use guarantees **regularity** of behavior and, therefore, interoperability."

– Elements of Programming, Section 1.5

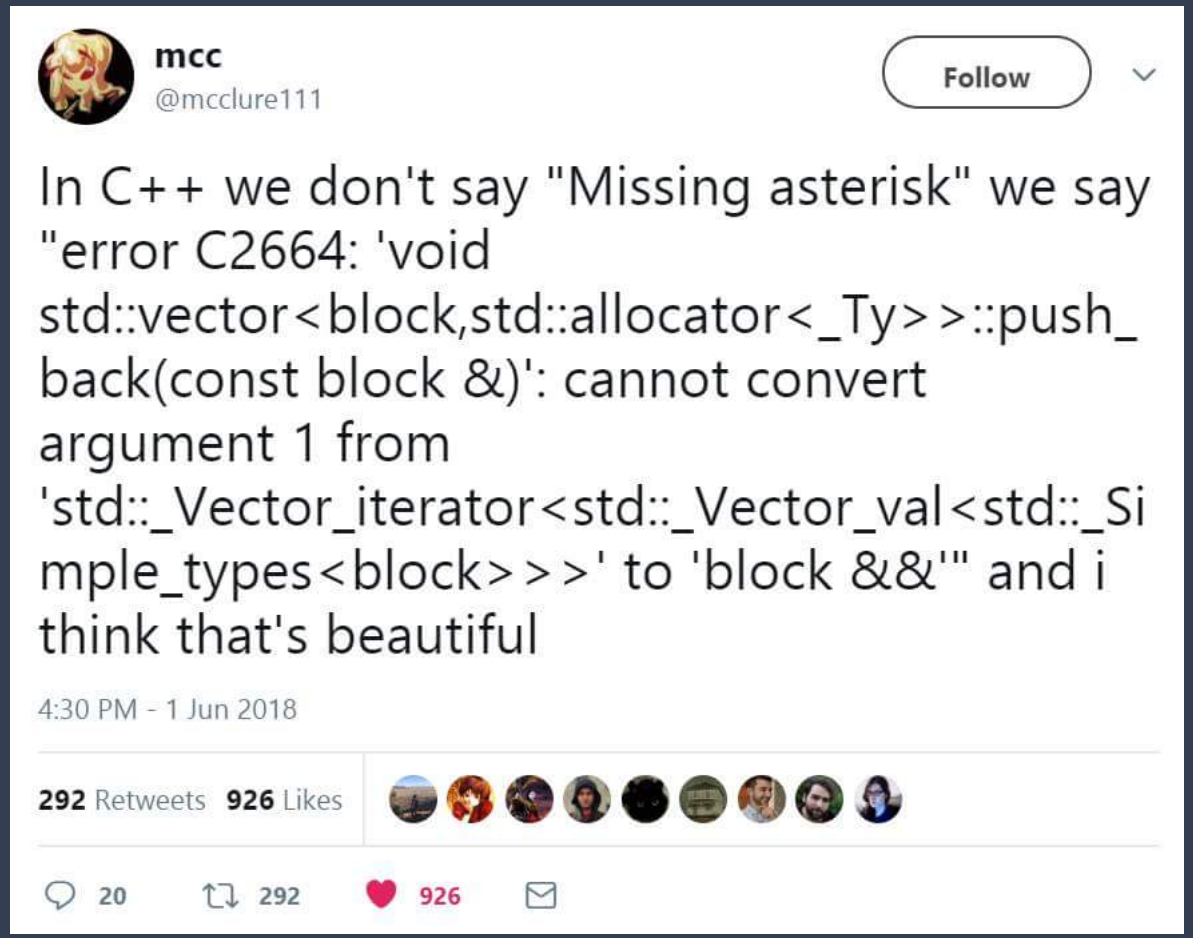
Limitation




```
template<typename T>
void swap(T& lhs, T &rhs) {
  T tmp = lhs; // copy constructor.
  lhs = rhs; // assignment operator.
  rhs = tmp;
}
```

Limitation

- Binary Compatibility
- Compiler Error / Debugging



Limitation

```
std::variant<A,B,C,D, ..... >
```

Limitation

- 수학의 함수 vs. 소프트웨어 함수

Conclusion

- How to design Types?
  - Over Abstraction <- GETTING THINGS RIGHT -> Under Abstraction

- Type을 보다 정확히 설계함으로써
  - Type Safety (Syntax)
  - Readability (Sematic)
  - Standard Supported Tool (Algebraic Data Type)
  - Testability (Total Function)
  - Reusability (Isomorphism)
  - Polymorphism w/o Inheritance (is-a relation)

Reference

- Regularity
  - [ppt] https://www.slideshare.net/ilio-catallo/regular-types-in-c
  - [book] Element Of Programming
  - [book] Mathmatics to Generic Programming
- Polymorphism
  - Better Code: Runtime Polymorphism
- Abstraction
  - Abstract Art: Getting Abstraction "Just Right"

Bonus

```
int validateImages(Image i1, Image i2) {

  int ducksInImage = countDucks(i1);

  int pianosInImage = countPianos(i2);

  return ducksInImage == pianosInImage;
}
```

Bonus

```
int answerToLifeTheUniverseAndEverything(Image i1, Image i2){

  int ducksInImage = countDucks(i1);

  int pianosInImage = countPianos(i2);

  return (ducksInImage + pianosInImage) * 7;
}

// const int lifeTheUniverseAndEverything = 42;
```