An unhinged introduction to Ada/SPARK

A real showcase of the Ada/SPARK language

Fernando Oleo Blanco - Irvise

Dis-fucking-claimer

Don’t take it too seriously
This is supposed to be entretaining too

My words are mine only

Join in for the ride!

Table of contents

1. Why a presentation like this?
2. A 60 seconds overview of Ada
3. Types, types, types aaaandddd... more types! Also, data!
4. Functions, procedures and exceptions
5. Packages
6. Generics
7. Tasks
8. Liquid types, contracts and proves! Fuck yeah!
9. Going beyond, more resources
10. Ada also sucks balls

Why a presentation like this?

Why a presentation like this?

- People really do not know how powerful Ada/SPARK is

Why a presentation like this?

- People really do not know how powerful Ada/SPARK is
- Most presentations are only introductory
  - The advance stuff is what makes Ada great!

Why a presentation like this?

- People really do not know how powerful Ada/SPARK is
- Most presentations are only introductory
  - The advance stuff is what makes Ada great!
- To make a direct comparison with modern C++ and Rust

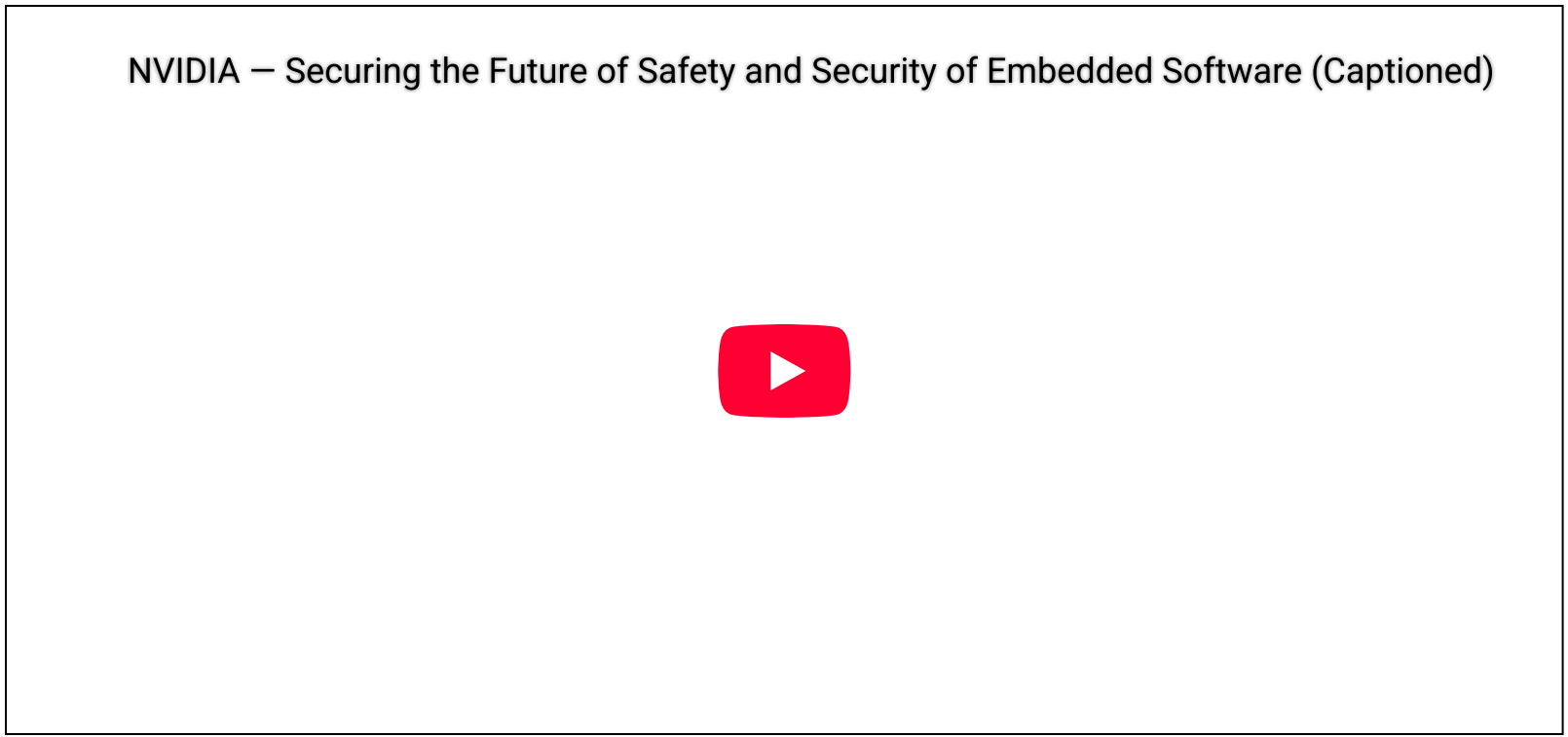
Why a presentation like this?

- People really do not know how powerful Ada/SPARK is
- Most presentations are only introductory
  - The advance stuff is what makes Ada great!
- To make a direct comparison with modern C++ and Rust
- Because Ada/SPARK is wonderful

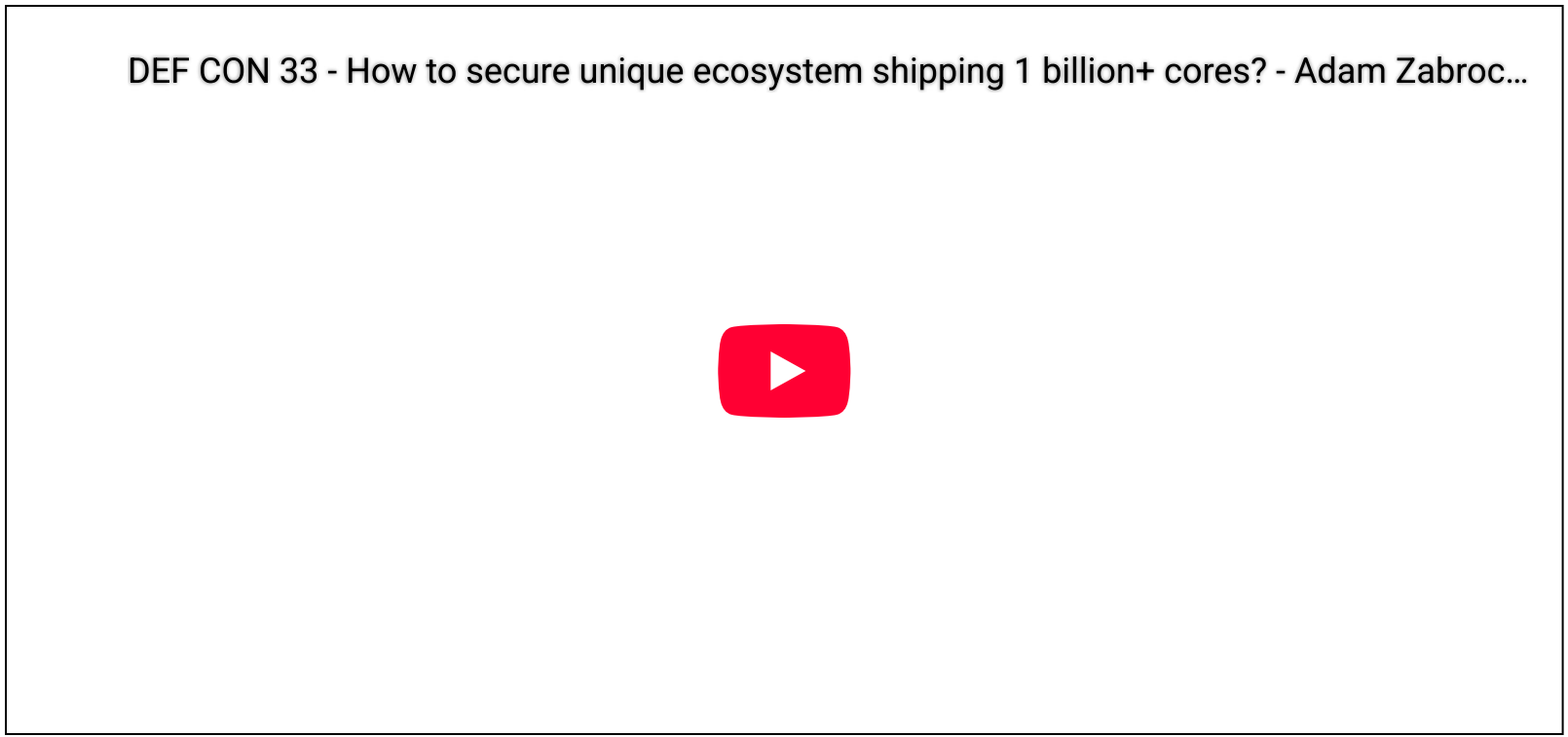
Why a presentation like this?

- People really do not know how powerful Ada/SPARK is
- Most presentations are only introductory
  - The advance stuff is what makes Ada great!
- To make a direct comparison with modern C++ and Rust
- Because Ada/SPARK is wonderful

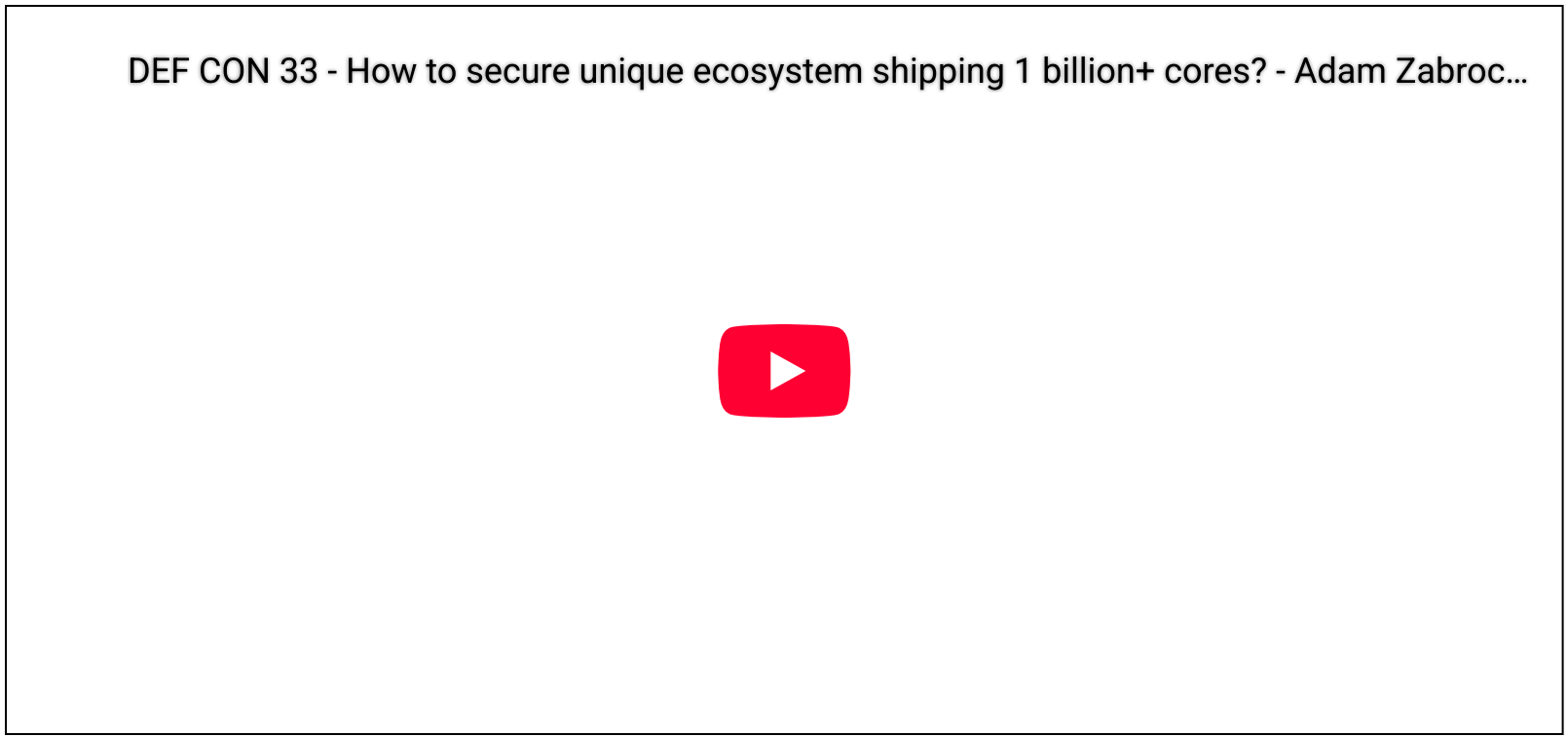
NVIDIA?


NVIDIA — Securing the Future of Safety and Security of Embedded Software (Captioned)

NVIDIA?


DEF CON 33 - How to secure unique ecosystem shipping 1 billion+ cores? - Adam Zabroc...

NVIDIA?



NVIDIA ISO-26262 SPARK guidelines. Also, NVIDIA certifies to SIL-4 a 7 MLOC OS written in Ada/SPARK (2025). More information in this ACM article (2024).

Follow along and test the features!

Learn.AdaCore

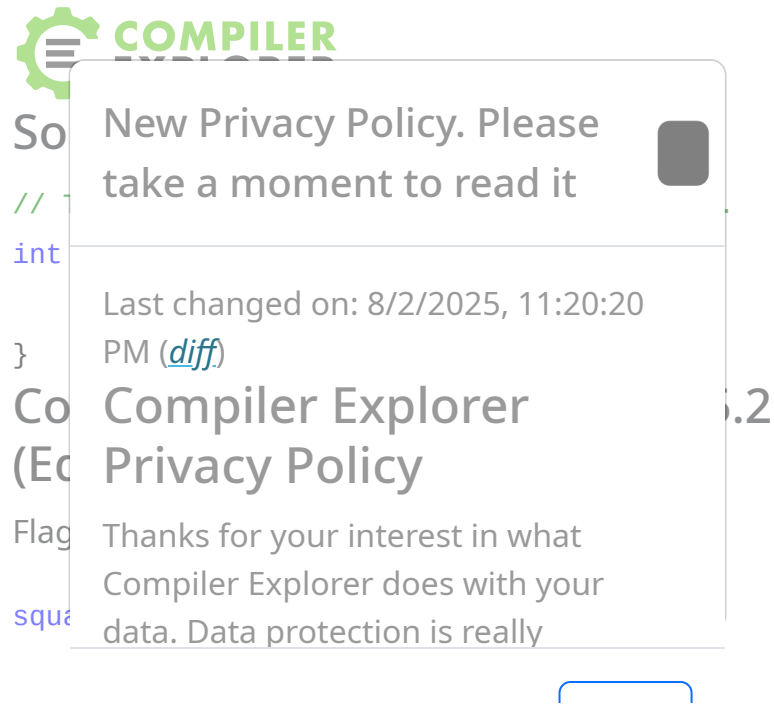
LEARN.ADACORE.COM

Edit on GitHub

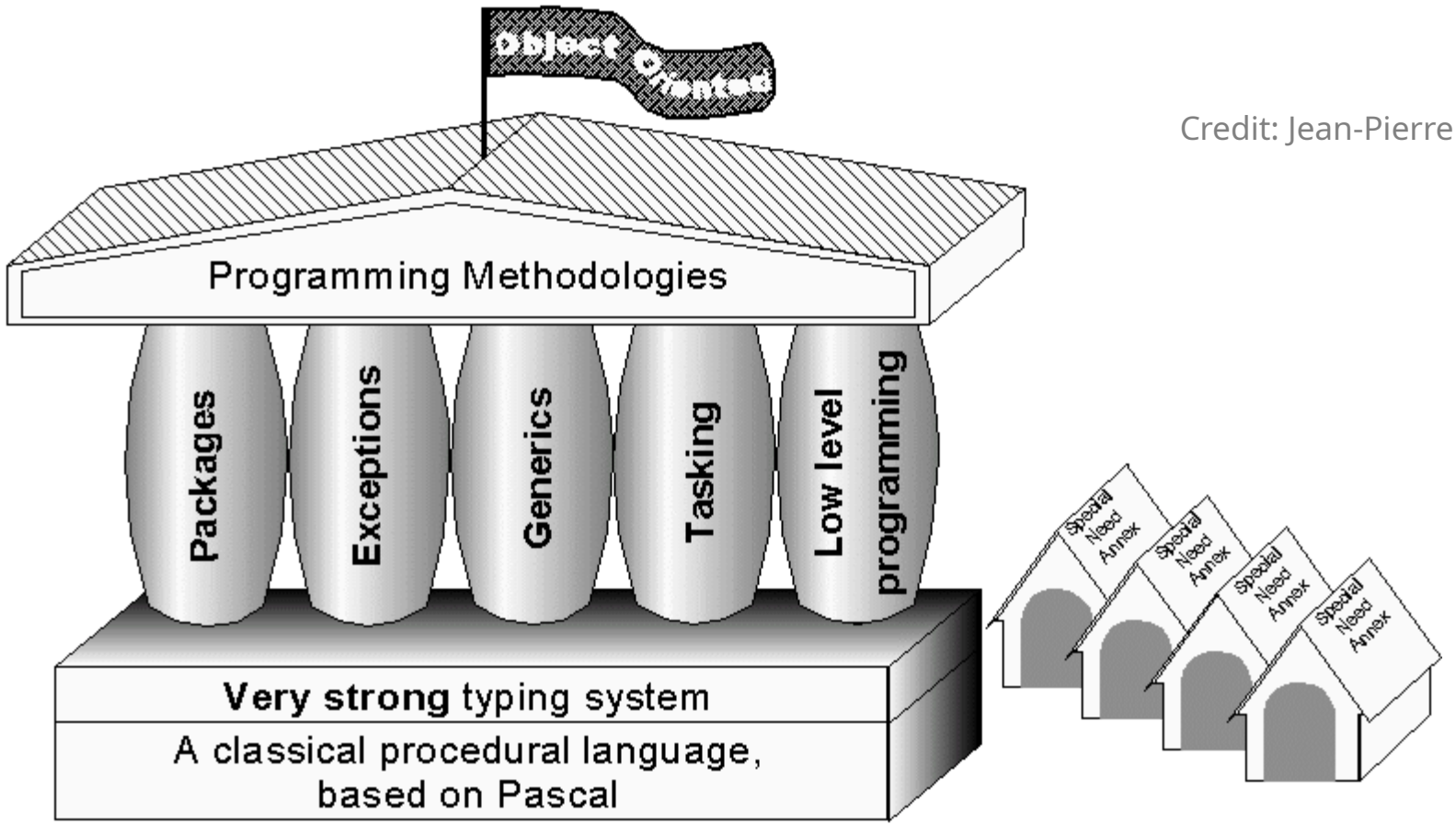
What is Ada and SPARK?

Ada is a state-of-the art programming language

Compiler Explorer/Godbolt



A 60 seconds overview of Ada



Credit: Jean-Pierre Rosen

A 60 seconds overview of Ada II

Ada is an open ISO standard (ISO/IEC 8652)

- Ada 83 (ANSI/MIL-STD-1815A): 348 pages (US letter)
- Ada 2012: 832 pages (A4)
- Ada 2022: 1048 pages (A4)
- There is a "Rationale"/Annotated (AARM) version that explains all decissions made
- **Meant to be used by all Ada programmers!!!**

A 60 seconds overview of Ada II

Ada is an open ISO standard (ISO/IEC 8652)

- Ada 83 (ANSI/MIL-STD-1815A): 348 pages (US letter)
- Ada 2012: 832 pages (A4)
- Ada 2022: 1048 pages (A4)
- There is a "Rationale"/Annotated (AARM) version that explains all decissions made
- **Meant to be used by all Ada programmers!!!**

C/C++ are ISO standards

- C90 (ISO/IEC 9899): 219 pages (A4)
- C23: 758 pages (A4)
- C++98 (ISO/IEC 14882, not open): 732 pages (A4)
- C++23: 2104 pages (A4)
- Standard not meant to be read by the general public (ISOCPP). Only drafts available in Github

A 60 seconds overview of Ada II

Ada is an open ISO standard (ISO/IEC 8652)

- Ada 83 (ANSI/MIL-STD-1815A): 348 pages (US letter)
- Ada 2012: 832 pages (A4)
- Ada 2022: 1048 pages (A4)
- There is a "Rationale"/Annotated (AARM) version that explains all decissions made
- **Meant to be used by all Ada programmers!!!**

C/C++ are ISO standards

- C90 (ISO/IEC 9899): 219 pages (A4)
- C23: 758 pages (A4)
- C++98 (ISO/IEC 14882, not open): 732 pages (A4)
- C++23: 2104 pages (A4)
- Standard not meant to be read by the general public (ISOCPP). Only drafts available in Github

Rust *is* standarised (no ISO)

- Added in 2023. Found in The Rust Reference.
- New releases every 6 weeks... Does not cover STL... 849 pages (no official pdf). There is Ferrocene...

Lets get started fucking dammit!

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player’s life in a videogame?

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    float life;
} Player;
```

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    float life = 100.0; // C++11
} Player;
```

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
```

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
```

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
```

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Player_T is record
   ...
   Life : Float; -- Naïve approach
end record;  -- Ada 83

Player : Player_T;
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T;
end record;  -- Ada 83

Player : Player_T;
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := 100.0;
end record;  -- Ada 83

Player : Player_T;
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
Player.Life :=  1000.0; -- Runtime exception
                        -- Comp-time warning
                        -- SPARK: error
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
Player.Life :=  1000.0; -- Runtime exception
                        -- Comp-time warning
                        -- SPARK: error
Player.Life := -1000.0; -- Same as above
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
Player.Life :=  1000.0; -- Runtime exception
                        -- Comp-time warning
                        -- SPARK: error
Player.Life := -1000.0; -- Same as above
Player.Life :=  100;    -- Compile time error
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```cpp
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```ada
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
Player.Life :=  1000.0; -- Runtime exception
                        -- Comp-time warning
                        -- SPARK: error
Player.Life := -1000.0; -- Same as above
Player.Life :=  Life_T(100);
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

Types, types, types aaaandddd... more types! Also, data!

Lets start with an example. How do you declare a player's life in a videogame?

C++

```
struct Player_t {
    ...
    const float  MAX_LIFE = 100.0;
    float life = MAX_LIFE; // C++11
} Player;
...

Player.life =  1000.0; // All good?
Player.life = -1000.0; // All good!?
Player.life =  100;    // Oh, yeah, that...
```

Ada

```
type Life_T is digits 6 range 0.0 .. 100.0;
type Player_T is record
   ...
   Life : Life_T := Life_T'Last;
end record;  -- Ada 83

Player : Player_T;
Player.Life :=  1000.0; -- Runtime exception
                        -- Comp-time warning
                        -- SPARK: error
Player.Life := -1000.0; -- Same as above
--  Note: runtime checks can be turned off,
--  but it is not recommended!
```

Implicit conversions considered harmful - Jason Turner - NDC TechTown 2025

The type atlas: the very basics I

Ada

```
-- Ada 83
type My_Enum  is (On, Off, Unknown);          -- Enum, it is not numeric!!!
```

The type atlas: the very basics I

Ada

```ada
-- Ada 83
type My_Enum  is (On, Off, Unknown);          -- Enum, it is not numeric!!!

type My_Int   is range -10 .. 2**8;           -- Integer, range must be increasing
type My_Float is digits 10;                   -- 100.00003, "digits" is the precission
type My_Fl2   is digits 10 range  0.0 .. 1.0; -- 0.3634
```

The type atlas: the very basics I

Ada

```
-- Ada 83
type My_Enum  is (On, Off, Unknown);          -- Enum, it is not numeric!!!

type My_Int   is range -10 .. 2**8;           -- Integer, range must be increasing
type My_Float is digits 10;                   -- 100.00003, "digits" is the precission
type My_Fl2   is digits 10 range  0.0 .. 1.0; -- 0.3634

type My_Fix   is delta 0.5 range -1.5 .. 2.0; -- Fixed point numbers!!
```

The type atlas: the very basics I

Ada

```ada
-- Ada 83
type My_Enum  is (On, Off, Unknown);          -- Enum, it is not numeric!!!

type My_Int   is range -10 .. 2**8;           -- Integer, range must be increasing
type My_Float is digits 10;                   -- 100.00003, "digits" is the precission
type My_Fl2   is digits 10 range  0.0 .. 1.0; -- 0.3634

type My_Fix   is delta 0.5 range -1.5 .. 2.0; -- Fixed point numbers!!

-- Ada 95
type My_Mod     is mod 3;                         -- 0, 1, 2, 0, 1, 2...
type My_Decimal is delta 10.0 ** (-4) digits 20; -- Decimal types!
```

The type atlas: the very basics I

Ada

```
-- Ada 83
type My_Enum  is (On, Off, Unknown);          -- Enum, it is not numeric!!!

type My_Int   is range -10 .. 2**8;           -- Integer, range must be increasing
type My_Float is digits 10;                   -- 100.00003, "digits" is the precission
type My_Fl2   is digits 10 range  0.0 .. 1.0; -- 0.3634

type My_Fix   is delta 0.5 range -1.5 .. 2.0; -- Fixed point numbers!!

-- Ada 95
type My_Mod     is mod 3;                         -- 0, 1, 2, 0, 1, 2...
type My_Decimal is delta 10.0 ** (-4) digits 20; -- Decimal types!
```

Notice that all type declarations are different!
This will be important when we get to generics

The type atlas: the very basics II

Some built in types

Ada

```
-- Ada does some basic type inference

Some_Big_Int : constant := 344_333_322_444; -- Notice the underscores!
Some_Float   : constant := 1.3566;

-- We can declare "anonymous" ranges too!
Some_Range   : Integer range 1 .. 10 := 4;

-- We can have multiple declarations too
F_1, F_2, F_3 : Float := 0.0;
```

The type atlas: the very basics II

Some built in types

Ada

```
type Boolean   is (True, False);            -- Enum

type Integer   is -- Implementation-defined, at least -2**15 + 1 .. 2**15 -1
type Natural   is range 0 .. Integer'Last;  -- Non-negative numbers
type Positive  is range 1 .. Integer'Last;  -- Default range for arrays

type Float     is -- Implementation-defined

type Character is (..., 'A', 'B', etc.);    -- ASCII
type Wide_Character is (...);
type Wide_Wide_Character is (...);
```

The type atlas: the very basics II

Some built in types

Ada

```
type Boolean   is (True, False);            -- Enum

type Integer   is -- Implementation-defined, at least -2**15 + 1 .. 2**15 -1
type Natural   is range 0 .. Integer'Last;  -- Non-negative numbers
type Positive  is range 1 .. Integer'Last;  -- Default range for arrays

type Float     is -- Implementation-defined

type Character is (..., 'A', 'B', etc.);    -- ASCII
type Wide_Character is (...);
type Wide_Wide_Character is (...);
```

For more info, see Ada Reference Manual (ARM) Appendix A.1.

Also, there are `Long_` and `Long_Long_` variants for `Integer` and `Float`.

The type atlas: the very basics III

Leveraging type relationships

Ada

```ada
-- Ada 83
type    Grade   is range 0 .. 10;
subtype Failure is Grade range 0 .. 4; -- Compatible with Grade, but restricted range
```

The type atlas: the very basics III

Leveraging type relationships

Ada

```ada
-- Ada 83
type    Grade   is range 0 .. 10;
subtype Failure is Grade range 0 .. 4; -- Compatible with Grade, but restricted range

type Math_Grade is new Grade;          -- Binary ideantical to Grade, incompatible
type Econ_Grade is new Grade;          -- Binary ideantical to Grade, incompatible
```

The type atlas: the very basics III

Leveraging type relationships

Ada

```ada
-- Ada 83
type    Grade   is range 0 .. 10;
subtype Failure is Grade range 0 .. 4; -- Compatible with Grade, but restricted range

type Math_Grade is new Grade;          -- Binary ideantical to Grade, incompatible
type Econ_Grade is new Grade;          -- Binary ideantical to Grade, incompatible

type Sports_Grade is new Grade range 1 .. 6; -- Binary identical to Grade,
                                             -- incompatible with all others
                                             -- restricted rage
```

The type atlas: the very basics III

Leveraging type relationships

Ada

```ada
-- Ada 83
type    Grade   is range 0 .. 10;
subtype Failure is Grade range 0 .. 4; -- Compatible with Grade, but restricted range

type Math_Grade is new Grade;          -- Binary ideantical to Grade, incompatible
type Econ_Grade is new Grade;          -- Binary ideantical to Grade, incompatible

type Sports_Grade is new Grade range 1 .. 6; -- Binary identical to Grade,
                                             -- incompatible with all others
                                             -- restricted rage


type    Week      is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Work_Days is Days range Mon .. Fri;  -- Yes, enums have ranges too!
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Grade : Grade   := 7;
Some_Fail  : Failure := 4; -- Reminder: a subtype

Some_Grade := Some_Fail;   -- Ok! It is always okay!
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Grade : Grade   := 7;
Some_Fail  : Failure := 4; -- Reminder: a subtype

Some_Grade := Some_Fail;   -- Ok! It is always okay!

Some_Grade := 7;
Some_Fail  := Some_Grade:  -- Allowed by compiler as the types are compatible
                           -- Can cause runtime exception!!
                           -- In this case, it will!
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Grade : Grade   := 7;
Some_Fail  : Failure := 4; -- Reminder: a subtype

Some_Grade := Some_Fail;   -- Ok! It is always okay!

Some_Grade := 7;
Some_Fail  := Some_Grade:  -- Allowed by compiler as the types are compatible
                           -- Can cause runtime exception!!
                           -- In this case, it will!

-- Ada 2012
Some_Fail := (if Some_Grade in Failure then Some_Grade else ...);
                            -- "in" is the most powerful keyword IMHO
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Math_Grade : Math_Grade := 7;  -- Reminder: "new" type of Grade
Some_Econ_Grade : Econ_Grade := 7;  -- Reminder: "new" type of Grade
Some_Grade      : Grade      := 7;
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Math_Grade : Math_Grade := 7;  -- Reminder: "new" type of Grade
Some_Econ_Grade : Econ_Grade := 7;  -- Reminder: "new" type of Grade
Some_Grade      : Grade      := 7;

Some_Econ_Grade := Some_Grade;      -- Compiler error!
                                    -- Incompatible types
Some_Math_Grade := Some_Econ_Grade; -- Compiler error!
                                    -- Incompatible types
```

The type atlas: the very basics IIII

Working with type relationships

Ada

```
-- Ada 83
Some_Math_Grade : Math_Grade := 7;  -- Reminder: "new" type of Grade
Some_Econ_Grade : Econ_Grade := 7;  -- Reminder: "new" type of Grade
Some_Grade      : Grade      := 7;

Some_Econ_Grade := Some_Grade;      -- Compiler error!
                                    -- Incompatible types
Some_Math_Grade := Some_Econ_Grade; -- Compiler error!
                                    -- Incompatible types

Some_Math_Grade := Math_Grade(Some_Grade); -- Explicit casting needed
                                          -- No issues if range was not restricted
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
// C (old C++)
int My_Integer = -100;
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
// C (old C++)
const int MIN_VAL = -100;
int My_Integer    = MIN_VAL;
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
// C (old C++)
const int MIN_VAL = -100;
const int MAX_VAL =  100;
int My_Integer    =  MIN_VAL;
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
// C++11
struct My_Int_T {
    const int MIN_VAL = -100;
    const int MAX_VAL =  100;
          int value   =  MIN_VAL;
} My_Integer; // Still no checks...
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
Okay, now... for real, I promise...
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
// C++ 20!!
template<int min, int max>
struct test
{
    consteval test(int v): value(v) {
        if(v < min || v > max) throw v;
    }
    int value, min, max;
};
int main()
{
    test<-100, 100> t1(2);
    test<-100, 100> t2(120); // Comp-time error!
} // Still no runtime checks
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
///// Runtime checks??
// Old C++
#include<limits>
std::numeric_limits<int>::min();
// Generic & machine related
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
///// Runtime checks??
// Old C++
#include<limits>
std::numeric_limits<int>::min();
// Generic & machine related

// C++20
#include<utility>
std::in_range<some_type>(some_value);
// Generic types
#include<ranges>
std::range::range;
// Ranged type, but not a "simple" type
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
///// Runtime checks??
// Old C++
#include<limits>
std::numeric_limits<int>::min();
// Generic & machine related

// C++20
#include<utility>
std::in_range<some_type>(some_value);
// Generic types
#include<ranges>
std::range::range;
// Ranged type, but not a "simple" type
// C++26 contracts are not for types!!
```

A first stop: comparison with C++

Ada

```ada
-- Ada 83
type My_Int_T is range -100 .. 100;
My_Integer : My_Int_T := My_Int_T'First;
```

cpp

```cpp
///// Runtime checks??
// Old C++
#include<limits>
std::numeric_limits<int>::min();
// Generic & machine related

// C++20
#include<utility>
std::in_range<some_type>(some_value);
// Generic types
#include<ranges>
std::range::range;
// Ranged type, but not a "simple" type
// C++26 contracts are not for types!!
```

A first (v2) stop: comparison with Rust

Rust ranged integers

- Available in nightly with the Ranged_Integers crate
  - Has been in development for years
  - There are still discussions going on
  - Will come to stable but... *WHEN?*

A first (v2) stop: comparison with Rust

Rust ranged integers

- Available in nightly with the Ranged_Integers crate
  - Has been in development for years
  - There are still discussions going on
  - Will come to stable but… *WHEN?*

Rust's other types?

- Nothing still for `floats` (`f16`, `f32`…)
- Discussions happened for fixed points, nothing official yet (though there are creates for them)
  - Obviously, no decimal
- `enum` in Rust is cool :)

Strongly typed (expect aliases 💪), and casting can be unexpected… See `1000 as u8`

See The state of Rust trying to catch up with Ada (FOSDEM, 2025)

The type atlas: the very basics V

Type/Variable/etc attributes

Ada

```ada
-- Ada 83
Integer'First;        -- Smallest number
Integer'Last;         -- Largest  number
Integer'Succ(3);      -- Returns 4
Integer'Pred(3);      -- Returns 2
Integer'Value("-10"); -- Returns -10
Boolean'Pos(False);   -- Normally 0, but not mandatory (specially in hardened systems)
Boolean'Val(0);       -- Would normally return False
My_Type'Size;         -- Size in _bits_ of the type!

My_Var'Image;         -- Returns the string representation of the type
My_Var'Access;        -- Returns an access (pointer) to the variable
```

Ada 2022 RM K.2 Language-Defined Attributes

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Ada 2012 (imho with better syntax, but they existed since 95)
type My_Atomic_Num is digits 15 with Atomic, Default_Value => 0.0;
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Ada 2012 (imho with better syntax, but they existed since 95)
type My_Atomic_Num is digits 15 with Atomic, Default_Value => 0.0;

type My_Smol_Enum is (Off, On) with Size => 1;  -- Of course we care about size
I2C1 : My_Smol_Enum with Address => 16#01000#;  -- Amazing for embedded
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Ada 2012 (imho with better syntax, but they existed since 95)
type My_Atomic_Num is digits 15 with Atomic, Default_Value => 0.0;

type My_Smol_Enum is (Off, On) with Size => 1;  -- Of course we care about size
I2C1 : My_Smol_Enum with Address => 16#01000#;  -- Amazing for embedded

Smol_Var : Bit_1 with Address => I2C1'Address;  -- Memory overlays for the win!
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Ada 2012 (imho with better syntax, but they existed since 95)
type My_Atomic_Num is digits 15 with Atomic, Default_Value => 0.0;

type My_Smol_Enum is (Off, On) with Size => 1;  -- Of course we care about size
I2C1 : My_Smol_Enum with Address => 16#01000#;  -- Amazing for embedded

Smol_Var : Bit_1 with Address => I2C1'Address;  -- Memory overlays for the win!

type My_Network  is xxxxx with Bit_Oder  => High_Order_First, -- Big Endian type!!
                               Alignment => 4;
                        -- for composite types, Scalar_Order may also be needed!!!
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```
-- Ada 2012 (imho with better syntax, but they existed since 95)
type My_Atomic_Num is digits 15 with Atomic, Default_Value => 0.0;

type My_Smol_Enum is (Off, On) with Size => 1;  -- Of course we care about size
I2C1 : My_Smol_Enum with Address => 16#01000#;  -- Amazing for embedded

Smol_Var : Bit_1 with Address => I2C1'Address;  -- Memory overlays for the win!

type My_Network  is xxxxx with Bit_Oder  => High_Order_First, -- Big Endian type!!
                               Alignment => 4;
                        -- for composite types, Scalar_Order may also be needed!!!

type API1_Access is access API1_Type with Storage_Pool => API1_Storage_Pool;
                                       -- Yup, we also have pools/arenas for "pointers"
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```
-- And there is so, so, so much more...
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```
function MemCopy
  (Destination : System.Address;
   Source      : System.Address;
   Length      : Natural)
return Address
with
  Import,                                        -- Calling from other binaries
  Convention => C,                               -- It is C code convention calls
  Link_Name => "memcpy",                         -- Symbol name
  Pre  => Source /= Null_Address      and then -- Contract based programming!
          Destination /= Null_Address and then -- Including when calling to
          not Overlapping (Destination, Source, Length),
  Post => MemCopy'Result = Destination;          -- external functions!
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!

type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!

type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);

-- Prime as a function
function Is_Prime (N : Positive) return Boolean is
  (for all J in Positive range 2 .. N - 1 => N mod J /= 0);
type Prime_2 is new Positive with Dynamic_Predicate => (Is_Prime(Prime_2));
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!

type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);

type Increasing_Array is array (Index) of Some_Number
  with Dynamic_Predicate => (for all I in Index =>
    (if I < Index'Last then Increasing_Array(I) < Increasing_Array(I+1))); -- Nice!
```

The type atlas: the very basics VI

Type/Variable/etc aspects

Ada

```ada
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!

type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);

type Increasing_Array is array (Index) of Some_Number
  with Dynamic_Predicate => (for all I in Index =>
    (if I < Index'Last then Increasing_Array(I) < Increasing_Array(I+1))); -- Nice!
```

Probably one of the coolest features of the language

The type atlas: the very basics VI

Type/Variable/etc aspects

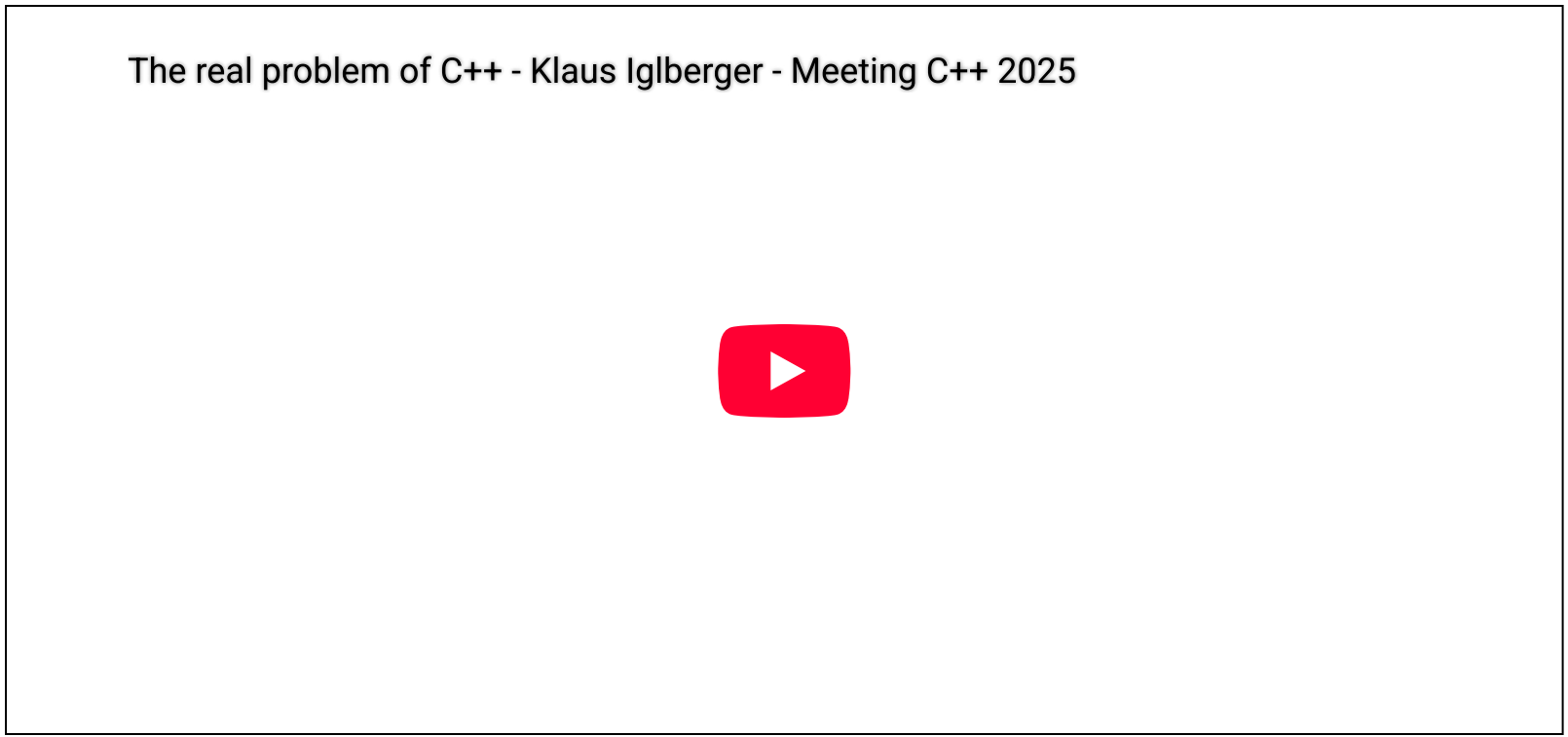
Ada

```ada
-- Formal specification/verification in Ada! Liquid types and contracts
-- ALL OF THIS IS ADA 2012!!!
type Even is new Natural with
  Dynamic_Predicate => Even mod 2 = 0; -- we write the actual properties of the data
type Odd  is new Natural with
  Dynamic_Predicate => not in Even; -- again, the "in" keyword is amazing!

type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);

type Increasing_Array is array (Index) of Some_Number
  with Dynamic_Predicate => (for all I in Index =>
    (if I < Index'Last then Increasing_Array(I) < Increasing_Array(I+1))); -- Nice!
```

Aspects decouple implementation details from the problem

Again, compare Ada to C++/Rust

Again, compare Ada to C++/Rust

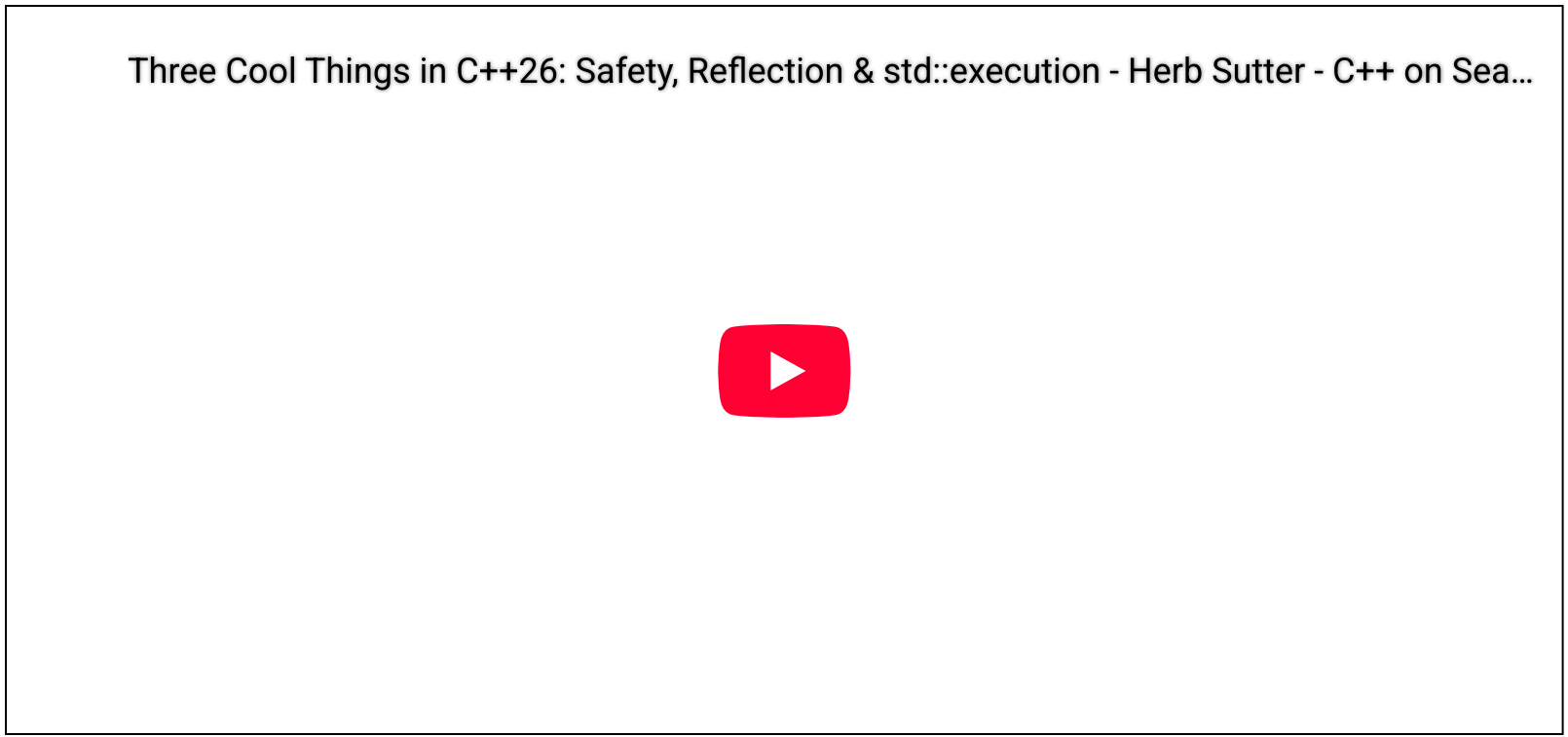


Bound safety? Type Safety? Initialization Safety? Lifetime safety? /—/ Now you care?

Again, compare Ada to C++/Rust

Catching bugs early (C++26) you say?? /—/ NOW you care? Do bugs cost money NOW???

Again, compare Ada to C++/Rust



Safety & UB? Reflection (much more advance than Ada's)? Concurrency and Parallelism???

You think that was a lot about types???

Composite types

The type atlas: the composite basics I

Arrays I

Ada

```ada
-- Ada 83
type My_Index is range -10 .. 10;
type Int_Arr  is array (My_Index) of Integer;
-- The Index can be any discrete type (Ints, Enums, Mods) and have any range
```

The type atlas: the composite basics I

Arrays I

Ada

```
-- Ada 83
type My_Index is range -10 .. 10;
type Int_Arr  is array (My_Index) of Integer;
-- The Index can be any discrete type (Ints, Enums, Mods) and have any range

My_Array1 : Int_Arr := []; -- Uninitialised. Braces are from Ada 2022
My_Array2 : Int_Arr := [1, 2, 1, 2...]; -- Sequential initialization
My_Array3 : Int_Arr := [-10 | -8 | -6 ... => 1, -9 | -7 ... => 2]; -- Positional init
```

The type atlas: the composite basics I

Arrays I

Ada

```
-- Ada 83
type My_Index is range -10 .. 10;
type Int_Arr  is array (My_Index) of Integer;
-- The Index can be any discrete type (Ints, Enums, Mods) and have any range

My_Array1 : Int_Arr := []; -- Uninitialised. Braces are from Ada 2022
My_Array2 : Int_Arr := [1, 2, 1, 2...]; -- Sequential initialization
My_Array3 : Int_Arr := [-10 | -8 | -6 ... => 1, -9 | -7 ... => 2]; -- Positional init
My_Array4 : Int_Arr := [others => 0]; -- Initialise all to 0
My_Array5 : Int_Arr := [-10 | -8 | -6 ... => 1, others => 2]; -- Mixed case
```

The type atlas: the composite basics I

Arrays I

Ada

```ada
-- Ada 83
type My_Index is range -10 .. 10;
type Int_Arr  is array (My_Index) of Integer;
-- The Index can be any discrete type (Ints, Enums, Mods) and have any range

My_Array1 : Int_Arr := []; -- Uninitialised. Braces are from Ada 2022
My_Array2 : Int_Arr := [1, 2, 1, 2...]; -- Sequential initialization
My_Array3 : Int_Arr := [-10 | -8 | -6 ... => 1, -9 | -7 ... => 2]; -- Positional init
My_Array4 : Int_Arr := [others => 0]; -- Initialise all to 0
My_Array5 : Int_Arr := [-10 | -8 | -6 ... => 1, others => 2]; -- Mixed case

-- Ada 2022
My_Array6 : Int_Arr := [for I in My_Index'Range => Integer(My_Index)]; -- -10, -9...
My_Array7 : Int_Arr := [for I in 1 .. 3 => Integer(I), 5 .. 10 => 10, others => 0];
```

The type atlas: the composite basics I

Arrays II: unconstrained arrays

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer; -- "<>" means ellipsis
```

The type atlas: the composite basics I

Arrays II: unconstrained arrays

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer; -- "<>" means ellipsis

My_Int_Arr_Var  : Int_Arr(1 .. 10) := [others => 0];  -- bounds given explicitly
My_Int_Arr_Var2 : Int_Arr          := [1, 2, 3, 4];   -- bounds given by data
My_Int_Arr_Var3 : Int_Arr(1 ..  A) := [others => 0];  -- bounds given at runtime!
```

The type atlas: the composite basics I

Arrays II: unconstrained arrays

Ada

```
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer; -- "<>" means ellipsis

My_Int_Arr_Var  : Int_Arr(1 .. 10) := [others => 0];  -- bounds given explicitly
My_Int_Arr_Var2 : Int_Arr          := [1, 2, 3, 4];   -- bounds given by data
My_Int_Arr_Var3 : Int_Arr(1 ..  A) := [others => 0];  -- bounds given at runtime!

type Arr_Int_Arr is array (Positive range <>) of Int_Arr (1 .. 20);
                                                    -- ^bounds need to be given!!
-- VERY IMPORTANT!! The bounds/range of arrays are an INTRINSIC PART of the type!!!
```

The type atlas: the composite basics I

Arrays II: unconstrained arrays

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer; -- "<>" means ellipsis

My_Int_Arr_Var  : Int_Arr(1 .. 10) := [others => 0];  -- bounds given explicitly
My_Int_Arr_Var2 : Int_Arr          := [1, 2, 3, 4];   -- bounds given by data
My_Int_Arr_Var3 : Int_Arr(1 ..  A) := [others => 0];  -- bounds given at runtime!

type Arr_Int_Arr is array (Positive range <>) of Int_Arr (1 .. 20);
                                                  -- ^bounds need to be given!!
-- VERY IMPORTANT!! The bounds/range of arrays are an INTRINSIC PART of the type!!!

type Bit_Matrix is array (Integer range <>, Integer range <>) of Boolean;
```

The type atlas: the composite basics I

Arrays II: unconstrained arrays

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer; -- "<>" means ellipsis

My_Int_Arr_Var  : Int_Arr(1 .. 10) := [others => 0];  -- bounds given explicitly
My_Int_Arr_Var2 : Int_Arr          := [1, 2, 3, 4];   -- bounds given by data
My_Int_Arr_Var3 : Int_Arr(1 ..  A) := [others => 0];  -- bounds given at runtime!

type Arr_Int_Arr is array (Positive range <>) of Int_Arr (1 .. 20);
                                                  -- ^bounds need to be given!!
-- VERY IMPORTANT!! The bounds/range of arrays are an INTRINSIC PART of the type!!!

type Bit_Matrix is array (Integer range <>, Integer range <>) of Boolean;
My_Mat'Range; My_Mat'Range(2); My_Arr'Length; My_Arr'First; My_Arr'Last;
-- Guess what all these attributes do?
```

The type atlas: the composite basics I

Arrays III: using `mod`s as index

Ada

```ada
-- Ada 95
type My_Mod   is mod 7; -- 0, 1, 2, 3, 4, 5, 6
type Int_Ring is array (My_Mod) of Integer;

Int_Ring_Var : Int_Ring := [for I in My_Mod => Integer(I)];
Index        : My_Mod   := 6; -- Int_Ring_Var(Index) = 6;

Index := Index + 1;
Int_Ring_Var(Index) = 0; -- True due to modular type arithmetic!!
```

The type atlas: the composite basics I

Arrays III: using `mod`s as index

Ada

```ada
-- Ada 95
type My_Mod   is mod 7; -- 0, 1, 2, 3, 4, 5, 6
type Int_Ring is array (My_Mod) of Integer;

Int_Ring_Var : Int_Ring := [for I in My_Mod => Integer(I)];
Index        : My_Mod   := 6; -- Int_Ring_Var(Index) = 6;

Index := Index + 1;
Int_Ring_Var(Index) = 0; -- True due to modular type arithmetic!!
```

With `mod` types, it is mathematicaly imposible to cause an out of bounds error!!!

There are no issues with ranges and we don't have to use clases/methods!

The type atlas: the composite basics I

Arrays IIII: using enums as index

Ada

```ada
type RGBA     is (Red, Green, Blue, Alpha);
type Pixel_T  is array (RGBA) of Mod_256;

Pixel : Pixel_T := [Alpha => Mod_256'Last; others => 0];

for C in RGBA range Red .. Blue loop
    Pixel(C) := 100;
end loop;
```

Enums allow for extremely easy state machines and precise iterations!

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
My_Int_Arr_Var2 : Int_Arr (1 .. My_Int_Arr_Var'Last / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
My_Int_Arr_Var2 : Int_Arr (1 .. My_Int_Arr_Var'Last / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. My_Int_Arr_Var'Last - 1) := [others => 2];
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
My_Int_Arr_Var2 : Int_Arr (1 .. My_Int_Arr_Var'Last / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. My_Int_Arr_Var'Last - 1) := [others => 2];

-- Copies
My_Ann_Arr_Var2 := My_Int_Arr_Var (1 .. My_Int_Arr_Var'Last / 2);
My_Int_Arr_Var3 := My_Int_Arr_Var (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);
-- Direct copies are allowed!
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```
-- This code is actually quite bad...
-- Lets refine it bit by bit and make it more idomatic
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
My_Int_Arr_Var2 : Int_Arr (1 .. My_Int_Arr_Var'Last / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. My_Int_Arr_Var'Last - 1) := [others => 2];

-- Copies
My_Ann_Arr_Var2 := My_Int_Arr_Var (1 .. My_Int_Arr_Var'Last / 2);
My_Int_Arr_Var3 := My_Int_Arr_Var (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
My_Int_Arr_Var2 : Int_Arr (1 .. My_Int_Arr_Var'Last / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (My_Int_Arr_Var'Last / 2 + 1 .. My_Int_Arr_Var'Last);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. My_Int_Arr_Var'Last - 1) := [others => 2];

-- Copies
My_Ann_Arr_Var2 := My_Int_Arr_Var (My_Int_Arr_Var2'Range);
My_Int_Arr_Var3 := My_Int_Arr_Var (My_Int_Arr_Var3'Range);
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
Last_Index renames My_Int_Arr_Var'Last;

My_Int_Arr_Var2 : Int_Arr (1 .. Last_Index / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (Last_Index / 2 + 1 .. Last_Index);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. Last_Index - 1) := [others => 2];

-- Copies
My_Ann_Arr_Var2 := My_Int_Arr_Var (My_Int_Arr_Var2'Range);
My_Int_Arr_Var3 := My_Int_Arr_Var (My_Int_Arr_Var3'Range);
```

The type atlas: the composite basics I

Arrays V: array slicing

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;
My_Int_Arr_Var  : Int_Arr (1 .. 50) := [others => 1];
Last_Index renames My_Int_Arr_Var'Last;

My_Int_Arr_Var2 : Int_Arr (1 .. Last_Index / 2); -- Integer division!
My_Int_Arr_Var3 : Int_Arr (Positive'Succ(Last_Index / 2) .. Last_Index);

-- Array slicing
My_Int_Arr_Var (My_Int_Arr_Var'First + 1 .. Last_Index - 1) := [others => 2];

-- Copies
My_Ann_Arr_Var2 := My_Int_Arr_Var (My_Int_Arr_Var2'Range);
My_Int_Arr_Var3 := My_Int_Arr_Var (My_Int_Arr_Var3'Range);
```

The type atlas: the composite basics I

Arrays VI: Strings

Ada

```ada
-- Ada 83
Text  : String; -- type String is (Positive range <>) of Character;

Text1 : String := "one";   -- Strings are not null terminated! C_Str is though
Text2 : String := "two";
Text3 : String := "three";
```

The type atlas: the composite basics I

Arrays VI: Strings

Ada

```ada
-- Ada 83
Text  : String; -- type String is (Positive range <>) of Character;

Text1 : String := "one";   -- Strings are not null terminated! C_Str is though
Text2 : String := "two";
Text3 : String := "three";

Text2 := Text1; -- All good without any special functions!!
Text2 := Text3; -- Compile time error!!!
-- ^Length=3, ^Length=5, incompatible types!!
```

The type atlas: the composite basics I

Arrays VI: Strings

Ada

```
-- Ada 83
Text  : String; -- type String is (Positive range <>) of Character;

Text1 : String := "one";   -- Strings are not null terminated! C_Str is though
Text2 : String := "two";
Text3 : String := "three";

Text2 := Text1; -- All good without any special functions!!
Text2 := Text3; -- Compile time error!!!
-- ^Length=3, ^Length=5, incompatible types!!

TextU : String := "こんにちは"; -- UTF-8 supported, but without special handling
                              -- TextU'Length = 15!!
```

The type atlas: the composite basics I

Arrays VII: Ada and the stack, part one

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;

My_Int_Arr_Var  : Int_Arr(1 .. B) := [others => 0]; -- B here is a VERY LARGE value
```

The type atlas: the composite basics I

Arrays VII: Ada and the stack, part one

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;

My_Int_Arr_Var  : Int_Arr(1 .. B) := [others => 0]; -- B here is a VERY LARGE value
-- raised STORAGE_ERROR : stack overflow or erroneous memory access
```

The type atlas: the composite basics I

Arrays VII: Ada and the stack, part one

Ada

```ada
-- Ada 83
type Int_Arr is array (Positive range <>) of Integer;

My_Int_Arr_Var  : Int_Arr(1 .. B) := [others => 0]; -- B here is a VERY LARGE value
-- raised STORAGE_ERROR : stack overflow or erroneous memory access
```

Ada (ab)uses the stack like crazy

It is great for performance, automatic memory management and ease of use but... It does have its limitations (see `Vector` for a solution).

Read GNAT's Stack Related Facilities for debugging, metrics and stack size configuration

The type atlas: the composite basics I

Arrays VIII: empty what?

Ada

```ada
-- Ada 83
type Some_Range is range 1 .. 0;  -- Ranges must be increasing!!
                                  -- There is no error nor warning by the compiler???
```

The type atlas: the composite basics I

Arrays VIII: empty what?

Ada

```ada
-- Ada 83
type Some_Range is range 1 .. 0;  -- Ranges must be increasing!!
                                  -- There is no error nor warning by the compiler???
                                  -- The empty set ∅
```

The type atlas: the composite basics I

Arrays VIII: empty what?

Ada

```ada
-- Ada 83
type Some_Range is range 1 .. 0;  -- Ranges must be increasing!!
                                  -- There is no error nor warning by the compiler???
                                  -- The empty set ∅
type Null_Arr is array (Some_Range) of Integer; -- Allowed?! Even at runtime??

-- An empty array cannot hold data. They tend to happen at runtime.
-- Operations on them (generally) have no effect
```

The type atlas: the composite basics I

Arrays VIII: empty what?

Ada

```ada
-- Ada 83
type Some_Range is range 1 .. 0;  -- Ranges must be increasing!!
                                  -- There is no error nor warning by the compiler???
                                  -- The empty set ∅
type Null_Arr is array (Some_Range) of Integer; -- Allowed?! Even at runtime??

-- An empty array cannot hold data. They tend to happen at runtime.
-- Operations on them (generally) have no effect

for I in Null_Arr'Range loop      -- Null loop, nothing happens!
...
end loop;
```

The type atlas: the composite basics II

Records I: aka, structs

Ada

```ada
-- Ada 83
type Data is record
   Day   : Integer range 1 .. 31;
   Month : Months;
   Year  : Integer range 1 .. 3000 := 2026; -- Default values allowed
end record; -- Aka, your typical struct in other languages
```

The type atlas: the composite basics II

Records I: aka, structs

Ada

```
-- Ada 83
type Data is record
   Day   : Integer range 1 .. 31;
   Month : Months;
   Year  : Integer range 1 .. 3000 := 2026; -- Default values allowed
end record; -- Aka, your typical struct in other languages

--  Positional components
Ada_Birthday    : Date := (10, December, 1815);
--  Named components
Leap_Day_2020   : Date := (Day   => 29,
                           Month => February,
                           Year  => 2020); -- Init by name
```

The type atlas: the composite basics II

Records I: aka, structs

Ada

```
-- Ada 83
type Data is record
   Day   : Integer range 1 .. 31;
   Month : Months;
   Year  : Integer range 1 .. 3000 := 2026; -- Default values allowed
end record; -- Aka, your typical struct in other languages

--  Positional components
Ada_Birthday    : Date := (10, December, 1815);
--  Named components
Leap_Day_2020   : Date := (Day   => 29,
                           Month => February,
                           Year  => 2020); -- Init by name
type Empty is null record; -- Empty "container", it is quite useful!
```

The type atlas: the composite basics II

Records II: dynamic record sizes

Ada

```ada
-- Just lik arrays, record sizes can also be dynamic
Max_Len : constant Natural := Compute_Max_Len;
                           -- ^ Not known at compile time

type Growable_Stack is record
   Items : Items_Array (1 .. Max_Len); -- Determined at runtime
   Len   : Natural;
end record;
-- Once Max_Len is known, all record sizes will stay the same, just like arrays
```

The type atlas: the composite basics II

Records III: record discriminants

Ada

```ada
-- Discriminated records: they are different/discriminated by their input
type Square_Mat (Size : Positive) is record       -- Unconstrained discriminant
      Mat : Matrix(1 .. Size, 1 .. Size);
end record;
type Buffer (Size : Buffer_Size := 100) is record -- Default initialized
   Pos   : Buffer_Size := 0;
   Value : String(1 .. Size);
end record;
```

The type atlas: the composite basics II

Records III: record discriminants

Ada

```
-- Discriminated records: they are different/discriminated by their input
type Square_Mat (Size : Positive) is record       -- Unconstrained discriminant
      Mat : Matrix(1 .. Size, 1 .. Size);
end record;
type Buffer (Size : Buffer_Size := 100) is record -- Default initialized
   Pos   : Buffer_Size := 0;
   Value : String(1 .. Size);
end record;

Basis   : Square_Mat(5); -- constrained, always 5 by 5
ILLEGAL : Square_Mat;    -- illegal, a Square_Mat must be constrained
```

The type atlas: the composite basics II

Records III: record discriminants

Ada

```ada
-- Discriminated records: they are different/discriminated by their input
type Square_Mat (Size : Positive) is record       -- Unconstrained discriminant
      Mat : Matrix(1 .. Size, 1 .. Size);
end record;
type Buffer (Size : Buffer_Size := 100) is record -- Default initialized
   Pos   : Buffer_Size := 0;
   Value : String(1 .. Size);
end record;

Basis   : Square_Mat(5); -- constrained, always 5 by 5
ILLEGAL : Square_Mat;    -- illegal, a Square_Mat must be constrained

Large   : Buffer(200);   -- constrained, always 200 characters (explicit discriminant)
Message : Buffer;        -- _unconstrained_, initially 100 characters, but can change
                         -- (default discriminant value)
```

The type atlas: the composite basics II

Records IIII: variant records (similar to Sum types in OCaml...)

Ada

```
type Expr_Kind_Type is (Bin_Op_Plus, Bin_Op_Minus, Num); -- Enum

type Expr (Kind : Expr_Kind_Type) is record
               -- ^ The discriminant is an enumeration value
   Current_Val : Float; -- This member is here allways
   case Kind is
      when Bin_Op_Plus | Bin_Op_Minus =>
         Left, Right : Operator;
      when Num =>
         Val : Integer;
   end case;
   --  Variant part. Only one, at the end of the record definition, but can be nested
end record; -- Very similar to C/C++/Rust union types
```

The type atlas: the composite basics II

Records IIII: variant records (similar to Sum types in OCaml...)

Ada

```ada
type Expr_Kind_Type is (Bin_Op_Plus, Bin_Op_Minus, Num); -- Enum

type Expr (Kind : Expr_Kind_Type) is record
   Current_Val : Float; -- This member is here allways
   case Kind is
      when Bin_Op_Plus | Bin_Op_Minus =>
         Left, Right : Operator;
      when Num =>
         Val : Integer;
   end case;
end record; -- Very similar to C/C++/Rust union types

E : Expr := (Num, 12.0, 133);
E.Left   := Some_Operation; --  Will compile but fail at runtime!
```

The type atlas: the composite basics II

Records IIII: variant records (similar to Sum types in OCaml...)

Ada

```
type Expr (Kind : Expr_Kind_Type) is record
   Current_Val : Float; -- This member is here allways
   case Kind is
      when Bin_Op_Plus | Bin_Op_Minus => Left, Right : Operator;
      when Num => Val : Integer;
   end case;
end record; -- Very similar to C/C++/Rust union types

E : Expr := (Num, 12);

case E.Kind is
   when Bin_Op_Plus  => Eval_Expr (E.Left) + Eval_Expr (E.Right),
   when Bin_Op_Minus => Eval_Expr (E.Left) - Eval_Expr (E.Right),
   when Num          => E.Val;
end case;
```

The type atlas: the composite basics II

Records V: variant records, common examples

Ada

```
-- Ada 83
type Option(Valid : Boolean := False) is record -- Rust: Option<Some|None>
   case Valid is
      when False => null;
      when True  => Value : My_Type;
   end case;
end record;

No_Data_You_Silly : exception;                  -- Rust: Result<V|E>
type Result is record
   My_Resutl : My_Type := raise No_Data_You_Silly; -- Exception thrown!
end record; -- If we don't get the data, we get a spook!
```

The type atlas: the composite basics II

Records VI: record subtyping

Ada

```ada
-- All type relationships seen for simple types also apply to arrays and records!!

type My_Fixed_Buffer is new Buffer(500);

subtype My_Num_Expr  is Expr(Num); -- Fixed variant record subtype

-- Subtyping of records can be useful when doing OOP and dealing with access types
```

The type atlas: the composite basics II.V

Low level aspects I: record and data representation

Ada

```
-- Enumeration numeric values
type UART_Baud is  (b_9600, b_115200, b_921600) with Size => 2;
for  UART_Baud use (b_9600 => 2#00#, b_115200 => 2#01#, b_921600 => 2#10#);
```

The type atlas: the composite basics II.V

Low level aspects I: record and data representation

Ada

```
-- Enumeration numeric values
type UART_Baud is  (b_9600, b_115200, b_921600) with Size => 2;
for  UART_Baud use (b_9600 => 2#00#, b_115200 => 2#01#, b_921600 => 2#10#);

type UART_Speed is record    -- Lets say that our board has 3 UARTs
   UART1 : UART_Baud;
   UART2 : UART_Baud;
   UART3 : UART_Baud;
end record with Bit_Order => Low_Order_First, Size => 16;
```

The type atlas: the composite basics II.V

Low level aspects I: record and data representation

Ada

```
-- Enumeration numeric values
type UART_Baud is  (b_9600, b_115200, b_921600) with Size => 2;
for  UART_Baud use (b_9600 => 2#00#, b_115200 => 2#01#, b_921600 => 2#10#);
type UART_Speed is record    -- Lets say that our board has 3 UARTs
   UART1 : UART_Baud;
   UART2 : UART_Baud;
   UART3 : UART_Baud;
end record with Bit_Order => Low_Order_First, Size => 16;

for UART_Speed use record    -- Bit-level data layout representation!
   UART1 at 0 range 0 .. 1;  -- Bit #2 and 3: reserved!
   UART2 at 0 range 4 .. 6;  -- Bit #7 and 8: reserved!
   UART3 at 1 range 0 .. 1;  -- Second byte, first two bits. Rest is unused
end record;                  -- Can your Rust do this so precisely? Can you C/C++??
```

The type atlas: the composite basics II.V

Low level aspects I: record and data representation

Ada

```
-- Enumeration numeric values
type UART_Baud is  (b_9600, b_115200, b_921600) with Size => 2;
for  UART_Baud use (b_9600 => 2#00#, b_115200 => 2#01#, b_921600 => 2#10#);
type UART_Speed is record    -- Lets say that our board has 3 UARTs
   UART1 : UART_Baud; UART2 : UART_Baud; UART3 : UART_Baud;
end record with Bit_Order => Low_Order_First, Size => 16;
for UART_Speed use record    -- Bit-level data layout representation!
   UART1 at 0 range 0 .. 1;  -- Bit #2 and 3: reserved!
   UART2 at 0 range 4 .. 6;  -- Bit #7 and 8: reserved!
   UART3 at 1 range 0 .. 1;  -- Second byte, first two bits. Rest is unused
end record;                  -- Can your Rust, C/C++ do this so precisely??

UART_Speed_Reg : aliased UART_Speed -- Aliased as it needs to live in memory/not stack
                 with Address => System'To_Address(16#8002001#), Volatile_Full_Access;
```

The type atlas: the composite basics II.V

Low level aspects I: record and data representation

Ada

```
type UART_Speed is record    -- Lets say that our board has 3 UARTs
   UART1 : UART_Baud; UART2 : UART_Baud; UART3 : UART_Baud;
end record with Bit_Order => Low_Order_First, Size => 16;
for UART_Speed use record    -- Bit-level data layout representation!
   UART1 at 0 range 0 .. 1;  -- Bit #2 and 3: reserved!
   UART2 at 0 range 4 .. 6;  -- Bit #7 and 8: reserved!
   UART3 at 1 range 0 .. 1;  -- Second byte, first two bits. Rest is unused
end record;                  -- Can your Rust, C/C++ do this so precisely??

UART_Speed_Reg : aliased UART_Speed -- Aliased as it needs to live in memory/not stack
                 with Address => System'To_Address(16#8002001#), Volatile_Full_Access;

UART_Speed_Reg.UART1 := b_115200; -- Note: no bit shifts, no function call, no objects!
-- No need for documentation, safe to reuse (different design -> error!)
```

The type atlas: the composite basics II.V

Low level aspects II: comparison to C/C++ & Rust

Ada

- Low-level access
- Low-level abstractions
- Perfect mapping: high-level ↔ low-level
- Readable
- Formally verifiable with SPARK

The type atlas: the composite basics II.V

Low level aspects II: comparison to C/C++ & Rust

Ada

- Low-level access
- Low-level abstractions
- Perfect mapping: high-level ↔ low-level
- Readable
- Formally verifiable with SPARK

Check out **SweetAda** for amazing low-level and high-level stuff in a ton of boards and arches (x86, ARM, RISC-V, M86K, SuperH, MicroBlaze, NiosII...)

Example: HiFive rev B (RISC-V) Alternate Low-Frequency Clock (LFALTCLK)

The type atlas: the composite basics II.V

Low level aspects II: comparison to C/C++ & Rust

Ada

- Low-level access
- Low-level abstractions
- Perfect mapping: high-level ↔ low-level
- Readable
- Formally verifiable with SPARK

C/C++

- Low-level access (bit fields, bit operations)
- Poor low-level abstractions (get/setters, OOP)
- High-level to low-level mapping is manual
- Not readable
- Bad typing checks

Check out **SweetAda** for amazing low-level and high-level stuff in a ton of boards and arches (x86, ARM, RISC-V, M86K, SuperH, MicroBlaze, NiosII...)

Example: HiFive rev B (RISC-V) Alternate Low-Frequency Clock (LFALTCLK)

The type atlas: the composite basics II.V

Low level aspects II: comparison to C/C++ & Rust

Ada

- Low-level access
- Low-level abstractions
- Perfect mapping: high-level ↔ low-level
- Readable
- Formally verifiable with SPARK

Check out **SweetAda** for amazing low-level and high-level stuff in a ton of boards and arches (x86, ARM, RISC-V, M86K, SuperH, MicroBlaze, NiosII...)

Example: HiFive rev B (RISC-V) Alternate Low-Frequency Clock (LFALTCLK)

C/C++

- Low-level access (bit fields, bit operations)
- Poor low-level abstractions (get/setters, OOP)
- High-level to low-level mapping is manual
- Not readable
- Bad typing checks

Rust

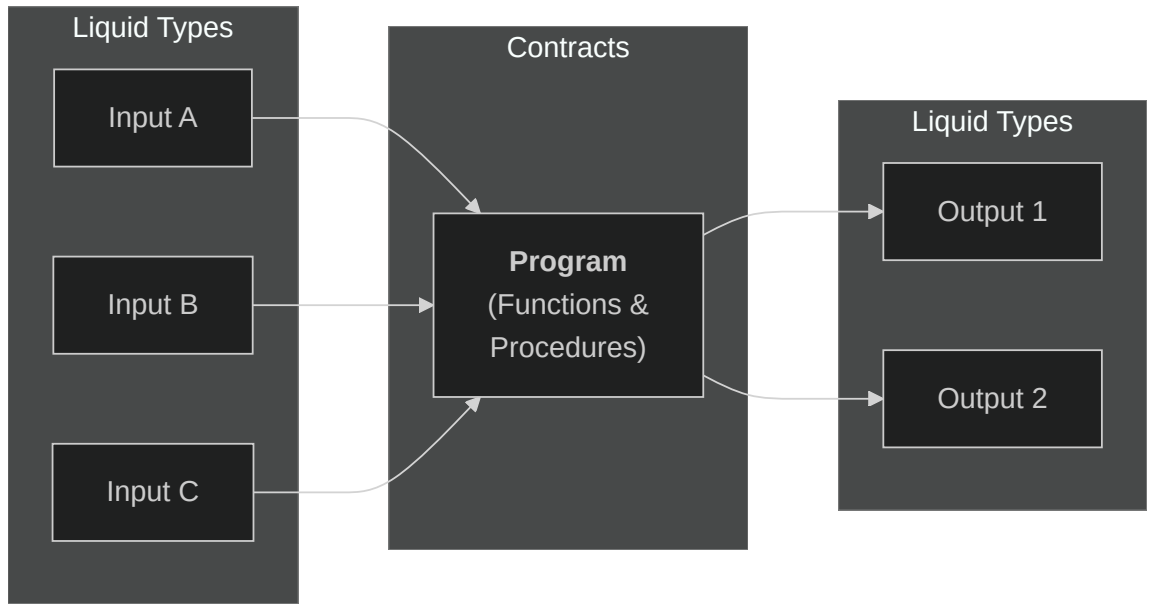
- Low-level access
- Poor low-level abstractions (get/setters, `impl`)
- High-level to low-level mapping is manual
- Not readable
- Gooder typing checks, but `unsafe`, `mut`s...

Before we continue…

Why do we care so much about data/Types?

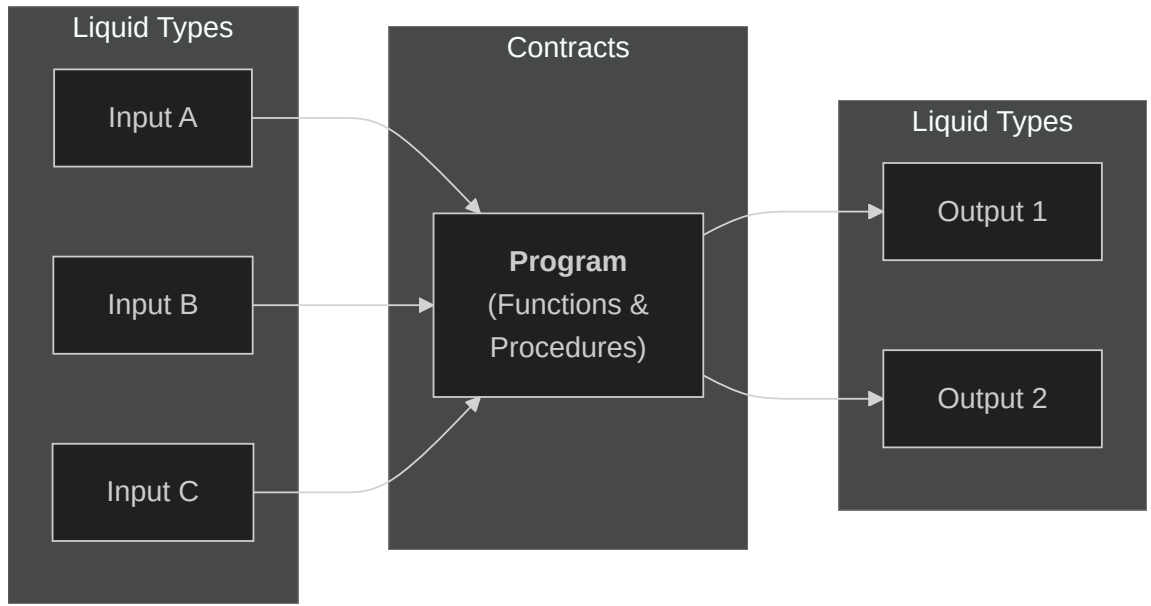
Why do we care so much about data/Types?

Data is "50%" of a program. Types help us improve that aspect!



Why do we care so much about data/Types?

Data is "50%" of a program. Types help us improve that aspect!



A good data foundation is the stone on which to build everything else!

Are we done with types?

No, fuck you, more types!!!

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```ada
-- Ada 83
type Int_Acc is access Integer;              -- Access (pointers)!
                                             -- No arithmetic operations allowed!
```

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```ada
-- Ada 83
type Int_Acc is access Integer;             -- Access (pointers)!
                                            -- No arithmetic operations allowed!
-- Ada 05
type NN_Int_Acc is not null access Integer; -- Must never be null!
type Better_Acc is new not null Int_Acc;    -- Just like with types
```

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```
-- Ada 83
type Int_Acc is access Integer;              -- Access (pointers)!
                                             -- No arithmetic operations allowed!
-- Ada 05
type NN_Int_Acc is not null access Integer; -- Must never be null!
type Better_Acc is new not null Int_Acc;    -- Just like with types

My_Var_Acc : not null Int_Acc := S'Access;  -- Yup, that is also possible
```

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```ada
-- Ada 83
type Int_Acc is access Integer;             -- Access (pointers)!
                                            -- No arithmetic operations allowed!
-- Ada 05
type NN_Int_Acc is not null access Integer; -- Must never be null!
type Better_Acc is new not null Int_Acc;    -- Just like with types

My_Var_Acc : not null Int_Acc := S'Access;  -- Yup, that is also possible

type Global_Int_Acc is access all  Integer; -- Can point to any memory!
```

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```ada
-- Ada 83
type Int_Acc is access Integer;             -- Access (pointers)!
                                            -- No arithmetic operations allowed!
-- Ada 05
type NN_Int_Acc is not null access Integer; -- Must never be null!
type Better_Acc is new not null Int_Acc;    -- Just like with types

My_Var_Acc : not null Int_Acc := S'Access;  -- Yup, that is also possible

type Global_Int_Acc is access all  Integer; -- Can point to any memory!
type Nice_Acc   is access constant Integer; -- Read only access! See Rust's & vs &mut
```

The type atlas: the not so very basics I

Access types I: pointers more or less

Ada

```ada
-- Ada 83
type Int_Acc is access Integer;             -- Access (pointers)!
                                            -- No arithmetic operations allowed!
-- Ada 05
type NN_Int_Acc is not null access Integer; -- Must never be null!
type Better_Acc is new not null Int_Acc;    -- Just like with types

My_Var_Acc : not null Int_Acc := S'Access;  -- Yup, that is also possible

type Global_Int_Acc is access all  Integer; -- Can point to any memory!
type Nice_Acc   is access constant Integer; -- Read only access! See Rust's & vs &mut

type Callback_F is access function  (A: Integer) return Natural;
type Callback_P is access procedure (A: in out Integer);
```

The type atlas: the not so very basics I

Access types II: working with `access`

Ada

```ada
-- Allocation of memory via an access
type Int_Acc is access Integer;
Int_1, Int_2, Int_3 : Int_Acc; -- Default value is always null
```

The type atlas: the not so very basics I

Access types II: working with `access`

Ada

```ada
-- Allocation of memory via an access
type Int_Acc is access Integer;
Int_1, Int_2, Int_3 : Int_Acc; -- Default value is always null

Int_1 := new Integer;          -- Allocate a new Int
Int_2 := new Integer'(10);     -- Allocate a new Int and initialize it
```

The type atlas: the not so very basics I

Access types II: working with `access`

Ada

```ada
-- Allocation of memory via an access
type Int_Acc is access Integer;
Int_1, Int_2, Int_3 : Int_Acc; -- Default value is always null

Int_1 := new Integer;          -- Allocate a new Int
Int_2 := new Integer'(10);     -- Allocate a new Int and initialize it

Int_3 := Int_1; Int_1 := null; -- Copy the access of 1 to 3, null 1
Int_3.all := Int_2.all;        -- Copy the Int value of 2 to 3
                               -- null derreference causes an exception!
```

The type atlas: the not so very basics I

Access types II: working with `access`

Ada

```
-- Allocation of memory via an access
type Int_Acc is access Integer;
Int_1, Int_2, Int_3 : Int_Acc; -- Default value is always null

Int_1 := new Integer;          -- Allocate a new Int
Int_2 := new Integer'(10);     -- Allocate a new Int and initialize it

Int_3 := Int_1; Int_1 := null; -- Copy the access of 1 to 3, null 1
Int_3.all := Int_2.all;        -- Copy the Int value of 2 to 3
                               -- null derreference causes an exception!

procedure Free is new Ada.Unchecked_Deallocation -- Unchecked as it may be dangerous
   (Object => Integer, Name => Int_Acc);         -- Typed deallocation, no "void*"!!
Free(Int_3); Free(Int_2); Free(Int_1); -- Deallocation nulls the values and it is safe
```

The type atlas: the not so very basics II

Limited types: skipping bugs and more data modelling!

Ada

```ada
-- Previously
Int_3 := Int_1; Int_1 := null; -- Copy the access of 1 to 3, null 1
Int_2.all := Int_1.all;        -- Would throw an exception!
```

The type atlas: the not so very basics II

Limited types: skipping bugs and more data modelling!

Ada

```
-- Previously
Int_3 := Int_1; Int_1 := null; -- Copy the access of 1 to 3, null 1
Int_2.all := Int_1.all;        -- Would throw an exception!

-- Generally we dont want to allow the copying of pointers/access!!!
-- Better memory management (less bugs), better performance (only one access!)
```

The type atlas: the not so very basics II

Limited types: skipping bugs and more data modelling!

Ada

```
-- Previously
Int_3 := Int_1; Int_1 := null; -- Copy the access of 1 to 3, null 1
Int_2.all := Int_1.all;        -- Would throw an exception!

-- Generally we dont want to allow the copying of pointers/access!!!
-- Better memory management (less bugs), better performance (only one access!)

type Lim_Int_Acc is limited record -- Limits copying and comparison!!!!
   V : Int_Acc;                    -- Limited is only available for record,
end record;                        -- interface and private types (more on that later)
```

The type atlas: the not so very basics II

Limited types: skipping bugs and more data modelling!

Ada

```
-- Generally we dont want to allow the copying of pointers/access!!!
-- Better memory management (less bugs), better performance (only one access!)

type Lim_Int_Acc is limited record -- Limits copying and comparison!!!!
   V : Int_Acc;                    -- Limited is only available for record
end record;                        -- interface and private types (more on that later)

Int_4, Int_5 : Lim_Int_Acc;
Int_4.V := new Integer'(5);

Int_5 := Int_4;       -- Error!! Not allowed by the compiler (custom procedure needed)
if Int_5 = Int_4 then -- Error! Comparison not allowed in limited!!
```

The type atlas: the not so very basics II

Limited types: skipping bugs and more data modelling!

Ada

```ada
-- Generally we dont want to allow the copying of pointers/access!!!
-- Better memory management (less bugs), better performance (only one access!)

type Lim_Int_Acc is limited record -- Limits copying and comparison!!!!
   V : Int_Acc;                    -- Limited is only available for record
end record;                        -- interface and private types (more on that later)

Int_4, Int_5 : Lim_Int_Acc;
Int_4.V := new Integer'(5);

Int_5 := Int_4;       -- Error!! Not allowed by the compiler (custom procedure needed)
if Int_5 = Int_4 then -- Error! Comparison not allowed in limited!!
```

There is so much more... See limited types in Learn.AdaCore

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
-- Ada 95
type My_Class is tagged null record; -- Just like a record with "tagged" keyword
                    -- "null" here just says there is no data associated
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
-- Ada 95
type My_Class is tagged null record; -- Just like a record with "tagged" keyword
                   -- "null" here just says there is no data associated

type Derived  is new My_Class with record
   A : Integer;     -- This derived class has some extra added data
end record;
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
-- Ada 95
type My_Class is tagged null record; -- Just like a record with "tagged" keyword
                   -- "null" here just says there is no data associated

type Derived  is new My_Class with record
   A : Integer;     -- This derived class has some extra added data
end record;

Obj1 : My_Class;
Obj2 : Derived        := (A => 12);
Obj3 : My_Class       := Obj2; -- ERROR, different type!!
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
-- Ada 95
type My_Class is tagged null record; -- Just like a record with "tagged" keyword
                   -- "null" here just says there is no data associated

type Derived  is new My_Class with record
   A : Integer;     -- This derived class has some extra added data
end record;

Obj1 : My_Class;
Obj2 : Derived        := (A => 12);
Obj3 : My_Class       := Obj2; -- ERROR, different type!!
Obj4 : My_Class'Class := Obj2; -- Ok! Any descendant of My_Class is good
Obj5 : My_Class'Class := Obj1;
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
type My_Class is tagged null record; -- Just like a record with "tagged" keyword
type Derived  is new My_Class with record
   A : Integer;     -- This derived class has some extra added data
end record;

Obj1 : My_Class;
Obj2 : Derived        := (A => 12);
Obj4 : My_Class'Class := Obj2; -- Ok! Any descendant of My_Class is good
Obj5 : My_Class'Class := Obj1;

procedure Foo (Self : in out My_Class); -- Any procedure of an object is a method

overriding                              -- Overriding is optional
procedure Foo (Self : in out Derived);  -- Ada allows overriding (more on that later)
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
Obj1 : My_Class;
Obj2 : Derived        := (A => 12);
Obj4 : My_Class'Class := Obj2; -- Ok! Any descendant of My_Class is good
Obj5 : My_Class'Class := Obj1;

procedure Foo (Self : in out My_Class); -- Any procedure of an object is a method
procedure Foo (Self : in out Derived);  -- Ada allows overriding (more on that later)

Foo(Obj1); -- Non dispatching, calls the function as normal  (My_Class.Foo)
Foo(Obj2); -- Non dispatching, calls the overridden function (Derived.Foo)
Foo(Obj4); -- Dispatching, calls the overridden function     (Derived.Foo)
Foo(Obj5); -- Dispatching!                                   (My_Class.Foo)
```

The type atlas: the not so very basics III

Tagged types (OOP) I: inheritance and runtime dispatch

Ada

```
Obj1 : My_Class;
Obj2 : Derived        := (A => 12);
Obj4 : My_Class'Class := Obj2; -- Ok! Any descendant of My_Class is good
Obj5 : My_Class'Class := Obj1;

procedure Foo (Self : in out My_Class); -- Any procedure of an object is a method
procedure Foo (Self : in out Derived);  -- Ada allows overriding (more on that later)

Obj1.Foo; -- Non dispatching, calls the function as normal  (My_Class.Foo)
Obj2.Foo; -- Non dispatching, calls the overridden function (Derived.Foo)
Obj4.Foo; -- Dispatching, calls the overridden function     (Derived.Foo)
Obj5.Foo; -- Dispatching!                                   (My_Class.Foo)
-- Dot notation only available for tagged types
-- There is a proposal to enable it on normal types). See GNAT language extensions!
```

The type atlas: the not so very basics III

Tagged types II: abstract data types and interfaces

Ada

```ada
-- Ada 95: abstract types/procedures cannot be used directly
type Set is abstract tagged null record; -- Abstract is meant to serve as ancestor
function Union(Left, Right : Set) return Set is abstract; -- Meant to be overridden
```

The type atlas: the not so very basics III

Tagged types II: abstract data types and interfaces

Ada

```
-- Ada 95: abstract types/procedures cannot be used directly
type Set is abstract tagged null record; -- Abstract is meant to serve as ancestor
function Union(Left, Right : Set) return Set is abstract; -- Meant to be overridden

-- Ada 05: interfaces are abstract types/Objects ala Java
type Queue is limited interface; -- there are protected, task, limited and
                                 -- syncronized interfaces. More on that (much) later
type Synchronized_Queue is synchronized interface and Queue; -- Multiple inheritance!
```

The type atlas: the not so very basics III

Tagged types II: abstract data types and interfaces

Ada

```
-- Ada 95: abstract types/procedures cannot be used directly
type Set is abstract tagged null record; -- Abstract is meant to serve as ancestor
function Union(Left, Right : Set) return Set is abstract; -- Meant to be overridden

-- Ada 05: interfaces are abstract types/Objects ala Java
type Queue is limited interface; -- there are protected, task, limited and
                                 -- syncronized interfaces. More on that (much) later
type Synchronized_Queue is synchronized interface and Queue; -- Multiple inheritance!

-- Another example, creating an API to access Devices and get data out
type Serial_Device is task interface;
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
```

The type atlas: the not so very basics III

Tagged types II: abstract data types and interfaces

Ada

```ada
-- Ada 95: abstract types/procedures cannot be used directly
type Set is abstract tagged null record; -- Abstract is meant to serve as ancestor
function Union(Left, Right : Set) return Set is abstract; -- Meant to be overridden

-- Ada 05: interfaces are abstract types/Objects ala Java
type Queue is limited interface; -- there are protected, task, limited and
                                 -- syncronized interfaces. More on that (much) later
type Synchronized_Queue is synchronized interface and Queue; -- Multiple inheritance!

-- Another example, creating an API to access Devices and get data out
type Serial_Device is task interface;
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
```

Recomended read: Ada 2005 rationale (interfaces)

The type atlas: the not so very basics III

Tagged types III: where is the encapsulation???

Ada's modular nature for the win!!

Ada

```ada
-- Ada's package system is used for encapsulation
package Person is
   type Object is tagged private;
   procedure Display (O : Object);
private
   type Object is tagged
     record
         Name   : String (1 .. 30);
         Gender : Gender_Type;
     end record;
end Person;
```

More on packages later

The type atlas: the not so very basics IIII

Controlled types (akin to RAII in C++): automatic management of data

Ada

```ada
with Ada.Finalization; -- We need to import the library to gain access to controlled

type T is new Ada.Finalization.Controlled with ... record; -- Add data to the type

procedure Initialize (E : in out T); -- Called after initializing the data
procedure Adjust     (E : in out T); -- Called every time the data is modified
procedure Finalize   (E : in out T); -- Called before the data is deallocated

-- Controlled types are great to encapsulate access, file handling
-- ease use of complex data or critical data adquisition
```

The type atlas: the not so very basics IIII

Controlled types (akin to RAII in C++): automatic management of data

Ada

```ada
with Ada.Finalization; -- We need to import the library to gain access to controlled

type T is new Ada.Finalization.Controlled with ... record; -- Add data to the type

procedure Initialize (E : in out T); -- Called after initializing the data
procedure Adjust     (E : in out T); -- Called every time the data is modified
procedure Finalize   (E : in out T); -- Called before the data is deallocated

-- Controlled types are great to encapsulate access, file handling
-- ease use of complex data or critical data adquisition
```

There is so much more... See controlled types in Learn.AdaCore.
Also, there is `Limited_Controlled`. For SPARK see lightweight finalization

The type atlas: more types in future sections

We still have not seen...

- `private` types → packages
- `task` types → tasking
- `protected` types → tasking
- `exception` "types" → exceptions

The type atlas: more types in future sections

We still have not seen...

- `private` types → packages
- `task` types → tasking
- `protected` types → tasking
- `exception` "types" → exceptions

Types matter, it is not just Ada

- JS → TypeScript
- Python → annotated Python
- Rust's efforts in better typing
- C++ efforts in better types and restrictions

The type atlas: more types in future sections

We still have not seen...

- `private` types → packages
- `task` types → tasking
- `protected` types → tasking
- `exception` "types" → exceptions

Types matter, it is not just Ada

- JS → TypeScript
- Python → annotated Python
- Rust's efforts in better typing
- C++ efforts in better types and restrictions

Remeber

- Data matters
- Model your problem space → 50% solved problem
- Leverage type relationships
- Leverage attributes and aspects!
  - They are incredibly powerful
- Good types make your code more readable and self-documenting
- Good data modelling (types) allows for much easier formal verification (SPARK)

Functions, procedures and exceptions

Packages

Generics

Tasks

Liquid types, contracts and proves! Fuck yeah!

Going beyond, more resources

Ada also sucks balls

Thanks to…

- Mona Sans & Monaspace Neon fonts
- The wider Ada community
- The sli.dev presentation engine
- The wider open source (libre) software community
- My fucking self
- Zemfira for being a cool girl

Thank you!

Questions?