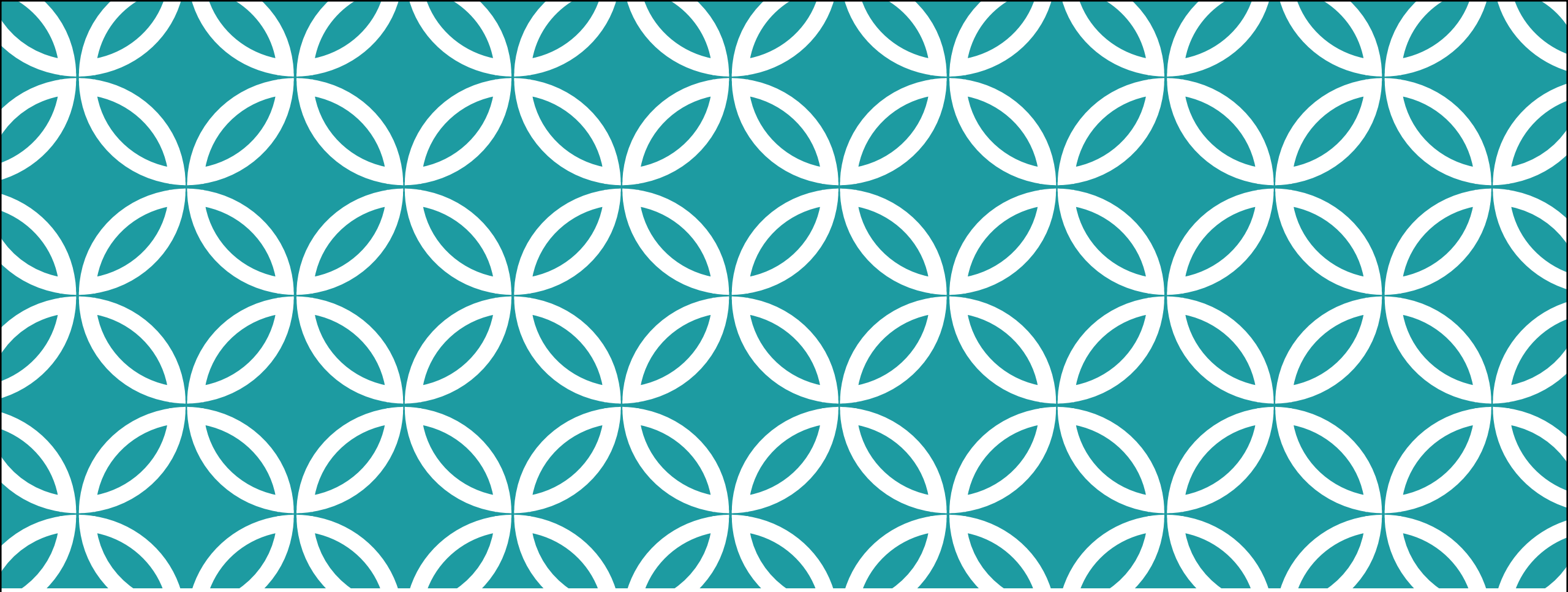


Rust Programming

Rust語言

- Rust 作為一個剛誕生不久的程式語言，從 2010 年公開到現在也才短短1x年的時間，但是在 Stack Overflow 的調查中卻是連續蟬聯了好幾年最受喜愛的程式語言排行榜的冠軍，可見 Rust 是相當有魅力而且前途一片光明的。
- 設計出 Rust 的目標之一，是要讓網路的客戶端跟伺服器在開發上更簡單，所以非常強調安全性、記憶體組態還有並行處理。這也是它的主要三大特點：**安全(safety)、性能優異(speed)、並行(concurrency)**
- Rust 是由 Mozilla 所主導的系統程式語言，旨在快速，同時保證記憶體與多執行緒的安全，這代表者使用 Rust 開發基本上不會再看到諸如：Segmentation Fault 等等的記憶體錯誤了，強大的 trait 系統，可以方便的擴充標準函式庫，這讓 Rust 雖然是**靜態的程式語言**，卻也有極大的靈活性，同時目前也有不少的應用，比如：網頁後端、系統程式還有 WebAssembly ，另外也因為其速度快與語法簡潔跟豐富的生態，也有不少公司用來處理極需要速度的部份，比如：Dropbox、npm …等等。

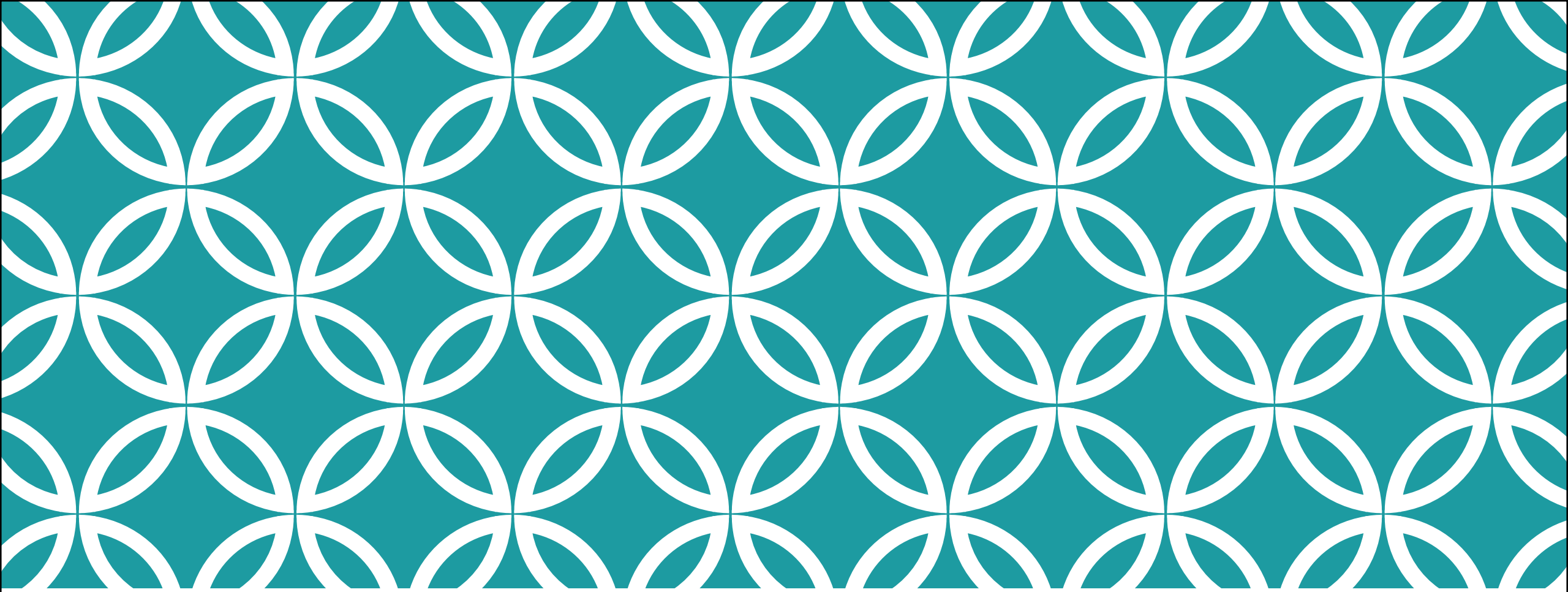
https://www.rust-lang.org/zh-TW/

Rust語言

- 重視速度與穩定性的開發者
  - Rust 適用於追求語言速度與穩定性的開發者。所謂的速度，指的是 Rust 程式碼的執行速度以及 Rust 開始撰寫程式碼的速度。Rust 編譯器的檢查能確保新增功能與重構時的穩定性。這與沒有這些檢查的語言形成對比，開發者通常不敢修改其脆弱的遺留程式碼。Rust 還力求無成本抽象化(zero-cost abstractions)，高階的特性編譯成底層程式碼後，執行的速度能像手刻一樣快。Rust 致力於讓安全的程式碼同時也能是執行迅速的程式碼。
  - Rust 語言期許能支援更多其他使用者。總體來說，Rust 最大的野心是消除數十年來開發者不得不作出的取捨，像是提供安全性與生產力、具有速度又易讀易用。
  - 已有大大小小數以百計的公司，在正式生產環境中使用 Rust 來處理各種任務，包含命令列工具、網路服務、DevOps 工具、嵌入式裝置、影音分析與轉碼、加密貨幣、生物資訊、搜尋引擎、物聯網應用、機器學習，甚至是 Firefox 瀏覽器的主要部分。

Rust語言

- Rust 被認定是一個有生產力的工具，讓能力不均的大型系統程式設計團隊能夠協同開發。底層程式碼容易產生難以察覺的錯誤，在多數其他語言中，只能靠大量的測試、以及經驗豐富的開發者小心翼翼地審核程式碼，才能找出它們。而在 Rust 中，編譯器扮演著守門員的角色阻擋這些難以捉摸的程式錯誤，包含並行(concurrency)的錯誤。透過與編譯器一同合作，開發團隊可以將他們的時間專注在程式邏輯，而不是成天追著錯誤跑。
- Rust 也將一些現代化的開發工具帶入系統程式設計的世界中：
  - Cargo：管理依賴函式庫與建構的工具，讓新增、編譯與管理依賴函式庫變得十分輕鬆，並在 Rust 生態系統維持一致性。
  - Rustfmt ：確保開發者遵循統一的程式碼風格。
  - Rust Language Server：為整合開發環境(IDE)提供了程式碼補全與行內錯誤訊息。
- 透過使用這些工具、以及其他Rust生態系統中的工具，開發者在寫系統層級的程式時更有生產力。

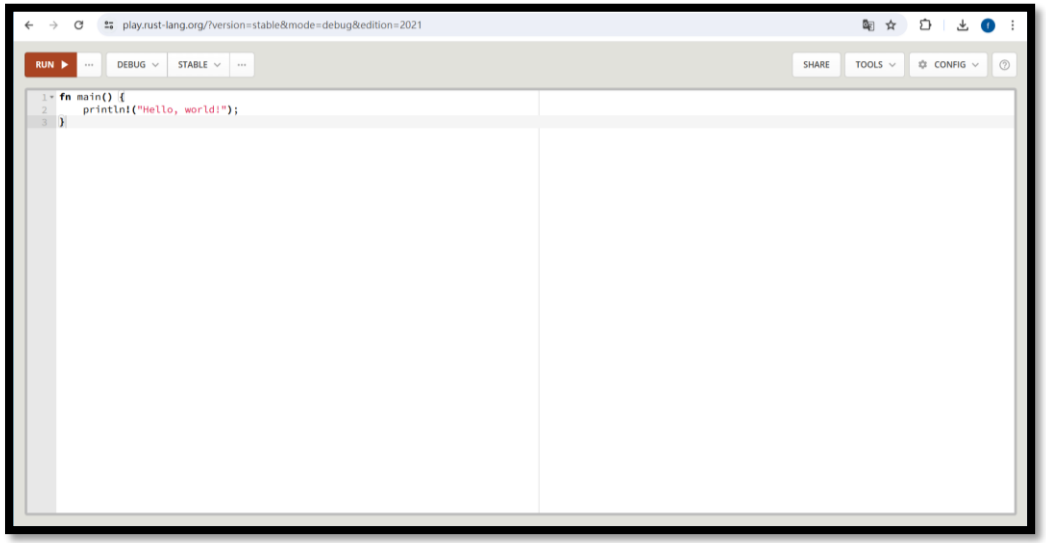


Start to Rust

Start to Rust

Rust Playground

- 連結點開來就有一隻現成的Hello World可以執行了。
- 有這個Rust Playground十分方便，可以在這平台測試跑小程式、也可以Share程式碼、複製網址貼給其他人看。並且還安裝好了很多常用的套件，如果臨時有什麼想測試的可以直接在這個網站使用。



https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=b524c6aab6bd50ee8bf790a7cad19762

Start to Rust

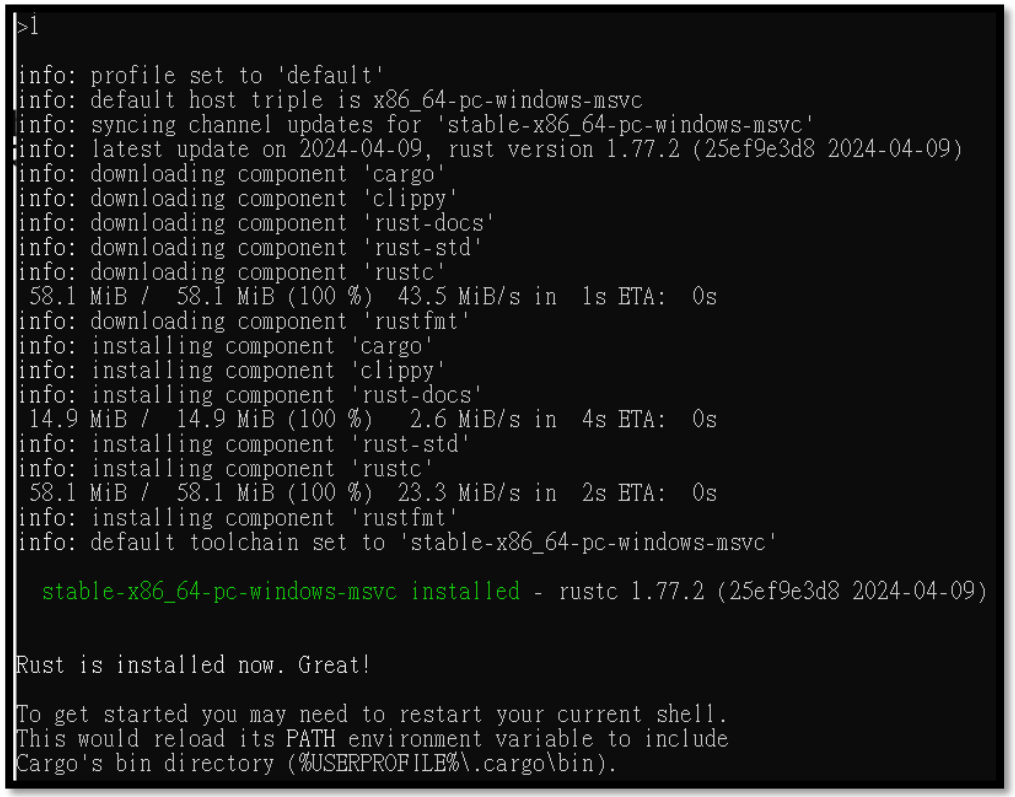
Rust 官網 | 到此下載檔案

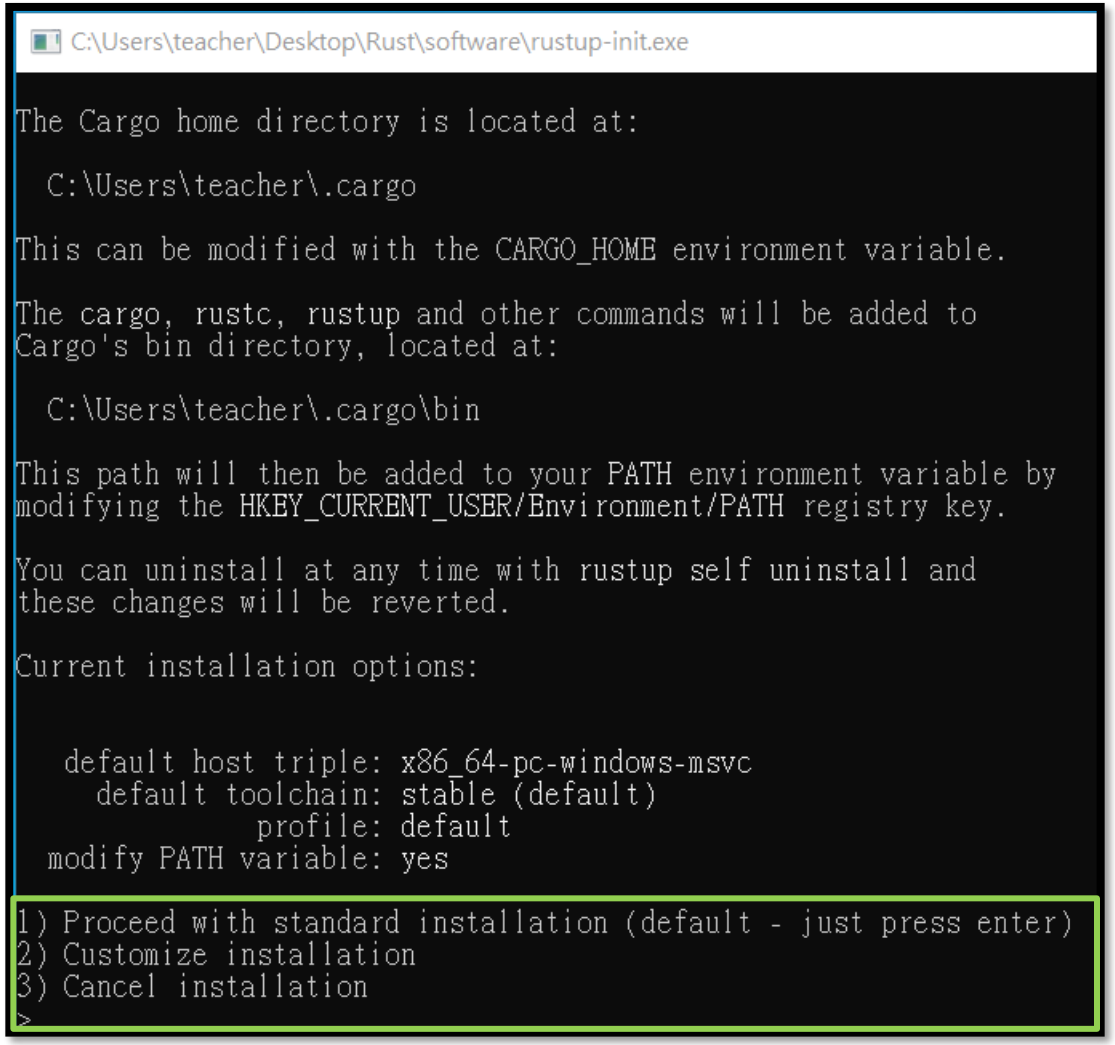
https://www.rust-lang.org/tools/install

Start to Rust

打開下載的檔案

- 選擇1

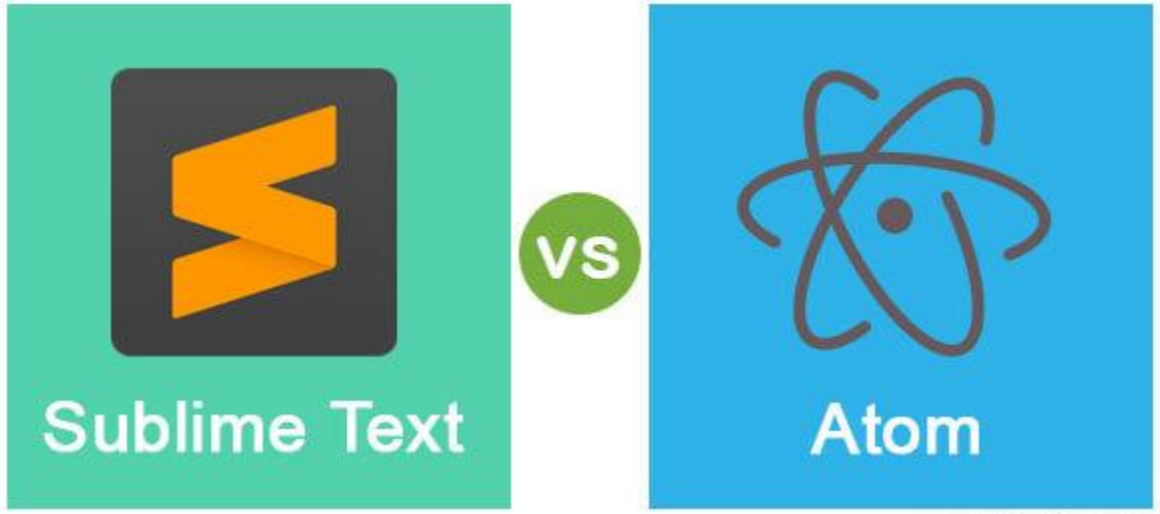




Start to Rust

安裝Rust語言：Editor-必先利其器

- 以下這些是免費的 Rust Editor，當然都是可以支援Windows和MacOS的
  - **Atom**：**https://atom.io/**
  - Sublime Text：https://www.sublimetext.com/



Start to Rust

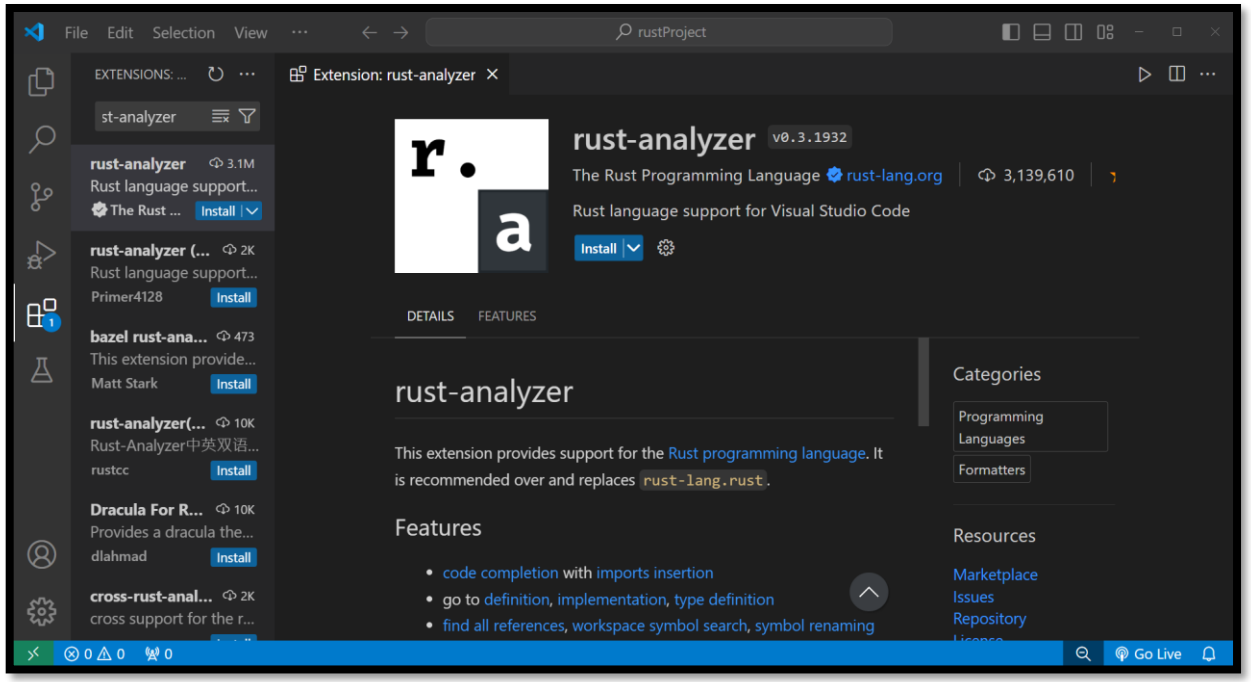
安裝Rust語言：IDE(Integrated Development Environment)-必先利其器

- 以下這些是免費的 Rust IDE，當然都是可以支援Windows和MacOS的
  - **VScode**：**https://code.visualstudio.com/**
  - Eclipse：https://www.eclipse.org/downloads/
- 付費的IDE當然比較好用，
  - **RustRover**：**https://www.jetbrains.com/rust/**
  - 新帳號註冊可免費使用30天。學生教育版信箱(Google教育版帳號也可以 .go.edu.tw)可以免費使用一年。一年繳199美金便可終身使用，只是一年期到以後，軟體要再更新、升版本要再繳下一個年度的費用。

Start to Rust

安裝Rust語言：IDE-Go必先利其器(VScode)

- 下載安裝完後，打開vs code，點選左側工具列的擴充(Extension)，搜尋rust-analyzer並且安裝它：
  - 提供了提示程式碼的部分，更重要的是會即時的提示哪邊有錯誤，可以幫助在後續寫 code 的時候會比較舒適。



Start to Rust

安裝Rust語言

- 確認Rust版本： `rustc --version`

```
C:\Users\teacher\Desktop\rustProject>rustc --version
rustc 1.77.2 (25ef9e3d8 2024-04-09)
```

  - rustc 的 c 是編譯(compiler)的意思。

- Rust 已經經常更新，所以也應該要執行這個命令： `rustup update`

```
C:\Users\teacher\Desktop\rustProject>rustup update
info: syncing channel updates for 'stable-x86_64-pc-windows-msvc'
info: checking for self-update

  stable-x86_64-pc-windows-msvc unchanged - rustc 1.77.2 (25ef9e3d8 2024-04-09)

info: cleaning up downloads & tmp directories
```

Start to Rust

用 Cargo 管理專案

- 什麼是 Cargo
  - Cargo 可以做到套件管理、建構系統、跑測試，甚至是產生文件，Cargo 的角色就像是 JavaScript 的 NPM，或是 Python 的 Pip。由於安裝 Rust 的時候，已經順便安裝了 Cargo，如果想確認有沒有安裝完成的話，可以在終端機輸入： `cargo --version` 或是： `cargo -V`

```
PS C:\Users\teacher\rustProject> cargo --version
cargo 1.77.2 (e52e36006 2024-03-26)
```

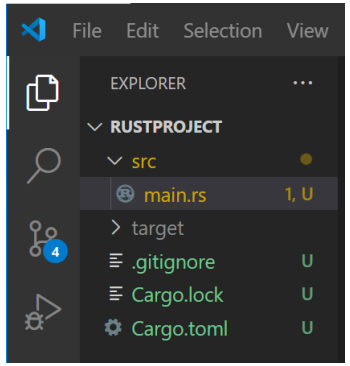
Start to Rust

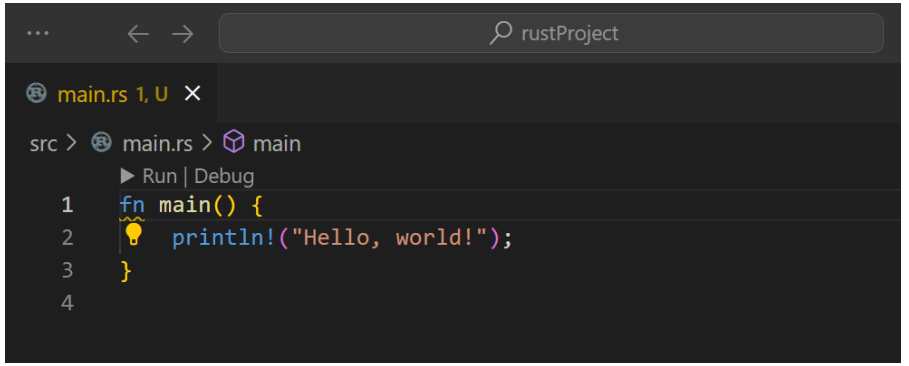
用 Cargo 管理專案

- 建立新專案：`cargo new rustProject`

```
(base) C:\Users\teacher>cargo new rustProject
warning: the name `rustProject` is not snake_case or kebab-case which is recommend
ed for package names, consider `rustproject`
     Created binary (application) `rustProject` package
```

- 可以看到 Cargo 們建立了資料夾 src，還有一個 Cargo.toml 檔案，並且們用 Git 來做版本控制。





- 資料夾 src 裡面有一個 main.rs 檔案，然後裡面理所當然地有一個主程式 main()，並且預設可以印出 "Hello, World"

Start to Rust

用 Cargo 管理專案

Cargo.toml

```toml
[package]
name = "rustProject"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

```

- 而 Cargo.toml 是一個專案的設定檔，就像 NPM 的 package.json。
- 簡單介紹一下 Cargo.toml 裡面有什麼內容：
  - [package]：代表一個區域，然後下面的內容是 Cargo 的一些設定。
    - name：是使用 Cargo 所建立這個專案的名稱。
    - version：是這個專案的版本。
    - edition：是目前所使用的 Rust 版本。
  - [dependencies]：代表另一個區域，下面則是會列出這個專案所安裝的函式庫，Rust 稱其為 crates。

Start to Rust

用 Cargo 管理專案

- Build專案： `cargo build`

  `cargo b`

```
PS C:\Users\teacher\rustProject> cargo build
   Compiling rustProject v0.1.0 (C:\Users\teacher\rustProject)
warning: crate `rustProject` should have a snake case name
  |
  = help: convert the identifier to snake case: `rust_project`
  = note: `#[warn(non_snake_case)]` on by default

warning: `rustProject` (bin "rustProject") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
```

- 這時 Cargo 就會自動產生出一個執行檔，路徑為 /target/debug/rustProject.exe
- 如果要執行結果的話，就在終端機輸入該檔案的路徑，則會顯示出運行的結果，就像下圖這樣：

```
PS C:\Users\teacher\rustProject> .\target\debug\rustProject
Hello, world!
```

Start to Rust

用 Cargo 管理專案

- Build & 執行專案
  - 如果想要 build 並且執行專案結果的話，Cargo 也有提供更快的方式：`cargo run` `cargo r`
  - Cargo 就會 build 這個專案，並且執行得到結果。
- cargo check
  - 如果只想要檢查這個專案的程式碼有無錯誤的話，可以省略產生執行檔的步驟：`cargo check` `cargo c`

Start to Rust

用 Cargo 管理專案

- cargo release
  - 如果專案已經準備好發佈的話，可以輸入： `cargo build --release` `cargo b -r`
  - 如果要順便執行的話，就輸入： `cargo run --release` `cargo r -r`
- 那麼 release 跟 build 有什麼差別呢？
  - 最大的不同是 Cargo 會最佳化編譯結果，並且產生的執行檔會在 /target/release。不過要注意的是，編譯的時間會變得更久，但可以讓該程式跑得更快。

Start to Rust

怎麼println!?

main.rs > ...

▶ Run | Debug

```rust
fn main() {
    println!("Hello, Alvin!");
}
```

OUTPUT

```
[Running] cd "c:\Users\teacher\rustProject\src\" && rustc main.rs &&
"c:\Users\teacher\rustProject\src\"main
Hello, Alvin!

[Done] exited with code=0 in 0.283 seconds
```

Start to Rust

```
main.rs > ...
▶ Run | Debug
fn main() {
    println!("Hello, Alvin!");
}
```

怎麼println!?

- fn main() {} 是 Rust 第一個優先執行的程式碼，是必需的一個函式，不會有參數，也不會回傳任何東西。**(fn是rust的關鍵字，function的簡寫、main是函式名稱，執行rust程式時將執行，如果沒有main哪可能是一個library)**
- Rust 的排版風格是 4 個空格，而不是一個 tab。
- println!() 是 Rust 拿來印出內容的一個巨集(marco)。那麼什麼是巨集呢？它跟函式不太一樣，先把它想成是一種 Rust 的功能集合，之後會來講解。**(println!是rust的巨集(macro) 如果沒有"!"表示是函式，有的話則是巨集)**
- 接著把"Hello World" 作為參數帶入，然後最後記得加上 ;，結尾非常重要，不加的話肯定會報錯。**rust是有分大小寫的。**

Start to Rust

Print! 用法

- 利用"{}"佔位符輸出**字串**，類似golang的fmt.Printf("%s, %s!", "Hello", "World")

```
print!("{}, {}!", "Hello", "World")

輸出
Hello, World!
```

- 利用"{}"佔位符輸出**數字**，類似golang的fmt.Printf("%s: %d", "Num", 9527)

```
print!("{}: {}", "Num", 9527)

輸出
Num: 9527
```

- 用"\n"輸出多行字串

```
print!("Hello, World!\nHello, World!\nHello, World!")

輸出
Hello, World!
Hello, World!
Hello, World!
```

Start to Rust

rustfmt 自動格式化

- 預設的rustfmt風格就很好用了，但如果團隊或是個人習慣想要不一樣排版風格也是可以，透過rustfmt.toml設定檔來改變。例如：二元運算子多行時要放在頭還是放在尾的部分

```
binop_separator = "Front" (默認)

let or = foofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
        || barbarbarbarbarbarbarbarbarbarbarbarbarbar;

------------------------------------------------------------
binop_separator = "Back"

let or = foofoofoofoofoofoofoofoofoofoofoofoofoofoofoo ||
        barbarbarbarbarbarbarbarbarbarbarbarbarbar;
```

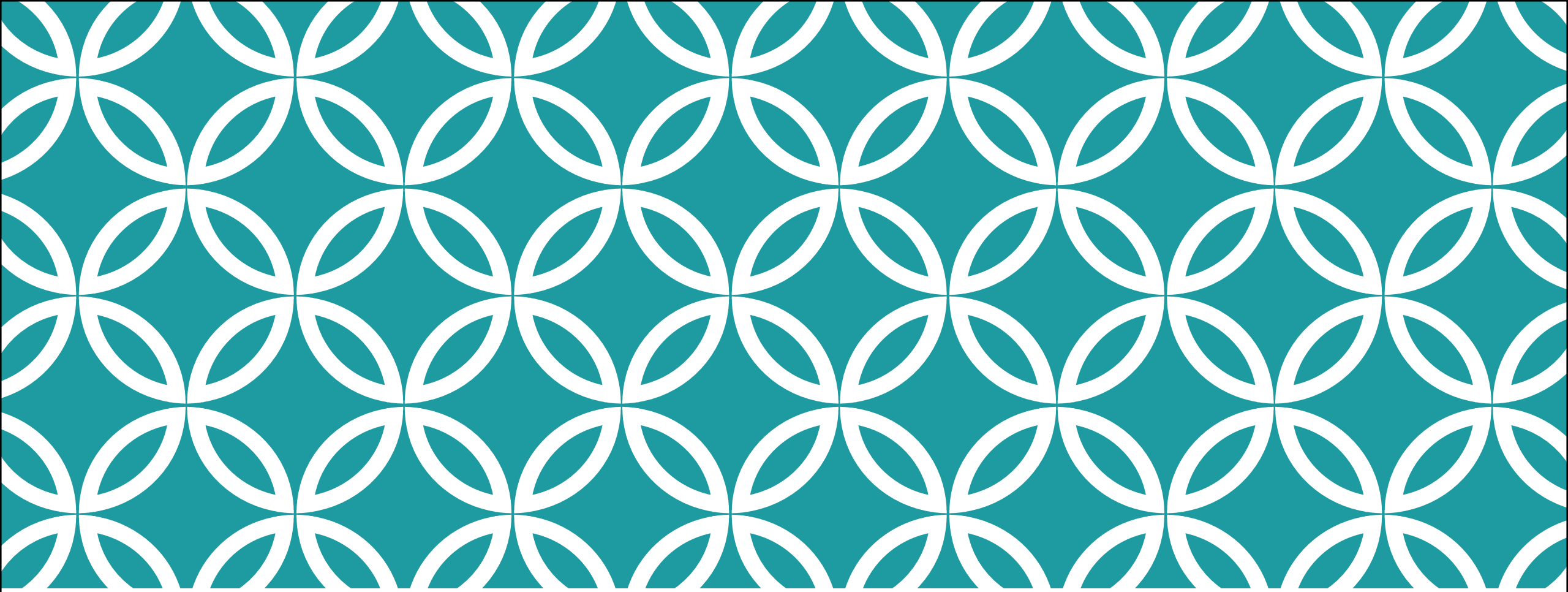
- 還有其他很多設定，可以參考這文件：
  - https://github.com/rust-lang/rustfmt/blob/master/Configurations.md
- 如果要讓存檔時自動格式化程式碼需要修改：
  - File > Preferences > Settings. 裡面的 editor.formatOnSave 打勾 就可以了

Start to Rust

rustfmt 自動格式化

- rustfmt是Rust官方提供自動格式化代碼的工具，用來統一代碼風格，避免有人用Tab有人用空格來縮排或是在大括號之後該換行之類的
- 透過rustup安裝：`rustup component add rustfmt`
- 透過指令來格式化程式碼：`rustfmt main.rs`
- 也可以透過cargo格式化整個專案：`cargo fmt`
- 也可以單純檢查並列出沒排好的地方：

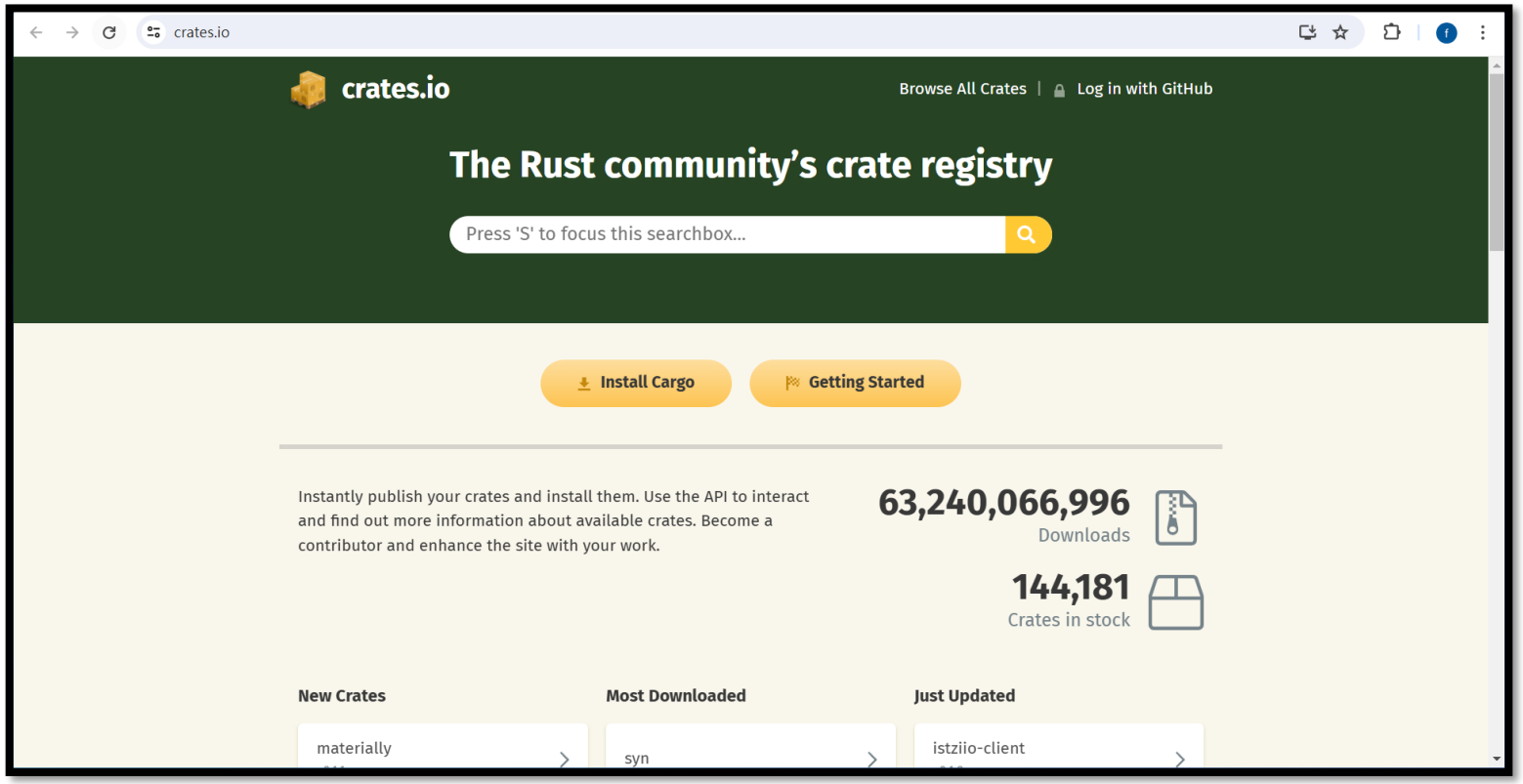
```
rustfmt --check main.rs

cargo fmt -- --check
```



套件管理工具 Cargo 與套件倉庫

套件管理工具 Cargo 與套件倉庫

Rust 的套件管理工具 Cargo 以及套件倉庫 crates.io，目前 crates.io 上有一萬八千多個套件，很多功能都可以在上面找到別人寫好的套件。



https://crates.io/

套件管理工具 Cargo 與套件倉庫

在 Rust 要使用別人寫好的套件，需要直接修改 Cargo.toml 這個檔案，把套件的名字以及需要的版本加進去，但無需擔心，它的寫法很簡單的，以下是個範例：

```
[dependencies]
rand = "0.5.5"
```

- Cargo.toml 跟 Node.js 的 package.json 的用途很像。
- 這個套件的功能是產生隨機的數字。
- 不過像這樣手動編輯檔案也挺容易出錯的，有個更好的方法，輸入底下的指令：

```
cargo install cargo-edit
```

- 這會幫 cargo 擴充新的功能，現在可以用底下的指令來加上套件了：`cargo add rand`

變數(Variable) &
資料型別(Data Type) &
註解(Comment)

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

什麼是變數(variable)?

- 變數就是分派資料(Assign Data)到一個暫時的記憶體位址(a temporary memory location)
- Rust 程式都會用到變數綁定(**名稱和內容綁在一起**)：
  - 變數綁定就是使用let宣告變數**(預設不可變)**
- 能將變數(variable)設定任何值和型別(value & type)
  - 變數預設為不可變性(immutable)
  - 可以設成可變性(mutable)
  - 常數(Constants)

```
let a = 1;
let b = 0.2;
let c = 'c';
let d = "Hola!"
let mut e = 50;
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

什麼是變數(variable)?

- 不可變性(immutable)：假如硬改不可變的變數x，這情況會...

```
fn main() {
    let x = 5;
    println!("x 的數值為：{x}");
    x = 6;
    println!("x 的數值為：{x}");
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/test-6)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("x 的數值為: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `my-project` due to previous error
exit status 101
```

- 將會收到如右錯誤「
- 這只是其中一個Rust推動使用Rust提供的安全性(safety)和簡易並行性(easy **concurrency**)寫程式的方法之一。
- 並行(Concurrency) 是什麼?
  - **在一段時間內做許多事，但每個時間點只做一件事**和**此相對的是Parallelism(平行)，每個時間點同時做許多事**。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

什麼是變數(variable)?

- 可變性(mutablility)

```
fn main() {
    let mut x = 5;
    println!("x 的數值為：{x}");
    x = 6;
    println!("x 的數值為：{x}");
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/test-6)
    Finished dev [unoptimized + debuginfo] target(s) in 1.80s
     Running `target/debug/my-project`
x 的數值為： 5
x 的數值為： 6
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

常數(constants)

- 和不可變變數一樣，常數會讓數值與名稱綁定且不允許被改變，但是不可變變數與常數還是有些差異。
- 無法在使用常數使用 mut，**常數不是預設不可變，它們永遠都不可變**
- 當使用 const 宣告而非 let 的話，必須指明型別
- **常數只能被常數表達式設置**

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- Rust 的常數命名規則為使用**全部英文大寫**並**用底線區隔每個單字**

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type

- Rust 是一門**靜態型別語言**(Statically Typed Language)，這代表它必須在編譯時知道所有變數的型別。
- 型別常見如下：
  - 整數(Integer)
  - 布林值(Boolean)
  - 浮點數(float: &32, &64)
  - 字元(char)
  - 字串(string)
- 複合型別(Compound Types)如下:
  - 元(數)組(tuples)
  - 陣列(arrays)

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：整數

長度	帶號	非帶號
8 位元	i8	u8
16 位元	i16	u16
32 位元	i32	u32
64 位元	i64	u64
128 位元	i128	u128
系統架構	isize	usize

數字字面值	範例
十進制	`98_222`
十六進制	`0xff`
八進制	`0o77`
二進制	`0b1111_0000`
位元組（僅限 `u8`）	`b'A'`

- 帶號與非帶號的區別是數字能不能有負數，換句話說就是數字能否帶有正負符號，如果沒有的話那就只會出現正整數而已。就像在紙上寫數字一樣：當需要考慮符號時，就會在數字前面加上正負號；但如果只在意正整數的話，那它可以不帶符號。**帶號數字是以二補數的方式儲存**。
- 數字字面值也可以加上底線 _ 分隔方便閱讀，比如說 1_000 其實就和指定 1000 的數值一樣。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**整數溢位**

- 假設有個變數型別是 u8 可以儲存 0 到 255 的數值。如果想要改變變數的值超出這個範圍的話，比方說像是 256，那麼就會發生整數溢位，這會產生兩種不同的結果。
  - 如果是在除錯模式編譯的話，Rust 會包含整數溢位的檢查，造成程式在執行時恐慌(panic)。Rust 使用恐慌來表示程式因錯誤而結束；當是在發佈模式下用 --release 來編譯的話，Rust 則不會加上整數溢位的檢查而造成恐慌。相反地，如果發生整數溢位的話，Rust 會作出二補數包裝的動作。
  - 簡單來說，超出最大值的數值可以被包裝成該型別的最低數值。**以 u8 為例的話，256 會變成 0、257 會變成 1**，以此類推。程式不會恐慌，但是該變數可能會得到一個不是原本預期的數值。通常依靠整數溢位的行為仍然會被視為邏輯錯誤。
- 要顯式處理可能的溢位的話，可以使用以下標準函式庫中基本型別提供的一系列方法：
  1. 將所有操作用 wrapping_* 方法包裝，像是 wrapping_add。
  2. 使用 checked_* 方法，如果有溢位的話其會回傳 None 數值。
  3. 使用 overflowing_* 方法，其會回傳數值與一個布林值來顯示是否有溢位發生。
  4. 屬於 saturating_* ，讓數值溢位時保持在最小或最大值。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**浮點數**

- 針對有小數點的浮點數提供兩種基本型別：f32 和 f64，分別佔有 32 位元與 64 位元的大小。而預設的型別為 f64，因為現代的電腦處理的速度幾乎和 f32 一樣卻還能擁有更高的精準度
- 所有的浮點數型別都是帶號的(signed)

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

- 浮點數是依照 IEEE-754 所定義的，f32 型別是單精度浮點數，而 f64 是倍精度浮點數。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**數值運算**

- 數值型別基本運算：加法、減法、乘法、除法和取餘數。**整數除法會取最接近零的下界數值**。

以下程式碼展示出如何在 let 陳述式使用這些運算：

```
fn main() {
    // 加法
    let sum = 5 + 10;

    // 減法
    let difference = 95.5 - 4.3;

    // 乘法
    let product = 4 * 30;

    // 除法
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // 結果爲 -1

    // 取餘
    let remainder = 43 % 5;
}
```

每一個陳述式中的表達式都使用了一個數學運算符號並計算出一個數值出來，賦值給該變數。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io

- 要取得使用者輸入並印出為輸出結果，需要將 io 輸入 / 輸出(input/output)函式庫引入作用域中。io 函式庫來自標準函式庫(常稱為 std)：`use std::io;`
- 在預設情況下，Rust 會將一些在標準函式庫定義的型別引入每個程式的作用域中。這樣的集合稱為 prelude，可以在標準函式庫的技術文件中看到這包含了那些型別。
- 如果想使用的型別不在 prelude 的話，需要顯式(explicit)地使用 use 陳述式(statement)將該型別引入作用域。std::io 函式庫能提供一系列實用的功能，這包含接收使用者輸入的能力。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io

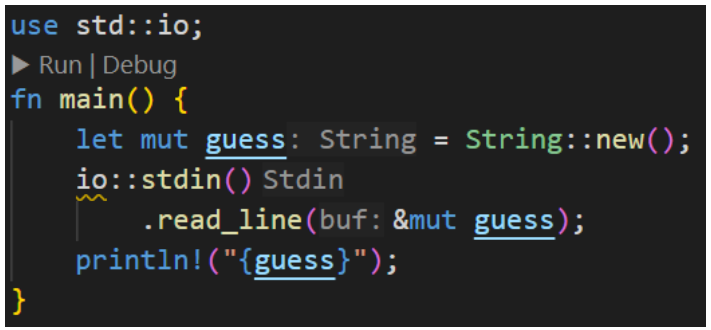
- 透過 use std::io; 來包含標準函式庫中的輸入 / 輸出功能。現在要從 io 模組(module)呼叫 stdin 函式來處理使用者的輸入：

```
use std::io;
▶ Run | Debug
fn main() {
    let mut guess: String = String::new();
    io::stdin() Stdin
        .read_line(buf: &mut guess);
    println!("{guess}");
}
```

- 如果沒有匯入 io 函式庫，也就是將 use std::io 這行置於程式最一開始的位置的話，還是能直接寫出 std::io::stdin 來呼叫函式。stdin 函式會回傳一個 std::io::Stdin 實例，這是代表終端機標準輸入控制代碼(handle)的型別。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io



- 接下來 .read_line(&mut guess) 這行會對標準輸入控制代碼呼叫 read_line 方法(method)來取得使用者的輸入。還傳遞了 &mut guess 作為引數(argument)給 read_line，來告訴它使用者輸入時該儲存什麼字串。整個 read_line 的任務就是取得使用者在標準輸入寫入的任何內容，並加入到字串中(不會覆寫原有內容)，使得可以傳遞該字串作為引數。字串引數需要是可變的，這樣該方法才能變更字串的內容。
- & 說明此引數是個參考(reference)，這讓程式中的多個部分可以取得此資料內容，但不需要每次都得複製資料到記憶體中。參考是個複雜的概念，而 Rust 其中一項主要優勢就是能夠輕鬆又安全地使用參考。現在還不用知道一堆細節才能完成程式。現在只需要知道參考和變數一樣，預設都是不可變的。因此必須寫 &mut guess 而不是 &guess 才能讓它成為可變的。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io

- **使用 Result 處理可能的錯誤：**

```
.expect("讀取該行失敗");
```

```
io::stdin().read_line(&mut guess).expect("讀取行數失敗");
```

  - 可以將程式碼寫成這樣：
  - 但是這麼長通常會很難閱讀，最好還是能夠分段。當透過 .method_name() 語法呼叫方法時，通常換行來寫並加上縮排，來拆開一串很長的程式碼會比較好閱讀。
  - read_line 會將使用者任何輸入轉換至傳入的字串，但它還回傳了一個 Result 數值。Result 是種列舉(enumerations)，常稱為 enums。列舉是種可能有數種狀態其中之一的型別，而每種可能的狀態稱之為列舉的變體(variants)。**Result 的變體有 Ok 和 Err。Ok 變體指的是該動作成功完成，且 Ok 內部會包含成功產生的數值。而 Err 變體代表動作失敗，且 Err 會包含該動作如何與為何會失敗的資訊。**
  - Result 型別的數值與任何型別的數值一樣，它們都有定義些方法。Result 的實例有 expect 方法能呼叫。如果此 Result 實例數值為 Err 的話，expect 會讓程式崩潰並顯示作為引數傳給 expect 的訊息。如果 read_line 回傳 Err 的話，這可能就是從底層作業系統傳來的錯誤結果。如果此 io::Result 實例數值為 Ok 的話，expect 會接收 Ok 的回傳值並只回傳該數值。在此例中，數值將為使用者輸入進標準輸入介面的位元組數字。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io

- **使用 Result 處理可能的錯誤：**
  - 如果沒有呼叫 expect，程式仍能編譯，但會收到一個警告：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled

warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```

  - Rust 警告沒有使用 read_line 回傳的 Result 數值，這意味著程式沒有處理可能發生的錯誤。
  - 要解決此警告的正確方式是實際進行錯誤處理，但因為只想要當問題發生時直接讓程式當掉，所以可以先使用 expect 就好。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

資料輸入：std::io

- 轉換文字型態：

```rust
use std::io;
▶ Run | Debug
fn main() {
    let mut guess: String = String::new();
    io::stdin() Stdin
        .read_line(buf: &mut guess);
    // println!("{guess}");
    let trimmed: &str = guess.trim();
    match trimmed.parse::<u32>() {
        Ok(i: u32) => println!("your integer input: {}", i),
        Err(..) => println!("this was not an integer: {}", trimmed),
    };
}
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**布林**

- 如同其他多數程式語言一樣，Rust 中的布林型別有兩個可能的值：true 和 false。布林值的大小為一個位元組。要在 Rust 中定義布林型別的話用 bool，如範例所示：

```
fn main() {
    let t = true;

    let f: bool = false; // 型別詮釋的方式
}
```

- 布林值最常使用的方式之一是作為條件判斷，像是在 if 表達式中使用。將會在「控制流程」段落介紹如何在 Rust 使用 if 表達式。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**字元(char)**

- Rust 的 char 型別是最基本的字母型別，以下程式碼顯示了使用它的方法：

```
fn main() {
    let something = 'a';
    let other_thing: char = 'A'; // 明確標註型別的寫法
    let heart_eyed_cat = '★';
}
```

  - 注意到 char 字面值是用**單引號**賦值，**宣告字串字面值時才是用雙引號**。Rust 的 char 型別大小為四個位元組並表示為一個 Unicode 純量數值，這代表它能擁有的字元比 ASCII 還來的多。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**字串String**

```
fn main() {
    let hello = String::from("Hello, world!");
    println!("{}", hello)
}
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別**

- 複合型別可以組合數個數值為一個型別，Rust 有兩個基本複合型別：**元組(tuples)和陣列(arrays)**
- 元組(Tuple)型別
  - 元組是個將許多**不同型別**的數值合成一個複合型別的常見方法。**元組擁有固定長度：一旦宣告好後，它們就無法增長或縮減。**
  - 建立一個元組的方法是寫一個用括號囊括起來的數值列表，每個值再用逗號分隔開來。元組的每一格都是一個獨立型別，不同數值不必是相同型別。以下範例也加上了型別詮釋，平時不一定要加上：

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

  - 此變數 tup 就是整個元組，因為一個元組就被視為單一複合元素。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-元組(Tuple)**

- 元組(Tuple)型別
  - 要拿到元組中的每個獨立數值的話，可以用模式配對(pattern matching)來解構一個元組的數值，如以下所示：

```
fn main() {
    let tup = (800, 4.5, 3);

    let (x, y, z) = tup;

    println!("y的值: {y}");
}
```

```
y的值: 4.5
```

  - 此程式先是建立了一個元組然後賦值給 tup，接著它用模式配對和 let 將 tup 拆成三個個別的變數 x、y 和 z。這就叫做解構(destructuring)，因為它將單一元組拆成了三個部分。最後程式將 y 的值印出來，也就是 4.5。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-元組(Tuple)**

- 元組(Tuple)型別
  - **也可以直接用句號（.）**再加上數值的索引來取得元組內的元素。舉例來說：

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

  - 此程式建立了元組 x，然後用它們個別的索引來存取元組的元素。和多數程式語言一樣，**元組的第一個索引是 0**。
  - 沒有任何數值的元組有一種特殊的名稱叫做**單元型別(Unit)，其數值與型別都寫作()，通常代表一個空的數值或空的回傳型別**。表達式要是沒有回傳任何數值的話，它們就會隱式回傳單元型別。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-陣列(Arrays)**

- 陣列(Arrays)型別
  - 另一種取得數個數值集合的方法是使用陣列。**和元組不一樣的是，陣列中的每個型別必須是一樣的**。和其他語言的陣列不同，**Rust 的陣列是固定長度的**。
  - 將數值寫在陣列中的括號內，每個數值再用逗號區隔開來：

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

    - 當想要的資料被配置在堆疊(stack)而不是堆積(heap)的話，使用陣列是很好的選擇(會在後面討論堆疊與堆積)。或者當想確定永遠會取得固定長度的元素時也是。**所以陣列不像向量(vector)型別那麼有彈性**，向量是標準函式庫提供的集合型別，**類似於陣列但允許變更長度大小**。如果不確定該用陣列或向量的話，通常應該用向量就好。
  - 不過如果知道元素的多寡不會變的話，陣列就是個不錯的選擇。舉例來說，如果想在程式中使用月份的話，可能就會選擇用陣列宣告，因為永遠只會有 12 個月份：

```
let months = ["一月", "二月", "三月", "四月", "五月", "六月", "七月",
              "八月", "九月", "十月", "十一月", "十二月"];
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-陣列(Arrays)**

- 陣列(Arrays)型別
  - 要詮釋陣列型別的話，可以在中括號寫出型別和元素個數，並用分號區隔開來，如以下所示：

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

  - i32 在此是每個元素的型別，在分號後面的數字 5 指的是此陣列有五個元素。
  - 如果想建立的陣列中每個元素數值都一樣的話，可以指定一個數值後加上分號，最後寫出元素個數。如以下所示：

```
let a = [3; 5];
```

    - 陣列 a 會包含 5 個元素，然後每個元素的初始化數值均為 3。這樣寫與 let a = [3, 3, 3, 3, 3]; 的寫法一樣，但比較簡潔。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-陣列(Arrays)**

- 陣列(Arrays)型別

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let months = [
        "一月",
        "二月",
        "三月",
        "四月",
        "五月",
        "六月",
        "七月",
        "八月",
        "九月",
        "十月",
        "十一月",
        "十二月",
    ];
    let b: [i32; 5] = [1, 2, 3, 4, 5];
    let c = [3; 5];

    println!("The value of a is: {:?}", a);
    println!("The value of months is: {:?}", months);
    println!("The value of b is: {:?}", b);
    println!("The value of c is: {:?}", c);
}
```

```
> cargo run
    Blocking waiting for file lock on build directory
   Compiling my-project v0.1.0 (/home/runner/test-6)
    Finished dev [unoptimized + debuginfo] target(s) in 2.59s
     Running `target/debug/my-project`
The value of a is: [1, 2, 3, 4, 5]
The value of months is: ["一月", "二月", "三月", "四月", "五月", "六月", "七月", "八月"
, "九月", "十月", "十一月", "十二月"]
The value of b is: [1, 2, 3, 4, 5]
The value of c is: [3, 3, 3, 3, 3]
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-陣列(Arrays)**

- 獲取陣列元素
  - 一個陣列是被配置在堆疊上且已知固定大小的一整塊記憶體，可以用索引來取得陣列的元素，比如：

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

  - 在此範例中，變數 first 會得到數值 1，因為這是陣列索引 [0] 的數值。變數 second 則會從陣列索引 [1] 得到數值 2。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

檔案(資料)型別Data Type：**複合型別-陣列(Arrays)**

- 無效的陣列元素存取
  - 如果存取陣列之後的元素會發生什麼事呢？假設修改成以下範例：

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("請輸入陣列索引");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("讀行失敗");

    let index: usize = index
        .trim()
        .parse()
        .expect("輸入的索引並非數字");

    let element = a[index];

    println!(
        "索引 {index} 元素的數值爲：{element}"
    );
}
```

此程式碼能編譯成功。如果透過 cargo run 執行此程式碼並輸入 0、1、2、3 或 4 的話，程式將會印出陣列索引對應的數值。但如果輸入超出陣列長度的數值，像是 10 的話，會看到像是這樣的輸出結果：

```
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 10', src/main.rs:19:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

此程式會在使用無效數值進行索引操作時產生執行時(runtime)錯誤。程式會退出並回傳錯誤訊息，且不會執行最後的 println!。
當嘗試使用索引存取元素時，Rust 會檢查索引是否小於陣列長度，如果索引大於或等於陣列長度的話，Rust 就會恐慌。這樣的檢查必須發生在執行時，尤其是在此例，因爲編譯器無法知道之後的使用者將會輸入哪些數值。

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

註解(Comment)

- 所有程式設計師均致力於讓程式碼易於閱讀，不過有時候額外的解釋還是需要的。這種情況下，開發者會在程式碼留下一些註解(comments)，編譯器會忽略這些字，但其他人在閱讀程式碼時就會覺得很有幫助。
- 在 Rust 中，慣用的註解風格是用兩行斜線再加上一個空格起頭，然後註解就能一直寫到該行結束為止。如果註解會超過一行的話，需要在每一行都加上 //
- 以下是一個簡單地註解：

```
// This is the entry point of the application
fn main() {
    println!("Hellllllo World!");
}
```

變數(Variable) & 資料型別(Data Type) & 註解(Comment)

註解(Comment)

- 註解也可以加在程式碼之後：

```
fn main() {
    let lucky_number = 7; // 幸運 777！
}
```

- 不過會更常看到它們用用以下格式，註解會位於要說明的程式碼上一行

```
fn main() {
    // 幸運 777！
    let lucky_number = 7;
}
```