

# CS 241: Systems Programming

## Lecture 10. Compound types and Loops

Spring 2024

Prof. Stephen Checkoway

# Compound types

Compound types group multiple values together

- Tuples
- Arrays
- Structures (we'll get to these later)
- Enumerations (we'll get to these later)

# Tuples

Tuples let us group multiple, heterogeneous types together

They have fixed size (no adding or removing elements)

```
let val: (i32, char, f64) = (42, '🦀', 6.022e23);
```

Tuples can be built from any types

# Accessing elements of the tuple

Use `.0`, `.1`, `.2`, etc. to access fields

```
let val: (i32, char, f64) = (42, '🦀', 6.022e23);  
let num = val.0;  
let ch = val.1;  
let float = val.2;
```

# Destructuring tuples

```
let val: (i32, char, f64) = (42, '🦀', 6.022e23);  
let (num, ch, float) = val;
```

Does what you want: assigns 42 to `num`, 🦀 to `ch`, and 6.022e23 to `float`

# Tuples in parameters

```
/// Compute the Euclidean distance between p1 and p2
fn distance(p1: (f64, f64, f64), p2: (f64, f64, f64)) -> f64 {
    let (x1, y1, z1) = p1;
    let (x2, y2, z2) = p2;
    let dx = x1 - x2;
    let dy = y1 - y2;
    let dz = z1 - z2;

    (dx * dx + dy * dy + dz * dz).sqrt()
}

fn main() {
    let p1 = (1.2, -8.6, 3.0);
    let p2 = (-10.2, -9.3, -6.1);
    println!("Distance = {}", distance(p1, p2))
}
```

# Returning tuples

```
fn origin() -> (f64, f64, f64) {  
    (0.0, 0.0, 0.0)  
}
```

# Printing tuples

```
fn main() {  
    let p1 = (1.2, -8.6, 3.0);  
    let p2 = (-10.2, -9.3, -6.1);  
    println!("{p1} - {p2} = {}", distance(p1, p2));  
}
```

```
error[E0277]: `(f64, f64, f64)` doesn't implement `std::fmt::Display`  
--> compoundtypes.rs:30:16  
30 |         println!("{p1} - {p2} = {}", distance(p1, p2))  
    |                   ^^^^^ `(f64, f64, f64)` cannot be formatted with the default formatter  
    = help: the trait `std::fmt::Display` is not implemented for `(f64, f64, f64)`  
    = note: in format strings you may be able to use `{:?}` (or `{:#?}` for pretty-print)  
instead
```



# Printing tuples' Debug representations

```
fn main() {  
    let p1 = (1.2, -8.6, 3.0);  
    let p2 = (-10.2, -9.3, -6.1);  
    println!("{p1:?} - {p2:?} = {}", distance(p1, p2));  
}
```

Output:

```
| (1.2, -8.6, 3.0) - (-10.2, -9.3, -6.1) | = 14.603424255975035
```

```
let val: (String, bool) = (String::from("clickers"), true);
```

how do we access the boolean component of `val`?

A. `val.1`

B. `val.2`

C. `val.bool`

D. `val.true`

E. `val.clickers`

# Modifying tuple components

```
fn main() {  
    let mut val: (String, bool) = (String::new(), false);  
    println!("{val:?}");  
  
    val.0.push_str("Rust is great!");  
    val.1 = true;  
  
    println!("{val:?}");  
}
```

Output:  
("", false)  
("Rust is great!", true)

# Unit — a zero-element tuple

There's zero-element tuple called **unit** which we write as `()`

The `()` type has exactly one value: `()`

```
let unit: () = ();
```

Or just

```
let unit = ();
```

This is often used with a `Result<(), E>` when a function has no meaningful value on success, but might return an error

# Unit—a zero-element tuple

```
fn could_error() -> Result<(), String> {  
    // ...  
    if some_error_condition {  
        return Err(String::from("Some error"));  
    }  
  
    Ok(())  
}
```

This returns `()` wrapped in an `Ok`

# Arrays

Arrays let us group multiple values of a single type together

```
let months: [&str; 12] = [  
    "January",  
    "February",  
    "March",  
    "April",  
    "May",  
    "June",  
    "July",  
    "August",  
    "September",  
    "October",  
    "November",  
    "December",  
];
```

Type

Count

# Accessing array elements

Standard array access notation: `arr[index]`

```
println!("{}", months[5]);  
let idx: usize = 8;  
println!("{}", months[idx]);  
println!("{}", months[idx + 1]);
```

# Accessing array elements

The type of the index must be a `usize`

```
let idx: i32 = 8;  
println!("{}", months[idx]);
```

```
error[E0277]: the type `[&str]` cannot be indexed by `i32`  
--> compoundtypes.rs:60:27  
60 |         println!("{}", months[idx]);  
   |                        ^^^ slice indices are of  
type `usize` or ranges of `usize`
```



# Casting i32 to usize

The type of the index must be a usize

```
let idx: i32 = 8;  
println!("{}", months[idx as usize]);
```

Must be careful: negative numbers wrap around to large positive numbers

```
let idx: i32 = -8;  
println!("{}", months[idx as usize]);
```

error: this operation will panic at runtime

--> compoundtypes.rs:60:20

```
60 |         println!("{}", months[idx as usize]);  
    |                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ index out of bounds: the  
length is 12 but the index is 18446744073709551608
```

# Rust prevents out-of-bounds access

Error at compile time, if the compiler can prove the index is out of bounds (as on the previous slide)

At run time, otherwise:

```
thread 'main' panicked at 'index out of bounds: the len
is 12 but the index is 12'
```

# Creating an array by copying an element

```
let arr: [i64; 20] = [42; 20];  
println!("{arr:?}");
```

Output:

```
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,  
42, 42, 42, 42, 42, 42]
```

Which of the following is the correct syntax to create an array of 5 u32 integers with values 1, 2, 3, 4, and 5?

- A. `let arr: [u32; 5] = [1..5];`
- B. `let arr: [5; u32] = [1, 2, 3, 4, 5];`
- C. `let arr = [1, 2, 3, 4, 5];`
- D. `let arr: [5; u32] = [1..5; 5];`
- E. `let arr: [u32; 5] = [1, 2, 3, 4, 5];`

# Vectors

Vectors are growable arrays (like ArrayList in Java or a list in Python)

```
let mut v: Vec<String> = Vec::new();  
v.push(String::from("Hi"));  
v.push(String::from("there!"));  
println!("{v:?}");
```

Output:

```
["Hi", "there!"]
```

# Accessing elements

Access elements just like you would an array

```
println!("{}", v[0], v[1]);
```

Output:

Hi there!

# Getting the length of a Vec

Use the len() method

```
let mut v: Vec<String> = Vec::new();  
println!("{}", v.len());  
v.push(String::from("Hi"));  
println!("{}", v.len());  
v.push(String::from("there!"));  
println!("{}", v.len());
```

Output:

0  
1  
2

# Create a vector with some elements

To create a vector with some elements, use `vec! []`

```
let v: Vec<i32> = vec![1, 2, 3, 4, 5];
```

To make it mutable, use `let mut` as usual



# Loop

```
fn main() {  
    let mut n = 17;  
    loop {  
        print!("{n} ");  
        if n == 1 {  
            break;  
        }  
        if n % 2 == 0 {  
            n = n / 2;  
        } else {  
            n = 3 * n + 1;  
        }  
    }  
    println!();  
}
```

Output: 17 52 26 13 40 20 10 5 16 8 4 2 1

# While loop

```
fn main() {  
    let mut n = 17;  
    while n != 1 {  
        print!("{n} ");  
        if n % 2 == 0 {  
            n = n / 2;  
        } else {  
            n = 3 * n + 1;  
        }  
    }  
    println!("1");  
}
```

# Looping over a vector (or array) by index

```
fn main() {  
    let v: Vec<i32> = vec![1, 2, 3, 4, 5];  
    let mut idx = 0;  
    while idx < v.len() {  
        println!("v[{idx}] = {}", v[idx]);  
        idx += 1;  
    }  
}
```

Output:

```
v[0] = 1  
v[1] = 2  
v[2] = 3  
v[3] = 4  
v[4] = 5
```

# Loop over a collection

We can use a for loop (similar to Python's for loop)

```
fn main() {  
    let v: Vec<i32> = vec![1, 2, 3, 4, 5];  
  
    for num in v {  
        println!("{}", num);  
    }  
}
```

# The for loop takes ownership of the vector!

```
for num in v {  
    println!("{num}");  
}  
for num in v {  
    println!("{num}");  
}
```

```
error[E0382]: use of moved value: `v`  
--> compoundtypes.rs:106:16
```

```
101 |         let v: Vec<i32> = vec![1, 2, 3, 4, 5];  
    |         - move occurs because `v` has type `Vec<i32>`, which does not  
implement the `Copy` trait  
102 |  
103 |         for num in v {  
    |         - `v` moved due to this implicit call to `.into_iter()`  
...  
106 |         for num in v {  
    |         ^ value used here after move
```

# Quick fix

```
for num_ref in &v {  
    println!("{num_ref}");  
}  
for num in v {  
    println!("{num}");  
}
```

By using `&v`, we're saying loop over a reference to `v`, this does not consume `v` so we can use it again later

When we loop over a reference, the variable is bound to a reference to the elements, rather than being the elements themselves (more about refs later!)

# Loop over a collection with index

We can use a for loop (similar to Python's for loop)

```
fn main() {  
    let v: Vec<i32> = vec![1, 2, 3, 4, 5];  
  
    for (idx, num) in v.iter().enumerate() {  
        println!("v[{idx}] = {num}");  
    }  
}
```

`v.iter()` returns an iterator over references to elements

`.enumerate()` returns an iterator over pairs (index, element)