# General coding

stirlingcodingclub.github.io/coding_types/slides.pdf

Stirling Coding Club

10 February 2021

# How does code actually work?

- ▶ Programmers work with **source code** (e.g., R, python)
- ▶ Computers execute **machine language** (binary)
- ▶ To get from source code to machine language, we can *compile* code or *interpret* it.

# Compiled versus interpreted code

## Compiled code

- ▶ Directly translates everything to machine language
- ▶ Transatlation must occur before running code
- ▶ C, C++, FORTRAN, Pascal (low-level languages)[1]
- ▶ Two steps to execute code
- ▶ Faster performance (executes immediately after compiling)
- ▶ Slower and more laborious to code

## Interpreted code

- ▶ Code is run bit by bit through an interpreter
- ▶ Interpreter breaks down and executes code 'on the fly'
- ▶ R, Perl, MATLAB, Python (high-level languages)[1]
- ▶ One step to execute code
- ▶ Slower performance (executes after interpreted)
- ▶ Faster and more intuitive to code

# Compiled versus interpreted code

**Compiled code**

- ▶ Directly translates everything to machine language
- ▶ Transatlation must occur before running code
- ▶ C, C++, FORTRAN, Pascal (low-level languages)[1]
- ▶ Two steps to execute code
- ▶ Faster performance (executes immediately after compiling)
- ▶ Slower and more laborious to code

**Interpreted code**

- ▶ Code is run bit by bit through an interpreter
- ▶ Interpreter breaks down and executes code 'on the fly'
- ▶ R, Perl, MATLAB, Python (high-level languages)[1]
- ▶ One step to execute code
- ▶ Slower performance (executes after interpreted)
- ▶ Faster and more intuitive to code

1. Note, techically, any language *could* be compiled or interpreted.

# R packages can include both compiled and intereted code



| Name | Size | Type | Date Modified ▲ |
|---|---|---|---|
| R | 4.1 kB | folder | 28/01/21 |
| notebook | 4.1 kB | folder | 28/01/21 |
| src | 4.1 kB | folder | 24/01/21 |
| docs | 4.1 kB | folder | 31/05/20 |
| vignettes | 4.1 kB | folder | 27/05/20 |
| tests | 4.1 kB | folder | 27/05/20 |
| man | 4.1 kB | folder | 27/05/20 |
| data | 4.1 kB | folder | 27/05/20 |
| gmse.Rproj | 303 bytes | R Project | 24/01/21 |
| DESCRIPTION | 2.5 kB | plain text document | 31/05/20 |
| NEWS.md | 2.4 kB | Markdown document | 27/05/20 |
| README.md | 5.7 kB | Markdown document | 27/05/20 |
| _pkgdown.yml | 477 bytes | YAML document | 27/05/20 |
| NAMESPACE | 2.0 kB | plain text document | 27/05/20 |
| cran-comments.md | 686 bytes | Markdown document | 27/05/20 |

# R packages can include both compiled and intereted code

**An R file named 'resource.R'**

```
run_resource <- function(RESOURCE_c, LANDSCAPE_c, PARAMETERS_c){
    .Call("resource", RESOURCE_c, LANDSCAPE_c, PARAMETERS_c);
}
```

# R packages can include both intereted code

**An R file named 'resource.R'**

```r
run_resource <- function(RESOURCE_c, LANDSCAPE_c, PARAMETERS_c){
    .Call("resource", RESOURCE_c, LANDSCAPE_c, PARAMETERS_c);
}
```

**A C file named 'resource.c'**

```c
SEXP resource(SEXP RESOURCE, SEXP LANDSCAPE, SEXP PARAMETERS){

    /* SOME STANDARD DECLARATIONS OF KEY VARIABLES, POINTERS   */
    /* ===================================================== */
    int xloc, yloc;      /* x and y locs in the RESOURCE array  */
    int land_x, land_y;  /* x and y maximum loc given LANDSCAPE */
    int zloc, land_z;    /* z locations                          */
    int resource;        /* Index for resource (rows RESOURCE)  */
    int resource_new;    /* Index for resource in new array      */
    int trait;           /* Index for resource traits            */
    int res_number;      /* Resources included (default = 1)     */
```

# The role of the processor and memory

▶ Machine code is a list of instructions written in binary (1s & 0s)

▶ Binary instructions sent to the *Central Processing Unit (CPU)*

    ▶ CPUs can read and write to memory, and do maths (and that's about it)

    ▶ Instructions tell the CPU to read and write information to memory

# The role of the processor and memory

- ▶ Machine code is a list of instructions written in binary (1s & 0s)
- ▶ Binary instructions sent to the *Central Processing Unit (CPU)*
    - ▶ CPUs can read and write to memory, and do maths (and that's about it)
    - ▶ Instructions tell the CPU to read and write information to memory
- ▶ Random-access memory (RAM, or just 'memory') is separate from the CPU, and holds data that can be read and changed
    - ▶ Memory exists as binary digits ('bits') of ones and zeros
    - ▶ Bits are grouped in chunks of eight to make one 'byte'

# Thinking about computer memory

In R, memory is allocated automatically when we assign values.

```r
array_1 <- 1:6; # Assign 'array_1' values 1 through 6
```

# Thinking about computer memory

In R, memory is allocated automatically when we assign values.

```r
array_1 <- 1:6; # Assign 'array_1' values 1 through 6
```

In C, memory needs to be allocated to a memory location.

| Location 1 | Location 2 | Location 3 | Location 4 | Location 5 | Location 6 |
|------------|------------|------------|------------|------------|------------|
| 0x7fc010 | 0x7fc011 | 0x7fc012 | 0x7fc013 | 0x7fc014 | 0x7fc015 |

# Thinking about computer memory

In R, memory is allocated automatically when we assign values.

```r
array_1 <- 1:6; # Assign 'array_1' values 1 through 6
```

In C, memory needs to be allocated to a memory location.

| Location 1 | Location 2 | Location 3 | Location 4 | Location 5 | Location 6 |
| --- | --- | --- | --- | --- | --- |
| 0x7fc010 | 0x7fc011 | 0x7fc012 | 0x7fc013 | 0x7fc014 | 0x7fc015 |

The `0x` indicates hexidecimal, but we can translate to decimal.

| Location 1 | Location 2 | Location 3 | Location 4 | Location 5 | Location 6 |
| --- | --- | --- | --- | --- | --- |
| 8372240 | 8372241 | 8372242 | 8372243 | 8372244 | 8372245 |

# Thinking about computer memory

In R, memory is allocated automatically when we assign values.

```r
array_1 <- 1:6; # Assign 'array_1' values 1 through 6
```

In C, memory needs to be allocated to a memory location.

| Location 1 | Location 2 | Location 3 | Location 4 | Location 5 | Location 6 |
|------------|------------|------------|------------|------------|------------|
| 0x7fc010   | 0x7fc011   | 0x7fc012   | 0x7fc013   | 0x7fc014   | 0x7fc015   |

The `0x` indicates hexidecimal, but we can translate to decimal.

| Location 1 | Location 2 | Location 3 | Location 4 | Location 5 | Location 6 |
|------------|------------|------------|------------|------------|------------|
| 8372240    | 8372241    | 8372242    | 8372243    | 8372244    | 8372245    |

```c
array_1 = malloc(6 * sizeof(double));
```

# Thinking about computer memory

|       | Column 2 | Column 3 | Column 4 |
|-------|----------|----------|----------|
| Row 1 | 3        | 5        | 4        |
| Row 2 | 1        | 7        | 6        |

# Thinking about computer memory

|         | Column 2 | Column 3 | Column 4 |
|---------|----------|----------|----------|
| Row 1   | 3        | 5        | 4        |
| Row 2   | 1        | 7        | 6        |

```
##      [,1] [,2] [,3]
## [1,]    3    5    4
## [2,]    1    7    6
```

# Thinking about computer memory

Pointer to pointer (\*\*array2D)

| Address: 7ffef08a8978 | Address: 7ffef08a8979 |
|---|---|
| Value: 150b010 | Value: 150b013 |

Addresses and values

| 150b010 | 150b011 | 150b012 | 150b013 | 150b014 | 150b015 |
|---|---|---|---|---|---|
| 3 | 5 | 4 | 1 | 7 | 6 |