# KIT107 PROGRAMMING

## *C Programming*

Dr Julian Dermoudy &
Dr Rainer Wasinger
School of Engineering and ICT
University of Tasmania

## Structure

- approximately 12 lectures (4 weeks)



## Preface

- 41 years old
- tied to the Unix operating system
- not a good first language (terse)
- very good for experienced programmers
- contains pointers!!

# 1. Introduction





## 1.1 C and Java/Python Similarities

- it has similar types
- it has similar variable declarations (to Java)
- it has identical control structures (to Java) and similar to Python
- it has similar comparison, logical, and arithmetic operators
- it supports the dynamic allocation of variables



## 1.1 C and Java/Python Similarities (continued)

- it supports the provision of ‘method’ declarations (interfaces/header files)
- it is case-sensitive
- lowercase is typically used — except for constant values — although underscores (_) are used in place of change of case: the Java/Python variable `myVar` would be `my_var` in C

## 1.1 C and Java/Python Similarities (continued)

- single statements end in a semi-colon (;)
- like Java, groups of statements may be collected together using braces ({}) to form compound statements or blocks

KIT107 © JRD 2014 Slide 7

## 1.1 C and Java/Python Similarities (continued)

- Unlike Python, groups of statements don't need to be indented (although they should be!)
- It is strongly typed (like Java but unlike Python)
- Unlike Python, arrays are fixed length

KIT107 © JRD 2014 Slide 8

## 1.2 C and Java/Python Differences

- it is not object-oriented (and therefore lacks encapsulation, information hiding, inheritance, instantiation of objects, etc.)
- it is procedural (and therefore contains types, variable declarations, and functions but these are not connected in any visible way)

KIT107 © JRD 2014 Slide 9

## 1.2 C and Java/Python Differences (continued)

- it has pointers, not references
  - pointers are an 'unsafe' form of references in which literal values and arithmetic are permitted on addresses
- it is compiled, not interpreted
- compiled C programs are not "architecture-neutral"

KIT107 © JRD 2014 Slide 10

## 1.2 C and Java/Python Differences (continued)

- it has only the traditional (application) form of program and no interactive command line
- it does not allow the importation of behaviour, only types
- it does not support exception handling
- there are no string or boolean types

KIT107 © JRD 2014 Slide 11

## 1.3 Reserved Words

- auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

KIT107 © JRD 2014 Slide 12

## 2. Example

2.1 Python Example
2.2 Java Example
2.3 C Example

KIT107 © JRD 2014 Slide 13

## 2.1 Python Example

```
# simple first program
print("Hello world")
```

KIT107 © JRD 2014 Slide 14

## 2.2 Java Example

```
import java.lang.*;

public class Example
{
	public static void main(String args[])
	{
		// simple first program
		System.out.println("Hello world");
	}
}
```

KIT107 © JRD 2014 Slide 15

## 2.3 C Example

return type
entry point
'import' statement

```
#include <stdio.h>

int main(int argc, char *argv[])
{
	// simple first program
	printf("Hello world\n");
}
```

parameter list
one-line comment
statement
block

KIT107 © JRD 2014 Slide 16

## 3. Program Structure



KIT107 © JRD 2014 Slide 17

## 3.1 Python Program Structure

- source files have the extension `.py`
- source files (*modules*) may contain *classes*
- each module can contain global variables, statements, and function definitions
- each class contains *instance variable* and *method* definitions

KIT107 © JRD 2014 Slide 18

## 3.1 Python Program Structure (continued)

- one file traditionally contains a function named `main()` which is the *entry-point* of the program
- user-defined methods/functions may be defined and *called*
- Methods/functions possess a *parameter list*
- Method parameter lists include *self*



## 3.1 Python Program Structure (continued)

- all parameter passing is *call-by-value* (but all parameters are objects)
- pre-compiled classes and/or modules may be *imported* and *linked*



## 3.1 Python Program Structure (continued)

- compilation is a two-stage process:
  - compilation proper
  - linking
- the compiler outputs Python *byte-code* (as a `.pyc` file)
- a runtime-environment is required to execute the byte-code

## 3.2 Java Program Structure

- source files have the extension `.java`
- programs contain *classes* and/or *interfaces*
- each class contains *instance variable* and *method* definitions
- each interface contains the heading of the public methods defined in the class

KIT107 © JRD 2014 Slide 22

## 3.2 Java Program Structure (continued)

- one class contains a method named `main()` which is the *entry-point* of the program
- user-defined methods may be defined and *invoked*
- methods possess a *parameter list*
- all parameter passing is *call-by-value*

KIT107 © JRD 2014 Slide 23

## 3.2 Java Program Structure (continued)

- collections of classes and/or interfaces may be compiled simultaneously and the compiler software can join these together (*link* them); or
- pre-compiled classes and/or interfaces may be *imported* and *linked*

KIT107 © JRD 2014 Slide 24

## 3.2 Java Program Structure (continued)

- compilation is a two-stage process:
  - compilation proper
  - linking
- the compiler outputs Java *byte-code* (either as a *class* — `.class` — or *Java archive* — `.jar` — file)
- a runtime-environment (*JVM*) is required to execute the byte-code

KIT107 © JRD 2014 Slide 25

## 3.3 C Program Structure

- source files have the extension `.c`
- programs contain *global variable* and *function* definitions
- function headings may be declared (these are usually declared in separate *header* — `.h` — *files* which are `#include`*d*)

KIT107 © JRD 2014 Slide 26

## 3.3 C Program Structure (continued)

- a function named `main()` must be defined which is the entry-point of the program
- user-defined functions may be defined and *called*
- functions possess a parameter list
- all parameter passing is call-by-value

KIT107 © JRD 2014 Slide 27

## 3.3 C Program Structure (continued)

- collections of files may be compiled simultaneously and *linked*; or
- pre-compiled (*object code*) — `.o` — files may simply be linked together

KIT107 © JRD 2014 Slide 28

## 3.3 C Program Structure (continued)

- compilation is a three-stage process:
  - pre-processing
  - compilation proper
  - linking
- the compiler outputs either *object code* or *machine code*

KIT107 © JRD 2014 Slide 29

## 3.3 C Program Structure (continued)

- compilation of a file without the `main()` function produces object code (which cannot be executed) and which must be linked with an executable file
- compilation of a file with the `main()` function produces a native machine code (or *binary code*) executable which is stand-alone

KIT107 © JRD 2014 Slide 30

## 3.4 Program Components

- *include files*
- definition of *new types*
- definition of *constants*
- definition of *global variables*
- definition of *user-defined functions*
- definition of the `main()` function

KIT107 © JRD 2014 Slide 31

## 3.5 `import` vs `#include`

- Java's import clause specifies classes to import
- an asterisk can be used to specify a package and imports all classes within the package, e.g.
  - `import java.awt.*;`
  - `import java.applet.Applet;`

KIT107 © JRD 2014 Slide 32

## 3.5 `import` vs `#include` (continued)

- Python's import clause specifies modules and/or classes to import
- an asterisk can be used to import all classes within a module, e.g.
  - `import math`
  - `from Tkinter import *`

KIT107 © JRD 2014 Slide 33

## 3.5 `import` vs `#include` (continued)

- C's include line specifies which header file to include
- header files may only include uncompiled code and usually contain *symbol* definininitions, type declarations, function declarations (headings), and sometimes constants

KIT107 © JRD 2014 Slide 34

## 3.5 `import` vs `#include` (continued)

- file inclusion is done by the pre-processor, e.g.
  - `#include <stdio.h>`
  - `#include "queue.h"`
- system header files are specified with angle brackets, user-defined (local) files are specified with double-quotes

KIT107 © JRD 2014 Slide 35

## 3.6 Libraries

- `<stdio.h>` — standard i/o facilities
- `<stdlib.h>` — memory allocation/ deallocation, type conversion, random number generation, and some system functions (e.g. `exit()`)
- `<stdbool.h>` — the C11 `bool` type and `false` (0) and `true` (1) literal values

KIT107 © JRD 2014 Slide 36

## 3.6 Libraries (continued)

- `<math.h>` — trigonometric and other mathematical functions
- `<ctype.h>` — character class tests (numeric, alphabetical, punctuation, white space, etc.)
- `<string.h>` — string functions (declaration, comparison, copying, concatenation, examination etc.)

KIT107 © JRD 2014 Slide 37

## 3.7 Header Files

- are the C equivalent of Java's interfaces
- existence is not necessary but good programming practice
- used to identify/advertise 'public' functions and constants
- have the same name as the program (`.c`) file but an extension of "`.h`"

KIT107 © JRD 2014 Slide 38

## 3.7 Header Files (continued)

- constants
  - constant variables may be defined inside or outside function definitions
  - they are defined using the `const` keyword (identical to Java's `final` keyword), e.g.
    `final double PI=3.1415926535;`

KIT107 © JRD 2014 Slide 39

## 3.7 Header Files (continued)

- symbols
  - *symbols* are compile-time definitions manipulated by the pre-processor e.g.

```
#define  TRUE     1
#define  FALSE    0
```

  - TRUE may now be used within the program as if it were a constant

KIT107 © JRD 2014 Slide 40

## 3.7 Header Files (continued)

  - each (non-quoted) symbol's name is textually replaced during pre-processing by its value, e.g.

```
if (TRUE)
```

becomes

```
if (1)
```

  - symbols may be defined without a value, e.g.

```
#define  SOLARIS
```

KIT107 © JRD 2014 Slide 41

## 3.7 Header Files (continued)

- conditional definition:
  - the existence of a symbol may then be checked:

```
#ifdef SOLARIS
#define JAVAC "/usr/local/bin/java/java1.3/javac"
#else
#define JAVAC "\\Program Files\sdk\java1.3\javac"
#endif
```

KIT107 © JRD 2014 Slide 42

## 3.7 Header Files (continued)

- `#ifndef` also exists enabling tests to see if a symbol is undefined e.g.

```
#ifndef  MYHEADER
#define  MYHEADER
… the rest of the header file…
#endif
```

KIT107 © JRD 2014 Slide 43

## 3.7 Header Files (continued)

- whitespace separates symbols from their value
- `#else` clauses may be omitted
- `#ifdef` and `#ifndef` constructs may be nested
- *macros* may also be defined, e.g.

```
#define sum(x,y) x+y
```

or functions may be `inlined`

KIT107 © JRD 2014 Slide 44

## 4. Types

4.1 Built-in Types
4.2 enum and Enumerated Types
4.3 Arrays
4.4 Pointers
4.5 Classes vs `structs`
4.6 `typedef` and Type Declarations
4.7 `unions`
4.8 Example

KIT107 © JRD 2014 Slide 45

## 4.1 Built-in Simple Types

- `char`
- `double`
- `enum`
- `float`
- `int`
- `long int`
- `long float`
- `short int`
- `void`
- (`bool`)

KIT107 © JRD 2014 Slide 46

## 4.2 `enum` and Enumerated Types

- the `enum` type allows the introduction of user-defined enumerated types, e.g.
  `typedef enum {FALSE,TRUE} boolean;`
- a new type is defined by listing (enumerating) all its values

KIT107 © JRD 2014 Slide 47

## 4.2 `enum` and Enumerated Types (continued)

- the first symbol declared receives `int` value `0`, the second `1`, (and so on)
- the above new type (`boolean`) can now be used as if it were a primitive (if C99/C11's `bool` from `<stdbool.h>` wasn't used…)

KIT107 © JRD 2014 Slide 48

## 4.3 Arrays

- consist of element-and-index pairs with a single name for the collection
- indices are contiguous non-negative `int` values
- all elements must be of the same type

KIT107 © JRD 2014 Slide 49

## 4.3 Arrays (continued)

- arrays are defined statically in C, e.g.
  `int x[15];`
  with a fixed length
- in Java and Python they are defined dynamically as objects
- C arrays don't know their length

KIT107 © JRD 2014 Slide 50

## 4.3 Arrays (continued)

- initialisations are also possible, e.g.
  `int a[]={10,20};`
- array use is similar to Java and Python, e.g. `a[1]=30;`
  but no slicing operators exist
- arrays are not objects in C, but the array variable is a pointer to the elements

KIT107 © JRD 2014 Slide 51

## 4.4 Pointers

- Java and Python possess *references*
- in Java, objects are created explicitly using new
- in Python, objects are created implicitly using =

KIT107 © JRD 2014 Slide 52

## 4.4 Pointers (continued)

- either way, the address (reference) of the object is assigned to a reference variable, e.g.

```
TextField x;
x=new TextField("hello");
```

or

```
x=str("a character string")
```

KIT107 © JRD 2014 Slide 53

## 4.4 Pointers (continued)

- the programmer has no access to the value of x
- x cannot be assigned a literal value (except `null` in Java or None in Python)
- arithmetic operations cannot be applied to x e.g. x+1 is not permitted

KIT107 © JRD 2014 Slide 54

## 4.4 Pointers (continued)

- all of these things are available in C, the resulting type is called a *pointer*
- pointer arithmetic often leads to runtime errors when the program attempts to access a part of memory which is used by another application

KIT107 © JRD 2014 Slide 55

## 4.4 Pointers (continued)

- in Java and Python, reference variables are defined automatically if the variable's type is a class, e.g.
  `i=7`
- in C, pointer variables must be explicitly defined, e.g.
  `int *ip;`

KIT107 © JRD 2014 Slide 56

## 4.4 Pointers (continued)

- in Java and Python, reference variables are dereferenced automatically if the variable's type is a class, e.g.
  `j=i`
- in C, pointer variables must be explicitly dereferenced, e.g.
  `j=*ip;`

KIT107 © JRD 2014 Slide 57

## 4.4 Pointers (continued)

- in Java and Python, you cannot find out what address a variable is stored at
- in C, you can do this by asking for the address of a variable with the & operator, e.g.

```
double d=13.7;
double *dp=&d;
```

KIT107 © JRD 2014 Slide 58

## 4.5 Classes vs structs

- a Java and Python encapsulates state and behaviour (instance variables and methods)
- C is not object-oriented and doesn't possess this idea

KIT107 © JRD 2014 Slide 59

## 4.5 Classes vs structs (continued)

- in C, fields may be collected together into a structure (`struct`) but there is no mechanism to encapsulate properties and methods together

KIT107 © JRD 2014 Slide 60

## 4.5 Classes vs `structs` (continued)

- `struct` introduces a new type e.g.

```
typedef struct {
    int hour;
    int minute;
    int second;
} time;
```

KIT107 © JRD 2014 Slide 61

## 4.5 Classes vs `structs` (continued)

- the components of a `struct` are called *fields*
- fields can be accessed, as in Java and Python, using the `.` operator, e.g.

```
time x;
…
x.hour=12;
```

KIT107 © JRD 2014 Slide 62

## 4.6 `typedef` and Type Definitions

- `typedef` has been used in the previous examples
- `typedef` is the easiest way to introduce new types
- the syntax is:

  `typedef` ***`type name`***`;`

  e.g.

```
typedef char *string;
```

KIT107 © JRD 2014 Slide 63

## 4.7 `union`s

- `union`s are similar to `struct`s — they possess fields
- unlike `struct`s, a variable of `union` type can only possess one of the declared fields at a time: it is an *or* relationship rather than an *and* relationship



## 4.7 `union`s (continued)

- the user is responsible for ensuring the correct values are in the fields — no run-time checks exist
- the total size of the `union` in memory is the size of the largest field



## 4.8 Type Example

```
enum item_kind {BOOK, VIDEO};
enum ratings {G, PG, M, MA, MAV,
              R};

typedef char *string;
```

## 4.8 Example (continued)

```
typedef struct {
    long int isbn;
    string name;
    string publisher;
    int year;
} book;
```

KIT107 © JRD 2014 Slide 67

## 4.8 Example (continued)

- ```
  typedef struct {
    string title;
    ratings rating;
    string studio;
    short int length;
  } video;
  ```

KIT107 © JRD 2014 Slide 68

## 4.8 Example (continued)

```
typedef struct {
    item_kind kind;
    union {
        book book_details;
        video video_details;
    } details;
} items;
```

KIT107 © JRD 2014 Slide 69
```