KIT107 PROGRAMMING

C Programming

Dr Julian Dermoudy
& Dr Shuxiang Xu
School of ICT

1

Structure

- approximately 8 lectures (3 weeks)

KIT107 ©JRD, 2021 Slide 2

2

KIT107 — C Programming

KIT107 ©JRD, 2021 Slide 3

3

Preface

- Released in 1972
- tied to the Unix operating system
- not a good first language (terse)
- very good for experienced programmers
- contains pointers!!

KIT107 ©JRD, 2021 Slide 4

4

0. Anatomy of Programming Languages

- Types, Variables, Literal Values
- Operators
- Control Structures
- Input/Output mechanisms
- Data Structures
- Abstraction (functions)
- Entry point
- Libraries

KIT107 ©JRD, 2021 Slide 5

5

1. Example

- 1.1 Python Example
- 1.2 Java Example
- 1.3 C Example

KIT107 ©JRD, 2021

6

1.1 Python Example

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

KIT107 ©JRD, 2021 Slide 7

7

1.2 Java Example

```
import java.lang.*;

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

KIT107 ©JRD, 2021 Slide 8

8

1.3 C Example

return type entry point 'import' statement

parameter list

```
#include <stdio.h>

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

one-line comment

statement

block

KIT107 ©JRD, 2021 Slide 9

9

2. Introduction

- 2.1 C and Java/Python Similarities
- 2.2 C and Java/Python Differences
- 2.3 Reserved Words

KIT107 ©JRD, 2021

10

2.1 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

KIT107 ©JRD, 2021 Slide 11

11

2.1 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 variable `myVar` would be `my_var` in C*

KIT107 ©JRD, 2021 Slide 12

* *I don't have the room for meaningful identifiers. I don't. So stop complaining.*

12

2.1 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, 2021 Slide 13

13

2.1 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, 2021 Slide 14

14

2.2 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, 2021 Slide 15

15

2.2 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, 2021 Slide 16

16

2.2 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, 2021 Slide 17

17

2.3 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, 2021 Slide 18

18

3. Program Structure

- 3.1 Python Program Structure
- 3.2 Java Program Structure
- 3.3 C Program Structure
- 3.4 Program Components
- 3.5 `import` vs `#include`
- 3.6 Libraries
- 3.7 Header files

KIT107 ©JRD, 2021

19

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, 2021 Slide 20

20

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*

KIT107 ©JRD, 2021 Slide 21

21

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*

KIT107 ©JRD, 2021 Slide 22

22

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

KIT107 ©JRD, 2021 Slide 23

23

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, 2021 Slide 24

24

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, 2021 Slide 25

25

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, 2021 Slide 26

26

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, 2021 Slide 27

27

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, 2021 Slide 28

28

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, 2021 Slide 29

29

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, 2021 Slide 30

30

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, 2021 Slide 31

31

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, 2021 Slide 32

32

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, 2021 Slide 33

33

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, 2021 Slide 34

34

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, 2021 Slide 35

35

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* definitions, type declarations, function declarations (headings), and sometimes constants

KIT107 ©JRD, 2021 Slide 36

36

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, 2021 Slide 37

37

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, 2021 Slide 38

38

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, 2021 Slide 39

39

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, 2021 Slide 40

40

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.

```
const double PI=3.1415926535;
```

KIT107 ©JRD, 2021 Slide 41

41

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, 2021 Slide 42

42

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  UNIX
```

KIT107 ©JRD, 2021 Slide 43

43

3.7 Header Files (continued)

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

```
#ifdef UNIX
#define JAVAC "/usr/local/bin/java/java1.8/javac"
#else
#define JAVAC "\\Program Files\sdk\java1.8\javac"
#endif
```

KIT107 ©JRD, 2021 Slide 44

44

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, 2021 Slide 45

45

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, 2021 Slide 46

46

4. Types

KIT107 ©JRD, 2021

47

4.1 Built-in Simple Types

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

KIT107 ©JRD, 2021 Slide 48

48

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, 2021 Slide 49

49

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 post C89's `bool` from `<stdbool.h>` wasn't used…)

KIT107 ©JRD, 2021 Slide 50

50

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, 2021 Slide 51

51

4.3 Arrays (continued)

- arrays are defined with a fixed length
  - statically in C (i.e. at compile time), e.g. `int x[15];`
  - dynamically from C11, e.g. (with an `int` variable n already defined) `int x[n];`
  - in Java and Python they are defined dynamically as objects
- C arrays don't know their length

KIT107 ©JRD, 2021 Slide 52

52

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, 2021 Slide 53

53

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, 2021 Slide 54

54

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, 2021 Slide 55

55

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, 2021 Slide 56

56

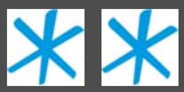
4.4 Pointers (continued)

- 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, 2021 Slide 57

57

4.4 Pointers (continued)



asterisks

asterixes

KIT107 ©JRD, 2021 Slide 58

58

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, 2021 Slide 59

59

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;
dp=&d;
```

KIT107 ©JRD, 2021 Slide 60

60

4.5 Classes vs structs

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

KIT107 ©JRD, 2021 Slide 61

61

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, 2021 Slide 62

62

4.5 Classes vs structs (continued)

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

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

KIT107 ©JRD, 2021 Slide 63

63

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, 2021 Slide 64

64

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, 2021 Slide 65

65

4.7 `unions`

- unions are similar to `structs` — they possess fields
- unlike `structs`, 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

KIT107 ©JRD, 2021 Slide 66

66

4.7 **unions** (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

KIT107 ©JRD, 2021 Slide 67

67

4.8 Type Example

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

typedef char *string;
```

KIT107 ©JRD, 2021 Slide 68

68

4.8 Example (continued)

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

KIT107 ©JRD, 2021 Slide 69

69

4.8 Example (continued)

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

KIT107 ©JRD, 2021 Slide 70

70

4.8 Example (continued)

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

KIT107 ©JRD, 2021 Slide 71

71

5. Variables

- 5.1 Default Initialisation
- 5.2 Declaration Syntax
- 5.3 Dynamic Variables
- 5.4 Arrays
- 5.5 Global Variables
- 5.6 Local Variables
- 5.7 Constants
- 5.8 Linked Lists
- 5.9 Strings
- 5.10 String Operations

KIT107 ©JRD, 2021

72

5.1 Default Initialisation

- variables must be declared before being used
- variables are strongly typed
- variables may be initialised at declaration
- no default initialisation provided

KIT107 ©JRD, 2021 Slide 73

73

5.2 Declaration Syntax

- variable declarations are syntactically as in Java
- variables can be declared:
  - following `#include` lines
  - within function parameter lists
  - anywhere within a block (post C89) — but good style is to place them immediately after a function header*

KIT107 ©JRD, 2021 Slide 74

* *and you will lose marks if you don't!*

74

5.2 Declaration Syntax (continued)

- Examples*:

```
int x=57;
double a,b;
char s[34];
int *p;
```

* *poor ones admittedly; see previous comments about me not doing it but you will lose marks if you don't!*

KIT107 ©JRD, 2021 Slide 75

75

5.2 Declaration Syntax (continued)

- the declaration of a pointer variable reserves space only for the pointer
- a pointer variable may hold any address value
- if a pointer variable is to not point anywhere, it may be given the symbolic address value `NULL`

KIT107 ©JRD, 2021 Slide 76

76

5.3 Dynamic Variables

- example: setting a pointer variable to `NULL`
```
p=NULL;
```
- example: reserving memory for an integer and pointing p at the new location
```
p=(int *) malloc(sizeof(int));
```

KIT107 ©JRD, 2021 Slide 77

77

5.3 Dynamic Variables (continued)

- `malloc()` is the C equivalent of `new` in Java
- `malloc()` is from `<stdlib.h>`
- `malloc()` returns values of type `(void *)`
- `malloc()` returns `NULL` if insufficient memory is available

KIT107 ©JRD, 2021 Slide 78

78

5.3 Dynamic Variables (continued)

- pointers must be explicitly followed (*deferenced*)
- pointer dereferencing is done using the asterisk (*), e.g.

```
*p=5;
x=*p+1;
```

- dereferencing a `NULL` pointer results in a run-time error (as in Java and Python)

KIT107 ©JRD, 2021 Slide 79

79

5.3 Dynamic Variables (continued)

- as in Java, there is no need to reserve space for pointer variables if none is needed, e.g. in Java

```
String s1;
String s2;
s1=new String("hello");
s2=s1;
```

- since s2 refers to the same object as s1, s2 doesn't require instantiation

KIT107 ©JRD, 2021 Slide 80

80

5.3 Dynamic Variables (continued)

- e.g. in C

```
char *c;
char *k;
c=(char *) malloc(sizeof(char));
*c='x';
k=c;
```

A char, not a string

- since k points to the same memory as c, space didn't need to be 'malloced' for k

KIT107 ©JRD, 2021 Slide 81

81

5.3 Dynamic Variables (continued)

- Java and Python possess a *garbage collector* to remove unreachable objects
- the programmer must reclaim memory explicitly in C using `free()` from `<stdlib.h>`, e.g.
  `free(c);`
- from the previous example, k is now a *dangling pointer*

KIT107 ©JRD, 2021 Slide 82

82

5.3 Dynamic Variables (continued)

- consider the following fragment:
  `(*a).b`
- there is a 'short-cut' for accessing fields from dereferenced pointers:
  `a->b`

KIT107 ©JRD, 2021 Slide 83

83

5.3 Dynamic Variables (continued)

- Example:

```
book *b;
b=(book *) malloc(sizeof(book));
```

- the following:
  `(*b).isbn=0131103628;`
- is equivalent to:
  `b->isbn=0131103628;`

KIT107 ©JRD, 2021 Slide 84

84

5.3 Dynamic Variables (continued)

- dereferencing is only required with pointer variables e.g.
```
book z;
z.isbn=0131103628;
```

KIT107 ©JRD, 2021 Slide 85

85

5.3 Dynamic Variables (continued)

- pointers and non-pointers may be mixed, e.g.
```
char c, *cp;
c='X';
cp=&c;
```
- cp now points to the memory location containing c
- & is the 'address of' operator

KIT107 ©JRD, 2021 Slide 86

86

5.3 Dynamic Variables (continued)

- `*` and `&` are now inverses
- ```
  c='X';
  cp=&c;
  ```
- ```
  *cp='A';
  c='A';
  ```
- `&(*cp)` is `cp`
- `*(&c)` is `c`

KIT107 ©JRD, 2021 Slide 87

87

5.4 Arrays

- (generally) declared statically only
- indexed from 0 upwards
- implemented using pointers
  i.e. `a[3]` and `*(a+3)` are synonymous
- can be created dynamically, e.g.

```
int *a;
a=(int *) malloc(10*sizeof(int));
```

Size of array

KIT107 ©JRD, 2021 Slide 88

88

5.5 Global Variables

- global variables are defined outside all functions
- such variables are in scope in all functions
- if multiple files are used, the variable can only be defined in one — it must be *externally defined* in all others, e.g.
  `extern int k;`

KIT107 ©JRD, 2021 Slide 89

89

5.6 Local Variables

- local variables may be defined and used as in Java, i.e. within parameter lists, as a loop counter, or at the start of functions*

** and you will lose marks if you don't!*

KIT107 ©JRD, 2021 Slide 90

90

5.7 Constants

- as in Java (with `const` replacing `final`), e.g.

```
const int PAGE=12;
const double PI=3.1415;
const char NAME[]="C Program";
const char X='X';
```

KIT107 ©JRD, 2021 Slide 91

91

5.8 Linked Lists

- (Introduced in Data Structures & Algorithms component of the course)

KIT107 ©JRD, 2021 Slide 92

92

5.9 Strings

- "String" is not a C type
- a string of characters is represented by a pointer to a block of memory containing `char` values
- (an array can be used instead)
- a sentinel *null* character (`'\0'`) — not `NULL` — indicates the end of the string

KIT107 ©JRD, 2021 Slide 93

93

5.10 String Operations

- some string routines are defined in `<string.h>`:
  - `strdup()` for copying strings
  - `strcat()` for joining strings
  - `strcmp()` for comparing strings
  - `strlen()` for finding the length of a string

KIT107 ©JRD, 2021 Slide 94

94

6. Input/Output

- 6.1 Introduction
- 6.2 Input
- 6.3 Output

KIT107 ©JRD, 2021

95

6.1 Introduction

- input/output is not a part of the language
- input/output routines are located in `<stdio.h>`

KIT107 ©JRD, 2021 Slide 96

96

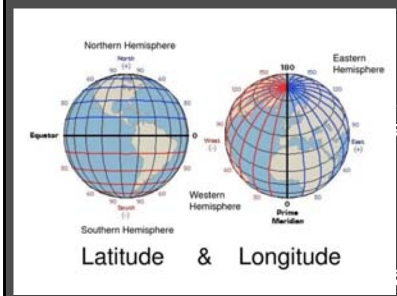
6.2 Input

- Simple input:
  - `char *gets()`
  - `char getchar()`

  - `atoi()`
  - `atof()`
  - (both from `<stdlib.h>`)

97

6.2 Input (continued)



…, *var*) or
ar, …, *var*)

- Hobart:
  - 42.8821° S
  - 147.3272° E

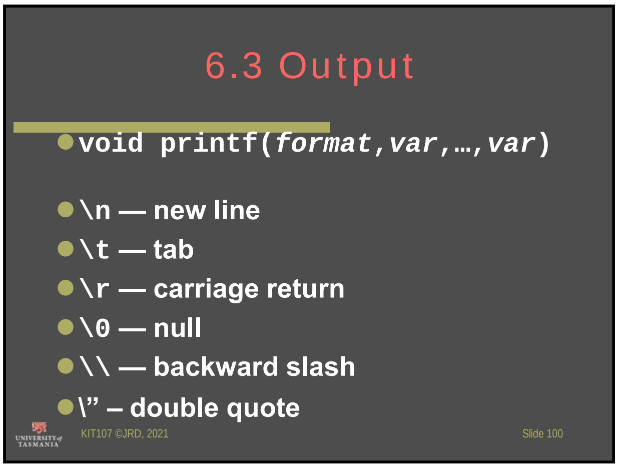
&x,&a,&y,&b);

98

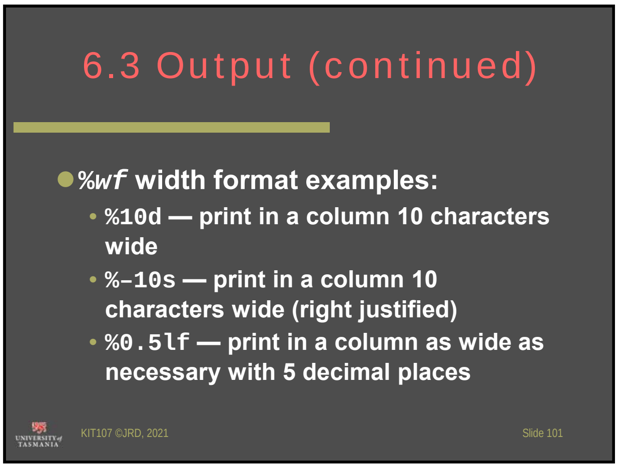
6.2 Input (continued)

- format strings:
  - `%d` — integer (or `%i`)
  - `%c` — character
  - `%s` — string
  - `%f` — float
  - `%lf` — double (long float)
  - `%p` — address
  - `%%` — percentage sign

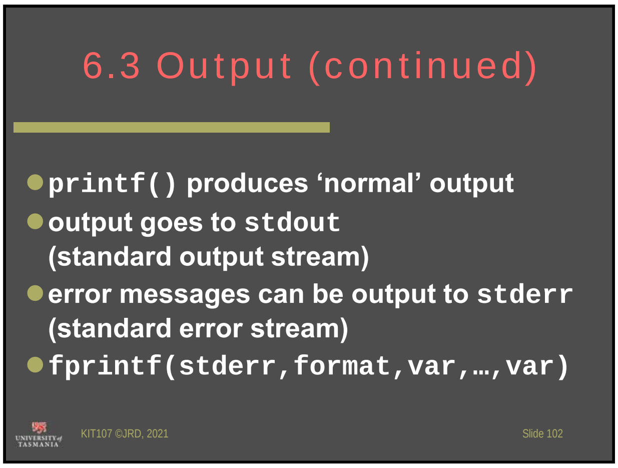
99


6.3 Output
void printf(format,var,…,var)
\n — new line
\t — tab
\r — carriage return
\0 — null
\\ — backward slash
\" – double quote
KIT107 ©JRD, 2021
Slide 100

100


6.3 Output (continued)
%wf width format examples:
%10d — print in a column 10 characters wide
%-10s — print in a column 10 characters wide (right justified)
%0.5lf — print in a column as wide as necessary with 5 decimal places
KIT107 ©JRD, 2021
Slide 101

101


6.3 Output (continued)
printf() produces 'normal' output
output goes to stdout (standard output stream)
error messages can be output to stderr (standard error stream)
fprintf(stderr,format,var,…,var)
KIT107 ©JRD, 2021
Slide 102

102

7. Operators

- 7.1 Arithmetic
- 7.2 Logical
- 7.3 Comparison
- 7.4 Assignment
- 7.5 Addresses etc.
- 7.6 Precedence

KIT107 ©JRD, 2021

103

7.1 Arithmetic Operators

- + — addition
- - — subtraction
- * — multiplication
- / — division
- % — modulus (remainder)

KIT107 ©JRD, 2021 Slide 104

104

7.2 Logical Operators

- && — and
- || — or
- ! — not

KIT107 ©JRD, 2021 Slide 105

105

7.3 Comparison Operators

- == — equals
- != — not equals
- < — less than
- <= — less than or equal to
- > — greater than
- >= — greater than or equal to

KIT107 ©JRD, 2021 Slide 106

106

7.4 Assignment

- = — read as “becomes” (no type error)
- +=, -=, *=, /=, %= — arithmetic assignment
- `var=(cond)?value:value` — conditional assignment
- ++, -- — increment and decrement assignment

KIT107 ©JRD, 2021 Slide 107

107

7.5 Address Operators etc.

- & — address operator
- * — dereference operator
- `[index]` — array index operator
- . — field operator
- -> equivalent to `(*var).field`

KIT107 ©JRD, 2021 Slide 108

108

7.6 Operator Precedence

- `() [] -> .`
- `! ++ -- + - * &`
  `(type) sizeof()`
- `* / %`
- `+ -`
- `< <= > >=`
- `== !=`
- `&&`
- `||`
- `?:`
- `= += -= *= /=`
  `%=`

KIT107 ©JRD, 2021 Slide 109

109

8. Control Structures

- 8.1 Statements
- 8.2 Conditions
- 8.3 Iteration
- 8.4 Comments

KIT107 ©JRD, 2021

110

8.1 Statements

- single statements must be followed by a semi-colon (`;`)
- multiple (compound) statements must be grouped into a block by braces (`{}`)

KIT107 ©JRD, 2021 Slide 111

111

8.2 Conditions

- conditions are implemented as integers:
  - 'false' is 0
  - 'true' is non-zero

KIT107 ©JRD, 2021 Slide 112

112

8.2 Conditions (continued)

```
if (condition) {
    statement
    …
}
else {
    statement
    …
}
```

- the 'else-clause' is optional

KIT107 ©JRD, 2021 Slide 113

113

8.2 Conditions (continued)

```
switch (expr) {
    case value: statement;
                …
                statement;
                break;
    …
    default:statement;
    …
}
```

KIT107 ©JRD, 2021 Slide 114

114

8.3 Iteration

```
while (condition)    do
{                    {
    statement            statement
    …                    …
}                    } while (condition);
```

KIT107 ©JRD, 2021 Slide 115

115

8.3 Iteration (continued)

```
for (init; condition; increment)
{
    statement
    …
}
```

KIT107 ©JRD, 2021 Slide 116

116

8.4 Comments

```
// single line comment(post C89)

/* multiple
    line
    comment */
```

KIT107 ©JRD, 2021 Slide 117

117

9. Functions

- 9.1 Function Form
- 9.2 Functions vs Methods
- 9.3 Parameter Passing
- 9.4 `main()`

KIT107 ©JRD, 2021

118

9.1 Function Form

```
int max(int x, int y)
{
	int r;

	if (x>y) {
		r=x;
	}
	else {
		r=y;
	}

	return r;
}
```

119

9.2 Functions vs Methods

- similar to Java and (less so to) Python syntax
- methods are *functions*
- method invocation becomes *function calling*
- must be declared before being used (called)
- no access modifiers

KIT107 ©JRD, 2021 Slide 120

120

9.2 Functions vs Methods (continued)

- definitions cannot be nested
- may be recursive
- may be passed as parameters
- cannot be called on a variable, variables can only be passed as parameters

KIT107 ©JRD, 2021 Slide 121

121

9.3 Parameter Passing

- parameter passing is strictly *call-by-value*:
  - the value of the actual parameter is calculated
  - a copy of this value is assigned to a newly created local variable (the formal parameter)
  - the function executes
  - the formal parameter is deleted

KIT107 ©JRD, 2021 Slide 122

122

9.3 Parameter Passing (continued)

```
void swap(int a, int b)
{
      int temp;

      temp=a;
      a=b;
      b=temp;
}

…
int x,y;
x=45; y=92;
swap(x,y);
```

123

9.3 Parameter Passing (continued)

- passing values out of a function using the parameter list can be achieved in C through the use of addresses
- declaring a parameter to be a pointer to the value required will achieve the desired result (at the cost of increased complexity)

KIT107 ©JRD, 2021 Slide 124

124

9.3 Parameter Passing (continued)

```
void swap(int *a, int *b)
{
     int temp;

     temp=*a;
     *a=*b;
     *b=temp;
}
```

KIT107 ©JRD, 2021 Slide 125

125

9.3 Parameter Passing (continued)

- to use the function, two pointer-to-integer (`int *`) values must be provided
- such values can be provided by:
  - passing an (`int *`) variable; or
  - passing the address of an `int` variable

KIT107 ©JRD, 2021 Slide 126

126

9.3 Parameter Passing (continued)

```
…
int *p,*q;
p=(int *) malloc(sizeof(int));
q=(int *) malloc(sizeof(int));
*p=45; *q=92;
swap(p,q);
```

or

```
…
int x,y;
x=45; y=92;
swap(&x,&y);
```

KIT107 ©JRD, 2021 Slide 127

127

9.4 `main()`

`int main(int argc, char *argv[]);`

- `main()` has two parameters:
  - a count of the command-line arguments (`argc`)
  - an array of the command-line arguments (`argv`)
- `main()`'s return type may be omitted (as may the parameters)

KIT107 ©JRD, 2021 Slide 128

128