RMIT UNIVERSITY Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Introduction

Course Introduction

About us:

- Ken Gardiner
  - Lecturer of this course
  - ken.gardiner@rmit.edu.au (but I reply **faster** to posts in the Canvas discussion board as email quickly gets buried under RMIT management emails)
  - Consultation time: Mondays TBA

Tutors:

- Dr. Shaahin Madani
- Matthew Bolger
- Dale Stanbrough

Course Introduction

General aims:

- introduce modular programming techniques using ANSI/ISO C
- to highlight a range of programming principles and techniques
- introduce dynamic memory allocation in C for the creation and manipulation of dynamic data structures.

- Programming requires both knowledge and experience. Practice is essential!

Assumed knowledge

- This course assumes you have done 1 to 2 semesters of programming in Java or another programming language.
- We **assume** you are familiar with:
  - variables
  - IF statements (including nested IF statements)
  - loops (e.g. WHILE, FOR, DO) including nested loops
  - arrays – 1D and 2D
  - functions (i.e. ‘methods’)
  - parameters (at least some familiarity of parameters)
- If you can’t use these concepts already, you are in the **wrong course**!

Main topics covered

- Portability
- Problem Solving
- Functional Abstraction
- Data Abstraction
- Pointer Types
- Debugging Techniques
- Managing C Programs
- Module Abstraction
- File Processing
- Program Readability
- Program Usability
- Generics
- Code Reusability
- Memory Management
- Dynamic Data Structures

Assessment for this course

- **Two programming assignments** to be completed individually and submitted for assessment throughout the semester/study period – on weeks 6 and 12. (20% + 30% = 50%)
- **One formal written examination** at the end of the semester/study period. (50%)
- There are no hurdles for this course.
- The recommended study schedule is presented in Canvas.

Advanced Programming Techniques

Module 01 – Getting Started

“If it wasn't for C, we'd be using BASI, PASAL and OBOL”

-- anon.

A Brief History of C

- Designed by Dennis Ritchie of Bell Laboratories.
- Implemented on PDP-11 in 1972, as a systems programming language for the UNIX operating system.
- Evolved from B and BCPL
  – BCPL = type-less PL developed by Martin Richards (Camb., ’67).
  – B was based on BCPL and was written by Ken Thompson in 1970 for the first UNIX system on a PDP-7.
- C was formed with low-level programming + structured programming issues in mind.

A Brief History of C

- C is a 3GL, permitting machine-oriented and problem-oriented solutions.
- Close to underlying hardware allowing detailed program control, yet supports advanced program and data structuring.
- Supports modular programming through use of storage-classes and functions within separate translation units.

Characteristics of C

- C revised by ANSI (X3J11) to form new standard (1989).
- Internationalised and became ANSI/ISO standard in 1990.
- "C99" standard finalised in 1999, and "C11" finalised in 2011
  - C99 was not well received by industry
- C11 has been better received – but not ubiquitous yet

Hello World

- First write this C program in a file named hello.c

```
#include <stdio.h>

int main(void)
{
   printf("Hello World\n");

   return 0;
}
```

- Next compile and run the program on a UNIX system as,

`gcc hello.c`	if the compilation was successful an executable named `a.out` will be created in the current directory
`./a.out`	to execute the program

Hello World (cont'd)

- If you want the executable to be called as `hello` instead of `a.out`,

```
gcc hello.c -o hello
```

- Here `-o` is a "flag" we pass to the C compiler `gcc` to indicate some options in the compilation process.
- We will look at more such options later in the lab and tute sessions.
- We will also look at a tool called `make` that can be used to simplify the compilation process later in the course.
- Your code is expected to compile without errors or warnings using the flags:

```
-ansi –pedantic -Wall
```

Hello World

```
#include <stdio.h>
```

- The #include ‘says’ the standard I/O (input/output) function library will be used. Function *printf* is an output function.
- You can include other files too. Often a *.c file will have a matching ‘header’ file *.h containing definitions used in the *.c file.
- To include such a file use “ ” and give the file name:

```
#include “hello.h”
```

Code comments in C

- A comment in ANSI C is as follows:

```
/* this is a comment */
```

- The // notation is **not** ANSI C compliant

```
// don't do this, you'll be marked down for it!
```

Revision: Variables, Types and Values

- A variable is a <u>named</u> memory location where a <u>value</u> can be stored
- Each variable has a <u>type</u> which defines the range of legal values for that variable
- The major pre-defined data types in C are:

int **char** **float** **double**

Variables, Types and Values

int	Integer values
	`3   -6   077   0x3f   3U`
char	Character (typically ASCII) values
	`'A'   'z'   '\n'   '\004'   '\0'`
float	Real (floating point) numbers
	`1.0F   273.33f   -1.1F   1.1e-02f`
double	`1.0   273.33   -1.1   1.1e-02`

I/O: printf and scanf

- Function *printf* is used for output, and is part of the C standard I/O (input/output) library.
  - A program using these functions must include the appropriate library header: *stdio.h*
- Function printf takes a ‘control string’ which may be just a string literal:

```
#include <stdio.h>
int main(void)
{
        printf("Hello World!\n");
        return 0;
}
```

I/O: printf and scanf

- The ‘control string’ may also contain ‘conversion specifications’ (indicated by a %), which specify how to print out a value (which must also be given to the function)
- E.g. ‘%d’ specifies ‘print as a decimal integer’

```
#include <stdio.h>
int main(void)
{
        int age = 20;
        printf(“I am %d today\n”,age);
        return 0;
}
```

I/O: printf and scanf

Conversion character	Prints as:
c	character
d	decimal integer
e	scientific floating-point number format
f	floating point
s	string

- We'll cover these in more detail later...

I/O: printf and scanf

- Function *scanf* is used for input
- The control string specifies how to interpret the input
- The function takes the address of a variable (specified by &) to read the result into

```
#include <stdio.h>
int main(void)
{
        int age;
        printf("How old are you?:");
        scanf("%d",&age);
        printf("You said you were %d years old\n",age);
        return 0;
}
```

Type limits and `sizeof`

- C does not specify max/min values or the storage space allocation for any of its types.
- The operator `sizeof` evaluates the number of bytes allocated for an object or type.
- Type `char` is always 1 byte (which is assumed to be 8 bits).

Type limits and `sizeof`

- Nor does C specify the internal representation for data values

  e.g. how negative numbers are stored, how real numbers are stored.

- Limits and internal representations are implementation dependent.

- Standard library headers `<limits.h>` and `<float.h>` have details of these.

Type qualifiers

C also has the following data types:

```
{signed | unsigned} {long | short} int
{signed | unsigned} char
{long} double
```

The actual effect of these is compiler dependent

- a `long int` <u>might</u> be larger (more bits) than an `int` but this is not guaranteed (it is guaranteed that it won't be smaller)

Program: to display size of the data types

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i;
    char c;
    float f;
    double d;

    printf(“Size of int = %lu bytes\n”, (unsigned long)sizeof(i));
    /* using the datatype like the following line is OK,
       but not recommended for most circumstances */
    printf(“Size of int = %lu bytes\n”, (unsigned long)sizeof(int));
    printf(“Size of char = %lu bytes\n”, (unsigned long)sizeof(c));
    printf(“Size of float = %lu bytes\n”, (unsigned long)sizeof{f));
    printf(“Size of double = %lu bytes\n”, (unsigned long)sizeof(d));

    return EXIT_SUCCESS;
}
```

Expressions and operators

- Each type has a set of <u>operators</u> for the legal operations that can be performed in <u>expressions</u> with those types:

integer arithmetic:	+ - * / % ++ --
real arithmetic:	+ - * /
relational:	== > < ! || && !=
assignment:	=
bit manipulation:	~ ^ | & << >>

NB: this is **not** a complete list of all the C operators

Expressions and operators (cont’d)

- Strict precedence (and associatively) rules are defined for all operators

- These determine which <u>operands</u> apply to which <u>operators</u> (not the order of evaluation – see later).

- C is not a strongly typed language, i.e. you can mix types in the one expression

  e.g. charVar = intVar + 2;

  potential pitfall – be careful – if in doubt, avoid it!

Expressions and operators (cont'd)

- Use casts to force explicit type conversions (and remove compiler warnings!)

```
float x;
int y = 5, z = 10;
x = y / z;                      /*  problem!  */
x = (float) y / (float) z; /*  solution  */
```

NB: strictly speaking, **only one** cast is required, which will "promote" the expression being evaluated to that type

Boolean data type

- Prior to C99, C had no type “Boolean”, instead using integer values:

  zero is interpreted as Boolean FALSE

  non-zero is interpreted as Boolean TRUE

- So the Boolean expression:

  (A==B)

  will evaluate to 0 if A and B are not equal

  otherwise to some non-zero value (typically 1) if they are equal

- C99 introduced type _Bool, with 0 for false and 1 for true

  <stdbool.h> has macros **bool**, **true** and **false**

The ++ and -- operators

- By itself: `x++;` `/* or ++x */` is equivalent to: `x = x + 1;`
- But in expressions with more than one operator:

	post-increment	**pre**-increment
	`m = 10;` `x = m++;`	`m = 10;` `x = ++m;`
yields:	`x == 10`	`x == 11`

(the same applies for the pre- and post- decrement operator **--**)

Beware of Side-Effects

- Unexpected and unwanted side-effects can occur when you have both:
  - multiple assignments in the one statement and
  - the same variable used more than once

Consider the following:

```
x = (y = 6) - (s = 3);          /*  OK       */
x = (y *= y) - (s -= y);        /* yikes     */
x = y++ + ++y;        /* legal but yikes2  */
```

RMIT UNIVERSITY Computer Science and Information Technology

Order of evaluation

- C does not always specify the order of evaluation of sub-expressions within a statement - e.g. given:

```
x = (y *= y) - (s -= y);
```

  the compiler could generate code to:

  – first evaluate the sub-expression `(y *= y)`
  – and then evaluate the sub-expression `(s -= y)`
  – or vice versa - which would yield a different result!!!

- **Implementation dependant, undefined, unpredictable ==> don't do it!!**

User defined types: enum

- *enum* is a keyword used to define an enumeration – i.e. a finite set of named integer constants.

```
enum day {sun, mon, tue, wed, thu, fri, sat};

enum day day1, day2;
day1 = mon;
if (day1 == day2) …
```

User defined types: enum

- By default, the value of the first enumerator is 0 (but that can be changed) and each subsequent set member increments by 1 from the previous one.

```
enum day { sun = 2, mon, tue, wed, thu, fri, sat};
```

  - sun is 2, mon is 3, tue is 4 etc.

- Enumerated types can improve the readability of code, as they self-document (*sun*, *mon* etc helps describe their purpose, rather than using arbitrary numbers)

typedef

- The *typedef* keyword is often used with enumerated types
  - It defines a new name for an existing data type
  - New type name may improve readability (by describing the purpose of the type), or may remove cumbersome syntax (such as having to say ‘enum day’ to define a variable of enumerated type ‘day’)

```
typedef enum day {sun, mon, tue, wed, thu, fri,
sat} Day;

Day day1, day2;
```

Sequence, selection and iteration

- Only 3 types of statements control the flow of execution of a program
  - *Sequence* – stepping through instructions
    - Do this, then that, then ...
  - *Selection* – branching
    - Do this **or** that depending on some condition
  - *Iteration* – looping
    - Do this while/until some condition is true

- Recursive functions definitions also supported; recursion can be considered a 4th construct / flow control mechanism.

Compound statements and blocks

- A block in C is one or more statements delimited by a {...}
- Blocks are used to define:
  - the start and end of a function
  - compound statements (see if statement for an example)
  - scope rules of variables

Boolean expressions

- An <u>expression</u> which evaluates to either

  FALSE – integer 0 in C

  TRUE – any non-zero integer value in C

- Used to control decisions in branching (selection) and loops (iteration)

Selection – if statement

<if-else-statement> ::=

if (<condition-expr>) <statement>

[else < statement >]

where:

- <condition-expr> is any expression
- <statement> can be a compound statement
- else part is optional

Selection – if statement examples

```
if ( x > 10 )
    a = 2;


if ( x > 10 )
    a = 2;
else
     b = 3;
```

```
if ( x > 10 && y < 20 )
{
    a = 2;
    b = 3;
}
else
{
    d = 4;
    e = 5;
}
```

Selection – if statement examples (cont'd)

Nested if statements:

```
if ( x > 10 )
   if ( y < 20 )
      c = 3;
   else
      d = 4;
else
   e = 3;
```

With compound statements:

```
if ( x > 10 )
{
   a = 2;
   if ( y < 20 )
      c = 3;
   else
      d = 4;
}
else
{
   e = 5;
   f = 6;
}
```

Selection – if statement examples (cont'd)

The dangling else problem:

```
if ( x > 10 )
   if ( y < 20 )
      c = 3;
else
   e = 3;
```

… and the solution:

```
if ( x > 10 )
{
   if ( y < 20 )
      c = 3;
}
else
   e = 3;
```

Selection – else if

```
if (ch == 'a')
{
   aCount++;
}
else if (ch == 'b')
{
   bCount++;
}
else if (ch == 'c')
{
   cCount++;
}
else
   other++;
```

Are all these brackets {…} really necessary?

No, but "defensive programming" says that it's a good idea!

NB. Counting frequency of occurrence of alphabetic characters can be written more efficiently with
an array of counters, and a simple arithmetic mapping from alphabetic characters onto array indexes.

Iteration – while

<while-statement> ::=

    while (<condition-expr>) <statement>

where:

- <condition-expr> is any expression
- <statement> can be a compound statement

Iteration – <u>while</u> examples

```
while (j > 0 )
{
   printf("%d\n", j);
   j--;
}
```

<u>Before each loop assume:</u>

j = 10;

```
while (j-- > 0 )
   printf("%d\n", j);
```

What is the value of j <u>after</u> each loop?

```
while (j-- )
   printf("%d\n", j);
```

Iteration – do-while

<do-while-statement> ::=
    do <statement>
    while (<condition-expr>)

where:

- <condition-expr> is any expression
- <statement> can be a compound statement

Iteration – do-while examples

```
do
{
    printf("%d\n", j);
    j--;
} while (j > 0 );
```

Before each loop assume:

j = 10;

```
do
    printf("%d\n", j);
while (j-- > 0 );
```

But what if j was already zero before the loop?

```
do
    printf("%d\n", j);
while (j-- );
```

Iteration - for

<for-statement> ::=

for (<init> ; <condition-expr> ; <step>)
<statement>

where:

- <init> is 0 or more expressions that are executed once only at the start of the loop
- <condition-expr> is any expression
- <step> is 0 or more expressions that are executed each time at the bottom of the loop
- <statement> can be a compound statement

Iteration - for examples

```
for (j = 10; j > 0; j-- )
   printf(“%d\n”, j);
```

Every for loop has a corresponding while loop

A typical use of a for loop is to perform a loop *count* number of times.

```
for (j = 0; j < count; j++ )
{
   …
}
```

```
for (j = 0, k = 5; j < count && k > 0; j++, k-- )
{
   …
}
```

… they can get quite complex

Structures built from basic data types

- It is possible to construct more complex (i.e. structured) data types from the basic types
- The 2 main structured types in C are:
  - arrays – where each element is of the same type
  - structs – where each element (known as a member) can be of a different type (discussed next week)
- These simple building blocks allow us to model complex real life entities

Arrays

A simple one-dimensional array:

```
float table[5];
int i = 1; j = 2;

table[2] = 3.24
table[i] = table[2] + 1;
table[i+j] = 18.0;
table[--i] = table[j] - 2;
```

Array declaration - subscript is the number of elements in the array

	[0]	[1]	[2]	[3]	[4]
table	1.24	4.24	3.24	18.0	???

No bounds checking in C

- A C implementation does **not** necessarily check for array boundary violations

```
table[5] = 12.2; /* syntactically legal ... */
table[-1] = 0.78;/* ... but not semantically */
```

	[0]	[1]	[2]	[3]	[4]	
0.78	1.24	4.24	3.24	18.0	???	12.2

memory has been corrupted!

- It is the programmer's responsibility to ensure that a program does not access outside an array's limits.

Arrays of arrays

- Multi-dimensional arrays are easily managed in C
- It is best to view a 2-D array as an array of arrays – this generalises to n-D arrays

```
int i, j, k = 0;
int matrix[NUM_ROWS][NUM_COLS];

for (i=0; i < NUM_ROWS; i++)
   for (j=0; j < NUM_COLS; j++)
      matrix[i][j] = k++;
```

Function calls as flow control

- A function call is a transfer of control to the called function
- A function exit (either return or just getting to the end of the function) is a transfer of control back to the statement after the function call
- In next week we'll focus in more detail on the use of functions in C.

Function definition

<return-type> <function-name> (<parameter-list>)
< block >

We have seen the one pre-defined function in C:

Return type

Function name

```
int main (void)
{
        ...
}
```

Block

Parameter list (empty)

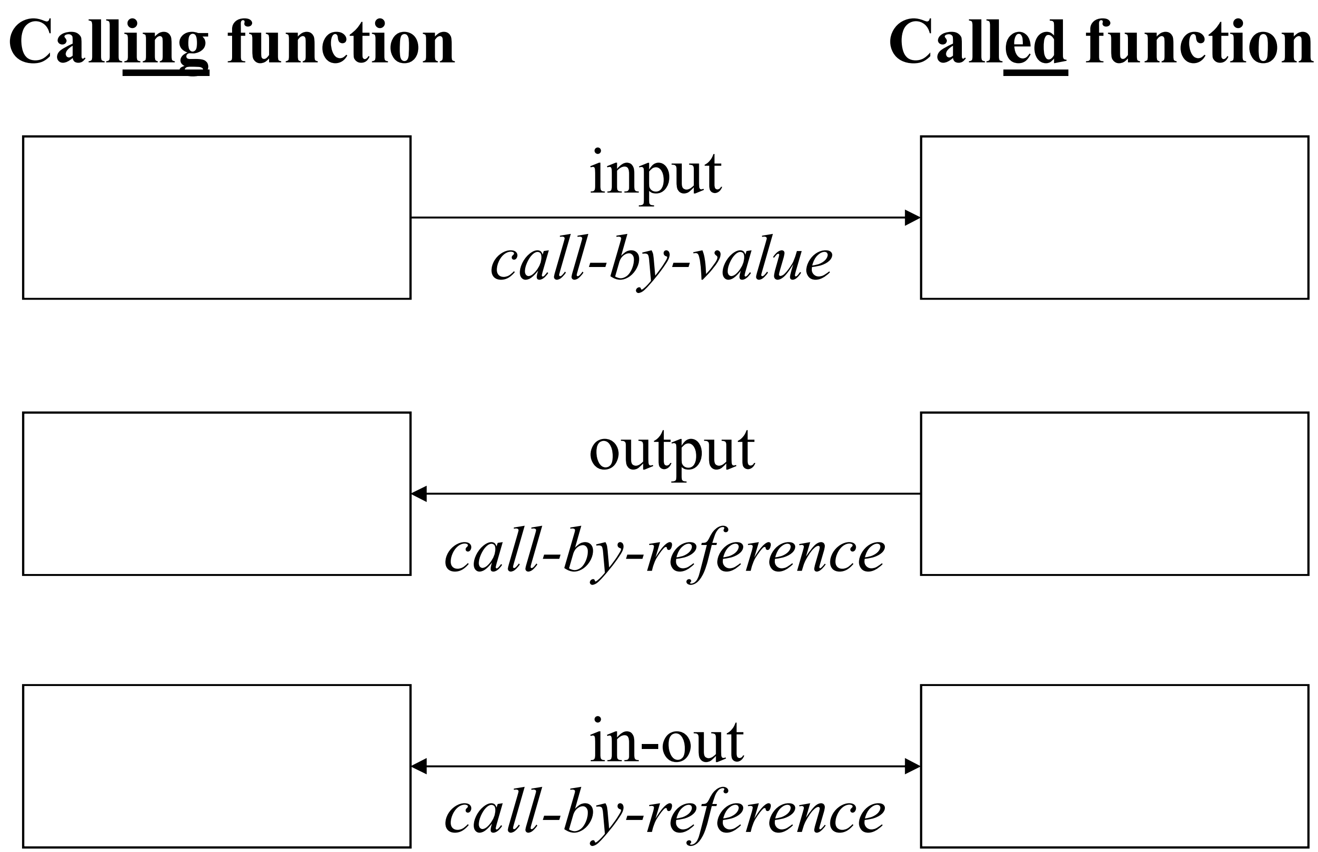
Sharing data between functions

1) Global variables:
- variables declared outside any function
- accessible to all functions
- dangerous and <u>not</u> recommended
- discussed later...

2) Parameter passing:
- data is sent to and received from functions via explicit parameters
- explicitly defines the interactions – i.e. <u>interfaces</u> – between modules/functions

3) return – useful but limited ... discussed later

Types of parameter passing



Parameters – call by value

```
int main(void)
{
    int x, y;
    x = 10;
    y = 4;
    display(x, y);
    return EXIT_SUCCESS;
}
```

```
void display (int a, int b)
{
  printf("First = %d\n", a);
  printf("Second = %d\n", b);
}
```

The values stored in x and y when `display()` is invoked are copied into a and b respectively

Parameters – call by value

```
int main(void)
{
    int x, y;
    x = 10;
    y = 4;
    swap(x, y);
    printf("x:%d y:%d\n",x,y);
    return EXIT_SUCCESS;
}
```

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

What is wrong with this?

Arrays as parameters

- Arrays can <u>only</u> be *call-by-reference* parameters, <u>never</u> *call-by-value*

- This means when you pass an array to a parameter, the function actually accesses the array (**not** a copy of the array)

- … because of the unique relationship between array names and pointers in C … a topic covered in future weeks

Arrays

```
#include <stdio.h>

void fun(int vector[3]){
    vector[1] = 9;
    vector[2] = 8;
    vector[3] = 7;
}

int main(void){
    int i, table[3];
    fun(table);
    for(i = 0; i < 3; i++)
        printf("d\n",table[i]);
    return 0;
}
```

Arrays

```
#include <stdio.h>

typedef int Vector[3];

void fun(Vector v){
   v[1] = 9;
   v[2] = 8;
   v[3] = 7;
}
int main(void){
   int i;
   Vector table;
   fun(table);
   for(i = 0; i < 3; i++)
       printf("d\n",table[i]);
   return 0;
}
```