RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

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

Functional Abstraction and Structs using C

"A computer program will always do what you tell it to do,
but rarely what you want it to do." -- anon.

Revision

- Only 3 types of structured statement 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: e.g. **IF** statement
  - *Iteration* – looping
    - Do this while/until some condition is true

Unstructured flow control (break, continue and goto)

break

Immediately terminates loop (or switch)

If in loop, control transfers to first statement after loop

continue

Immediately terminates current iteration of loop

Control transfers to top of loop and loop resumes

goto

Control transfers to … anywhere (within the function)!!!

In this course, **NEVER** use it!

Selection – switch statement

<switch-statement> ::=

switch (<integer-expr>) {
case <integer-constant> : <statement>
: : :
[default : < statement >] }

where:

- <integer-expr> is any integer expression
- default part is optional(but recommended)

(for full specification see the textbook or reference manual)

Selection – switch statement example

```
switch (option)
{
   case 1 : printf("option 1 chosen\n");
            break;
   case 2 : printf("option 2 chosen\n");
            break;
   case 3 : printf("option 3 chosen\n");
            break;
   default : printf("invalid option\n");
}
```

Problem Solving

- Problem Solving != Debugging.
- Problem Solving != ad-hoc compile-and-tweak.
- There are usually many possible solutions to a problem.
- Typical steps involved in problem solving include: define and analyse, investigate and understand, select best option, design solution, and then implement solution.
- Design and implementation are our prime concerns at the moment. Design of suitable algorithms requires understanding the constructs available. We will review the available constructs.
- A designers three most important tools are knowledge, experience, and good judgement.

Problem Solving (cont'd)

- Sometimes new problems can be approached by starting with known solutions to similar problems, hence the value of knowledge and experience.
- Programming in the small vs. programming in the large.
- Top-down design & Step-wise refinement (Wirth, 1972)
- Decomposition & Abstraction.
- Decomposition is the process of breaking a problem down into individual manageable sub-problems. Each sub-problem then can be decomposed further until it can be solved directly.
- Abstraction involves considering sub-problems in isolation to reduce complexity.

C.A.R. Hoare on “Software Design”

- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

  -- C.A.R. Hoare.

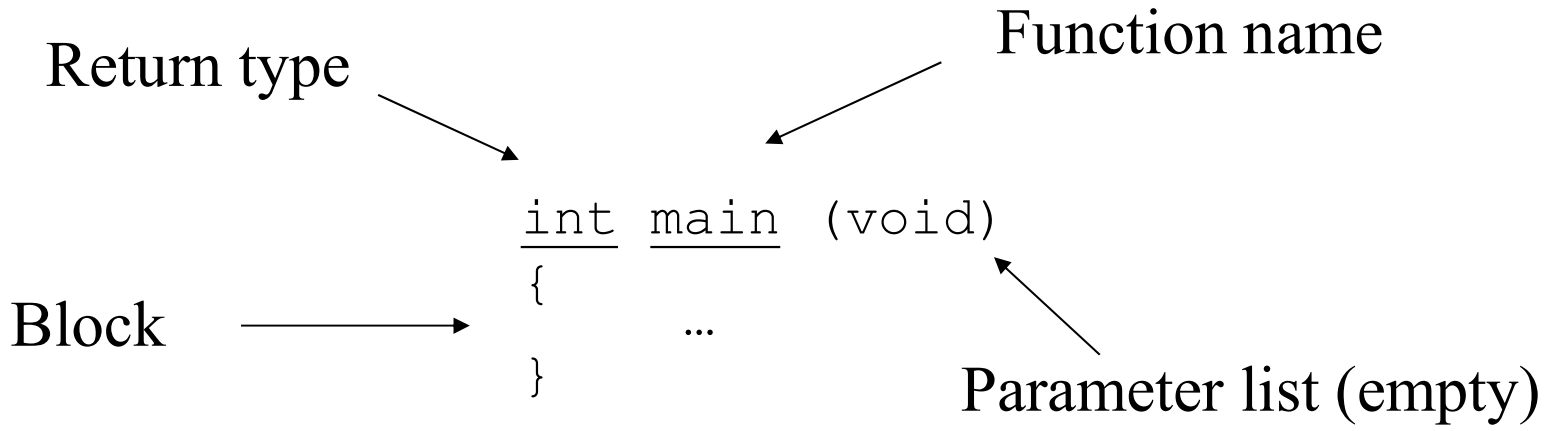
Modularity

- Large, complex programs are built using small, self-contained modules – functions in C
- Each module has (should have) a single, logical and clearly defined purpose
- Critical issue is the interaction – i.e. sharing data – between functions can be done in 3 ways:
  - global variables – not recommended
  - parameter passing – preferred and better technique
  - return statement – limited to single value only

Revision: Function definition

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

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



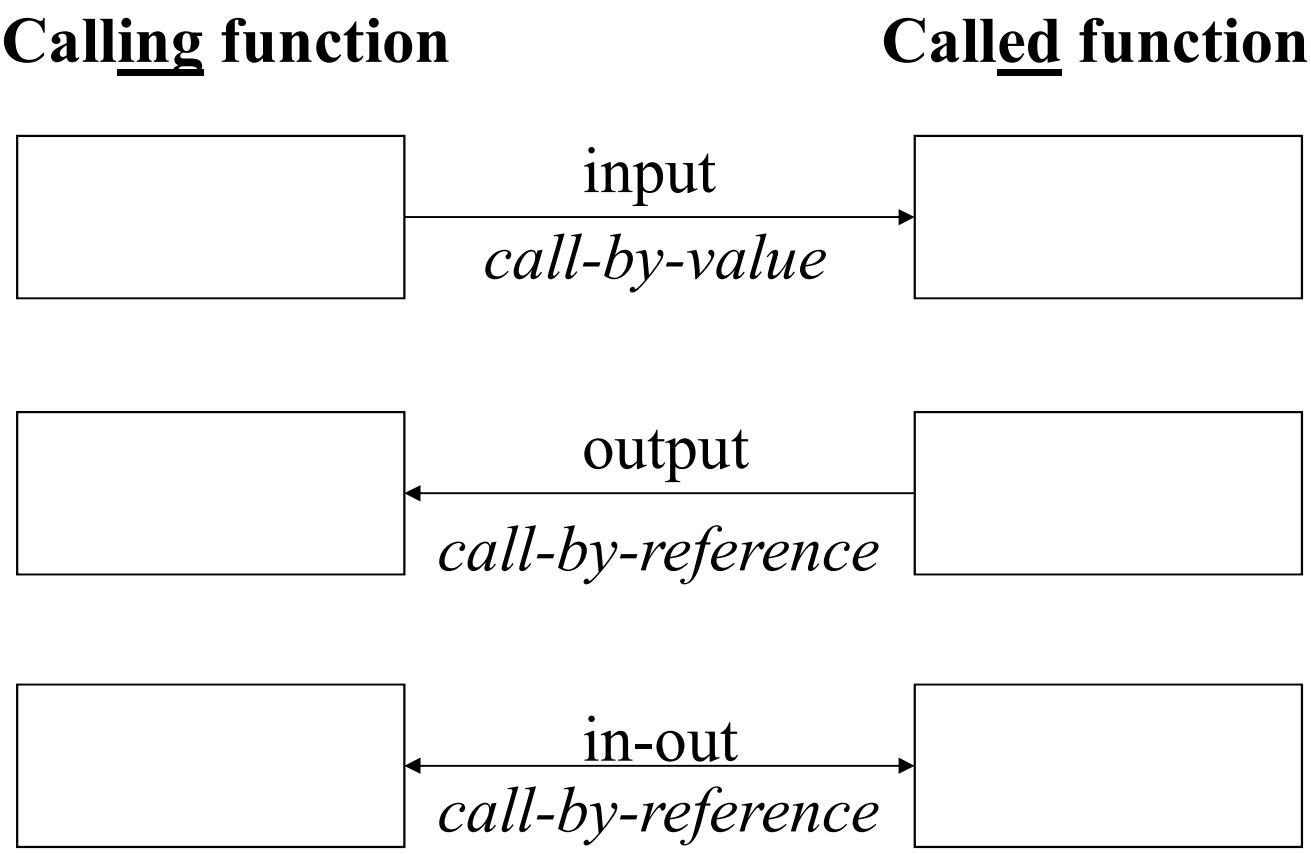
Revision: Sharing data between functions

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

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

3) return – useful but limited

Revision: Types of parameter passing



Rules for passing parameters

- The sending (a.k.a. actual) parameters and the receiving (a.k.a. formal) parameters must match in:
  - data type and ...
  - number
- Names do not need to match (but they are allowed to)
- Position in the parameter list is important
  - the first parameter in function call maps onto the first parameter in the called function … and so on ...
  - i.e. there is *positional correspondence*

Revision: 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);
}
```

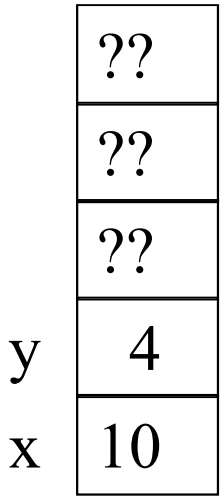
The values stored in x and y when `display()` is invoked are copied into a and b respectively

Parameters – call by value (cont'd)

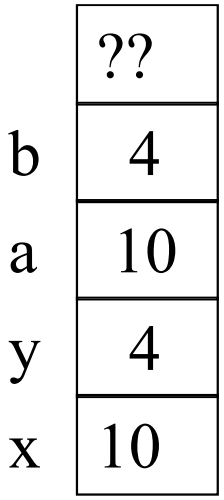
- In this example there are exactly two parameters:
  x maps onto a which must be an int
  y maps onto b which must be an int

- Data values in x and y are passed to function display() by making a copy of them into a and b
- This is known as *call-by-value* or input parameters
- It is not possible for display() to modify x or y in main() using this technique – a good thing!

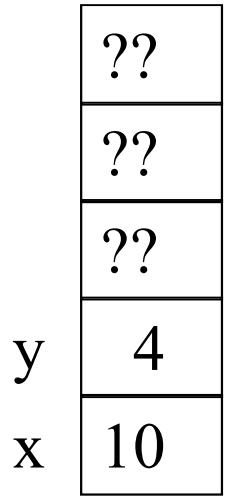
Parameters – call by value (cont'd)



before `display()` is called



`a` and `b` created and initialised



after exiting `display()`

The return statement

- A technique by which it is possible to pass just a single item back to the *calling* function – the return statement

- Requires:
  - defining a non-void return type for the function and ...
  - using the return statement

The return statement (cont'd)

```
int sum (int a, int b)
{
   return (a + b);
}
```

```
int main(void)
{
   int x, y, z;
   x = 10;
   y = 4;
   z = sum(x, y);
   printf(" %d + %d = %d\n",
        x, y, z);
   ...
   return 0;
}
```

Function prototypes

- We can "pre-declare" the essential characteristics of a function:
  - return type
  - function name
  - type, number and position of parameters

  so that the compiler can check the syntax of any function calls – very useful!

- These are called function prototypes.

Function prototypes (cont'd)

```
int sum(int, int);
int main(void)
{
       ...
       z = sum(x, y);
       ...
}
int sum(int a, int b)
{
         …
}
```

Function prototype

note that the:

- name
- return type
- number of parameters
- type of each parameter

match the corresponding:

- function call … and ...
- function definition

[NB also that prototype parameters are not named (optional) and also the semi-colon]

Function prototypes

- If a C source file needs to know about a function prototype, we could put the prototype in a *.h file and include it.

main.h

```
int sum(int, int);
```

main.c

```
#include "main.h"

int main(void)
{
        ...
        z = sum(x, y);
        ...
}
int sum(int a, int b)
{
          …
}
```

Revision: 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
- These simple building blocks allow us to model complex real life entities

Revision: 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;
```

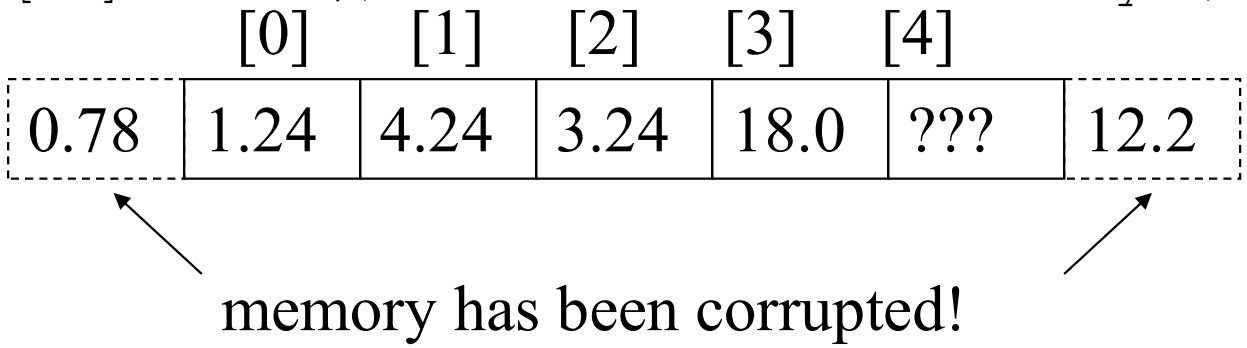
Array declaration - subscript is the number of elements in the array

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

Revision: 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 */
```



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

Initialising arrays

- When an array variable is being defined (and only then), it can be initialised as follows:

```
int topScores[5] = {4,3,5,9,0};
```

or equivalently:

```
int topScores[] = {4,3,5,9,0};
```

Structs

- A struct in C is a complex data type, like an array, except each element can be of a different type
- Each element, known as a member, has its own name (not a subscript) and its own type
- A member can itself be a structured type
- Corresponds to the “record” structure in other languages

structs (cont'd)

```
struct studentResults
{
        unsigned int  ass1, ass2, exam;
        float  total;
        char grade;
};
```

structure tag (optional)

struct members

We can then declare variables of this type by:

```
struct studentResults rohini, james;
```

structs (cont'd)

- Alternatively, variables can be declared with the struct definition:

```
struct studentResults
{
    unsigned int ass1, ass2, exam;
    float  total;
    char   grade;
} rohini, james;
```

- … but we'll see a better method soon …

structs (cont'd)

- Once variables have been declared, we reference individual members using the "dot" (.) operator:

```
rohini.ass1 = 75;
rohini.ass2 = 20;
rohini.exam = 60;
rohini.total = rohini.ass1 * 0.2 +
               rohini.ass2 * 0.3 +
               rohini.exam * 0.5;
if (rohini.total < 50)
  rohini.grade = 'N';
```

User-defined types using typedef

- The typedef statement allows us to assign a user-defined name to a type definition - e.g.

```
typedef int Time;
Time hrs, mins, secs;
```

we now have a type name `Time`

declares variables hrs, mins and secs of type `Time` (i.e. int)

- This is most commonly used with structs to simplify data type names and to build more complex structs of structs (… of structs …)

User-defined types using typedef (cont'd)

The recommended way …

```
typedef struct studentResults
{
    int    ass1, ass2, exam;
    float  total;
    char   grade;
} StudentType;


StudentType rohini, james;
```

Nested structs

```
typedef struct dateStruct
{
        int day;
        int month;
        int year;
} DateType;

typedef struct studentRecord
{
        DateType  birthdate;
        int     ass1, ass2, exam;
        float   total;
        char    grade;

        :
      :
} StudentType;
```

type name DateType

nested struct

type name StudentType

Nested structs (cont'd)

Again we just use the (repeated) dot notation:

```
StudentType student1;

student1.birthdate.day = 6;
student1.birthdate.month = 2;
student1.birthdate.year = 1999;
student1.ass1 = 70;
```

simpler, clearer declaration of variable's type

a "dot" (.) for each level of nesting

Arrays of structs

- We can also easily have arrays where each element of the array is a struct

```
StudentType student[10];

student[i].birthdate.day = 12;
student[i].birthdate.month = 11;
student[i].birthdate.year = 2000;
```

Structs as parameters – call by value

- consider function printStudentData() which accepts a struct
- We could use call-by value
  - i.e. the student struct is copied to the function parameter

```
int main(void)
{
    StudentType student;
    :
    printStudentData(student);
    :
}
```

Structs as parameters (cont'd)

```
void printStudentData(StudentType student)
{
  printf("Birthdate: %d - %d - %d\n",
         student.birthdate.day,
         student.birthdate.month,
         student.birthdate.year);
    ...
}
```

- If the struct is large, call-by-value would require copying a lot of data and use a lot of stack space
- Also, if we use call-by-value the function can't alter values of the original structure
  - how to do call-by-reference is the topic of next week's lecture