CM 10227: Lecture 4

Dr Rachid Hourizi and Dr. Michael Wright

October 25, 2016

Resources

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs
  - ... **Wednesday 11:15-13:05 EB0.7**
  - ... Fridays 17:15 to 19:15 in CB 5.13)
  - The Drop in Session
  - ... booked 20 min appointments
  - ... **Friday 11.15-13.05 1E 3.9**
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)

- If you are finding the pace a little slow on the other hand, you can now sign up to
  - The Advanced Programming Labs
  - ... Wednesday 11.15-13.05 EB 0.7

Questions?

Last Week

- Iteration
- Collections: Strings and Arrays

This Week

- Complex Collections
- Abstract Data Types

Recap

- Programming can be thought of as the process of breaking a large, complex real world task up into smaller and smaller sub-tasks until eventually the sub-tasks are simple enough to be performed with a simple (programmable) instruction. e.g.
  - Get data from the keyboard, or a file, or some other device.
  - Perform basic mathematical operations e.g. addition and/or multiplication.
  - Check for certain conditions and execute an appropriate sequence of statements in line with those condition
  - Perform some action repeatedly, usually with some variation
  - Display data on the screen or send data to a file or other device.

Recap

- Over the first three weeks of the course we have looked at...
- ... and used

- Basic data types
  - int
  - double
  - char

Recap

- Functions
  - main
  - library functions

```
1.     #include <stdio.h>
2.
3.     int main(void) {
4.         int number;
5.         number = 10227;
6.         printf("%d\n", number);
7.         return 0;
8.     }
```

```
$ gcc example.c

$ ./a.out
10227
```

```
4.          int number;
5.          number = 10227;
```

- line 4 - declared a variable **number**
- line 4 - its type is **int**
- line 5 - initialised **number** to **10227**

- C is a statically typed language i.e. types matter
- ... by declaring that **number** is an int
- ... C reserves some memory for us to store an int value
- ... by initialising it **10227** we are storing that value in the memory allocated to **number**
- ... and by using **number** in our code we can get at the value stored there

Question!

What happens if you do not initialise a variable?

```
3.      int main(void) {

7.          return 0;
8.      }
```

- line 3 - declares the **main** function ...
- ... every program must have a main function to say "this is what to run first"
- line 3 - **main** returns an **int**
- line 3 - **main** does not have any parameters
- line 7 - return 0 to indicate that "everything is OK" ...
- ... if we return anything other than 0 it indicates that something went wrong

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

6.         printf("%d\n", number);
```

- line 1 - tells the compiler that we want to use code from another file
- line 6 - **printf** function from the stdio library ...
- ... **printf** has the signature

```
int printf(const char *format, ...)
```

- ... it **returns an int**
- ... takes in as a **parameter** what you want printed including format specifiers

Recap

- Functions
  - written our won
- Iteration
  - e.g. while loops

```
1.      #include <stdio.h>
2.
3.      int main(void) {
4.          int number;
5.          number = 10;
6.          count_down(number);
7.          return 0;
8.      }
9.
10.      void count_down(int from){
11.          while(from >= 0){
12.              printf("%d\n",from);
13.              from= i - 1;
14.          }
15      }
```

```
$ gcc example.c

$ ./a.out
10
9
8
7
6
5
4
3
2
1
0
```

```
10.    void count_down(int from){

15     }
```

- line 10 - function signature...
- ... return type - in this case it does not return anything (**void**)
- ... name of function - **count_down**
- ... parameters - this function expects an **int** to be passed to it when called
- ... in this function we will refer to this parameter as **from**

```
3.      int main(void) {
4.          int number;
5.          number = 10;
6.          count_down(number);
7.          return 0;
8.      }
```

- line 4 - declared a variable **number**
- line 5 - initialise the variable **number** to **10**
- line 6 - we call out **count_down** function
- ... passing in as its parameter the value stored in **number**

```
10.     void count_down(int from){
11.         while(from >= 0){
12.             printf("%d\n",from);
13.             from= i - 1;
14.         }
15     }
```

- line 11 - while loop ...
- ... the code inside the while loop will be executed as long as ...
- ... **from** is greater than or equal to **0**
- line 12 - call the function **printf** in the **stdio** library ...
- ... to print to the standard out the current value of **from**
- line 13 - decrease the value stored in **from** by **1**

Recap

- Last week we started to consider slightly more complex data structures
  - Arrays
  - Strings

- Arrays allow you to group together elements of the same type

```
1.      int multiples_of_two[10];
2.
3.      void set_multiples_array(){
4.          int count = 0;
5.          while(count<10){
6.              multiple_of_two[count] = mult(count, 2);
7.          }
8.
9.      int mult(int number, int by){
10.          return number*by;
11.      }
```

```
1.      int multiples_of_two[10];
```

- line 1 - global variable **multiples_of_two** ...
- ... is an array of ints
- ... has space for 10 elements
- ... accessed by their index which starts at 0 and ends at 9

```
3.     void set_multiples_array(){

5.         while(count<10){
6.             multiple_of_two[count] = mult(count, 2);
7.         }
```

- line 5 - loop 10 times
- line 6 - set the value at the index **count**
- ... to the value returned from the function **mult**
- ... where **mult** takes as parameters **count** and **2**

Recap

- Functions

- Encapsulate functionality
- ... wrap a piece of code in a function
- Aim for as generalised a function as possible
- ... e.g. starting with code that prints multiples of 2
- ... changing it so that it prints multiples of any integer

Questions?

Complex Collections and Abstract Data Types

- We have looked at basic variable types
- We have looked at functions which can manipulate them
- We have looked at arrays that allow us to store a fixed number values of the same type

- In some situations, however, we might want to use yet more complex data types
  - Lists
  - Queues
  - Stacks

- Or even
  - Telephone books
  - Library catalogues
  - Family trees
  - etc.

- In C (and other languages) we can create our own abstract data types
- Abstract Data Types (ADT)
  - Data representation
  - Functions to provide operations on that data representation

Data Representations

- In C we use **struct** to define our own data representations (e.g. Date)
- Which can be comprised of one or more basic data types (e.g. int's and char's)
- Don't have to be the same basic data type
- And can also contain other data representations

Operations

- Define functions to provide operations on these data representations
- e.g. new_date(), increment_date();

- Lets create our own ADT
- Date

```
21 OCT 2015
```

- Data representation for our Date ADT

```
struct Date
{
    int day;
    char month[3];
    int year;
};
```

Aside: Why Our Own Data Representations?

- We could just declare these as variables

```
int day;
char month[3];
int year;
```

- But if we had multiple dates we wanted to keep track of
- Our code quickly becomes long, difficult to read, difficult to maintain etc.

- For example, if we want to store the dates for today, tomorrow and yesterday...

```c
#include <stdio.h>

int today_day;
char today_month[3];
int today_year;

int tomorrow_day;
char tomorrow_month[3];
int tomorrow_year;

int yesterday_day;
char yesterday_month[3];
int yesterday_year;

int main(void) {
        /* Additional Code */
        return 0;
}
```

```c
#include <stdio.h>

struct Date
{
    int day;
    char month[3];
    int year;
};

int main(void) {
        struct Date today;
        struct Date tomorrow;
        struct Date yesterday;
        /* Additional Code */
        return 0;
}
```

- Lets instantiate a Date

```c
#include <stdio.h>

struct Date {
    int day;
    char month[3];
    int year;
};

int main(void) {
        struct Date today;

        today.day = 21;
        today.month[0] = 'O';
        today.month[1] = 'C';
        today.month[2] = 'T';
        today.year = 2015;

        return 0;
}
```

- Now lets provide functions to operate on our data to complete our ADT definition of Date

- Create a new Date
- Increment Date
- etc.

```
struct Date new_date(int d, char m[], int y)
{
    struct Date date;
    date.day = d;
    date.month[0] = m[0];
    date.month[1] = m[1];
    date.month[2] = m[2];
    date.year = y;

    return date;
}
```

- We have created out ADT (Date)
- We could create any number of Date variables as above

```
int main(void) {
        struct Date today;
        struct Date tomorrow;
        struct Date yesterday;

        today = new_date(21,"OCT",2015);
        tomorrow = new_date(22,"OCT",2015);
        yesterday = new_date(20,"OCT",2015);
        return 0;
}
```

- We only need to know about the functions that manipulate the data representation

```
struct Date increment_date(struct Date d)
{
    struct Date date;
    date.day = d.day+1;
    date.month[0] = d.month[0];
    date.month[1] = d.month[1];
    date.month[2] = d.month[2];
    date.year = d.year;

    return date;
}
```

```c
int main(void) {
        struct Date today;
        struct Date tomorrow;

        today = new_date(21,"OCT",2015);
        tomorrow = increment_date(today);
        return 0;
}
```

- If we now change the implementation of increment_date

```
tomorrow = increment_date(today);
```

- Lets look at some more examples which use our Date abstract data type

- Personal Information
  - Name
  - Date-of-Birth
  - etc.

```
struct PersonalInfo
{
    char name[50];
    struct Date dob;
};
```

- As with Date lets provide functions to perform operations on our data representation

```
struct PersonalInfo new_pi(char n[], struct Date d)
{
    struct PersonalInfo pi;
    int i = 0;
    while(i < strlen(n)){
        pi.name[i] = n[i];
        i++;
    }
    pi.dob = d;
    return pi;
}
```

```c
int main(void) {
    struct Date birthday;
    birthday = new_date(26,"OCT", 1985);

    struct PersonalInfo info;
    info = new_pi("Michael",birthday);

    return 0;
}
```

- In pairs (or threes) how could you define ADT's for...
- Netflix?
- Spotify?
- Amazon?

- What data representations (**struct**) might you need?
- What functions would you implement?

- Another common example of the use of ADTs are Stacks

- Ordered collection of data
- We discussed Stacks during the lecture on recursion
- Stacks hold data much like a pile of plates
- The first item added is the last item removed
- First In Last Out (FILO)

- A stack can be described as a collection of elements or items, for which the following operations are defined:
  - isEmpty() is a predicate that returns “true” if the stack is empty, and “false” otherwise;
  - push(item) adds the given item to the stack
  - pop() removes the most recently added
  - top () simply returns the last item to be added to the stack item from the stack S and returns it as the value of the function;
  - The primitive isEmpty is needed to avoid calling pop on an empty stack, which should cause an error.

```c
#include <stdio.h>

struct Stack
{
    int head;
    int max_size;
    int filo[1000];
};

struct Stack my_stack;
```

```
void initialise()
{
    my_stack.head = 0;
    my_stack.max_size = 999;
}

int isEmpty()
{
    return (my_stack.head == 0);

}
```

```
void push(int i)
{
    if(my_stack.head < my_stack.max_size){
        my_stack.filo[my_stack.head] = i;
        my_stack.head = my_stack.head + 1;
    }
}

int pop(){
     if(isEmpty() == 0){
        int popped = my_stack.filo[my_stack.head-1];
        my_stack.head = my_stack.head - 1;
        return popped;
    }
    return -1;
}
```

```
int peek(){
    if(isEmpty() == 0){
        int peeked = my_stack.filo[my_stack.head-1];
        return peeked;
    }
    return -1;
}
```

```c
int main(void) {
    initialise();

    push(10);
    push(20);
    push(30);
    push(40);

    printf("%d\n", pop());
    printf("%d\n", pop());
    printf("%d\n", peek());
    printf("%d\n", pop());

    return 0;
}
```

- What will be printed?

```
$ gcc test_stack test_stack.c

$ ./test_stack
40
30
20
20
```

- Another data structure which we might find interesting is a Map or Dictionary
- When we consider a real world dictionary, however, referring to each of the elements in that dictionary by numerical index seems unwieldy
- E.g. referring to the entry for "Aardvark" as OxfordEnglishDictionary[27] doesn't seem very helpful

- However, we can use the word itself as the "key" to be searched for within the dictionary
- And the entry for that word as the 'value" corresponding to that "key"

- The combination of each key in the dictionary and the corresponding value it refers to are (unsurprisingly) called key-value pairs
- Conceptually, each of those key value pairs form a Tuple

- Tuples are ordered collections of values of different types.
- Unlike arrays or lists, which may contain several values of the same type, tuples may "contain" values of mixed types e.g.
  - (100, 7, "abc", 5.0)

- Commonly used tuples are given specific names

- a tuple of two elements is generally called a pair (or double, couple, dual, twin) and
- a tuple of three elements is generally called a triple.
- There are also such names as quadruple, quintuple and so on.
- Note that there are also tuples of 0 and 1 element.
  - A tuple of one element is rarely used as such, since the element itself may be used instead.
  - However, for theoretical discussion, it may be useful to consider such tuples.
  - A tuple of one element is sometimes called a single.
- It is sometimes easier to use the name tuplen with n being the number of elements.

- Lets create an ADT for a Map
- So we can store ASCII codes for different letters (chars)

```c
#include <stdio.h>

struct KeyValuePair
{
    char key;
    int value;
};

struct Map
{
    int size;
    struct KeyValuePair kvp[1000];
};

struct Map hm;
```

```
int get(char find){
    int i;
    for(i=0; i < hm.size; i++){
        if(hm.kvp[i].key == find){
            return hm.kvp[i].value;
        }
    }
    return 0;
}
```

```
void put(char key, int value){
    int next = hm.size;
    struct KeyValuePair x;
    x.key = key;
    x.value = value;
    hm.kvp[next] = x;
    hm.size = hm.size + 1;
}
```

```
int main(void) {
        put('a',10);
        put('b',11);
        put('c',12);
        put('d',13);
        put('e',14);
        put('f',15);

        printf("%d\n", get('a'));
        printf("%d\n", get('c'));
        printf("%d\n", get('f'));
        return 0;
}
```

```
$ gcc test_map test_map.c

$ ./test_map
10
12
15
```

Questions?

- In pairs (or threes)
- How would you create your own ADT for a Queue?
- First In First Out

- What would be your data representation?
- What functions would you provide?