Lecturer 3: Functions and how to play with them

Henry

Functions are really useful when writing programs. They can help you to decompose the solution into clear steps and this can make the program easier to read and easier to reason about.

However, there are some rules for using functions that we must know and understand.

Today we will start with a review of the rules and then look at some examples.

Function rules

- A function is like a mini-program that lives inside your main program.

- From the main program we should not be allowed to look inside the function to see how it works or to use any variables that live inside it.

- From inside a function we are not allowed to look outside at the main program to see how it works or to use any variables it has.

Function Rules

- The main program can communicate with the function by passing in some values using the function parameters (sometimes called the function arguments). A function can have as many parameters as we want.

- The function can only communicate with the main program by returning a value when the function finishes.

- There should not be any other way in which they communicate.

Function Rules

- Every function needs to have a name.
- Every function needs to have a return type. This is the type of the value which the function returns to the main program when it finishes. (E.g int, float, char).
- Every function should have a list of the parameters it takes, this list should give the name and type of each parameter.

Function Rules

- When you use a function you give the function the actual parameter values that you want it to use. These values can be variables or expressions of the correct type.

- When you use a function it will return a value which can be used in an expression in your main program.

Practical solutions

- Construct a program to take in a date since 1800 and to compute and output the next date.

- We will use variables d, m, y to represent the date. They are all int variables.

Practical solutions

- When the problem involves a few different possibilities it can be useful to try to solve the easiest ones first.

- This is often a good guideline to help us construct a solution.

Practical solutions

- If the date is somewhere in the “middle of the month” (I mean it isn’t the last day in the month) then we just need to add 1 to the day number

- Otherwise.....

Practical solutions

- Otherwise, we will need to move on to the next month..
- If we are not at the end of the year then we set the day number to 1 and we add 1 to the month number.

- Otherwise, we are at the last day of the year and then we must set both the day number and the month number to 1 and add 1 to the year number.

```
if ( d < maxdays(m, y))
  {
    d = d+1 ;
  }
else if ( d == maxdays(m, y))
  {
    if (m != 12)
      {
        d = d+1 ;
        m = m+1 ;
      }
    else if ( m == 12)
      {
        d = 1 ;
        m = 1 ;
        y = y+1 ;
      }
}
```

EASY

EASY

MORE DIFFICULT

MORE DIFFICULT

This is nice, but we need the function maxdays.

```
 int maxdays( int month, int year)
 {
   if ((month == 4) || (month == 6) || (month == 9) || (month == 11))

     { return 30 ; }

   else if ((month == 1) || (month = 3) || (month == 5) || (month == 7) ||
            (month == 8) || (month == 10) || (month == 12))

     { return 31 ; }

   else if ((month == 2))

     { if leap(year)

       { return 29; }

     else if (!leap(year))

       { return 28 ;}
     }
}
```

Practical solutions

- Construct a program to take in a date since 1800 and to compute and output the previous date.

- We will use variables d, m, y to represent the date. They are all int variables.

- We will try to use the same style of thinking as in the last example; solve the easy parts first...

Practical solutions

- If the date is somewhere in the “middle of the month” (I mean it isn’t the first day in the month) then we just need to subtract 1 from the day number

- Otherwise.....

Practical solutions

- Otherwise, we will need to move back to the previous month..
  - If we are not at the start of the year then we set the day number to the last day of the previous month and we subtract 1 from the month number.
  - Otherwise, we are at the first day of the year and then we must set the day number to 31 and the month number to 12 and subtract 1 from the year number.

```
if ( d > 1)
   {
      d = d-1 ;
   }
else if ( d == 1)
   {
      if (m != 1)
         {
            d = maxdays(m-1, y) ;
            m = m-1 ;
         }
      else if ( m == 12)
         {
            d = 31 ;
            m = 12 ;
            y = y-1 ;
         }
}
```

Another problem

- Suppose we want to count the number of **even** digits in an integer.
- First we ask what the answer would be if we only had a single digit.
- If that digit was **even** then we would want to count 1, and of it was **odd** we would want to count 0.
- We could write a little function to take a digit and decide what to count for it.

Another problem

- Now we can use that function to write the actual function which counts the number of even digits.

```
int count_even_digits (int n)
      {
        if (n < 10)
            {return what_to_count_for_this(n);}
        else
            {return what_to_count_for_this(n%10) + count_even_digits(n/10);}
      }
```

Count even digits; Loop

```
int count_even_digits_loop (int n)
        {
          int result = 0;
          while (!(n<10))
            {
              result = result + what_to_count_for_this(n%10) ;
              n = n/10 ;
            }

      result = result + what_to_count_for_this(n) ;
      return result ;
}
```

More problems

- Suppose we asked you to add up the odd digits in an integer.

- How might you do that?

```
int what_to_add_for_this (int x)
      {
        if ( x % 2 == 0)
            {return 0;}
        else
            {return x;}
      }

int add_odd_digits (int n)
      {
  if (n < 10)
      {return what_to_add_for_this(n);}
  else
      {return what_to_add_for_this(n%10) + add_odd_digits(n/10);}
}
```

Sum Odd digits; Loop

You should try to write the loop version of the function to sum the odd digits in an integer.

Factorial

- The factorial is a function defined for natural numbers.
- It is usually defined like this
- fact(0) = 1
- fact(n) = n*fact(n-1) if n>0
- This is already a recursive definition so it is very easy to write it as a function in C

```
int fact (int n)
       {
         if (n == 0)
              {return 1;}
         else
              {return n * fact(n-1);}
       }
```

Many functions in Maths are defined recursively, this makes it very easy to program them.

An “interesting” number

- 145 is an interesting number because it has the following property
- fact(1) + fact(4) + fact(5) = 145
- let us write a function to check if a particular integer is interesting in this way.

```
int is_interesting (int n)
  {
if (n == sum_of_factorials_of_digits(n))
  { return 1;}
else
  { return 0;}
}
```

```
int is_interesting (int n)
  {
  if (n == sum_of_factorials_of_digits(n))
    { return 1;}
  else
    { return 0;}
  }
```

```
int sum_of_factorials_of_digits (int n)
  {
  if (n < 10)
    { return fact (n);}
  else
    { return fact(n%10) + sum_of_factorials_of_digits(n/10);}
  }
```

Some advice

- Knowing how to write and use functions correctly is a key skill you need to know.

- If you have any questions or if you don’t feel confident about writing and using functions please ask me or a TA at the practicals.

A challenge

- The first day of January in 1900 was a Monday.

- Write a program to read in a valid date since then and output what day of the week it is.

A challenge

- Write a function which takes an integer and replaces the digit 3 with the digit 7 anywhere that 3 occurs in the number.

- replace(32532) = 72572

- replace(12456) = 12456)