UCL COMPUTER SCIENCE

COMP0002 Programming Principles

Programming Notes and Exercises 3

Example Answers to selected questions

Core Questions

Q3.1 Write a program to input the length of the sides of a triangle, and prints the area and length of the perimeter of the triangle. If the input values do not represent a triangle then display an error message instead.

If a, b and c are the sides of the triangle then:

perimeter = a + b + c

and

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

where s is the *semiperimeter*

$$s = \frac{1}{2}(a+b+c)$$

The calculation of the perimeter, semi-perimeter and area should be done by separate functions.

Hint: the standard C maths library provides a function called `sqrt` to calculate square roots.

Example Answers:

Here is one way to write the program:

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

double side1, side2, side3;
double area, semiPerimeter, perimeter;

void emptyInputBuffer()
{
  char ch;
  while ((ch = getchar()) != '\n' && ch != EOF);
}

double inputDouble(const char *prompt)
{
  double inputValue = 0.0;
  while(true)
  {
    printf("%s ", prompt);
    if (scanf("%lf", &inputValue)) { break; }
    puts("Unable to recognise a number, please try again.");
    emptyInputBuffer();
  }
  return inputValue;
}

void inputSideLengths()
{
  side1 = inputDouble("Enter length of first side: ");
  side2 = inputDouble("Enter length of second side: ");
  side3 = inputDouble("Enter length of third side: ");
}

bool isValidTriangle()
{
  return ((side1 < (side2 + side3)) &&
          (side2 < (side1 + side3)) &&
          (side3 < (side1 + side2)));
}

void calculatePerimeter()
{
  perimeter = side1 + side2 + side3;
}

void calculateSemiPerimeter()
{
  semiPerimeter = 0.5 * (side1 + side2 + side3);
}

void calculateArea()
{
  area = sqrt(semiPerimeter * (semiPerimeter - side1) *
           (semiPerimeter - side2) * (semiPerimeter - side3));
}

void doCalculations()
{
   calculatePerimeter();
   calculateSemiPerimeter();
   calculateArea();
}

void printTriangleDetails()
{
  printf("Perimeter = %.2f, Area = %.2f\n", perimeter, area);
}

void printErrorMessage()
{
  puts("The input did not specify a valid triangle.");
}
```

```
int main(void)
{
  inputSideLengths();
  if (isValidTriangle())
  {
    doCalculations();
    printTriangleDetails();
  }
  else
  {
    printErrorMessage();
  }
  return 0;
}
```

This program makes use of global variables[1], declared outside of any of the functions, to store the input and the calculated results. Each function is kept as short and cohesive as possible, focusing on doing one task. Also, as far as possible each function operates at the same level of abstraction, meaning that a function either performs a basic task or acts as a control function by calling other more basic functions to carry out a multiple-step task. The functions called by a control function should always be more basic than the control function in terms of how they call other functions. Each function in the program should have its own consistent level of abstraction, and not mix abstraction levels. In a more complex program there will be multiple layers of control functions.

Look at the `main` function and notice that the code can be read in a straightforward way, primarily using the function names, to get a clear idea of what the program as a whole does without having to look at the implementation of each function called. This is an example of how good code readability can be achieved. Further, comments are not needed to explain what the `main` function is doing as it is obvious from reading the code inside it. Good code should explain itself.

What about the use of *global variables* in the program above? A global variable is one that is in scope throughout an entire program, hence global to the program. Global variables are generally considered to be a “bad thing” and their use is to be avoided. There are good reasons for not using them; we should always aim to limit the scope of a name to a minimal region of a program to reduce the inter-dependencies, or *coupling*, that can otherwise result.

For example, if a global variable is used in many functions, then all those functions have to use the variable correctly and any change to the variable, such as its name or type, results in having to modify lots of functions. As the code becomes more inter-dependent, or coupled, it becomes harder to read, write, debug and maintain. As the program gets larger, and parts of it are written by different people, the problems caused by these issues just get worse. You reach the point where you are afraid to change anything, as you don’t know what will break.

Below is a revision of the program that replaces the use of global variables, with local variables and parameters. Is it actually a better version, though?

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

void emptyInputBuffer()
{
  char ch;
  while ((ch = getchar()) != '\n' && ch != EOF);
}
```

[1] Strictly these are external variables, but the discussion will focus on the use, or misuse, of global variables.

```
double inputDouble(const char *prompt)
{
  double inputValue = 0.0;
  while(true)
  {
    printf("%s ", prompt);
    if (scanf("%lf", &inputValue)) { break; }
    puts("Unable to recognise a number, please try again.");
    emptyInputBuffer();
  }
  return inputValue;
}

bool isValidTriangle(double side1, double side2, double side3)
{
  return ((side1 < (side2 + side3)) &&
          (side2 < (side1 + side3)) &&
          (side3 < (side1 + side2)));
}

double calculatePerimeter(double side1, double side2, double side3)
{
  return side1 + side2 + side3;
}

double calculateSemiPerimeter(double side1, double side2, double side3)
{
  return 0.5 * (side1 + side2 + side3);
}

double calculateArea(double semiPerimeter, double side1, double side2, double
side3)
{
  return sqrt(semiPerimeter *
              (semiPerimeter - side1) *
              (semiPerimeter - side2) *
              (semiPerimeter - side3));
}

void printTriangleDetails(double perimeter, double area)
{
  printf("Perimeter = %.2f, Area = %.2f\n", perimeter, area);
}

void printErrorMessage()
{
  puts("The input did not specify a valid triangle.");
}

int main(void)
{
  double side1, side2, side3;
  double perimeter, semiPerimeter, area;

  side1 = inputDouble("Enter length of first side: ");
  side2 = inputDouble("Enter length of second side: ");
  side3 = inputDouble("Enter length of third side: ");

  if (isValidTriangle(side1, side2, side3))
  {
    perimeter = calculatePerimeter(side1, side2, side3);
    semiPerimeter = calculateSemiPerimeter(side1, side2, side3);
    area = calculateArea(semiPerimeter, side1, side2, side3);
    printTriangleDetails(perimeter, area);
  }
  else
  {
    printErrorMessage();
  }
  return 0;
}
```

Without the global variables, most of the functions now have parameters to pass around the values being used. The `main` function declares the required variables as local variables, which is why the other functions now have parameters, as the values of the variables need to be available outside the scope of the main function (don't forget the values of the variables are passed to other functions not the variables themselves — this is call-by-value parameter passing). As a side-effect, though, the `main` function itself has got longer and less readable, due to the code inputting the lengths of the sides now being in the function, replacing the `inputSideLengths` function.

While thinking about function parameters, a good design guideline is to minimise the number of parameters that a function has. One or two parameters are reasonable, three are tolerable, four or five are sometimes needed, six should be an upper limit, while more parameters indicate something is probably going wrong. As with all design guidelines, sometimes you may find a reason for a larger number of parameters, but it needs to be a really good reason. And you need to come back later to have a look to see if you can reduce the number.

We could try to simplify `main` by adding back the `inputSideLengths` function, but there is a problem as there are three side values to return and a function can return a single value only. One messy way of solving this is to start using pointers to store values into the side variables that are now local variables in `main`. Pointers are a topic covered later in the module, so you may want to come back to this section once you are familiar with them. This gives:

```
void inputSideLengths(double *side1, double *side2, double *side3)
{
  *side1 = inputDouble("Enter length of first side: ");
  *side2 = inputDouble("Enter length of second side: ");
  *side3 = inputDouble("Enter length of third side: ");
}
```

and this function call in `main`:

```
inputSideLengths(&side1, &side2, &side3);
```

Basically, `main` passes pointers to its local variables to the `inputSideLengths` function, which now takes three pointers as parameters. Each pointer is dereferenced, using the '`*`' dereference operator, to store a value in the variable being pointed at. When the function call returns the variables back in `main` will have been updated.

If you are thinking that all this looks like a nasty work-around, you are right! This is going down the route of making the solution unnecessarily complicated for no benefit. In fact you should not write a function like `inputSideLengths` that is using pointer parameter variables to effectively return values. Parameters should be for passing values into a function, not getting values back out. With this version of the function the parameters are pointers to variables where the values input from the keyboard are being written to, not read from. They are being used as what is referred to as output parameters.

Except… Nothing is straightforward and sometimes it is useful, even necessary, to write a function that does use pointer parameters to return values! An array parameter is actually a pointer to an array, even if the square bracket notation is used and it looks like the entire array is being passed not just a pointer. This means that the array is not copied and in the function the pointer is being used to access the array, which of course is the same array as the code calling the function is using. If the array is modified by assignment in the function, the modification is effectively passed back out of the function, as the array in the calling code is being modified.

This, for example, is how an array sorting function works. A pointer to the array to be sorted is passed to the function and it sorts the array. No copy is made, unless the function first makes a copy and sorts that, so the array in the calling code gets sorted and the unsorted version gets overwritten.

Learn to recognise when you are forcing an unnecessarily complex solution on what should be a simpler problem. Design is about making trade-offs and finding a balance amongst competing goals, in this case the desire to avoid global variables versus accepting that in this context they provide a good enough solution.

Wait though, not done yet! There is a way to return more than one value from a function by using a *struct*, which is a mechanism for grouping several variables together as a single entity in the same block of memory. For example,

```
struct account {
  unsigned int accountNumber;
  char *name;
  int balance;
}
```

This declares a struct type called `struct account` that consists of three variables, each with their own name within the struct, allowing variables of the new type to be declared:

```
struct code anAccount;
```

The variable `anAccount` contains the three variables declared in the struct, which can then be accessed using the dot ('.') operator:

```
anAccount.accountNumber = 10;
```

See https://en.wikipedia.org/wiki/Struct_(C_programming_language) to find out more.

Note that the type name when declaring a struct is actually composed of two words of tokens: *struct account*. This is in contrast to the built in type names like int, double, etc.

So where does this leave the solution to this question? Is the first program with the global variables actually a better solution? Yes, as a solution to the question asked it is.

Aren't global variables bad? Yes, but as with all design problems you have to look at the context and trade-offs. In this case we have a small program, less than one hundred lines of code, for which a simple solution is sufficient. The really bad things about using global variables are not going to affect a small program like this. Trying to eliminate the global variables actually resulted in a less satisfactory design. Scale matters when making design decisions and trade-offs.

If this program were to evolve into something larger and more complex (a super-duper do-everything triangle calculating program), then we would have to revisit the global variable decision. If possible, we would switch to a different programming language such as C++ or Java. Both support object-oriented programming, which gives a much better range of design solutions than C.

Q3.2 Write a program that defines and calls a function to raise an integer to a positive integer power (e.g., x^y). Provide a version of the function that uses a loop and another that uses recursion.

Example Answer:

Here is a program giving three example functions, one iterative, two recursive. The second recursive function, *recursivePower2*, is the same algorithm as the first recursive version but uses the conditional operator instead of an if statement.

The conditional operator uses two symbols, '?' and ':', and has three arguments or operands, making it a *ternary* operator. When evaluated the conditional operator yields a value, which in this case is returned as the value of the function call. The boolean expression in parentheses before the '?' is evaluated first, if it is true (not zero) then the expression following the '?' is evaluated and is returned as the value of the conditional operator expression. Otherwise if false (zero) the expression following the ':' is the value of the conditional operator expression.

```
#include<stdio.h>

int iterativePower(int n, int power)
{
  int i;
  int result = 1;
  for (i = 0 ; i < power ; i++)
  {
    result *= n;
  }
  return result;
}

int recursivePower(int n, int power)
{
  if (power == 0) return 1;
  return n * recursivePower(n, power - 1);
}

int recursivePower2(int n, int power)
{
  return (power == 0) ? 1 : n * recursivePower(n, power - 1);
}

int main(void)
{
  int n;
  for (n = 0 ; n < 17 ; n++)
  {
    printf("Iterative   2^%d = %d\n", n, iterativePower(2,n));
    printf("Recursive   2^%d = %d\n", n, recursivePower(2,n));
    printf("Recursive2  2^%d = %d\n", n, recursivePower2(2,n));
  }
}
```

Q3.3 Write a program to determine if a long int is a palindrome (i.e., represents the same value when reversed, for example 123454321).

Example Answer:

This program will keep asking for input until a valid long int is entered, then it tests to see if the number is a palindrome by reversing the digits in the number and checking if the reversed number is the same as the original.

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

long reverse(long number)
{
  long reversed = 0;
  do
  {
    int rightDigit = number % 10;
    reversed = reversed * 10 + rightDigit;
    number = number / 10;
  }
  while ( number != 0 );
  return reversed;
}

void emptyInputBuffer()
{
  char ch;
  while ((ch = getchar()) != '\n' && ch != EOF);
}

long inputLong()
{
  long number = 0;
  while(true)
  {
    printf("Enter your long integer: ");
    if (scanf("%li", &number)) { break; }
    puts("Unable to recognise a long integer, please try again.");
    emptyInputBuffer();
  }
  return number;
}

int main(void)
{
  long number = inputLong();
  long reversedNumber = reverse(number);
  char *result = (number == reversedNumber) ? "is" : "is not";
  printf("The long integer %li %s a palindrome\n", number, result);
  return 0;
}
```

Take a look at the `inputLong` function, and notice that the while loop boolean expression is simply the value `true`. Instead of using the loop boolean expression to terminate the loop a `break` statement is used to exit the loop body when the `scanf` function is able to find a valid integer in the input character sequence.

A `break` statement will jump out of the enclosing loop, with program execution continuing with the statement following the loop. If the loop is a nested loop then only that loop is terminated, not the outer nesting loop.

This style of loop termination is useful when there is no convenient loop boolean expression possible, and can also simplify the statements needed in the loop body. As an exercise try re-writing the loop without using `break`.

Q3.4 Write a program that uses a function to calculate the product of a sequence of numbers specified by the user. For example, if the user specifies 4 to 8, the function calculates 4*5*6*7*8. Any range can be used, including the use of negative numbers, and the program must correctly determine the values in the range.

Example Answer:

Here is a version of the program. The `calculateSum` function is doing the main work, and uses the conditional operator to determine which of the input numbers is the lowest and highest, in order to calculate the product correctly in the loop.

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

void emptyInputBuffer()
{
  char ch;
  while ((ch = getchar()) != '\n' && ch != EOF);
}

long inputLong(char* prompt)
{
  long number = 0;
  while(true)
  {
    printf("%s: ", prompt);
    if (scanf("%li", &number)) { break; }
    puts("Unable to recognise the input, please try again.");
  }
  emptyInputBuffer();
  return number;
}

long calculateSum(long first, long last)
{
  long lowest = (first <= last) ? first : last;
  long highest = (last >= first) ? last : first;
  long product = lowest;
  while (++lowest <= highest)
  {
     product *= lowest;
  }
  return product;
}

int main(void)
{
  long firstNumber = inputLong("Enter first number of the range");
  long lastNumber = inputLong("Enter last number of the range");
  long product = calculateSum(firstNumber, lastNumber);
  printf("The product of the range is: %li\n", product);
  return 0;
}
```

Notice that any sequence that includes the number zero will result in a product of zero. In theory the calculation loop could be stopped early if zero occurs, or some `if` statements might added to check whether zero occurs in the range and skip the loop altogether. However, in practice this program simply doesn't need that level of sophistication, and we can easily trade-off the cost of occasionally doing unnecessary calculations against keeping the code simple.

You may have also noticed that when the product of a longer sequence of numbers is calculated it rapidly becomes very large unless the range includes zero! In fact, the product of just `1` to `21` is enough to cause a numeric overflow, meaning that the result is too big to store in a 64-bit long value (assuming a long on your computer is 64 bits, which is usually the case). If an overflow does occur it doesn't cause the program to crash, an invalid result is calculated and displayed.

Q3.5 Write a program that reads an integer between 0 and 999 and "verbalises it". For example, if the program is given 123 it would display "one hundred and twenty-three".

Hint: Write functions to deal with ranges of numbers, such as single digits between "zero" and "nine", numbers between 10 and 19 and so on.

Example Answer:

This is a bit more complicated than previous questions, so we will work on it in stages, following the principle of getting something simple working first and then extend it step by step until the solution is reached. The first stage is to verbalise the digits zero to nine, giving this program:

```
#include <stdio.h>

void verbaliseDigit(int n) {
  char *digit;
  switch (n)
  {
    case 0 : digit = "zero"; break;
    case 1 : digit = "one"; break;
    case 2 : digit = "two"; break;
    case 3 : digit = "three"; break;
    case 4 : digit = "four"; break;
    case 5 : digit = "five"; break;
    case 6 : digit = "six"; break;
    case 7 : digit = "seven"; break;
    case 8 : digit = "eight"; break;
    case 9 : digit = "nine"; break;
    default : digit = "unknown";
  }
  printf("%s", digit);
}
```

```
void verbaliseNumber(int n) {
  printf("The number is ");
  verbaliseDigit(n);
  printf("\n");
}

int main(void)
{
  verbaliseNumber(0);
  verbaliseNumber(1);
  verbaliseNumber(9);
  return 0;
}
```

As we are focussing on the verbalisation part this program doesn't do any input, we will add that once verbalisation works. The `main` function is acting as a very basic test function to check that verbalisation is working.

Can this be improved? Well consider the `verbaliseDigit` function. It uses a switch statement but look at the repetitive pattern of each case in the switch. This is a form of duplication and duplication is bad! To improve the design of code we identify and remove duplication, such as that repetitive pattern we see in the switch statement.

The words 'zero' through 'nine' are data, so should be stored in a data structure. We can use a file scope (global to the file) array:

```
const char* const digits[] = {"zero", "one", "two", "three", "four", "five",
                              "six", "seven", "eight", "nine"};
```

Then rewrite the `verbaliseDigit` function to this:

```
void verbaliseDigit(int n) {
  printf("%s", digits[n]);
}
```

That is a much better, and shorter, function!

The keyword `const` means that a variable is a constant and cannot be changed by assignment. Using the keyword allows the programmer to explicitly state that the variable value should not change and the compiler will enforce this, reporting an error if an assignment is attempted. In a language like C, of course, pointers can always to mis-used to get around this, but that would be bad design and bad practice.

The type of the array may look odd as it includes the keyword `const` twice. It actually means that the array is a sequence of constant pointers to constant characters, expressing the intention that then contents of the array should not be changed by either assigning a new pointer to a different string, or changing any of the characters in a string.

With a basic program in place, next we can consider the numbers from ten to nineteen. All of these numbers have a name that is used only in that number range, so need to be handled separately. Once we get to twenty we can start reusing the numbers from one to nine. The following code adds support for ten to nineteen, using a second array to store the number names.

```
const char* const digits[] = {"zero", "one", "two", "three", "four", "five",
                              "six", "seven", "eight", "nine"};


const char* const teens[] = {"ten", "eleven", "twelve", "thirteen", "fourteen",
                             "fifteen", "sixteen ", "seventeen", "eighteen",
                             "nineteen"};
```

```
void verbaliseDigit(unsigned short n) {
  printf("%s", digits[n]);
}

void verbaliseTeens(unsigned short n) {
  printf("%s", teens[n-10]);
}

void verbaliseNumber(unsigned short n) {
  printf("The number is ");
  if (n < 10) {
    verbaliseDigit(n);
  } else {
    verbaliseTeens(n);
  }
  printf("\n");
}
```

Notice that the type of the number being verbalised has been changed to `unsigned short`. This recognises that no negative integers are being handled and the maximum number, as specified in the question, can easily fit into a `short`.

Now what about the numbers from twenty to ninety-nine? We can follow the same approach and add this array and function:

```
const char* const tens[] = {"twenty", "thirty", "forty", "fifty", "sixty",
                            "seventy", "eighty ", “ninety”};

void verbaliseTwentyToNinetyNine(unsigned short n) {
  printf("%s ", tens[(n/10)-2]);
  if ((n % 10) != 0)
    verbaliseDigit(n%10);
}
```

And modify `verbaliseNumber` like this:

```
void verbaliseNumber(unsigned short n) {
  printf("The number is ");
  if (n < 10) {
    verbaliseDigit(n);
  } else
  if (n < 20) {
    verbaliseTeens(n);
  } else {
    verbaliseTwentyToNinetyNine(n);
  }
  printf("\n");
}
```

So far, so good, but it is a bit annoying that the nested if statements are appearing due to the numbers from ten to ninety-nine requiring two functions to verbalise them, a consequence of the way numbers are named.

If we push ahead and add the ability to verbalise up to nine hundred and ninety-nine, we get:

```
void verbaliseHundreds(unsigned short n) {
  verbaliseDigit(n / 100);
  printf(" hundred ");
  if ((n % 100) != 0) {
    printf("and ");
    if ((n % 100) > 19) {
      verbaliseTwentyToNinetyNine(n % 100);
    } else {
      if ((n % 100) > 9) {
        verbaliseTeens(n % 100);
      } else {
        verbaliseDigit(n % 100);
      }
    }
  }
}

void verbaliseNumber(unsigned short n) {
  printf("The number is ");
  if (n < 10) {
    verbaliseDigit(n);
  } else
  if (n < 20) {
    verbaliseTeens(n);
  } else
  if (n < 100) {
    verbaliseTwentyToNinetyNine(n);
  } else {
    verbaliseHundreds(n);
  }
  printf("\n");
}
```

Well it works, but time for some cleaning up to remove the nested if-else statements and make the code more readable.

Here is a revised version:

```
const char* const digits[] = {"zero", "one", "two", "three", "four", "five",
                              "six", "seven", "eight", "nine"};

const char* const teens[] = {"ten", "eleven", "twelve", "thirteen", "fourteen",
                             "fifteen", "sixteen ", "seventeen", "eighteen",
                             "nineteen"};

const char* const tens[] = {"twenty", "thirty", "forty", "fifty", "sixty",
                             "seventy", "eighty ", "ninety"};

void verbalise(unsigned short);

void verbaliseDigit(unsigned short n) {
  printf("%s", digits[n]);
}

void verbaliseTeens(unsigned short n) {
  printf("%s", teens[n-10]);
}

void verbaliseTens(unsigned short n) {
  printf("%s", tens[(n/10)-2]);
  if ((n % 10) != 0) {
    printf("-");
    verbaliseDigit(n % 10);
  }
}

void verbaliseHundreds(unsigned short n) {
  verbaliseDigit(n / 100);
  printf(" hundred ");
  if ((n % 100) != 0) {
    printf("and ");
    verbalise(n % 100);
  }
}

void verbalise(unsigned short n) {
  if (n < 10) verbaliseDigit(n);
  if (n > 9 && n < 20) verbaliseTeens(n);
  if (n > 19 && n < 100) verbaliseTens(n);
  if (n > 99) verbaliseHundreds(n);
}

void verbaliseNumber(unsigned short n) {
  printf("The number is ");
  verbalise(n);
  printf("\n");
}
```

This version has a control function called `verbalise` that calls a function for each number range. The `verbaliseHundreds` function calls `verbalise` recursively to avoid duplicating any of the `verbalise` function behaviour. Note this has the side-effect of requiring the `verbalise` function to be declared earlier in the file using a function signature. A declaration names a function and specifics the return and argument types but does not define a function body. If the declaration is removed then there will be a compilation error as `verbalise` is called before it is defined, generating a conflicting signature, as the compiler will assume it takes a parameter of type `int`.

There is still some cleaning up of the code that might be done, such as removing the 'magic' numbers, and introducing a few more functions to help readability. However, for the purposes of these exercises this is good enough.

Finally here is the complete program, with the code added to read a number to be verbalised from the input.

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

const char* const digits[] = {"zero", "one", "two", "three", "four", "five",
                              "six", "seven", "eight", "nine"};

const char* const teens[] = {"ten", "eleven", "twelve", "thirteen", "fourteen",
                             "fifteen", "sixteen ", "seventeen", "eighteen",
                             "nineteen"};

const char* const tens[] = {"twenty", "thirty", "forty", "fifty", "sixty",
                             "seventy", "eighty ", "ninety"};

void verbalise(unsigned short);

void verbaliseDigit(unsigned short n) {
  printf("%s", digits[n]);
}

void verbaliseTeens(unsigned short n) {
  printf("%s", teens[n-10]);
}

void verbaliseTens(unsigned short n) {
  printf("%s", tens[(n/10)-2]);
  if ((n % 10) != 0) {
    printf("-");
    verbaliseDigit(n % 10);
  }
}

void verbaliseHundreds(unsigned short n) {
  verbaliseDigit(n / 100);
  printf(" hundred ");
  if ((n % 100) != 0) {
    printf("and ");
    verbalise(n % 100);
  }
}

void verbalise(unsigned short n) {
  if (n < 10) verbaliseDigit(n);
  if (n > 9 && n < 20) verbaliseTeens(n);
  if (n > 19 && n < 100) verbaliseTens(n);
  if (n > 99) verbaliseHundreds(n);
}

void verbaliseNumber(unsigned short n) {
  printf("The number is ");
  verbalise(n);
  printf("\n");
}

void emptyInputBuffer()
{
  char ch;
  while ((ch = getchar()) != '\n' && ch != EOF);
}
```

```
unsigned int inputNumber(char* prompt)
{
  unsigned short number = 0;
  while(true)
  {
    printf("%s: ", prompt);
    if (scanf("%hu", &number)) { break; }
    puts("Unable to recognise the input, please try again.");
    emptyInputBuffer();
  }
  return number;
}

int main(void)
{
  unsigned int number = inputNumber("Enter number in the range 0-999: ");
  verbaliseNumber(number);
  return 0;
}
```

And for one last thing, here is the code to verbalise up to 9999.

```
void verbaliseThousands(unsigned short n) {
  verbaliseDigit(n / 1000);
  printf(" thousand ");
  if ((n % 1000) != 0) {
    if ((n % 1000) < 100) { printf("and "); }
    verbalise(n % 1000);
  }
}

void verbalise(unsigned short n) {
  if (n < 10) verbaliseDigit(n);
  if (n > 9 && n < 20) verbaliseTeens(n);
  if (n > 19 && n < 100) verbaliseTens(n);
  if (n > 99 && n < 1000) verbaliseHundreds(n);
  if (n > 999) verbaliseThousands(n);
}
```

Q3.6 (This is a simpler version of the program than asked for in the exercise question sheet). Write a program that uses a function to determine if an integer is a prime number.

Example Answer:

There are a number of algorithms for finding prime numbers. The algorithm used in the program below is one of the simplest but also least efficient, working by dividing the number being tested by all the odd numbers up to the square root of the number. This can be classified as a ‘brute force algorithm’, as it simply tries everything until an answer is reached.

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

bool isPrime(long n) {
  if (n < 2L) { return false; }
  if (n == 2L) { return true; }
  if ((n % 2L) == 0L) { return false; }
  long divisor = 3L;
  const long limit = sqrt(n);
  while (divisor <= limit) {
    if ((n % divisor) == 0L) { return false; }
    divisor += 2L;
  }
  return true;
}

int main(void)
{
  int count = 0;
  for (long i = 2L; i < 2000000L; i++) {
    if (isPrime(i)) {
      printf("%ld is prime\n", i);
      count++;
    }
  }
  printf("%d primes found\n", count);
  return 0;
}
```

The program finds all the prime numbers less than two million. You may notice that it uses type `long` to represent integers, and that the literal values `2` and `3` are written as `2L` and `3L`. The ‘`L`’ specifies the literal value is of type `long` rather than the default of type `int`. Actually, as no negative numbers are involved type `unsigned long` could be used to increase the maximum prime number that can be represented.

The current generation of processors is so fast that this program will complete in a relatively few seconds. In fact, most of the runtime will be taken up printing the prime numbers as they are found! If the `printf` in the loop is commented out the program will finish in under a second on a reasonable performance desktop or laptop machine (on Unix use the `time` command to find out how long a program runs for). Even on a tablet or phone the runtime is quick. However, a value of type `long` can hold a pretty large number, up to +9223372036854775807 for a 64-bit 2’s complement value, and finding all the primes in that range will take a great deal longer.

There are much more efficient algorithms for finding prime numbers. One that is commonly referenced is The Sieve of Eratosthenes. (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes).

Q3.7 Twin prime numbers occur when the value of the two prime numbers differ by less than or equal to two. For example, 2 and 3, 3 and 5, and 11 and 13 are twin prime numbers. Write a program to print all the twin prime numbers between a specified range.

Example Answer:

The previous question includes a function to test if a `long` integer is prime so it can be reused here. The main loop simply keeps track of the last prime found and uses it to compare with the next prime found, with two added.

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

bool isPrime(long n) {
  if (n < 2L) { return false; }
  if (n == 2L) { return true; }
  if ((n % 2L) == 0L) { return false; }
  long divisor = 3L;
  const long limit = sqrt(n);
  while (divisor <= limit) {
    if ((n % divisor) == 0L) { return false; }
    divisor += 2L;
  }
  return true;
}

int main(void)
{
  long lastPrime = 0L;
  for (long i = 3L; i <= 10000; i++) {
    if (isPrime(i)) {
      if ((lastPrime + 2L) == i) {
        printf("Found %ld  and %ld\n", lastPrime, i);
      }
      lastPrime = i;
    }
  }
  return 0;
}
```

There is a useful list of twin primes here: https://primes.utm.edu/lists/small/100ktwins.txt if you want to check the right numbers are being found!

Q3.8 A strong number is defined as an integer where the sum of the factorials of the digits forming the number is equal to the original number. For example: 1! + 4! + 5! = 1 + 24 + 120 = 145. Write a program to find all the strong numbers within a specified range.

Example Answer:

Here is a straightforward program that tests each `int` value up to the maximum that can be represented (2147483647).

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

int factorial(int n)
{
  if (n < 1)
    return 1;
  else
    return n * factorial(n-1);
}

int factorialSum(int n)
{
  int sum = 0;
  while (n != 0)
  {
    int digit = n % 10;
    n = n / 10;
    sum += factorial(digit);
  }
  return sum;
}

void findStrongNumbers(int limit)
{
  for (int x = 1 ; x < limit ; x++)
  {
    if (factorialSum(x) == x)
      printf("%d is a strong number\n", x);
  }
}

int main(void)
{
  findStrongNumbers(INT_MAX);
  return 0;
}
```

The maximum `int` value is represented by `INT_MAX`, which is declared in the header file limit.h. This gives the maximum and minimum values that can be represented for all the C types, specific to the compiler and machine being used.

If you run this you code you will discover that a) it takes a long time to finish, and b) there are not very many strong numbers! You may well start to realise that this is all a bit odd and the code is seriously inefficient.

The most obvious inefficiency is that the factorial function is called for every digit in every number. In fact, as only the values 0! through 9! are needed they may as well be stored as literal values and not calculated by the program at all.

Here is a version that eliminates the factorial function:

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

int factorials[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};

int factorialSum(int n)
{
  int sum = 0;
  while (n != 0)
  {
    int digit = n % 10;
    n = n / 10;
    sum += factorials[digit];
  }
  return sum;
}

void findStrongNumbers(int limit)
```

```
{
  for (int x = 1 ; x < limit ; x++)
  {
    if (factorialSum(x) == x)
      printf("%d is a strong number\n", x);
  }
}

int main(void)
{
  findStrongNumbers(INT_MAX);
  return 0;
}
```

This version runs a lot faster but can it be improved further? Well think about what is the largest number that can be found by adding together factorials, in particular 9!. If the largest `int` value is actually 2,147,483,647 with 10 digits then 9! x 10, which is 3,628,800, exceeds the largest strong number that can be represented by an `int`. Limiting the loop to that number dramatically improves the speed and the strong numbers are found almost instantly.

If you dig a bit deeper then you will find this article on Wikipedia:

https://en.wikipedia.org/wiki/Factorion

This uses the name Factorion for a number that is the sum of the factorials of its digits, and there are only four of them in base 10 — yes four! For some reason factorion numbers are often referred to as strong numbers, probably a confusion with another set of ideas altogether, but small example programs and exercises to write a program to find the factorions seem to be a common programming exercise.

In conclusion, the real value of this exercise question is to review the code you are writing and ask yourself how can it be improved. In this case the excessive number of calculations being made, and the time taken for the naive version of the program to run, all pointed to looking more closely at the problem specified and working out a better way to solve it. Can you optimise the code further?

Well…. Given that there are only four numbers, which are well known, you could always do this!

```
#include <stdio.h>

int main(void)
{
  printf("1\n");
  printf("2\n");
  printf("145\n");
  printf("40585\n");
  return 0;
}
```

Nonetheless can you find a quicker way of finding the numbers by searching? On Unix systems you can use the time command to see how long it takes a program to run:

```
$ time ./a.out
1 is a strong number
2 is a strong number
145 is a strong number
40585 is a strong number

real  0m0.220s
```

```
user  0m0.217s
sys   0m0.002s
```

The real time is the total time the program takes to run from the command line. If your computer is running other programs then the real time will be longer as the program will be competing for runtime resources with the others. User time is the actual CPU time taken by the program, sys is the time taken for system calls (where the operating system is invoked to do things like output to the screen).

Q3.10 Consider displaying a large digit formed from star characters:

```
******
     *
******
*
******
```

Write a program that includes a set of functions for displaying each of the digits 0 to 9 and minus, in the way shown above. When one of the functions is called it should display *one line* of a large digit, with the line to display being given as a parameter, e.g., big2(3) would display the 3rd line of a big 2. A further function should be included that takes an integer argument and displays the integer in big digits, for example, 123 would appear as:

```
 *  ****** ******
**       *      *
 *  ****** ******
 *  *           *
*** ****** ******
```

Note: You can only output normal characters left-to-right, line-by-line, so think carefully about how your functions work.

Example Answer:

The question suggests having a separate function for each digit, `big0`, `big1`, `big2` and so on. Let's start by trying this and see what we get.

The design chosen for the large-size digits uses five rows by six columns, giving this set of patterns:

```
     *  ******  ******  *    *  ******  ******  ******  ******  ******  ******
     *       *       *  *    *  *       *            *  *    *  *    *  *    *
     *  ******  ******  ******  ******  ******       *  ******  ******  *    *
     *  *            *       *       *  *    *       *  *    *       *  *    *
     *  ******  ******       *  ******  ******       *  ******       *  ******
```

As the first part of the program design here are some functions to split up an `int` entered via the keyboard to get each digit in the correct order, left to right, and then display the `int` in the large size format row by row.

```
const int rows = 5;

void displayRowForDigit(int row, int digit)
{
  // Still need to write this
}

void displayRowForNumber(int row, int number)
```

```
{
  if (number != 0)
  {
    displayRowForNumber(row, number / 10);
    displayRowForDigit(row, number % 10);
  }
}

void displayNumber(int number)
{
  for (int row = 0 ; row < rows; row++)
  {
    if (number == 0)
      displayRowForDigit(row, 0);
    else
      displayRowForNumber(row, number);
    printf("\n");
  }
}

int main(void)
{
  int number;
  printf("Enter a number: ");
  scanf("%d", &number);
  printf("\n\n");
  displayNumber(number);
  return 0;
}
```

This is straightforward. The `main` function inputs a number and calls `displayNumber` to get it displayed as large character pattern digits.

The `displayNumber` function iterates through each row that will make up the large size digits in order to display a complete line of output made up of the row from each digit, like a slice through the number, done by calling `displayRowForNumber`.

The `displayRowForNumber` function splits the input number into digits in forwards order (e.g., 123 becomes 1,2,3) and calls `displayRowForDigit` for each digit displayed in the row. This means that if we have the input `int 1234567890`, then the first row output will be:

```
    *  ******  ******  *    *  ******  ******  ******  ******  ******  ******
```

The second row:

```
    *       *       *  *    *  *       *            *  *    *  *    *  *    *
```

And so on.

Note that the `displayRowForNumber` is recursive, as that is an easy way of getting each digit in forwards order. You don’t need to know how many digits the number has, for example. It does have the side-effect that it cannot display the number zero, so an extra `if` statement is added in `displayNumber` to deal with that case.

Now we need to think about the functions to display rows from a larger size digit. Starting with the function `big2` we get this:

```
const char *const gap = "  ";
const char *const fullRow  = "******";
const char *const leftRow  = "*     ";
const char *const rightRow = "     *";
const char *const sidesRow = "*    *";

void big2(int row)
{
  switch(row)
  {
    case 0:
    case 2:
    case 4:
      printf("%s", fullRow); break;
    case 1:
      printf("%s", rightRow); break;
    case 3:
      printf("%s", leftRow); break;
   }
}
```

This function uses a `switch` statement to determine what to print for each row of the large digit. It can also be noted that for the large digit representation chosen there are only four different row patterns, so they are declared using four file scope ("global") variables. An additional variable `gap` gives the number of spaces between two digits. The type of these variables is `const char *const`, which means constant pointer to a constant character. Neither the pointer or the string being pointed at can be changed by assignment. The compiler will check this as best it can, but you can, of course, change a string by sneaky use of pointers if you wanted. That is not recommended!

As well as writing the rest of the big number functions we can now go back and complete the `displayRowForDigit` function that needs to call the number functions. The function will look like this:

```
void displayRowForDigit(int row, int digit)
{
  switch (digit)
  {
    case 0: big0(row); break;
    case 1: big1(row); break;
    case 2: big2(row); break;
    case 3: big3(row); break;
    // etc.
  }
  printf("%s", gap);
}
```

Putting everything together we have a complete program:

```
#include <stdio.h>

const int rows = 5;

const char *const gap = "  ";
const char *const fullRow  = "******";
const char *const leftRow  = "*     ";
const char *const rightRow = "     *";
const char *const sidesRow = "*    *";
```

```
void big0(int row)
{
  switch(row)
  {
      case 0:
      case 4:
        printf("%s", fullRow); break;
      default:
        printf("%s", sidesRow);
   }
}

void big1(int row)
{
  printf("%s", rightRow);
}

void big2(int row)
{
  switch(row)
  {
    case 1:
      printf("%s", rightRow); break;
    case 3:
      printf("%s", leftRow); break;
    default:
      printf("%s", fullRow);

   }
}

void big3(int row)
{
  switch(row)
  {
    case 0:
    case 2:
    case 4:
      printf("%s", fullRow); break;
    case 1:
      printf("%s", rightRow); break;
    case 3:
      printf("%s", rightRow); break;
   }
}

void big4(int row)
{
  switch(row)
  {
    case 0:
```

```
    case 1:
      printf("%s", sidesRow); break;
    case 2:
      printf("%s", fullRow); break;
    default:
      printf("%s", rightRow);
   }
}

void big5(int row)
{
  switch(row)
  {
    case 1:
      printf("%s", leftRow); break;
    case 3:
      printf("%s", rightRow); break;
    default:
      printf("%s", fullRow);
   }
}

void big6(int row)
{
  switch(row)
  {
    case 1:
      printf("%s", leftRow); break;
    case 3:
      printf("%s", sidesRow); break;
    default:
      printf("%s", fullRow);
   }
}

void big7(int row)
{
  switch(row)
  {
    case 0:
      printf("%s", fullRow); break;
    default:
      printf("%s", rightRow);
   }
}

void big8(int row)
{
  switch(row)
  {
    case 1:
    case 3:
      printf("%s", sidesRow); break;
    default:
      printf("%s", fullRow);
   }
}

void big9(int row)
{
  switch(row)
  {
    case 0:
    case 2:
      printf("%s", fullRow); break;
    case 1:
      printf("%s", sidesRow); break;
    default:
      printf("%s", rightRow);
   }
}

void displayRowForDigit(int row, int digit)
{
  switch (digit)
  {
    case 0: big0(row); break;
    case 1: big1(row); break;
    case 2: big2(row); break;
    case 3: big3(row); break;
    case 4: big4(row); break;
    case 5: big5(row); break;
    case 6: big6(row); break;
    case 7: big7(row); break;
    case 8: big8(row); break;
    case 9: big9(row); break;
  }
  printf("%s", gap);
}

void displayRowForNumber(int row, int number)
{
  if (number != 0)
  {
      displayRowForNumber(row, number / 10);
      displayRowForDigit(row, number % 10);
  }
}

void displayNumber(int number)
{
  for (int row = 0 ; row < rows; row++)
  {
    if (number == 0)
```

```
      displayRowForDigit(row, 0);
    else
        displayRowForNumber(row, number);
      printf("\n");
  }
}

int main(void)
{
  int number;
  printf("Enter a number: ");
  scanf("%d", &number);
  printf("\n\n");
  displayNumber(number);
  return 0;
}
```

There you have it. And I don't like it! There is a lot of code duplication in the functions for printing the large digits, the functions make it difficult to visualise the large digit patterns, and the whole approach is too awkward if the large digits change patterns. Let's have another go.

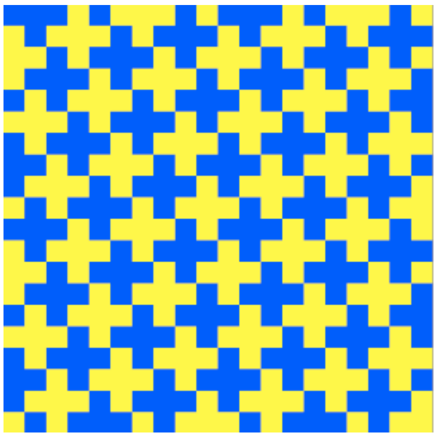
This time a data structure is used to store the large digit patterns, allowing the code to be simplified significantly.

```
#include <stdio.h>

const char *const gap = "  ";
const int rows = 5;

const char *const numbers[][5] =
{
  {
      "******",
      "*    *",
      "*    *",
      "*    *",
      "******"
  },
  {
    "     *",
    "     *",
    "     *",
    "     *",
    "     *",
  },
  {
      "******",
      "     *",
      "******",
      "*     ",
      "******"
  },
  {
      "******",
      "     *",
      "******",
      "     *",
      "******"
  },
  {
      "*    *",
      "*    *",
      "******",
      "     *",
      "     *"
  },
  {
      "******",
      "*     ",
      "******",
      "     *",
      "******"
  },
  {
      "******",
      "*     ",
      "******",
      "*    *",
      "******"
  },
  {
      "******",
      "     *",
      "     *",
      "     *",
      "     *"
  },
  {
      "******",
      "*    *",
      "******",
      "*    *",
      "******"
  },
  {
      "******",
      "*    *",
      "******",
      "     *",
      "     *"
  }
};

void displayRowForDigit(int row, int digit)
{
  printf("%s%s", numbers[digit][row],gap);
```

```
}

void displayRowForNumber(int row, int number)
{
  if (number != 0)
  {
      displayRowForNumber(row, number / 10);
      displayRowForDigit(row, number % 10);
  }
}

void displayNumber(int number)
{
  for (int row = 0 ; row < rows; row++)
  {
    if (number == 0)
      displayRowForDigit(row, 0);
    else
      displayRowForNumber(row, number);
    printf("\n");
  }
}

int main(void)
{
  int number;
  printf("Enter a number: ");
  scanf("%d", &number);
  printf(“\n\n");
  displayNumber(number);
  return 0;
}
```

The digit pattern data is stored in an array of arrays of pointers to `char*` strings, which can be treated as a 2D array using the square bracket syntax. Actually you might argue that this is a 3D array structure, but we don't need to explicitly treat the strings being pointed at as arrays. The `displayRowForDigit` function is now down to one line, and it is a lot easier to see and edit the large digit patterns.

Q3.11 Write a drawing program to display this pattern:



Example Answer:

The ‘trick’ for writing a concise version of this program is to identify what the pattern actually is. It may look like a pattern of cross shapes but it is really a repeating pattern of different coloured squares in rows. Looking at the top row the basic pattern is blue, blue, blue, yellow, blue, yellow, yellow, yellow, blue, yellow, then it repeats. The next row is the same except the pattern starts with the first yellow square, the following row with the fourth yellow square, and so on. Each row starts three steps later than the previous one in the colour cycle, with the cycle restarting as necessary.

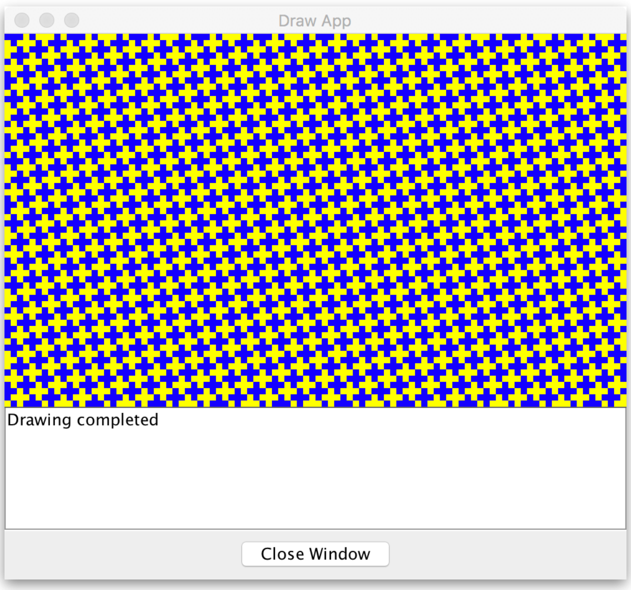
The result is this program:

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

void pattern() {
  const int squareSize = 15 ;
  const int width = 20 ;
  const int height = 20 ;

  colour colourCycle[] = { blue, blue, blue, yellow, blue,
                           yellow, yellow, yellow, blue, yellow } ;

  int cycleLength = sizeof(colourCycle) / sizeof(colour) ;
  int rowStartColourIndex = 0 ;
  for (int row = 0 ; row < height ; row++) {
    int nextColourIndex = rowStartColourIndex ;
    rowStartColourIndex = (rowStartColourIndex + 3) % cycleLength ;
    for (int column = 0 ; column < width ; column++) {
      setColour(colourCycle[nextColourIndex++ % cycleLength]) ;
      fillRect(column * squareSize, row * squareSize, squareSize, squareSize) ;
    }
  }
}

int main(void) {
  pattern();
}
```

The colour cycle is represented by an array of type `colour`, where colour is defined in `graphics.h`. The start position in the cycle is calculated for each row and stored in the `rowStartColourIndex` variable. The `nextColourIndex` variable keeps track of the position in the cycle as each row is drawn. When a colour index reaches the end of the array it is wrapped around to the start of the array by setting it to zero using the modulus operator (‘`%`’).

Notice that the size of the colour array is calculated using the `sizeof` operator. The size of the array in bytes is divided by the size of a single colour value in bytes.

By modifying the values of the `squareSize`, `width` and `height` variables it is easy to draw patterns of different sizes, for example:



Drawing completed