CO516 Introduction to C++ Programming

Department of Computing

Imperial College London

Personal Summary

Author:
Alexander Nederegger

February 8, 2020

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

1.1 A basic C++ program

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Hello World!" << endl;
   return 0;
}
```

Listing 1: 'Hello World!' code

. . . where . . .

- *#include <iostream>* is the include directive that tells the compiler and linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (i.e. the cin and cout statements). The header file "iostream" constains the information about this library
- *using namespace std;* is the using directive
  - C++ divides (e.g. cin and cout) into subcollections of names called 'namespaces'
  - Here it says, the program will be using names that have a meaning in the 'std' namespace
- *return 0;* returns 0 to the OS of the computer to signal that the program has completed successfully (the statement is optional)

1.2 Some terminology and special characters

- *Variable declaration*: signals the compiler to set aside enough space for a particular type – the variable has a random/unpredictable value.
- *Variable assignment*: assigns a value to a variable (also reassign).
- *Variable initialisation*: assigns a value to a variable at the point of declaration (note that 'cont' variables can only be initialised and not assigned and are thus *not* reassignable).
- *Function declaration*: tells the compiler the existence of the function (before the *main* function).
- *Function definition*: defines the behaviour of the function (usually after the *main* function unless function is defined at point of declaration).
- *n* or *endl* . . . are new-line characters (but *endl* also flushes the output buffer.
- *t* . . . is the tab character.
- *0* . . . is the sentinel character (used as last character in a *char array* to mark it as a c-string variable.

1.3 Some commonly used predefined functions

Name	**Description**	**Library header**
sqrt(d)	Takes the square root of a *real* number	cmath
pow(n, m)	Result of n to power of m	cmath
abs(i)	Absolute value of an integer	cstdlib
labs(i)	Absolute value of a long int	cstdlib
fabs(d)	Absolute value of a double	cmath
ceil(d)	Rounds UP a double	cmath
floor(d)	Rounds DOWN a double	cmath
srand()	Seed random number generator	cstdlib
rand()	Random number generator	cstdlib

Table 1: Commonly used predefined functions

2 Variables, Types, and Expressions

2.1 Data Types

Integer

- Can use *int*, *short int*, or *long int*.
- Adding the prefix *unsigned* to any of the types means only positive integers are stored.
- Special prefixes to control the base of the number system
  - *Octal* has a leading *0*
  - *Hexadecimal* has a leading *0x*
  - *Binary* has a leading *0b*

Real numbers

Can use *float*, *double*, as well as the respective *short, long* and *unsigned* prefixes.

Characters

- Use *char*
- Must be in single quotes . . . such as char 'someChar'
- Characters are interpreted as integers inside the computer - can even use arithmetic statements on characters such as . . .
  - . . . 'Z' – 'A' . . . would evaluate to the number of letters in the alphabet.
- The most commonly used collection of characters is the ASCII set (where each character has a number).

- The code below prints out the whole ASCII table.

```
int main()
{
   int number;
   char character;

   for (number = 32 ; number <= 126 ; number = number + 1)
   {
      character = number;
      cout << "The character '" << character;
      cout << "' is represented as the number ";
      cout << dec << number << " decimal or " << hex << number<< " hex.\n";
   }
  return 0;
}
```

Listing 2: Print out the ASCII table

Strings

Strings are discussed later - but we will use two key ways to handle strings.

- *c-string variables* which are arrays of characters that end with a sentinel character.
- Using the *string class*.

Booleans

C++ implicitly includes the named enumeration *enum bool* {*false, true*}, hence you can define a variable such as . . .

```
bool is_true = false;
```

Type casting integers to real numbers and vv.

- New version which should be used is a static cast.

```
static_cast<int>(14.53)  //changes 14.53 to be 14
```

- Older version is simply *int(*14.53*)*, however, this is not recommended to use.

2.2 Formatting numbers

Precision

- Use the below two lines to output only 2 digits of the next number.

```
cout.setf(ios::fixed);
cout.precision(2);
```

- While the two lines below output the a number in *scientific* form.

```
cout.setf(ios::scientific);
cout.precision(2);
```

Fake an output table

Use tabbing of the output with the statement ...

```
#include <iomanip> //need this header to make output left-justified

int main()
{
   cout.setf(ios::left) //need this because of the output is by default right justified
   cout.width(20);  /*The output of the next 'cout' statement will be at
   least 20 characters*/
}
```

Listing 3: Fake an output table using setf(ios::left)

2.3 Constants and enumerations

Enumerations are essentially of type const int. Constants are usually declared as global variables to make them accessible to all functions.

```
enum color {RED, GREEN, BLUE, YELLOW}; //or
enum color {RED=2, GREEN, BLUE=5, YELLOW};
```

In the above, the first line is shorthand for ...

```
const int RED = 0, GREEN = 1, BLUE = 2, YELLOW = 3;
```

... and the second line is shorthand for ...

```
const int RED = 2, GREEN = 3, BLUE = 5, YELLOW = 6;
```

A quick example

```
enum Seasons {spring, summer, autumn, winter}
int main()
{
   Seasons season_1 = winter;
      //'Seasons' is the type name
      //'season_1' is the variable name
      //'winter' is the value (which would be the const int 3
}
```

Listing 4: Basic enum example

+, -, *, /	are basic arithmetic operators
%	Modulo operator (note if numerator is negative the result is negative)
number += 1	Adds one to number (this also works with the other operators above
a *= c1 + c2	Is same as a = a * (c1 + c2)
$n++$ OR $n--$	Is same as n += 1 (n is in/de-cremented by 1, AFTER the line of code has executed
$++n$ OR $--n$	Is same as n-= 1 (n is in/de-cremented BEFORE the line of code has executed

Table 2: Summary of arithmetic operators

2.4 Expressions

Arithmetic

Boolean operators

- >, <, >=, <= . . . used for greater, smaller, etc.
- == . . . *equals*
- != . . . *does not equal*
- && . . . *and*
- ||. . . *or*

Precedence of operations

When in doubt - or to make program more readable - just use brackets.

- The unary operators . . . +, -, !, ++, and $--$
- The binary arithmetic operations . . . *, /, %
- The binary arithmetic operations . . . +, -
- The boolean operations . . . >, <, >=, <=
- The boolean operations . . . ==, !=
- The boolean operation . . . &&
- The boolean operation . . . ||

3 Functions

3.1 Declaration and definition

- A function needs to be *declared* at the top of the program (or included as in a respective header file (pre-supplied or personally created) – otherwise the function cannot be called in any other function bodies (such as in *main*).
- A function has one *return type* (such as int, double, int*, int&, char, char**, ...) but can also have a return nothing using *void*.
  - A void function must not have a return statement - often used when getting user input.
  - A non-void needs to have at least one return statement.
- A function can have multiple return statements, however, the function will stop its execution as soon as the first return statement is reached.
  - It is good practice to only have ONE return statements and not using either *continue* or *break* statements.
- Can use 0 or more parameters - which can be either *value* or *reference* parameters
- If you want to return more than one value by a function you need to use a reference parameters.

```
char* function_that_returns_a_pointer_to_a_char(int some_int, double& ref_parameter);
   //This is the function 'declaration'
   //char* could also be an array of characters
int main()
{
   cout << function_that_returns_a_pointer_to_a_char(3);
            //This is the function call
      //prints out the address the first element of the array
}

char* function_that_returns_a_pointer_to_a_char(int some_int, double& ref_parameter);
{
   //This is the function definition and here goes the function body
}
```

Listing 5: An illustration of how to declare, define, and call a function

Value vs reference parameters

A value parameter makes a copy of the value of the variable (it is thus the same but not not the self-same). Hence the variables are unique to the scope which calls the function and the scope of the calling function. The two variables have two different memory addresses and changing the value of the one, does not change the value of the other.

If a variable is passed by reference, it is the self-same variable that is used in the new scope. Hence, it is only one variable with one memory location – changing the value of in the new function body, also changes its value in the scope of the calling scope. You pass a parameter by reference by appending the *&* to its type in both the function declaration, and definition (shown in above example).

Function overloading

Can use the same function name if the functions are distinguishable by the return type and the number of parameters.

3.2 Splitting programs into different files

Usually use . . .

- . . . a header file for the function *declaration*
- . . . and an implementation file for the function *definition*

Do . . .

- Make a header file that just includes the function *declarations* which should also include all comments about how the function is used.
- Make an implementation file that includes the function *definition*.
- Include the header file in both the implementation file **and** the file where the function is called
  - It is convention to delimit user defined file-names with double quotations (see example).

Example (1/3) - Header file

```
#ifndef NAME_OF_HEADER_FILE_H
#define NAME_OF_HEADER_FILE_H

   //Add commentary here
int declaration_of_a_function{double number, char name, int integer};

#endif
```

Listing 6: Example of splitting a function across files 1/3

Example (2/3) - Implementation file

```
#include "name_of_header_file.hpp"
   //Don't forget to include the header file

int declaration_of_a_function{double number, char name, int integer}
{
   //do something here
}
```

Listing 7: Example of splitting a function across files 2/3

Example (3/3) - File in which function is called

```cpp
#include "name_of_header_file.hpp"
    //Don't forget to include the header file

int main()
{
    int number = declaration_of_a_function(2.0, 'A', 5);
    return 0;
}
```

Listing 8: Example of splitting a function across files 3/3

4 Branch and loop statements

4.1 Loops

- We use the three loop statements: *for*, *while*, *do-while*.
- Any for-loop can be rewritten as while-loop and vv.
- A do-while-loop differs in that the statement in the braces are executed at least once (before the repetition condition is even checked) - they are useful to check if a user's keyboard input is of the correct format (can help to avoid writing duplicate lines

For loop syntax

```
for (int i = 0; i <= 100; i++)
{
   //for the values of i between 0 to 100 (starting at 0 and including 100),
   //do something
   //then increment i by 1
}
```

Listing 9: For loop syntax example

While loop syntax

```
int i = 0;
while (i <= 100)
{
   //if i < 100
   //do something
   i++;
   //then increment i by 1 (and re-check again if i < 100)
}
```

Listing 10: While loop syntax example

Do-while loop syntax

```
int i = 0, candidate_score;
do
{ //This part is at least executed once
   cout << Enter candidate score;
   cin >> candidate_score;
   if (candidate_score < 50)
   {
      cout << "Failed!";
   }

   i++;
}
while (i < 100);
```

Listing 11: Do-while loop syntax example

4.2 If, else if, else, and switch statements

General if, else statement

```
int score = 75;
if (score <=  100 && score >= 70)
{
   //do
}

else if (score >= 60 && score < 70)
{
   //do
}

else
{
   //do
}
```

Listing 12: If, else if, else syntax example

Switch statement

- The statements that are executed are those between the first label that matches the value of selector and the first break after this matching label.
- The *break* statements are optional but help in program clarity
- The selector can have any ordinal type (such as char or int) but cannot be a float or double.
- The default is optional but a good safety measure.

```
int score = 8;
switch (score)
{
   case 0:
   case 1:
   case 2:
   case 3:
   case 4:
      cout << "You are a failure!";
      break; //don't forget the break statement
   case 5:
      cout << "Marginally passed!";
      break;
   case 6:
   case 7:
   case 8:
   case 9:
   case 10:
      cout << "Pass!"   ;
      break;
   default:
      cout << "Incorrect score!"
      //No break statement needed here
}
```

Listing 13: Switch statement syntax example

4.3 One-liner if statement using the ternary operator

The two statements below are equivalent . . .

```
//Syntax
x = (condition) ? (value if true) : (value if false);

//Example
z = (x > y) ? z : y;

//The above is the same as writing ...
if (x > y)
{
   z = z;
}

else
{
   z = y;
}
```

Listing 14: One liner if using the ternary operator

4.4 Blocks and scoping

- Variables declared within a block have this block as their scope.
- While inside a block, the program will assume that the identifier refers to the inner variable.

- If the variable can't be found in the block, then it looks one or more scopes outside to find the variable.

4.5 Nested loops

To loops more clearly try to write them as functions (particularly for nested loops). The 3 examples below illustrate this.

Nested loops without making them subfunctions - not clear

```
//A program that outputs a multiplication table.
int main()
{
   int number;

   for (number = 1 ; number <= 10 ; number++)
   {
      int multiplier;
      for (multiplier = 1 ; multiplier <= 10 ; multiplier++)
      {
         cout << number << " x " << multiplier << " = ";
         cout << number * multiplier << "\n";
      }
      cout << "\n";
   }
   return 0;
}
```

Listing 15: Nested loops illustration 1/3

Nested loops as subfunction - clearer

```
//A program that outputs a multiplication table (using a function call for the nested loop)
void print_times_table(int value, int lower, int upper);

int main()
{
   int number;

   for (number = 1 ; number <= 10 ; number++)
   {
      print_times_table(number,1,10); //call to nested loop function
      cout << endl;
   }
   return 0;
}

void print_times_table(int value, int lower, int upper)
{
   int multiplier;
   for (multiplier = lower ; multiplier <= upper ; multiplier++)
   {
      cout << value << " x " << multiplier << " = ";
      cout << value * multiplier << endl;
   }
}
```

Listing 16: Nested loops illustration 2/3

Nested loops as subfunctions - very clear

```
//Same example but wrapping each loop into a function
void print_tables(int smallest, int largest);

void print_times_table(int value, int lower, int upper);

int main()
{
   print_tables(1,10);
   return 0;
}

void print_tables(int smallest, int largest)
{
   int number;
   for (number = smallest ; number <= largest ; number++)
   {
      print_times_table(number,1,10);
      cout << endl;
   }
}

void print_times_table(int value, int lower, int upper)
{
   int multiplier;
   for (multiplier = lower ; multiplier <= upper ; multiplier++)
   {
      cout << value << " x " << multiplier << " = ";
      cout << value * multiplier << endl;
   }
}
```

Listing 17: Nested loops illustration 3/3

5 Files and streams

- A file is just a linear sequence of *characters*.
- A stream is a channel on which data is passed from senders to receivers.
- Streams allow travel in only one direction (out from the program on an output stream or received from the program on an input stream.
- The standard input stream *cin* is connected to the keyboard and the standard output stream is connected to the monitor *cout*.
- To use streams we need to include the *fstream* header file.

5.1 Creating & opening streams and checking for failure

Creating a stream

Creating a stream is a bid like a variable declaration. The below creates an instance of the the class *ifstream*, called in_stream, and *ofstream*, called out_stream.

```
#include <fstream>

ifstream in_stream;
ofstream out_stream;
```

Listing 18: Creating a stream

Connecting and disconnecting streams to files

Use the member functions *open* and *close* - remember to always close. Also note that when opening an output stream, the contents of the file are deleted and is then ready for new input. Closing an output stream will also put an EOF marker at the end.

```
in_stream.open("filename.txt");
out_stream.open("other_filename.txt");

//Remember to always close ...
in_stream.close();
out_stream.close();
```

Listing 19: Connecting and disconnecting streams

To *append* to an output stream (thus not to overwrite), open it using the following line:

```
out_stream.open("some_file.txt', ios::app); //requires iostream
```

Checking for failure with file commands

Use the member function *.fail( )* – can do this with input and output streams. Place this statement immediately after opening a stream.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
   ifstream in;

   in.open("some_text.csv");
   if (in_stream.fail())
   {
      cout << "Sorry, the file was not opened!" << endl;
      exit(1);
   }
   ...
}
```

Listing 20: Checking for failure when opening a stream

5.2 Character input and output and checking for end-of-file

You can get or put characters one at a time using the member functions *get* and *put* (or also *putback*. Alternatively you can use the operators << or >> to read in or put out blocks of characters (useful for numbers).

Input/output with get, put, putback

The functions take a single argument of type *char* and always only handle one character at a time (where an empty space or new line etc. are also all chars.

```
char character;
in_stream.get(character);
   //gets the next character of the input file whose value is assigned to the character

out_stream.put('4'); //puts 4 as a type character into the output file
//OR
out_stream.put(character); //puts a character into the output file


in_stream.putback(character);
   //Puts the character implicitly back into the input file and repositions to this character
in_stream.putback('7');
   //Can also put another character pack to the input file
   //NOTE that both do not alter the actual input file (this is just implicit).
```

Listing 21: Illustration of character input/output using get and put

Input/output with the operators <<and >>

The problem we have is that some data types, such as int, double etc., have to be converted into character sequences before they can be written to a file, and these character sequences have to be converted back again when they are input. The operators <<and >>do some of this conversion automatically.

```
out_stream << 437 << ' ';
    //Here the characters '4' '3' '7' are output to the file and end with an empty character
```

For input streams you could also read in numbers with it - note that the <<skips over blank space (irrespective of the data type).

```
int number;
in_stream >> number;
```

You can also read in ever other item using the syntax below . . .

```
int number;
in_stream >> number >> number;
//if you use 'number' now in a loop it will be every other number ....
```

Here you see an example where you try to read in five (two-digit) numbers as either ints or chars. When reading them in as ints the count is 5 . . . using chars the count is 10 - in both cases whitespace is stripped and empty characters are not counted.

```
int main()
{
    char character;
    int number = 51;
    int count = 0;
    ofstream out_stream;
    ifstream in_stream1;   //Stream for counting integers
    ifstream in_stream2;   //Stream for counting characters

    //Create the file
    out_stream.open("Integers");
    for (count = 1 ; count <= 5 ; count++)
        out_stream << number++ << ' ';
    out_stream.close();

    //Count the integers
    in_stream1.open("Integers");
    count = 0;
    in_stream1 >> number;
    while (!in_stream1.fail())
    {
        count++;
        in_stream1 >> number;
    }
    in_stream1.close();
    cout << "There are " << count << " integers in the file,\n";

    //Count the characters
    in_stream2.open("Integers");
    count = 0;
    in_stream2 >> character;
    while (!in_stream2.fail())
    {
        count++;
        in_stream2 >> character;
    }
    in_stream2.close();
    cout << "represented using " << count << " characters.\n";
}
```

Listing 22: Illustration of character input/output using <<and >>

Checking for the end of and input file (eof)

Once an in-stream reaches the eof, no attempt should be made to read from the file, since the results will be unpredictable. To check if the eof is reached, the boolean expression *in_stream.eof()* would be true.

The below is an important example with a common loop structure. The code reads in a text file and outputs each character to the screen and to a copy file.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   char character;
   ifstream in_stream;
   ofstream out_stream;

   in_stream.open("file.cpp");
   out_stream.open("Copy_of_file");

   // ... could check for failure to open stream here ...

   in_stream.get(character); // !!! get char here first once
   while (!in_stream.eof())
   {  //while we are not at the end-of-file
      cout << character;
      out_stream.put(character);
      in_stream.get(character); // !!! get char here again
   }

   out_stream.close(); //always close
   in_stream.close(); //always close
}
```

Listing 23: Checking for EOF when using input streams

5.3 Streams as arguments to functions

Streams *must* be reference parameters in functions.

Below is a the same code, that reads a text-file and copies the content to a new file and outputs the contents to the monitor - but here a function *copy_to(...)* is used.

```cpp
#include <iostream>
#include <fstream>

using namespace std;

void copy_to(ifstream& in, ofstream& out);

int main()
{
   ifstream in_stream;
   ofstream out_stream;

   in_stream.open("4-6-1.cpp");
   out_stream.open("Copy_of_4");

   copy_to(in_stream, out_stream);

   out_stream.close();
   in_stream.close();

   return 0;
}

void copy_to(ifstream& in, ofstream& out)
{
   char character;
   in.get(character);
   while (!in.fail())
   {
      cout << character;
      out.put(character);
      in.get(character);
   }
}
```

Listing 24: Streams as arguments to functions - use as reference parameters

5.4 Formatting when using output streams

Member function - .precision(integer)

Using the *precision* member function will result in all numbers output on that stream to be shown to a certain digit (argument used for integer).

```cpp
out_stream.precision(2);
```

Flag member function - .setf(some argument here)

Flag	Meaning
ios::fixed	Floating point numbers are not written in scientific notation.
ios::scientific	Floating point numbers are not written in e-notation (if neither this or the above is set, the system will decide which to use).
ios::showpoint	A decimal point and trailing zeros are always shown for floats.
ios::showpos	A plus sign is output before positive integer values
ios::right	Item will be at right end of the space (default case) - thus right-justified
ios::left	Item will be left-justified

Table 3: Common flags for output streams

6 Arrays and c-strings

6.1 Basics of arrays

Syntax for declaring an array is ...

```
//Declaration:
<type of elements> <variable name>[integer value];

//Initialisation - v01:
int numbers[3] = {1, 10, 15};
//Initialisation - v02:
int numbers[] = {1, 10, 15};

//Some examples
int numbers[10];
   //an array that holds 10 integers

char characters[5];
   //an array that holds 5 characters

double* real_numbers[3];
   //an array that holds 3 pointers to variables of type double
   //... since an array is like a pointer variable, it could also be
   //... an array of 3 arrays that hold doubles

int** some_array[8];
   //an array that holds 8 pointers to pointers to variables of type int
   //is like an array of arrays of arrays
   // ... similar to a two-dimensional array
```

Could use a *typedef* if we declare the same array structure many times.

```
const int NUMBER = 6;
typedef int Hours_array[NUMBER]; //Hours_array is then like a type

Hours_array hours; //hours is now an array that holds 6 integers
Hours_array hours_w2; //hours_w2 is now an array that holds 6 integers
```

Listing 25: Using a typedef for arrays

Example of assigning the elements of an array with user input

Note that you cannot simply assign an array but only its elements (typically using a loop).

```
int main()
{
   int hours[6];
   int count;

   for (count = 1 ; count <= NO_OF_EMPLOYEES ; count++)
   {
      cout << "Enter hours for employee number " << count << ": ";
      cin >> hours[count - 1];
   }
}
```

Listing 26: Assigning each element of an array with user input

Note that C++ does not do range bound error checking - thus does not warn you when you try to access an element outside of the array (given an array is just the address of the first element of the array). To avoid the possibility of a range bound error you can make a condition in the loops that does not modify anything below or above the number of elements of the array …such as …

```
for (int i = 0; !in.eof() && i < MAX; i++)
{
   //Do something here
}
```

Range based for-loop for arrays

```
int array[] = {2, 4, 6, 8};
for (int x : array)
{
   cout << x;
}
```

Listing 27: Range based for-loop for arrays

This can also be based by reference …

```
int array[] = {2, 4, 6, 8};
for (int& x : array)
{
   x++;
}
   for (auto x : array)
{
   cout << x;
}
// will output 3579
```

Listing 28: Range based for-loop for arrays passed by reference

Example of a code that prints itself out (uses files and arrays of chars)

```
#include <fstream>

const int MAX = 1000;

int main()
{
   int count;
   char character;
   int file[MAX];
   ifstream in_stream;

   in_stream.open("6-1-2.cpp");
   in_stream.get(character);
   for (count = 0 ; ! in_stream.fail() && count < MAXIMUM_FILE_LENGTH ; count++)
   {
      file[count] = character;
      in_stream.get(character);
   }
   in_stream.close();

   while (count > 0) //count is now at last character of array
   {
      cout << file[--count];
   }
}
```

Listing 29: Code that reads itself in and prints out to screen using streams and arrays

6.2 Arrays as parameters of functions

You can simply use an array as a parameter in a function. Often you specify two parameters, one is the array and the second is the length of the array - see example. Array parameters are effectively reference parameters - no copy of the array is made and hence can be permanently changed within the function.

- In the declaration and definition of the function you need two parameters:
   1. The parameter of the array requires the correct base type (e.g. double) and requires empty squarebrackets (note if we would include a number in the brackets the compiler would ignore it).

2. Second parameter needs to be an 'int' which gives the size of the array.

- To call the function we need to declare the array of the correct base type and the arguments are passed as:
  1. Name of the array without the squarebrackets.
  2. Size (i.e. length of the array).

```
double average(int some_array[], int length_of_the_array)
{
   int total = 0;
   for (int count = 0; count < length_of_the_array; count++)
   {
      total += some_array[count];
   }
   return (total/length);
}
```

Listing 30: Array as parameter of a function

Therefore, we could just pass the array we wish to alter as parameter to a void function. In the below we permanently change the array *total*. Also with by using the *const* modifier on the first and second array we guarantee that they will not be changed by the function (this is a useful safety measure).

```
void add_lists(const int first[], const int second[], int total[], int length)
{
   int count;
   for (count = 0 ; count < length ; count++)
      total[count] = first[count] + second[count];
}
```

6.3 Function that returns an array

1. Since arrays as parameters to functions are effectively reference parameters (w/o explicitly writing the &, you could just make a void function and change the array within it.

2. Make a function that returns a dynamic pointer (an address on the heap).

3. Declare a local pointer variable in the frame where the function will be called and then make a call to a function that returns the pointer (where you use the pointer variable also as the argument).

Version 1 - using a call by reference parameter

```
//1) Returning and array of integers (using a call by reference param.)
//   - PARAMETER is a pointer called by reference
void return_int_array(int*& array)
{
   for (int i = 0; i < 3; i++)
   {
      array[i] = 1;
   }
```

```
}

int main()
{
	int* a;
	return_int_array(a);
}
```

Version 2 - returning a dynamic pointer

```
//2) Returning and a dynamic pointer
int* return_dynamic_pointer()
{
   int* int_array = new int[10];

   for (int i = 0; i < 10; i++)
   {
      int_array[i] = 1 + i;
   }
   return int_array;
}

int main()
{
   int* a;
   a = return_dynamic_pointer();
}
```

Version 3 - return a local pointer that was declared before in other scope

```
//3) Returning and array of integers (by returning a pointer, using also argument)
int* return_int_array(int* array)
{
   for (int i = 0; i < 3; i++)
   {
      array[i] = 1;
   }
   return array;
}

int main()
{
   int* a;
   a = return_int_array(a);
   for (int i = 0; i < 3; i++)
   {
      cout << a[i] << " ";
   }
}
```

6.4 Sorting arrays using the selection sort

The algorithm in words

1. At every current position of an array (and starting at 0) . . .

   - find the minimum
   - swap it with the current position
   - go to next position at array
   - end when the array ends

2. Where the minimum function should return the index of the min.

3. Use a swap function that swaps the two values of the indices (min and current).

```
void sort_array(int array[], int length);
int find_min_index(int array[], int position, int length);

int main()
{
    int array[3] = {2, 3, 1};
    sort_array(array, 3);
}

void sort_array(int array[], int length)
{
    for (int i = 0; i < length; i++)
    {
        int min = 0;
        min = find_min_index(array, i, length);

        swap(array[i], array[min]);
    }
}

int find_min_index(int array[], int position, int length)
{
    int min_index = position;

    for (int i = min_index; i < length; i++)
    {
        if (array[i] < array[min_index])
        {
            min_index = i;
        }
    }
    return min_index;
}

void swap_values(int current_index, int min_index)
{
    int temporary_value;

    temporary_value = current_index;
    current_index = min_index;
    min_index = temporary_value;
}
```

Listing 31: Selection sort

6.5 Sorting arrays using the bubble sort

Bubble sort used two loops where the first loops from the last element up until (but excluding) the first element. Let the current index (at first iteration the last index) be i.

In the second loop you always start at the beginning and compare the value at index 0 with the one at 0+1 ...0+1 with 0+2 etc. If the value at 0 is larger than the one at 1, then swap the two. Stop the loop when it has reached i from the first loop.

```
void bubbleSort(int array[], int length)
{
    int temp;
    for (int i = length - 1; i > 0; i--)
    {
        for (int j = 0; j < i; j++)
        {
            if (array[j] > array[j+1])
            {
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
}
```

Listing 32: Bubble sort

6.6 Multi dimensional arrays

Often used for screen bitmaps or n x m matrices of integers. Other examples can be a chess-board or sudoku. A multidimensional array is just an array of arrays.

Example of a 2D array

```
double two_D_array[3][6];
   //Is an array with 3 rows and 6 columns
   //Think of it as being a 1D array of size 3 whose base type is a one dimensional array
   //of doubles of size 6
```

2D as parameter of function

The length of the first dimension (rows) is not given inside the brackets, but the size of all other dimensions is given in the brackets - see below.

```
void some_function{int bitmap[][nr_of_columns], int nr_of_rows};
```

Looping over a 2D array

```
double two_D_array[3][6];
for (int row = 0; row < 3; row++)
{
   for (int column = 0; column < 6; column++)
   {
      // Do something
   }
}
```

Listing 33: Looping over each field of a 2D array

6.7 C-strings

C-string variables are arrays of chars with a sentinel string character *\0*. Hence, you always need to reflect the sentinel character in the length, when creating a c-string. Even if you have other characters after the sentinel character, these will be ignored by the string functions (shown below).

Declaration and assignment

Declare it just like an array of chars. Can also be initialised just like other arrays - but cannot be assigned and not compared using simple operators.

```
//Declare a c-string
char some_string[14];

//Initialise version 1
char some_string[14] = {'E', 'n', 't', 'e', 'r', ' ', 'a', 'g', 'e', ':', ' ', '\0'};

//Initialise version 2 (is equivalent)
char some_string[14] = "Enter age: ";

//Initialise version 3 (omitting the length, makes an array that is just large enough)
char some_string[] = "Enter age: ";
```

When looping through a c-string it is good to add the following safety measures ...

```
char my_name = "Alex";
for (int i = 0; (i != '\0' && i < SIZE); i++)
{
   //do something
}
```

Predefined functions

The below functions are contained in the *c-string* library.

String input using *getline(...)*

Can use the getline function for user input on the standard input-stream and all other input-streams. Getting an input with the operator >>assumes the input is complete when the first

Function	**Return type**	**Description**
strcpy(a_string, b_string)	void	Copies the string value stored in *b_string* onto *a_string*. But beware, that the a string is large enough to contain the b string. You can use also a text like *"Some text here!!!"* instead of *b_string*.
strncpy(a_string, b_string, limit)	void	Same as above but copies at most as many characters as specified with the int *limit*.
strlen(a_string)	int	Returns the length of the string excluding the the sentinel character (not counted).
strcmp(a_string, b_string)	int	Returns 0 if the two string arguments are the same, a negative number if a string is smaller than b string, and positive if it is the other way round.
strcat(a_string, b_string)	void	Concatenates the b string onto the a string. Beware that the a string is large enough to contain both strings.
strncat(a_string, b_string, limit)	void	Same as above but appends at most *limit* characters.

Table 4: C-string functions in cstring library header

empty space character is encountered. For example, cin >>Rob Miller, will only get 'Rob'. However, using cin.getline(some_string, 80), will allow the user to type in a string of up to 79 characters. The function *getline* reads up to the sentinel character.

```
ifstream in_stream;
in_stream.getline(a_string, 80);
```

The example below illustrates the use of c-string functions and getline.

```
#include <cstring>

const int MAXIMUM_LENGTH = 80;

int main()
{
   char first_string[MAXIMUM_LENGTH];
   char second_string[MAXIMUM_LENGTH];

   cout << "Enter first string: ";
   cin.getline(first_string,MAXIMUM_LENGTH);
   cout << "Enter second string: ";
   cin.getline(second_string,MAXIMUM_LENGTH);

   cout << "Before copying the strings were ";
   if (strcmp(first_string,second_string))
      cout << "not ";
   cout << "the same.\n";

   strcpy(first_string,second_string);

   cout << "After copying the strings were ";
   if (strcmp(first_string,second_string))
      cout << "not ";
   cout << "the same.\n";

   strcat(first_string,second_string);

   cout << "After concatenating, the first string is: " << first_string;
}
```

Listing 34: Illustration of c-string functions an getline(. . .)

C-string as parameter to a function

Same logic as for arrays but particular care has to be taken not to overwrite the sentinel character.

C-string to number conversion

- Use the functions *atoi*, *atol*, and *atof* to convert a c-string variable to an integer, long, or double, respectively.
- These functions require the header-file *cstdlib*.
- if the argument is such that the conversion cannot be made then the function returns zero.

```
//Example
int x = atoi("676");
double y = atof("#546.3");
```

Good illustrative example of manipulating and working with c-strings (book)

- Uses call by reference parameters.
- Input using get and >>.

  - Shows that if you prompt user to type something and after that (before using cin >>or get you call a new void function - then in stackframe of new function you use cin >>etc. - then whatever you type in will be assigned to a variable within this new stackframe.

- Illustrates how to get rid characters that are no digits within a c-string variable (uses *isdigit*).

- Illustrates design of function that prompts user to change his input up until the user says he's happy.

```
// MAIN PART AND FUNCTION DECLARATIONS
#include <iostream>
#include <cstdlib>
#include <cctype>

using namespace std;

void get_user_input(int& input_number);
//gets user to input a number

void read_and_clean(int& n);
// Reads a line, discards all symbols that are no digis
// Converts the string to an integer and sets n equal to the value
// of this integer

void new_line();
//Discards all the input remaining on the current input line
//Also discards the '\n' at the end of the line

//---------------
int main()
{
    int n;

    get_user_input(n);
    cout << "Final value read in = " << n << endl;

    return 0;
}
```

Listing 35: Long example of using c-strings 1/4

```
// FUNCTION: get_user_input
void get_user_input(int& input_number)
{
    char ans;
    do
    {
        cout << "Enter an integer and press RETURN: ";

        read_and_clean(input_number);

        cout << "You entered: " << input_number;
        cout << " ... fine? ";
        cin >> ans;
        cin >> ans;
        new_line();

    } while ( (ans != 'y') && (ans != 'Y') );

}
```

Listing 36: Long example of using c-strings 2/4

```
//FUNCTION: read_and_clean
void read_and_clean(int& n)
{
    const int SIZE = 6;
    char digit_string[SIZE];

    char next;
    int index = 0;

    cin.get(next); // reads in next 'char' even if its a whitespace

    while ( next != '\n')
    {
        if ((isdigit(next)) && (index < (SIZE - 1)))
        {
            digit_string[index] = next;
            index++;
        }
        cin.get(next);
    }
    digit_string[index] = '\0';
    n = atoi(digit_string);
}
```

Listing 37: Long example of using c-strings 3/4

```
// FUNCTION new line
void new_line()
{
    char symbol;
    do
    {
        cin.get(symbol);
    } while (symbol != '\n');
}
```

Listing 38: Long example of using c-strings 4/4

7 The standard *string* class

7.1 Introduction to the string class

- The class *string* is defined in the library *string* and the definitions are placed in the std namespace.
- Can assign a string variable with = (note that with c-strings we cannot assign but only initialise).
- Can concatenate using +.

Simple example

```
#include <string> //needs to be included
using namespace std; //required

int main()
{
   string phrase; // initialised to empty string w. default constructor
   string adjective("fried"), noun("ants"); // initialise two string

   //Use overloaded operators '=' and '+' to assign a new string value to phrase
   phrase = "I love " + adjective + " " + noun << endl;
}
```

Listing 39: Basic handling of string objects

7.2 I/O with the string class

I/O is very similar to c-strings but the *geline* function differs.

- Use << on output streams.
- Use >> on input streams (as before you only read in a string up to, and excluding, the next whitespace).
- Use function *getline* to input an entire line of text into a string object.
  - Syntax differs from c-string variables: *getline(in_stream, string)*.
  - getline(...) stops reading when it encounters the end-of-line marker \n.

```
string greeting("Hello"), response, next_word;
cout << greeting <<endl;

getline(cin, response); // gets entire line
cin >> next_word; // only gets text up to next white-space
```

Listing 40: I/O with string objects

- If you want to read in up to a certain character use the following syntax ...
  - *getline(inStream, stringToWriteTo, charWhereToStop);*
  - e.g. stop reading when '?' is encountered: *getline(cin, question, '?');*

Short illustrative example

```
#include <string>

void new_line();

int main()
{
   string first_name, last_name, record_name;
   string motto = "Your records are our records.";

   cout << "Enter your first and last name:\n";
   cin >> first_name >> last_name; // Reads in first and last name separated by whitespace

   new_line(); // call to function 'new_line'

   //String concatenation and assignment works with overloaded operators
   record_name = last_name + ", " + first_name;

   cout << "Please suggest a better motto:\n";
   getline(cin, motto); // read in new string from keyboard and assign it to motto
}

void new_line()
{
   char next_char; //declare a character variable
   do
       {
      cin.get(next_char); // gets some character until the character
      //is the new line character
   } while (next_char != '\n');
}
```

Listing 41: Short illustrative example of handling string objects

Note that when mixing >>and getline(...), you may encounter the problem that >>only leaves a '\n' in the line. When getline(...) comes in, it may only see the '\n' and then stop. To avoid this problem to happen us either of the two ...

- The new_line function from the previous example.
- The function *ignore* from the iostream library
  - cin.ignore(1000, '\n');
  - Will read and discard the entire rest of the line up to and including the '\n' ...
  - or until it discards 1000 characters if it does not find the end of the line after 1000 characters.

7.3 Member functions of the class string

Member function *at(integer)*

Checks if *integer* evaluates to an illegal index.

The following table is an illustration of some commonly used functions of the string class.

Function	Description
Accessors	
str[i]	Returns read/write reference to character in *str* at index i.
str.at(i)	Returns read/write reference to character in str at index i. Same as str[i], but this version checks for illegal index.
str.substr(position, length)	Returns the substring of the calling object starting at *position* and having *length* characters.
str.length()	Returns the length of str.
Assignment/Modifiers	
str1 = str2;	Initialises str1 to str2's data.
str1 += str2;	Character data of str2 is concatenated to the end of str1.
str.emphy()	Returns *true* if str is an empty string and false otherwise.
str1 + str2	Returns a string that has str2's data concatenated to the end of str1's data.
str.insert(pos, str2);	Inserts str2 into str beginning at position pos.
str.erase(pos, length);	Removes substring of size length, starting at position pos.
Comparison	
str1 == str2 OR str1 != str2	Compare for equality or inequality; returns a bool value.
str1 ¡ str2 OR str1 ¿= str2	Lexocographical comparison.
Finds	
str.find(str1)	Returns index of the first occurrence of str1 in str. If str1 is *not* found, then the special value *string::npos* is returned.
str.find(str1, pos)	Returns index of the first occurence of string str1 in str; the search starts at position *pos*.
str.find_first_of(str1, pos)	Returns the index of the first instance in str of any character in str1, starting the search at position *pos*.
str.find_first_not_of(str1, pos)	Returns the index of the first instance in str of any character not in str1, starting the search at position *pos*.

Table 5: Commonly used member functions of string class

7.4 Converting from string objects

There is no automatic conversion of string objects to c-strings. We must explicitly perform the type conversion which can be done with the string member function *c_str()*. Also given the = operator does not work on c-strings we need to copy the value of a string object to a c-string with *strcpy*.

```
//For example
strcpy(c_string, string_object.c_str());
```

Converting to numbers

- Use *stof*, *stod*, *stoi*, or *stol* to convert a string to a float, double, int, or long.
- Use *to_string* to convert a number type to a string.

```
//For example
int i;
string s;

i = stoi("35"); // converts string "35" to an integer 35
s = to_string(2.5 * 2); // converts 5.0 to a tring
```

8 Vectors

Vectors have the same purpose as arrays but they can grow and shrink. Vectors are part of the Standard Template Library (STL). Requires use of the *vector* library.

```
#include <vector>
```

8.1 Vector basics

Declaration and basic member functions

```
//Syntax
vector<base_type> v_name;
//Example of a vector that stores ints
vector<int> v_name;
```

- The example above invokes a call to the default constructor for the class vector<int>which creates a vector object that is empty.
- Vector elements can be accessed with the square brackets (once they were already assigned a value once).
- You cannot initialise vector elements using the square brackets (only reassign).
- You initialise elements using the member functino *pushback*.

```
vector<double> sample_v;
sample_v.pushback(23.1);
```

- Can initialise a vector with ...

```
vector<double> sample_v = {12.1, 14.34, 15.2};
```

- Access the length of a vector using the *size* member function (which returns an unsigned int as type).
- Note that whenever a vector runs out of capacity, it is automatically increased.
  - Note that increasing the capacity in small chunks is inefficient and thus by default whenever the capacity needs to increase it doubles.
  - If you need to manage memory you may wish to change this.
  - In particular you may need to use the member functions *reserve* or *resize*.

```
unsigned int integer = sample_v.size();
```

9 Pointers and references

9.1 Overview of pointers and references

References

Note that references are typically safer to use than pointers because they cannot be reassigned.

- Is an alias for something (i.e. another name for an already existing variable.
- Need to be initialised and cannot be reassigned.
- All performed on the reference variable also happens to the original.
- Declaration with the ampersand, *&* ...

```
int& referance_v_as_alias_of = my_name;
```

- The double meaning of *&* ...
  - In a declaration (i.e. at initialisation or as a function parameter) it is a reference parameter.
  - Not in a declaration it is an address operator.

Pointers

- Stores a memory address.
- Pointers are reassignable.
- Pointer has their own memory address while references do not (i.e. you can have a pointer to a pointer but not a reference of a reference).
- Declaration ...

```
int* pointer_name = &some_variable;

//OR

int* pointer_name;
pointer_name = &some_variable;

//OR using a typedef
typedef int* int_pointer;
int_pointer a, b, c; //declares 3 pointer variables
```

Overview of fetures of pointers, references, and const pointers

Concept	nullable	(re)assignable	arithmetic
Pointer	yes	yes	yes
const Pointer	yes	no	yes
Reference	no	no	no

Table 6: Overview of features of pointers, const pointers, and references

Note ((think) that if you call a function with an & next to the argument, then the declaration of this function requires the corresponding parameter to have a (i.e. to be a pointer) . . . but check this again.

9.2 Illustration of when to use a reference, const ref, pointer, or pass by value

Parameter is *not* optional

Parameter is optional

9.3 Pointers

The *new* operatator

- Used to create dynamic variables (have no identifiers to serve as their variable names).
- Dynamic variables are stored on the heap.
- Refer to the values of these variables using dereferenced pointers.

```
p1 = new int; //creates new dynamic variable (has no variable name)
cin >> *p1; // assign a number to the variable
*p1 = *p1 + 7; // add 7 to the variable
```

The *delete* operatator - return memory to the heap

- When deleting, the variable is then undefined - ensure to check if a pointer actually points to something before applying the dereferencing operator.

```
// Make the following check before trying to assign a value to a dangling pointer
delete pointer;
pointer = NULL; //once it is NULL you cannot assign a value to the
//dereferenced pointer

if (pointer != NULL){
    *pointer = 40;
} else {
    cout << "Dangling pointer";
    exit(1);
}
```

Safety measure if there is not sufficient memory on heap to create a dynamic variable

- C++ would throw an exception called std::bad_alloc.
- You can catch the error in two ways:
  1. *try ... catch* or
  2. Call to *new (nowthrow)* and then set the corresponding pointer to NULL in case of allocation failure.

1. Try catch

```
try
{
    ptr_a = new int;
}
catch
{
    cout << "Sorry, ran out of memory.";
    ptr_a = NULL;
    exit(1);
}
```

Listing 42: Check that there is enough memory to create a dynamic pointer - v1

2. Call to nothrow

```
ptr_a = new (nothrow) int;
if (ptr_a == NULL)
{
    cout << "Sorry ran out of memory."
}
```

Listing 43: Check that there is enough memory to create a dynamic pointer - v2

But it would be more elegant to wrap this around a function (using a call by reference parameter).

```
int main()
{
    char* c_pointer;
    assign_new_int(c_pointer);
}

// In function definition
void assign_new_int(char*& some_pointer)
{ // function parameter is a pointer that is called by reference?!
    some_pointer = new (nothrow) char;
    if (some_pointer == NULL)
    {
        cout << "Bad allocation."
        exit(1);
    }
}
```

9.4 Dynamic arrays

Note that you can assign an array to a pointer variable (provided that they have the same base type) - but the other way round is illegal.

```
int a[10]; // declare array
int* pointer; // declare a pointer

pointer = a; // is legal

//BUT the following is NOT legal
a = pointer;
```

Declare and delete a dynamic array

```
double* pointer;
pointer = new double[10];

// When deleting a dynamic array do NOT forget the '[]'
delete [] pointer;
```

Can also use pointer arithmetic on arrays.

```
int* d;

d = new int[10];

// The two below are equivalent:
d[5];
*(d+5);
```

Summary on using a dynamic array

- Define a pointer type (optional): a type for a pointer to variables of the same type as the elements of the array (e.g. typedef double* double_array_pointer;).
- Declare a pointer variable: that will point to the dynamic array in memory and will serve as the name of the dynamic array (e.g. double_array_pointer a;)
- Call new: to create a dynamic array (e.g. a = new double[array_size];)
- Use like ordinary array: Note that the pointer variable should not have any other pointer value assigned to it, but should be used like an array variable only (otherwise that may confuse the system).
- Call delete: When your program is finished with the dynamic variable, use *delete* and empty [] along with the pointer variable to return memory to the heap (e.g. delete [] a;).

```
/* This program illustrates the use of dynamic arrays. It prompts
   the user for a list of integers, then outputs their average to
   the screen. */

#include <iostream>
#include <cstdlib>
using namespace std;

//Function to compute the average value of the integer
//elements in an array "list[]" of length "length"
float average(int list[], int length);

int main()
{
   int no_of_integers, *number_ptr;

   cout << "Enter number of integers in the list: ";
   cin >> no_of_integers;

   number_ptr = new (nothrow) int[no_of_integers];
   if (number_ptr == NULL)
   {
      cout << "Sorry, ran out of memory.\n";
      exit(1);
   }

   cout << "type in " << no_of_integers;
   cout << " integers separated by spaces:\n";
   for (int count = 0 ; count < no_of_integers ; count++)
      cin >> number_ptr[count];
   cout << "Average: " << average(number_ptr,no_of_integers) << "\n";

   delete [] number_ptr;
}

float average(int list[], int length)
{
   float total = 0;
   int count;
   for (count = 0 ; count < length ; count++)
      total += float(list[count]);
   return (total / length);
}
```

Listing 44: Illustration of use of dynamic arrays

10 Recursion

10.1 The basic idea

A function is recursive, if the function definition includes a call to itself. A familiar mathematical example of a recursive function is the factorial function. The definition includes a base case (the definition of 0!) and a recursive part. The recursive call to the function always needs to be embedded in a branch statement with at least one non-recursive branch (i.e. the base case, to avoid infinite loop).

C++ arranges the memory spaces needed for each function call in a stack. The memory area for each new call is placed on the top of the stack, and then taken off again when the execution of the call is completed. Basically if you have 3 calls you start with the first call, then on top comes the 2nd and the 3d, then the second and then the first again (LIFO structure).

Note that any function that can be defined recursively can also be defined iteratively. Because of extra stack manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts - function calls are expensive. But often recursive definitions are easier to read.

Recursive functions are often useful when manipulating recursive data structures (e.g. a node in a linked list).

A basic example

The following inputs a series of characters from the keyboard, terminated with a full-stop character, and then prints it backward on the screen.

```
void print_backwards();

int main()
{
   print_backwards();
   cout << endl;

   return 0;
}

void print_backwards()
{
   char character;

   cout << "Enter a character ('.' to end program): ";
   cin >> character;
   if (character != '.')
   {
      print_backwards();
      cout << character;
   }
   else
   {
      ;
   }
}
```

Listing 45: Basic example of a recursive function

10.2 Three more examples

Factorial

```
int factorial(int number)
{
   if (number < 0)
   {
      cout << "\nError - negative argument to factorial\n";
      exit(1);
   }
   else if (number == 0)
      return 1;
   else
      return (number * factorial(number - 1));
}
```

Listing 46: Factorial as a recursive function

Power

```
float raised_to_power(float number, int power)
{
   if (power < 0)
   {
      cout << "\nError - can't raise to a negative power\n";
      exit(1);
   }
   else if (power == 0)
      return (1.0);
   else
      return (number * raised_to_power(number, power - 1));
}
```

Listing 47: Recursive power function

Sum of first n elements of an array

```
const int NO_OF_ELEMENTS = 10;
int sum_of(int a[], int n);

int main()
{
   int list[NO_OF_ELEMENTS];
   int count;
   int no_of_elements_to_sum = 5;

   for (count = 0 ; count < NO_OF_ELEMENTS ; count++)
   {
      cout << "Enter value of element " << count << ": ";
      cin >> list[count];
   }

   cout << "\nHow many elements do you want to add up? ";
   cin >> no_of_elements_to_sum;

   cout << "\n\n";

   cout << "The sum of the first " << no_of_elements_to_sum << " element";
   if (no_of_elements_to_sum > 1)
      cout << "s";
   cout << " is " << sum_of(list, no_of_elements_to_sum) << ".\n";

   return 0;
}

int sum_of(int a[], int n)
{
   if (n < 1 || n > NO_OF_ELEMENTS)
   {
      cout << "\nError - can only sum 1 to ";
      cout << NO_OF_ELEMENTS << " elements\n";
      exit(1);
   }
   else if (n == 1)
      return a[0];
   else
      return (a[n-1] + sum_of(a,n-1));
}
```

Listing 48: Recursive sum of first n elements of an array

10.3 Quick sort - a recursive procedure for sorting

Can sort in ascending or descending way. Logic is as follows: you select a *pivot* (which can be any number of the array) and put all numbers smaller than the pivot to the left and larger than the pivot to the right. After that you treat the numbers to the left and right of the pivot as two new independent arrays for which two pivots are selected - then sorted. Repeat this process until the numbers are all single independent arrays.

Suppose you have the following 11 digit int array.

$$14, 3, 2, 11, 5, 8, 0, 2, 9, 4, 20$$

Steps:

- Select a pivot with the index *(first index + last index) / 2* note that you select the value here and not the index.
- Identify a left arrow at index 0.
- identify a right arrow at end index of the array.
- Now starting from the right, the right arrow is moved to the left until until a value less-than or equal to the pivot is encountered - (i.e. the '4' in the example).
- Similarly the left arrow is moved to the right until a value greater than or equal to the pivot is encountered (i.e. already the 14 in the example).
- Now swap the two values at the indices where the left and right arrows are and we have ...

$$4, 3, 2, 11, 5, 8, 0, 2, 9, 14, 20$$

- Now move the right arrow left again and the left arrow right again (2 and 11).
- Exchange the values and we have ...

$$4, 3, 2, 2, 5, 8, 0, 11, 9, 14, 20$$

- This part only stops when the condition left arrow >right arrow becomes true.
- Note that it is acceptable to exchange the pivot because you swap the values and not the index.

Below is the C++ procedure for a recursive quick sort algorithm.

```
//Function declarations and other headers go here...
int main()
{
    int list[5] = {14, 3, 2, 11, 5};

    //Call to quick_sort ...
    quick_sort(list, 0, 5 -1);
}

void quick_sort(int list[], int left, int right)
{
    int pivot, left_arrow, right_arrow;
    left_arrow = left;
    right_arrow = right;
    pivot = list[(left + right)/2];

    do
    {
        //Move the right arrow to left until the value is smaller than or equal to the pivot
        while (list[right_arrow] > pivot)
        { right_arrow--; }

        //Move the left arrow to right until the value is larger than or equal to the pivot
        while (list[left_arrow] < pivot)
        { left_arrow++; }

        if (left_arrow <= right_arrow)
        {
            swap(list[left_arrow], list[right_arrow]);
            left_arrow++;
            right_arrow--;
        }
    }
    while (right_arrow >= left_arrow);

    //Stopping condition ... for each call to the quick sort need to have two more
    //calls to the quick sort (as the array is splitted in two)
    if (left < right_arrow)
    {
        quick_sort(list, left, right_arrow);
    }
    if (left_arrow < right)
    {
        quick_sort(list, left_arrow, right);
    }
}

void swap(int& first, int& second)
{
   //use a temp to swap the values
}
```

Listing 49: Recursive quick sort algorithm

11 Structs

11.1 Struct basics

Structs are like classes, used to create own datatypes. Structs and classes have member variables and member functions. The difference is that all member variables for structs are public whereas for classes they are private by default.

```
struct CDAccount
{
   double balance;
   double interestRate;
   int term;
}; //don't forget the semicolon here
```

Listing 50: Struct definition

In the above, *CDAccount* is a new *type* with three member variables. To make an instance of the object and access the member variables, type ...

```
//Create instances
CDAccount myAccount, yourAccount;

//Access member variables
myAccount.balance = 500.00;
int someInt = yourAccount.term;
```

Note that you can assign structure values using the assignment operator (see below). In this case all of the member values of myAccount are set equal to those of yourAccount (this can pose a big problem if some of the member variables are pointers a/o dynamic variables are used).

```
myAccount = yourAccount;
```

11.2 Structures and functions

- Can be a call by value or call by reference parameter.
- Can also be the type returned by a function (as in example below).

```
CDAccount a_function(double theBalance, double theRate, int theTerm)
{
   //Local variable of type CDAccount is used to build up a complete
   //structure value
   CDAccount temp;

   temp.balance = theBalance;
   temp.interestRate = theRate;
   temp.term = theTerm;

   return temp;
}
```

Listing 51: Function that returns a struct (a temporary instance)

Now make an actual instance and call the function on it . . .

```
CDAccount = new_account;
new_account = a_function(100.0, 5.1, 11);
```

11.3 Structures whose members are struct instances

In the below the struct *PersonInfo* contains an instance of a struct called *Date*.

```
struct Date
{
   int month;
   int day;
   int year;
};

struct PersonInfo
{
   double height;
   double weight;
   int* int_pointer;
   Date birthday; //this is the instance birthday which is of type 'Date'
};
```

Listing 52: Struct that contains a struct instance

You can access the member variables of *birthday* using . . .

```
//Assume you have an instance of 'PersonInfo' called person1
PersonInfo person1; //has some values assigned to members

//Access the height
person1.height

//Access the birth month
person1.birthday.month

//Access the the address stored in a pointer
person1.int_pointer

//Access the the value stored behind the pointer
// !!! --> NEED TO CHECK THIS
```

11.4 Basic initialisation of a struct

At declaration we can write the required numbers in curly brackets. The numbers need to be in the same orders as the member variables in the struct definition.

```
Date today = {12, 31, 2004};
```

11.5 A pointer variable to an object as instance

You can also declare a pointer to an object as for other more general types. You can declare and access the data members as per below ...

```
//Declare
Date* pointer_to_a_date;

//Access (the two are equivalent
(*pointer_to_date).month = 2; //need the brackets because the '.' has precedence over the '*'
pointer_to_date->month = 2; //this is the preferred syntax
```

12 Linked lists and binary trees

You implement linked lists and binary trees usually using struct or also class objects. These objects have various data members – one of these members is a pointer to the next node.

12.1 Basics of linked lists

The linked list is then a list of nodes where you have one pointer pointing to the first node, and each node has a member pointer that points to the next node. Finally the member pointer of the last node points to NULL.

Particular care needs to be taken not to lose nodes so not to have memory leaks.

12.2 Basics steps to build linked lists

Set up struct and first node

1. Create struct object.
2. Declare a head pointer that points to the first node.
3. Assign this variable to a *dynamic node* using the new operator.
4. Give this first node values - also setting the pointer to the next node to NULL.

```
//The structure of a node
struct Node
{
   int data;
   Node* link; //pointer to next node
};

//Headpointer
Node* head;
head = new Node;

//Assign values to the data members
head->data = 3;
head->link = nullptr;
```

Listing 53: Set up a linked list

Insert a node at the beginning

The steps such a function should perform are as follows ...

1. Create a new dynamic node (this is referred to using the dereferencing operator).
2. Assign the values to the data members of the node.
3. Make the pointer of this new node point to the (current) *head* node.
   - Set the variable equal to the value of the *head pointer*.
4. Make head pointer point to the new node.

```
void headInsert(Node*& head, int theNumber)
{ //head is a pointer that is used as a call by reference parameter ...

   //Declare the new node
   Node* temporary_pointer;
   temporary_pointer = new Node;

   //Assign member variables
   temporary_pointer->data = theNumber;
   temporary_pointer->link = head; //now the pointer in the new node
      //points to the node that was previously at start

   //Make the head pointer point to the new node
   head = temporary_pointer;
}
```

Listing 54: Add a node at the beginning of a linked list (as function)

Empty list

To indicate that linked list is empty, set the head pointer to NULL. When creating functions that manipulate linked lists, also check if they work when the list is empty.

12.3 Using the pointer in the nodes as iterators

```
Node* iter; //a pointer to variables of type 'Node'

for (iter = head; iter != NULL; iter = iter->link)
{
   //For every node (i.e. for every pointer that points
   //to a node) ...
   //... until the pointer is a NULL (i.e. at end of linked list) ...

   //Do something

   //And then reset the pointer to the address in that node
}
```

Listing 55: Using pointer as the iterator in a linked list

12.4 Searching a linked list

Use the following function definition that returns the pointer to the node that contains the target value we are looking for.

```
//Use same node definition as in previous example
#include <cstdef>

Node* search(Node* head, int target)
{
   Node* here = head; //here now points to start

   //If the list is empty
   if (here == nullptr)
   {
      return nullptr;
   }

   //If list includes at least one node
   else
   {
      while (here->data != target && here->link != nullptr)
      {
         here = here->link;
      }
      if (here->data == target)
      {
         return here;
      }
      else
      {
         return nullptr;
      }
   }
}
```

Listing 56: Searching a linked list

12.5 Inserting and removing nodes within a list

Insert after a specific node

The code below works for inserting at end and middle but not at the beginning of the linked list.

```
void insert(Node* after_me, int the_number)
{
   Node* temporary_ptr;
   temporary_ptr = new Node;

   temporary_ptr->data = the_number;
   temporary_ptr->link = after_me->link;

   after_me->link = temporary_ptr;
}
```

Listing 57: Insert a node after a specific node within a linked list

Removing a node

!!! To be checked from Will's notes !!!

12.6 Double linked list

In such lists, nodes have two links – one points to the next node and the other points to the previous node. Also we not only have a *head* pointer that points to the first node, but we also have a *rear* pointer that points to the last node.

```
struct Node
{
   int data;
   Node* forward_link;
   Node* backward_link;
};
Node* head;
Node* rear;
```

Listing 58: Structure of a double linked list

12.7 Intro to binary trees

Binary trees are not linked lists per se but they use the same basic concept of being nodes that consist pointers that point to other nodes. For trees, you have a root pointer that points to the first (the root) node of the tree. Each node then has one pointer pointing to the left and one pointing to the right (i.e. at each level the tree is doubling).

```
struct TreeNode;
{
   int data;
   TreeNode* leftLink;
   TreeNode* rightLink;
};
```

Listing 59: Basic structure of a binary tree

12.8 Routines from lecture slides

Set up node

```
struct Node
{
   char word[80];
   Node* next;
};
```

Initialise linked list

Sets up an empty list.

```
void initialise_list(Node*& front, Node*& rear)
{
   front = rear = nullptr;
}
```

Create new node

Returns a pointer to a new node and the pointer within the node is set to NULL.

```
Node* create_new_node(const char* constents)
{
   Node* new_node = new (nothrow) Node;
   assert(new_node != NULL);

   strcpy(new_node->word, contents);
   new_node->next = NULL;
   return new_node;
}
```

Count items in list

Returns an integer that is the number of items in list.

```
int count_items(Node* front)
{
   int length = 0;
   for (Node* current = front; current; current = current->next)
   {
      length++;
   }
   return length;
}
```

Add a node at front of the list

Note that if *front* is NULL the boolean expression *if (front)* would evaluate to false (given false is 0). Hence the boolean expression *if (!front)* tests if the list is empty (i.e. when front equals the NULL).

```
void add_to_front(Node*& front, Node*& rear, const char* word)
{
   Node* item = create_new_node(word);

   //Case if list is empty
   if (!front)
   {
      front = rear = item;
      return;
   }

   //Default case (if there are items in the list)
   item->next = front;
   front = item;
}
```

Add a node at rear of the list

```
void add_to_rear(Node* front, Node*& rear, const char* word)
{
```

```
    Node* item = create_new_node(word);

    //Empty list
    if (!front)
    {
        front = rear = item;
        return;
    }

    //Default case
    rear->next = item;
    rear = item;
}
```

Add a node in a sorted list

```
void add_sorted(Node*& front, Node*& rear, const char* new_word)
{
    Node* p = front;
    Node* q = rear;

    //Find the node after which you wish to insert
        //For every pointer starting at the beginning...
        //... until is NULL & the word in the node is alphabetically smaller
        //than the word we want to insert as node.
    for (p = front; p && (strcmp(p->word, new_word) < 0); p = p->next)
    {
        q = p;
    }
    //q now points to the node we wish to insert AFTER
    //p now points to the node we wish to insert BEFORE

    //In case we need to add it to the front (i.e. the new node comes alphabetically first)
    if (q == nullptr)
    {
        add_to_front(front, rear, new_word);
        return;
    }

    //In case we need to add it to the rear (i.e. the new node comes alphabetically last)
    if (p == nullptr)
    {
        add_to_rear(front, rear, now_word);
        return;
    }

    //Otherwise create the new node and insert it after q and before p
    new_node->next = p;
    q->next = new_node;
}
```

Print the list to the screen

```
void print_list(Node* front)
{
    for (Node* current = front; current; current = current->next)
```

```
    {
        cout << "[" << current->word << "]" << endl;
    }
    cout << endl;
}
```

Print one element of the list

```
void print_element(Node* front, int item_number)
{
    int count = 0;
    Node* current;

    for (current = front; current && count < item_number; current = current->next)
    {
        count++;
    }

    if (current)
    {
        cout << "The current word is: " << current->word << endl;
    }
}
```

Print list backwards (recursively)

```
void print_list_backwards(Node* front)
{
    //Base case
    if (!front)
    {
        return;
    }

    //Recursive step
    print_list_backwards(front->next);
    cout << "The word is " << front->word << endl;
}
```

Print list backwards (iteratively)

```
void print_list_backwards(Node* front)
{
    int count = count_items(front);
    for (int target = count - 1; target >= 0; target--)
    {
        print_element(front, target);
    }
    cout << endl;
}
```

Delete a node containing the target string

```
void delete_node(Node*& front, Node*& rear, const char* target)
{
   Node* p = front;
   Node* q = rear;

   for (p = front; p && strcmp(p->word, target); p = p->next)
   {
      q = p;
   }
   //q now points to the node BEFORE the one we want to delete
   //p points to the node we wish to delete

   if (p == nullptr)
   {
      cout << "Could not find the word " << target << endl;
      return;
   }

   // !!! Store the target's 'next' pointer
   Node* link = p->next;
   delete p;

   //Case if target is the front (i.e. q is NULL)
   if (p == front)
   {
      front = link;
      if (front == nullptr)
      {
         rear = nullptr;
      }
      return;
   }

   //Update the next pointer of the node before the one being deleted
   q->next = link;

   //Handle case that the target is the rear
   if (p == rear)
   {
      rear = q;
   }
}
```

Delete all nodes in the list

```
void delete_list(Node*& front, Node*& rear)
{
   while (front != nullptr)
   {
      Node* link = front->next;
      delete front;
      front = link;
   }
   rear = nullptr;
```

```

}
```

12.9 Binary trees – lecture routines

Note for the below you need to declare a root node in the main function (clearly after the struct for the nodes has been declared).

```
Node* root = nullptr;
```

Set up the struct

```
struct Node
{
   int number;
   Node* left;
   Node* right;
};
```

Create a new node

```
Node* create_new_node(Node* l, Node* r, int value)
{
   Node* item = new Node;
   assert(item);
   item->number = value;
   item->left = l;
   item->right = r;

   return item;
}
```

Insert – uses pass by value for root (recursive)

```
Node* insert(Node* root, int value)
{
   if (!root)
   {
      root = create_new_node(value, nullptr, nullptr)
   }

   if (root->number == value)
   {
      return root;
   }

   if (value < root->number)
   {
      root->left = insert(root->left, value);
   }

   else
   {
```

```
        root->right = insert(root->right, value);
    }
    return root;
}
```

Output the tree (recursive)

```
void output(Node* current)
{
    if (current)
    {
        output(current-> left);
        cout << current->number << " ";
        output(current->right);
    }
}
```