ACM Computing Seminar C++ Guide

Matt Hancock

Contents

1 Introduction **2**
- 1.1 A little about the language . . . 3
- 1.2 Additional references . . . 3
- 1.3 License, source, etc. . . . 3

2 Getting started **3**
- 2.1 Text editors . . . 4
- 2.2 Compilers . . . 4
- 2.3 Writing a program . . . 4

3 Data types **5**
- 3.1 The `bool` type . . . 5
- 3.2 `int` types . . . 6
- 3.3 Floating point types . . . 7
- 3.4 Casting . . . 7
- 3.5 The `const` modifier . . . 8
- 3.6 The `typedef` keyword . . . 8
- 3.7 Pointers and references . . . 9
  - 3.7.1 Pointers . . . 9
  - 3.7.2 References . . . 10
- 3.8 Arrays . . . 11
  - 3.8.1 Fixed length arrays . . . 11
  - 3.8.2 Dynamic length arrays . . . 11

4 Control structures **12**
- 4.1 Conditionals . . . 12
  - 4.1.1 Example: `if` / `else` and random number generation . . . 12
  - 4.1.2 Example: `if` / `else if` / `else` . . . 13
- 4.2 Loops . . . 14
  - 4.2.1 The `for` loop . . . 14

4.2.2 The `while` loop . . . 16
4.2.3 The `break` keyword . . . 18

5 Input / Output **18**
5.1 Inputs to `main`: `argc` and `argv` . . . 18
5.2 Filestreams . . . 19
5.2.1 Reading data from a file . . . 19
5.2.2 Writing data to a file . . . 20

6 Functions **21**
6.1 Writing a function . . . 21
6.1.1 Example: `linspace`: generating a set of equally-spaced points . . . 22
6.2 Header and implementation files . . . 23
6.2.1 The header file . . . 23
6.2.2 The implementation file . . . 24
6.2.3 The file containing `main` . . . 25
6.3 Function pointers . . . 25
6.3.1 Example: Newton's method for rootfinding . . . 26
6.3.2 Example: The midpoint rule for definite integrals . . . 27

7 Object-oriented programming **29**
7.1 Example: a vector `class` . . . 29
7.1.1 The header file . . . 29
7.1.2 The implementation file . . . 31
7.1.3 Example usage . . . 32
7.1.4 Operator overloading . . . 33
7.1.5 The copy constructor . . . 36
7.1.6 Friend functions . . . 37
7.2 Templating: a matrix class . . . 40
7.2.1 Overloading `operator*` . . . 43
7.3 Inheritance . . . 45

1 Introduction

This manual is a guide for quickly learning C++ for mathematical and scientific computing applications. The goal of this guide is not to make you a C++ expert, but to quickly teach you enough of the C++ fundamentals and design patterns to help you off the ground. If you should like to go beyond this guide, a few references are listed below.

1.1 A little about the language

Before you dive in, here is a little about the C++ programming language:

C++ is an extension of the C programming language. Both C and C++ are **statically-typed** and **compiled** languages, which means that the **type** of variables used in your source code must be declared explicitly and is checked when the program is compiled (i.e., translated into a machine executable file).

One key difference between C++ and C, however, is that C++ provides many mechanisms to allow for the object-oriented programming paradigm. This essentially allows the software writer to create custom, complex, reusable data structures. The object-oriented paradigm is extremely useful, but we will only touch the surface of it in this guide.

1.2 Additional references

- C++ reference
- C++ tutorials
- C++ wiki
- A compiled list of C++ textbooks from stackoverflow

1.3 License, source, etc.

This document was created using Emacs org mode with some custom css and javascript. You can find the license, view the source, and contribute to this document here:

`https://github.com/notmatthancock/acm-computing-seminar`

2 Getting started

The particular programming tools that you choose to use will likely be largely influenced by the operating system that you use. We will use free tools (often developed for GNU / Linux systems) in this guide. These tools are mostly available in other operating systems as well. For example, on Windows, you could use Cygwin, or install a dual boot with some Linux distribution (e.g., Ubuntu). On the other hand, MAC OSX, being a BSD-derived system, has many of the required tools already available (although, a command line utility, Brew, makes building and installing other tools very simple).

In the following two sections, we'll talk about the two basic types of software that you'll need to begin writing C++ programs.

2.1 Text editors

The text editor that you choose to use should be be any program capable of editing plain text files. However, you may find that it's more productive to write in an editor that offers features such as syntax highlighting, code-completion, bracket-matching, or other features. Here are some popular free text editors:

- Atom is a recently open-sourced GUI editor which some have compared to the very popular non-free editor, Sublime Text.
- Emacs is another powerful editor, which allows for highly optimized workflows.
- Gedit is a nice and simple GUI editor, which is the default in GNOME desktop environment.
- Kate is a another simple GUI editor, which is the default in the KDE desktop environment.
- Vim is a modal editor with a steep learning curve. It offers highly efficient means to edit text, and is available (or it's predecessor, vi) by default on nearly all UNIX-like operating systems.

2.2 Compilers

Second, you'll need a program called a **compiler**. A compiler translates the high-level C++ language into an executable program. In this guide, we will use the `g++` compiler which is freely available through the gnu compiler collection (gcc).

`g++` is a program which you typically call from the command line, which takes as input, your C++ source code file, and produces as output, a binary executable file.

2.3 Writing a program

Let's create our first C++ program, the obligatory "Hello, world!". First, fire-up your text editor and create a file called, `hello.cpp`, with the following contents:

```
1  #include <iostream>
2
3  int main() {
4    std::cout << "Hello, world!";
5    return 0;
6  }
```

Now, to compile the program, execute:

```
g++ hello.cpp
```

Followed by:

```
./a.out
```

By default, `g++` names the resulting binary executable, `a.out`, but you can specify a different output file name by using the `-o` flag:

```
g++ hello.cpp -o my_executable_file.bin
```

Note that in Windows, executable files typically end in `.exe`. In Unix-like systems, there is no particular file-extension type for executables.

3 Data types

As we mentioned previously, you must explicitly declare the type of a variable. So, in this section, we'll talk about the main variable types you'll use. In the section on object-oriented programming, we'll discuss how to build more complex data types.

3.1 The `bool` type

A boolean data type is either `true` or `false`. There are a number of operators between these types, illustrated in the code snippet below (note that lines starting with `//` are comments and are ignored by the compiler):

```
1  bool a,b,c; // Declare the type of variables a, b, and c.
2  a = true;
3  b = false;
4
5  // !  is logical negation when applied to a single variable.
```

```
 6  c = !a; // c is false.
 7
 8  // && is logical and.
 9  c = (a && b); // c is false.
10
11  // || is logical or.
12  c = (a || b); // c is true.
```

We don't often use boolean variables by themselves, but rather as a result of comparing two other data types (such as comparing if one integer is less than another integer). Expressions that result in boolean values are mostly used in conditional control structures.

3.2 `int` types

There are a variety of integer types in C++. Below, we illustrate a couple. These can be modified further using the `short` and `long` keywords, changing the number of bytes occupied by the variable (and hence the maximum and minimum size the variable can take on).

```
 1  int a = 6; // initialize a to 6.
 2  unsigned int b = 7; // initialize b to 7.
 3  int c; // declare c to be an integer variable.
 4
 5  a = 6;
 6  b = 7;
 7
 8  c = a / b; // c is 0
 9  c = b / a; // c is 1
10  c = b % a; // c is 1 (% is the integer remainder or modulo operator)
11  c = a - b; // c is -1
12  c = a > b; // c is 0 (boolean gets cast to integer)
13  c = a < b; // c is 1 (boolean gets cast to integer)
14  c++;       // c is 2 (++ is shorthand for c = c + 1)
15
16  b = a - b; // b is 4294967295 (-1 gets cast to unsigned)
17  b = b + 1; // b is 0 (b was previously the largest unsigned,
18             // so adding one circles it back to zero.)
19  b += 7;    // b is 7 (+= is shorthand for b = b + 7;
```

In the above, we've illustrated the use of signed and unsigned integer types and the operators between them. It is important to take care when

you assign a result to a variable that doesn't match the type of the result. In many cases, the result gets implicitly cast to the type of variable being assigned to. The result may or may not match your expectations, as shown above.

3.3 Floating point types

There are two main floating point data types in C++, `float` and `double`, which correspond to IEEE 32- and 64-bit floating point types.

```
 1  #include <iostream>
 2  #include <limits>
 3
 4  int main() {
 5    float a; //  Declare a single precision float.
 6    double b; // Declare a double precision float.
 7
 8    // Print the max value of a float type.
 9    std::cout << std::numeric_limits<float>::max() << std::endl;
10
11    // Print the max value of a double type.
12    std::cout << std::numeric_limits<double>::max() << std::endl;
13
14    // Print machine epsilon of a float type.
15    std::cout << std::numeric_limits<float>::epsilon() << std::endl;
16
17    // Print machine epsilon of a double type.
18    std::cout << std::numeric_limits<double>::epsilon() << std::endl;
19
20    return 0;
21  }
```

```
3.40282e+38
1.79769e+308
1.19209e-07
2.22045e-16
```

3.4 Casting

Sometimes it is useful to explicitly cast one variable type as another. This can be done like the following:

```
1  int a; double b = 3.14159;
2
3  a = (int) b;
4
5  std::cout << a << std::endl;
```

```
3
```

3.5 The `const` modifier

If the value of some variable should not change, you can use the `const` keyword to protect its status. It is typical to denote `const` variables with all caps. Try to compile the following program:

```
1  const double PI = 3.14159;
2
3  PI = 3.0;
```

You will see an error like, `error: assignment of read-only variable 'PI'`.

3.6 The `typedef` keyword

Suppose you have a large numerical experiment, where all your code used floating point of type `double`. Your curious about how the results will be affected by changing the floating point type to single precision `float` type. One solution would be to run a "find and replace" in your editor, but something about that doesn't feel right.

Instead, we can use the `typedef` statement to define types:

```
 1  // Define "int_type" to be a short int.
 2  typedef short int int_type;
 3
 4  // Define "float_type" to be single precision float.
 5  typedef float float_type;
 6
 7  // Define "array_index_type" to be unsigned long int.
 8  typedef unsigned long int array_index_type;
 9
10  int_type a = -17;
11  float_type b = 1.14;
12  array_index_type c = 9;
```

3.7 Pointers and references

3.7.1 Pointers

Pointers are variables that hold the **memory address** for a variable of a specific type. Pointers are declared by specifying the variable type, followed by the `*` symbol, followed by the name of the pointer variable, e.g., `double * x` defines a "pointer to double" variable. The variable, `x`, therefore, does not hold the value of a `double` type, but rather, the memory address for a variable of type, `double`. The memory address for a variable can be obtained by the `&` operator.

```
1  double * a;
2  double b = 7;
3
4  // This obtains the memory address of ‘b‘.
5  a = &b;
6
7  // Prints some memory address (starts with 0x)
8  std::cout << a << std::endl;
```

```
0x7ffe0d98f7b8
```

Similar to obtaining the memory address from a regular variable, using the `&` operator, you can use the `*` symbol before a pointer to access the variable value held at the memory location of the pointer. In this context, the `*` symbol is called the **dereference operator**. This is probably better understood with a short example:

```
 1  double * a;
 2  double b = 7.3;
 3  double c;
 4
 5  // Now ‘a‘ holds the memory address of ‘b‘.
 6  a = &b;
 7
 8  // ‘*a‘ obtains the value of the variable
 9  // at the memory address held by ‘a‘.
10  // So, ‘c‘ is 7.3.
11  c = *a;
12
13  std::cout << c << "\n";
```

7.3

3.7.2 References

A reference is a sort of like a pointer, but not quite. There are differences. A good analogy, which you can find in the previous link, is that a reference is similar to a symbolic link, or "shortcut" if you're on Windows. You can treat it more-or-less like the original variable, but it's not the original.

```
 1  double a = 1.1;
 2  // ‘b‘ is a reference to ‘a‘.
 3  double & b = a;
 4
 5  std::cout << "a: " << a << ", b: " << b << "\n";
 6
 7  a = 2.1;
 8
 9  std::cout << "a: " << a << ", b: " << b << "\n";
10
11  b = 3.1;
12
13  std::cout << "a: " << a << ", b: " << b << "\n";
14
15  std::cout << "\n\n";
16  std::cout << "&a: " << &a << "\n" << "&b: " << &b << "\n";
```

```
a: 1.1, b: 1.1
a: 2.1, b: 2.1
a: 3.1, b: 3.1


&a: 0x7ffcfbe7b1e8
&b: 0x7ffcfbe7b1e8
```

References are useful for passing around large objects, so that the object doesn't need to be copied. References are also useful as a return type for functions (to be discussed later) because it allows to assign to assign a value to a function, which is useful if the function, for example, returns a reference to the element of an array.

3.8 Arrays

The length of an array can be fixed or dynamic, and how you declare the array depends on this. Array indexing starts at 0 in C++ (compared to start at 1, for example, in Fortran or Matlab).

3.8.1 Fixed length arrays

```
1  double a[5];
2
3  a[0] = 1.0;
4  // etc.
```

3.8.2 Dynamic length arrays

Dynamic length arrays are made possible through pointers:

```
 1  // This allocates memory for 5 double types.
 2  double * a = new double[5];
 3
 4  // Afterwards, you can treat ‘a‘ like a normal array.
 5  a[0] = 1.0;
 6  // etc...
 7
 8  // Whenever you use the ‘new‘ keyword, you must
 9  // delete the memory allocated when you’re done by hand.
10  delete [] a;
11
12  // We can change the size of ‘a‘.
13  a = new double [10];
14
15  a[0] = 2.0;
16  // etc...
17
18  delete [] a;
```

Note that omitting the first `delete` statement will cause no error. However, the memory allocated by the first `new` statement will not be freed, and thus inaccessible. This is bad because the memory cannot be allocated to other resources. You should generally try to avoid manually memory management when possible, but a good tool for debugging memory problems is called valgrind.

4 Control structures

4.1 Conditionals

4.1.1 Example: `if` / `else` and random number generation

Often a code block should only be executed if some condition is true. Below, we generate a random number between 0 and 1; print the number; and, print whether or not the number was greater than 0.5.

```
 1  #include <iostream>
 2  #include <stdlib.h>
 3  #include <time.h>
 4
 5  int main() {
 6    // Seed a random number generator.
 7    srand(123);
 8
 9    // rand() produces a random integer between 0 and RAND_MAX.
10    double num = rand() / ((double) RAND_MAX);
11
12    std::cout << "num: " << num << "\n";
13
14    if (num < 0.5) {
15      std::cout << "num was less than 0.5.\n";
16    }
17    else {
18      std::cout << "num was greater than 0.5.\n";
19    }
20
21    // Do it again.
22    num = rand() / ((double) RAND_MAX);
23
24    std::cout << "num: " << num << "\n";
25
26    if (num < 0.5) {
27      std::cout << "num was less than 0.5.\n";
28    }
29    else {
30      std::cout << "num was greater than 0.5.\n";
31    }
```

```
32
33    return 0;
34  }
```

```
num: 0.0600514
num was less than 0.5.
num: 0.788318
num was greater than 0.5.
```

4.1.2 Example: `if` / `else if` / `else`

You can follow `else` immediate by another `if` to have multiple mutually-exclusive blocks:

```
 1  #include <iostream>
 2  #include <stdlib.h>
 3  #include <time.h>
 4
 5  int main() {
 6    // Seed the random number generator based on the current time.
 7    srand(time(NULL));
 8
 9    // rand() produces a random integer between 0 and RAND_MAX.
10    double num = rand() / ((double) RAND_MAX);
11
12    std::cout << "num: " << num << "\n";
13
14    if (num >= 0.75) {
15      std::cout << "num was between 0.75 and 1.\n";
16    }
17    else if (num >= 0.5) {
18      std::cout << "num was between 0.5 and 0.75.";
19    }
20    else if (num >= 0.25) {
21      std::cout << "num was between 0.25 and 0.5.";
22    }
23    else {
24      std::cout << "num was between 0 and 0.25";
25    }
26
27    return 0;
```

```
28  }
```

```
num: 0.0456405
num was between 0 and 0.25
```

The conditions are checked in the order that they're written. So, for example, in the second condition, we don't need to specify `num >= 0.5 && num < 0.75` because we know that this condition will only be checked if the previous was false.

4.2 Loops

We discuss two main structures for iterating – the `for` and `while` loops.

4.2.1 The `for` loop

The `for` loop requires three specifications – the iteration variable initialization, the termination condition, and the update rule. The body of the loop follows these three specifications. Shown below, we declare an array; assign to its components; and, print the current component to the screen.

```
 1  int length = 11;
 2  double x[length];
 3
 4  for(int i=0; i < length; i++) {
 5    // Assign to each array component.
 6    x[i] = (double) i / (length - 1);
 7
 8    // Print the current component.
 9    std::cout << "x[" << i << "] = " << x[i] << std::endl;
10  }
```

```
x[0] = 0
x[1] = 0.1
x[2] = 0.2
x[3] = 0.3
x[4] = 0.4
x[5] = 0.5
x[6] = 0.6
x[7] = 0.7
x[8] = 0.8
x[9] = 0.9
x[10] = 1
```

Example: row-major matrix You can nest loops, i.e., loops inside of loops, etc.

Below, is an example of a double loop for creating and accessing matrix data stored in a flat array. The matrix data is stored in row-major order. This means the first `n_cols` elements of the array named, `matrix`, will contain the first row of the matrix, the second `n_cols` elements of `matrix` will contain the second row, etc.

```
 1  int n_rows = 4;
 2  int n_cols = 3;
 3
 4  // Row-major matrix array.
 5  double matrix [n_rows*n_cols];
 6
 7  // temporary index.
 8  int k;
 9
10  for(int i=0; i < n_rows; i++) {
11    for(int j=0; j < n_cols; j++) {
12      // Convert the (i,j) matrix index to the "flat" row-major index.
13      k = i*n_cols + j;
14
15      // Assign a value of 1.0 to the diagonal,
16      // 2 to the off-diagonal, and 0 otherwise.
17      if (i == j) {
18        matrix[k] = 1.0;
19      }
20      else if ((i == (j+1)) || (i == (j-1))){
21        matrix[k] = 2.0;
22      }
23      else {
24        matrix[k] = 0.0;
25      }
26    }
27  }
28
29
30  // Print the matrix elements.
31  for(int i=0; i < n_rows; i++) {
32    for(int j=0; j < n_cols; j++) {
```

```
33      k = i*n_cols + j;
34
35      std::cout << matrix[k];
36      if (j != (n_cols-1)) {
37        std::cout << ", ";
38      }
39    }
40
41    if (i != (n_rows-1)) {
42      std::cout << "\n";
43    }
44  }
```

```
1, 2, 0
2, 1, 2
0, 2, 1
0, 0, 2
```

4.2.2 The `while` loop

A `while` loop iterates while a condition is `true`. Essentially, it is a `for` loop without an update variable.

Example: truncated sum In the following example, we approximate the geometric series:
$1 = \sum_{n=1}^{\infty} \left(\frac{1}{2}\right)^n$
The loop exits when the absolute error,
absolute error $:= 1 - \sum_{n=1}^{N} \left(\frac{1}{2}\right)^n$
is less than some specified tolerance, `tol`.

```
 1  double sum = 0.0;
 2  double base = 0.5;
 3  double pow = base; // initialize to base^1
 4  double tol = 1e-4;
 5  int iter = 1;
 6
 7  while((1-sum) >= tol) {
 8    // Add ‘pow‘ to ‘sum‘.
 9    sum += pow;
10    // Update ‘pow‘ by one power of ‘base‘.
```

```
11    pow *= base;
12
13    printf("Iter: %03d, Sum: %.5f, Abs Err: %.5f\n", iter, sum, 1-sum);
14
15    // Update the ‘iter‘ val by 1.
16    iter += 1;
17  }
```

```
Iter: 001, Sum: 0.50000, Abs Err: 0.50000
Iter: 002, Sum: 0.75000, Abs Err: 0.25000
Iter: 003, Sum: 0.87500, Abs Err: 0.12500
Iter: 004, Sum: 0.93750, Abs Err: 0.06250
Iter: 005, Sum: 0.96875, Abs Err: 0.03125
Iter: 006, Sum: 0.98438, Abs Err: 0.01562
Iter: 007, Sum: 0.99219, Abs Err: 0.00781
Iter: 008, Sum: 0.99609, Abs Err: 0.00391
Iter: 009, Sum: 0.99805, Abs Err: 0.00195
Iter: 010, Sum: 0.99902, Abs Err: 0.00098
Iter: 011, Sum: 0.99951, Abs Err: 0.00049
Iter: 012, Sum: 0.99976, Abs Err: 0.00024
Iter: 013, Sum: 0.99988, Abs Err: 0.00012
Iter: 014, Sum: 0.99994, Abs Err: 0.00006
```

Example: estimating machine epsilon

```
 1  double eps = 1;
 2  int count = 1;
 3
 4  while(1.0 + eps*0.5 > 1.0) {
 5      eps *= 0.5;
 6      count += 1;
 7  }
 8
 9  std::cout << eps << ", " << std::numeric_limits<double>::epsilon() << "\n"
10            << count << ", " << std::numeric_limits<double>::digits;
```

```
2.22045e-16, 2.22045e-16
53, 53
```

4.2.3 The `break` keyword

The `break` keyword provides a mechanism for exiting the direct parent loop for which the `break` statement is placed. For example:

```
1  for(int i=0; i < 3; i++) {
2    while(true) {
3      std::cout << "Entering infinite loop number " << (i+1) << "\n";
4      break;
5    }
6    std::cout << "We escaped the infinite loop!\n";
7  }
```

```
Entering infinite loop number 1
We escaped the infinite loop!
Entering infinite loop number 2
We escaped the infinite loop!
Entering infinite loop number 3
We escaped the infinite loop!
```

The previous example is contrived, but there are situations, where you might find the break statement within an infinite loop useful. Of course, you should avoid this sort of thing if there is a more straight-forward approach.

5 Input / Output

We have already used the `<iostream>` library to print results to the console. However, in many cases, we'd like to read in lots of data from a file, pass option flags to the program from the command line, or save the results of some computation to a file for further analysis.

5.1 Inputs to `main`: `argc` and `argv`

The `main` function has two optional arguments which we have thus far omitted, `argc` and `argv`. These arguments allow arguments to passed to the `main` function when the program is run. This is how flags and other arguments are passed to programs you use from the command line. The first, `argc`, is of type, `int`, and stands for arg count. It gives the number of arguments to the program. The arg count is always at least 1 because the program's name is always the first argument. The second, `argv`, is a double pointer to `char`. In essence, `argv` is an array of strings.

```
 1  #include <iostream>
 2
 3  int main(int argc, char ** argv) {
 4    std::cout << "argc = " << argc << "\n";
 5
 6    for(int i=0; i < argc; i++) {
 7      std::cout << "argv[" << i << "] = " << argv[i] << "\n";
 8    }
 9    return 0;
10  }
```

Compile this program, and run, for example:

```
g++ main.cpp
./a.out hello 1 2 34
```

```
argc = 5
argv[0] = ./a.out
argv[1] = hello
argv[2] = 1
argv[3] = 2
argv[4] = 34
```

`argc` and `argv` are handy for setting up large experiments. You could, for example, set up your main function so that different functions or parameters or used based on the arguments of `arcv`. Then, you could set up a shell script that loops through the desired arguments to be supplied to the main function.

5.2 Filestreams

File input and output is crucial for numerical experiments with lots of data. In this section, we see how to read and write data to files.

5.2.1 Reading data from a file

In general, how data is read in depends heavily on how the data is stored. Nevertheless, we will give an example of reading in a vector stored in a particular fashion. Suppose a text file exists in the directory, `./data/vector.txt`, containing

```
1 2 3.14 4 5 6.28
```

```
 1  #include <iostream>
 2  #include <fstream>
 3
 4  int main() {
 5      std::fstream fin("./data/vector.txt", std::ios_base::in);
 6      double vector[6];
 7      int i = 0;
 8      while(fin >> vector[i]) {
 9        std::cout << vector[i] << " ";
10        i++;
11      }
12      return 0;
13  }
```

```
g++ main.cpp && ./a.out

1 2 3.14 4 5 6.28
```

This simply prints the data in the file back out to the console. Note, however, that the data is read into an array of type `double`, so it can be processed numerically thereafter.

In this example dealt with simply stored data, and it was assumed that the number of data entries was known beforehand. Parsing data can become quite complicated depending on how it is stored, and depending on the intended format of the data.

5.2.2 Writing data to a file

Writing to a file is similar, using the `<fstream>` library.

```
 1  #include <fstream>
 2  #include <cmath>
 3
 4  int main() {
 5    std::fstream fout("./data/new_shiny_data.txt", std::ios_base::out);
 6    double x;
 7
 8    fout << "x\tsin(x)\n";
 9
10    for(int i=0; i < 11; i++) {
11      x = i / 10.0;
```

```
12      fout << x << "\t" << sin(x) << "\n";
13    }
14
15    fout.close();
16
17    return 0;
18  }
```

This produces a file called `new_shiny_data.txt` in the folder, `data`, containing:

```
x   sin(x)
0   0
0.1 0.0998334
0.2 0.198669
0.3 0.29552
0.4 0.389418
0.5 0.479426
0.6 0.564642
0.7 0.644218
0.8 0.717356
0.9 0.783327
1   0.841471
```

6 Functions

So far, we've piled everything into the `main` function. When we have a block of code used for a specific subtask, we can offload it to a function. This promotes code which is separated based on the subtasks each block is intended to perform. This, in turn, makes your code easier to debug and easier to understand.

6.1 Writing a function

A function must be declared before use. Thus, a function usual consists of two parts, a declaration and an implementation. You must declare the return type of a function as well as the types of all the function's arguments. If the function is defined in the same file as the `main` function, you should write the declaration before `main` and the implementation after `main`.

6.1.1 Example: `linspace`: generating a set of equally-spaced points

```
 1  #include <iostream>
 2
 3  // This is the function declaration.
 4  // You should describe the functions arguments
 5  // and what is returned by the function in comments
 6  // near the declaration.
 7  //
 8  // `linspace` returns an array of doubles containing
 9  // `n_points` entries which are equally-spaced, starting
10  // at `start` and ending at `stop`.
11  double * linspace(double start, double stop, int n_points);
12
13  // `void` is a function with no return type.
14  // `print_array` takes an array and prints it to std out.
15  void print_array(double * arr, int arr_len);
16
17  int main() {
18    double * xs = linspace(-1, 1, 5);
19    print_array(xs, 5);
20    delete [] xs;
21
22    return 0;
23  }
24
25  // Implementation of `linspace`.
26  double * linspace(double start, double stop, int n_points) {
27    double * arr = new double [n_points];
28    double dx = (stop-start) / (n_points-1.0);
29
30    for(int i=0; i < n_points; i++) {
31      arr[i] = start + i*dx;
32    }
33
34    return arr;
35  }
36
37  // Implementation of `print_array`.
38  void print_array(double * arr, int arr_len) {
```

```
39    for(int i=0; i < arr_len; i++) {
40      std::cout << arr[i] << "\n";
41    }
42  }

-1
-0.5
0
0.5
1
```

6.2 Header and implementation files

The example in the previous section certainly made the `main` function cleaner and simpler to understand, having only two function calls. However, the file itself was still pretty messy. Thankfully, there is a way to modularize further, by creating header and implementation files. Here is how we do it:

6.2.1 The header file

Put the declarations from the into a header file, called `my_library.h`:

```
 1  #ifndef MY_LIBRARY_H
 2  #define MY_LIBRARY_H
 3
 4  #include <iostream>
 5
 6  namespace my_namespace {
 7      // ‘linspace‘ returns an array of doubles containing
 8      // ‘n_points‘ entries which are equally-spaced, starting
 9      // at ‘start‘ and ending at ‘stop‘.
10      double * linspace(double start, double stop, int n_points);
11
12      // ‘void‘ is a function with no return type.
13      // ‘print_array‘ takes an array and prints it to std out.
14      void print_array(double * arr, int arr_len);
15  }
16
17  #endif
```

Note the the function declarations are wrapped in conditional "macro" statments, `#ifndef`, `#define`, and `#endif`. You can think of this as protecting your library from being imported twice.

We have also introduced the notion of a `namespace` above. Namespaces help to prevent naming clashes between separate libraries. When calling a function from a particular namespace, you must write the namespace followed by `::` and then the function name. This is why many standard library functions like `<iostream>` begin with `std::`.

6.2.2 The implementation file

Create a file called `my_library.cpp` containing the implementations as follows:

```
 1  #include "my_library.h"
 2
 3  // Implementation of ‘linspace‘.
 4  double * my_namespace::linspace(double start, double stop, int n_points) {
 5    double * arr = new double [n_points];
 6    double dx = (stop-start) / (n_points-1.0);
 7
 8    for(int i=0; i < n_points; i++) {
 9      arr[i] = start + i*dx;
10    }
11
12    return arr;
13  }
14
15  // Implementation of ‘print_array‘.
16  void my_namespace::print_array(double * arr, int arr_len) {
17    for(int i=0; i < arr_len; i++) {
18      std::cout << arr[i] << "\n";
19    }
20  }
```

Note that we have to include the header file in quotations at the beginning, and the names of the functions must be prepended by the namespace that we've given in the header file.

6.2.3 The file containing `main`

Create a file with the main function, say `main.cpp`:

```
 1  #include <iostream>
 2  #include "my_library.h"
 3
 4  int main() {
 5      double * xs = my_namespace::linspace(-1,1,5);
 6      my_namespace::print_array(xs, 5);
 7      delete [] xs;
 8
 9      return 0;
10  }
```

Now the main function is very nice and clean, but now we 3 separate files we must compile into one executable. This is done as follows:

```
# Convert the library into an object file.
g++ -c my_library.cpp
# Compile the main to an executable.
g++ my_library.o main.cpp
# Run it.
./a.out

-1
-0.5
0
0.5
1
```

If successful, you will see the same output as previously.

6.3 Function pointers

Pointers can be made to functions, and these function pointers can be used as arguments to other functions. We'll look at two functions that accept a function pointer as one of their arguments.

6.3.1 Example: Newton's method for rootfinding

Suppose $f : \mathbb{R} \to \mathbb{R}$, and we'd like to find a root of f. Newton's method is an iterative method for finding roots, which, starting from some initial guess, x_0, iterates:

$x_{n+1} \leftarrow x_n - \frac{f(x_n)}{f'(x_n)}$

For simplicity, we'll dump everything into the file containing `main`, but you could imagine a libary with many methods for finding roots, which would contain Newton's method.

Let's consider $f(x) = x^2 - 2$.

```
 1  #include <cmath>
 2  #include <iostream>
 3
 4  // The function to find the root of.
 5  double func(double x);
 6  // Its derivative.
 7  double dfunc(double x);
 8
 9  // Find the root of ‘f‘ using Newton's method,
10  // starting from ‘x0‘ until |f(x)| < ‘tol‘ or ‘max_iters‘
11  // is reached.
12  //
13  // Note the first and second arguments are function pointers.
14  double newton_root(double (*f)(double), double (*df)(double), double x0,
15                     double tol, int max_iters, bool print_iters);
16
17  int main() {
18    double x = newton_root(&func, &dfunc, 1.0, 1e-6, 1000, true);
19
20    return 0;
21  }
22
23  double func( double x) { return x*x - 2; }
24  double dfunc(double x) { return 2*x; }
25
26  double newton_root(double (*f)(double), double (*df)(double), double x0,
27                     double tol, int max_iters, bool print_iters) {
28    double x  = x0;
29    int iter  = 0;
```

```
30
31    while (std::abs(f(x)) > tol && iter < max_iters) {
32      if (print_iters) {
33        std::cout << "f(" << x << ") = " << f(x) << "\n";
34      }
35
36      // Newton's method update.
37      x -= f(x) / df(x);
38      iter++;
39    }
40
41    // One last print if necessary.
42    if (print_iters) {
43      std::cout << "f(" << x << ") = " << f(x) << "\n";
44    }
45
46    return x;
47  }
```

```
f(1) = -1
f(1.5) = 0.25
f(1.41667) = 0.00694444
f(1.41422) = 6.0073e-06
f(1.41421) = 4.51061e-12
```

6.3.2 Example: The midpoint rule for definite integrals

The midpoint rule is a numerical integration method which approximates the definite integral of a specified function over a specified interval using a specified number of subintervals where on each subinterval, the area under the curve is approximated by a rectangle whose width is the width of the subinterval and whose height is the height of the function at the midpoint between the points defining the end points of the subinterval.

Specifically, if n equally-sized subintervals are used on $[a, b]$, then the midpoint rule approximation, M_n, to the definite integral of $f(x)$ on $[a, b]$ is:

$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f\left(\frac{x_{i-1}+x_i}{2}\right)\Delta x =: M_n$

where $\Delta x = \frac{b-a}{n}$, and $x_i = a + i \cdot \Delta x, \;\; i = 0, 1, \ldots, n$.

Let's consider $f(x) = \frac{1}{x}$ on $[1, e]$.

```
1  #include <iostream>
```

```
 2  #include <cmath>
 3
 4  const double E = std::exp(1.0);
 5
 6  // The function to be integrated.
 7  double func(double x);
 8
 9  // Compute the midpoint rule approximation to
10  // the definite integral of ‘f‘ from ‘a‘ to ‘b‘
11  // using ‘n‘ subintervals.
12  double midpoint_rule(double (*f)(double), double a, double b, int n);
13
14
15  int main() {
16    for(int n=2; n <= 20; n += 2) {
17      std::cout << "n = " << n << ", "
18                << "M_n = " << midpoint_rule(&func, 1, E, n) << "\n";
19    }
20
21    return 0;
22  }
23
24  double func(double x) { return 1.0 / x; }
25
26  double midpoint_rule(double (*f)(double), double a, double b, int n) {
27    double xi;
28    double xi_prev = a;
29    double dx = (b-a) / n;
30    double sum;
31
32    for(int i=1; i <= n; i++) {
33      xi = a + i*dx;
34      sum += f(0.5*(xi_prev + xi));
35      xi_prev = xi;
36    }
37
38    return sum*dx;
39  }
```

```
n = 2, M_n = 0.97636
```

```
n = 4, M_n = 0.993575
n = 6, M_n = 0.997091
n = 8, M_n = 0.998353
n = 10, M_n = 0.998942
n = 12, M_n = 0.999264
n = 14, M_n = 0.999459
n = 16, M_n = 0.999585
n = 18, M_n = 0.999672
n = 20, M_n = 0.999734
```

7 Object-oriented programming

New data types can be created by writing a new `class`. A `class` has state variables and functions that act on the state variables. An instance of a `class` is called an **object**. Let's write a `vector` class that improves upon the default `double` array.

7.1 Example: a vector `class`

7.1.1 The header file

Create the header file, `vector.h`:

```
 1  #ifndef VECTOR_H
 2  #define VECTOR_H
 3
 4  namespace vec {
 5    class vector {
 6    public:
 7      // Constructor. This function is called when the object is created.
 8      vector(unsigned len);
 9
10      // Destructor. This function is called when the object is destroyed.
11      ~vector();
12
13      // length accessor.
14      unsigned len() const;
15
16      // data accessor.
17      double & element(unsigned i) const;
```

```
18
19      // Simple print function.
20      void print() const;
21
22    private:
23      unsigned length;
24      double * data;
25      void check_index(unsigned i) const;
26    };
27  }
28  #endif
```

First note the macro guards, `#ifndef`, `#define`, and `#endif`, as well as the namespace, `vec`, wrapping the `class` declaration. Macro guards and namespaces were previously discussed when we initially introduced how to write header and implementation files.

Now, within the namespace, we've declared a class, `vector`, which contains `public` and `private` variables and function declarations. Private functions and variables may only be accessed through the public methods. This means if you created an instance of the class, `vector`, you would not be able to access the private variable directly. You could only call the **public member-functions**, which, in turn, may manipulate the **private member-variables**, or call the **private member-functions**. Splitting variables and functions into public and private helps to ensure that other libraries and programs use your class as intended.

Thus far, this class has 5 public member-functions, 2 private member-variables, and 1 private member-function. The first two member functions are special, the **constructor** and **destructor**, respectively. The constructor is called explicitly when you declare a new instance of this class, while the destructor is usually called implicitly when the object is deleted or when it goes out of scope.

Notice that the method for accessing elements of `vector` is called `element` and its return type is a **reference** (discussed previously). This allows us to use this function on both the left side of assignment operators, i.e., to assign values to `vector` components, and on the right side of assignments, i.e., to treat it as the value of the component.

Finally, notice that some member function declarations end with the keyword, `const`. Functions with such a signature are not allowed to modify member variables, and they are also not allowed to call other non `const` member functions.

7.1.2 The implementation file

Create the implementation file, `vector.cpp`:

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include "vector.h"
 4
 5  namespace vec {
 6    vector::vector(unsigned len) {
 7      this->length = len;
 8      this->data = new double[len];
 9      // Initialize data to zeros.
10      for(int i=0; i < this->len(); i++) { this->data[i] = 0.0; }
11    }
12
13    vector::~vector() {
14      delete [] this->data;
15    }
16
17    unsigned vector::len() const {
18      return this->length;
19    }
20
21    double & vector::element(unsigned i) const {
22      #ifndef NDEBUG
23      check_index(i);
24      #endif
25      return this->data[i];
26    }
27
28    void vector::print() const {
29      for(int i=0; i < this->len(); i++) {
30        std::cout << this->data[i] << '\n';
31      }
32    }
33
34    void vector::check_index(unsigned i) const {
35      if (i < 0 || i >= this->length) {
36        std::cerr << "ERROR: index, " << i << ", is out-of-bounds.\n"
37                  << "(valid indices are 0-" << (this->length-1) << ")\n";
```

```
38        exit(1);
39      }
40    }
41  }
```

Note that we again wrap the implementations in the same namespace as wrapped by the class declaration. Also observe how each member-function is prepended by `vector::`.

The keyword, `this`, is a pointer to the calling object. Writing, `this->`, is equivalent to `(*this).`, and in fact, can be used for any pointer. Thus, `this->length` is equivalent to `(*this).length`.

Observe how the private member function, `check_index`, is used in the public `element` accessor function. If this library is compiled with the flag, `-DNEDUBG`, then the check function will not be called. You could read this flag as "define no debug". Thus, when this flag is present, the debug function `check_index` is called whenever the element accessor is called. The `check_index` function simply checks if the provided index is out-of-bounds for the vector. If it is, an informative message is printed, and the program terminates prematurely by calling `exit(1)`. Such assertions with informative messages are a good practice, and will likely save you lots of headaches in the future.

7.1.3 Example usage

Ok. Let's see some example usage, by creating a `main.cpp`, containing:

```
 1  #include <iostream>
 2  #include "vector.h"
 3
 4  int main() {
 5      vec::vector v(5);
 6
 7      std::cout << "‘v‘ has length = " << v.len() << "\n";
 8
 9      v.element(0) = -1.27;
10      v.element(3) = 3.1;
11
12      v.print();
13
14      v.element(5) = 1234.0;
```

```
15
16      return 0;
17  }
```

Let's first compile with our `check_index` debugger function in place:

```
g++ -c vector.cpp
g++ vector.o main.cpp
./a.out
```

If successful, you should see:

```
‘v‘ has length = 5
-1.27
0
0
3.1
0
ERROR: index, 5, is out-of-bounds.
(valid indices are 0-4)
```

Now let's run without `check_index`:

```
g++ -DNDEBUG -c vector.cpp
g++ vector.o main.cpp
./a.out
```

Upon running, you will likely see some extensive list of errors when the element beyond the array's length is attempted to be accessed. Again, by liberally sprinkling these sorts of assertions through your code, you will (sometimes) find debugging much easier. After you're fairly certain that your code is working, you can simply compile with `-DNEDUBG`.

7.1.4 Operator overloading

The `v.element(i)` accessor is a bit clunky. We can replace this with the more natural, `v[i]`, by **overloading** the `[]` operator. Indeed, we can overload many of the normal C++ operators, e.g. `+`, `-`, `=`, etc. Some of these operators require more careful consideration when implementing class that utilizes dynamic memory allocation, such as our `vector` class.

Overloading `operator[]` In the header file, replace the `element` function declaration with:

```
1  // data accessor.
2  double & operator[](unsigned i) const;
```

and in the implementation file, replace the `element` implementation with:

```
1  double & vector::operator[](unsigned i) const {
2    #ifndef NDEBUG
3    check_index(i);
4    #endif
5    return this->data[i];
6  }
```

Just think of `operator[]` as the new name of the function, `element`. We can now use the overloaded operator just like how we would use the `[]` for operator for accessing array components, expect now our overloaded operator function is called instead.

```
1  v[0] = -1.27;
2  v[3] = 3.1;
```

Compile and run:

```
g++ -c vector.cpp && g++ vector.o main.cpp && ./a.out
```

and you should see:

```
-1.27
0
0
3.1
0
```

just like before.

Overloading `operator=` Let's overload the `=` operator so we can assign one vector to another. We'll write in a way such that the vector on the left hand side is overwritten by the one on the right.

Let's add a declaration to the header file,

```
1  // assignment operator.
2  vector & operator=(const vector & src);
```

and let's add to the implementation file,

```
 1  vector & vector::operator=(const vector & src) {
 2    // Delete the old data.
 3    delete [] this->data;
 4
 5    // Initialize the new data.
 6    this->length = src.len();
 7    this->data = new double[this->len()];
 8
 9    // Copy over the new data.
10    for(int i=0; i < this->len(); i++) {
11      this->data[i] = src[i];
12    }
13
14    return *this;
15  }
```

Now, let's assume the `vector` instance, `v`, from above is still defined, and we'll create a new vector:

```
1  vec::vector w(14);
2  w = v;
3  w.print();
```

This should print,

```
-1.27
0
0
3.1
0
```

Notice that `w` is intially defined to be of length `14`, but this is overwritten, and its new length is the length of `v`. Also note that all of `w`'s old data is deleted.

7.1.5 The copy constructor

It may be tempting at this point to attempt to initialize `w` from `v` directly:

```
1  vec::vector w = v;
```

If you attempt this currently, you will see all sorts of errors. This is because this type of intialization does not call the assignment operator. It calls the **copy constructor**. The assignment operator is only called when the object has already been intialized. Writing the previous line of code is essentially equivalent to

```
1  vec::vector w(v);
```

In other words, the constructor is called with the existing vector, `v`, as the argument, but we have not written a constructor yet with such a call signature.

The constructor can be overloaded, i.e., we can write multiple versions of the constructor function, and the one that matches the correct call signature will be used. This function overloading behavior actually applies to all functions in C++.

Let's add the copy constructor declaration to the header file:

```
1  // Copy constructor.
2  vector(const vector & src);
```

and let's add its implementation:

```
1  vector::vector(const vector & src) {
2    this->length = src.len();
3    this->data = new double[this->len()];
4
5    // Copy over the data.
6    for(int i=0; i < this->len(); i++) {
7      this->data[i] = src[i];
8    }
9  }
```

Now we compile and run something like:

```
1  vec::vector w = v;
2  w.print();
```

we will see:

```
-1.27
0
0
3.1
0
```

7.1.6 Friend functions

Non-member functions may access private member variables and private member functions of by labeling them as `friend`. This is useful in situations where it is not clear that function should be "called on" an object, i.e. `object.method(params)`. Friend functions should be declared in the class declaration, but their implementation is not prepended with `class::`, which is necessary for member functions.

Overloading `operator*` As an example, we'll overload `operator*` to implement scalar multiplication.

Because scalar multiplication should commute, let's add the following two declarations to our header file:

```
1  friend vector operator*(const vector & v, double s);
2  friend vector operator*(double s, const vector & v);
```

Let's implement the first by adding to the declaration file:

```
1  vector operator*(const vector & v, double s) {
2      // Copy v to start.
3      vector result = v;
4      // Then multiply all entries by scalar, s.
5      for(int i=0; i < v.len(); i++) {
6          result[i] *= s;
7      }
8      return result;
9  }
```

Observe how we didn't prepend `vector::` before `operator*` because these are `friend` functions.

Now, we can use the first implementation to achieve the second:

```
1  vector operator*(double s, const vector & v) {
2      return v*s;
3  }
```

And let's try it out:

```
 1  #include <iostream>
 2  #include "vector.h"
 3
 4  int main() {
 5      vec::vector v(5);
 6
 7      v[0] = -1.27;
 8      v[3] = 3.1;
 9
10      vec::vector w = 2*v;
11
12      w.print();
13
14      std::cout << "\n";
15
16      w = w*0.5;
17
18      w.print();
19
20      return 0;
21  }
```

prints:

```
-2.54
0
0
6.2
0

-1.27
0
0
3.1
0
```

Other binary arithmetic operators could also be implements as `friend` functions, e.g., vector addition and subtraction and component-wise multiplication and division. Component-wise multiplication would overload `operator*` for a third time but would accept two vectors as function arguments.

Overloading `operator«` We may overload `operator«` to send things to the output stream. This is a more C++ way to print than our current `print` function. We overload this operator by adding the following `friend` declaration to the header:

```
1  friend std::ostream & operator<<(std::ostream & outs, const vector & v);
```

Note that the `ostream` object belongs to the `std` namespace. Next, we add to the implementation file:

```
1  std::ostream & operator<<(std::ostream & outs, const vector & v) {
2      for(int i=0; i < v.len(); i++) {
3          outs << v.data[i] << "\n";
4      }
```

So now we stream the vector to `std::cout` just as we do for printing numbers and strings to the screen:

```
 1  #include <iostream>
 2  #include "vector.h"
 3
 4  int main() {
 5      vec::vector v(5);
 6
 7      v[0] = -1.27;
 8      v[3] = 3.1;
 9
10      std::cout << v;
11
12      return 0;
13  }
```

which prints

```
-1.27
0
```

```
0
3.1
0
```

7.2 Templating: a matrix class

Template classes allow you to generalize your classes. We will introduce the concept of templating by creating a templated matrix class. Below is a bare-bones template class for a matrix type. Note that the implementations for templates must all go in the header file; they cannot be split into seperate implementation files.

```
 1  #ifndef MATRIX_H
 2  #define MATRIX_H
 3
 4  #include <iostream>
 5  #include <cstdlib>
 6
 7  namespace mtx {
 8      template<class T>
 9      class matrix {
10      public:
11          // Default empty constructor.
12          matrix() {}
13
14          // Constructor from matrix dimensions.
15          matrix(int m, int n) {
16              this->m = m; this->n = n;
17              this->data = new T[m*n];
18              // Initialize to zeros.
19              for(int i=0; i < m*n; i++) this->data[i] = T(0);
20          }
21
22          // Copy constructor.
23          matrix(const matrix & src) : matrix(src.m, src.n) {
24              for(int i=0; i < m; i++) {
25                  for(int j=0; j < n; j++) {
26                      (*this)(i,j) = src(i,j);
27                  }
28              }
```

```
29          }
30
31          // Destructor.
32          ~matrix() { delete [] this->data; }
33
34          int n_rows() const { return this->m; }
35          int n_cols() const { return this->n; }
36
37          // Element accessor.
38          T & operator()(int i, int j) const {
39              #ifndef NDEBUG
40              check_indices(i,j);
41              #endif
42
43              // Data stored in row-major.
44              int k = i*this->n + j;
45
46              return this->data[k];
47          }
48      protected:
49          T * data;
50          int m; // n_rows
51          int n; // n_cols
52          void check_indices(int i, int j) const {
53              if (i < 0 || i >= this->m) {
54                  std::cerr << "Invalid row index, " << i << ".\n"
55                            << "Valid row indices are: 0-" << (this->m-1)
56                            << ".\n";
57                  exit(1);
58              }
59              if (j < 0 || j >= this->n) {
60                  std::cerr << "Invalid column index, " << j << ".\n"
61                            << "Valid column indices are: 0-" << (this->n-1)
62                            << ".\n";
63                  exit(1);
64              }
65          }
66      };
67  }
68
```

```
69  #endif
```

Observe how the class declaration begins with the statement, `template<class T>`. When new instances of the class, `matrix`, are instantiated, the type, `T`, must be specified. Notice that the private variable now holds a pointer to type, `T`, and the element accessor returns a reference to to type, `T`. Compare this to our previous `vector` class where each vector instance held type, `double`.

Also notice that the copy constructor uses something called an argument list. We've used it to call the initial constructor before proceeding with the copying of components from the `matrix` instance, `src`. To use this type of syntax (i.e., initializer lists) you may have to compiel with the flag `-std=c++11`, e.g., `g++ -std=c++11 main.cpp`.

Finally, observe that rather the keyword, `private`, we have used the keyword `protected`. This will allow inherited classes to use the private member variables and functions. We will talk about inheritance later, but for now, it essentially the same as marking these things as `private`.

Here is some example usage:

```
 1  #include <iostream>
 2  #include "matrix.h"
 3
 4  int main() {
 5      mtx::matrix<float>  A(10, 10);
 6      mtx::matrix<double> B(10, 12);
 7
 8      A(0,0) = 3.14;
 9      B(2,4) = -2*A(0,0);
10
11      std::cout << A(0,0) << "\n";
12      std::cout << B(2,4) << "\n";
13
14      mtx::matrix<double> C = B;
15      mtx::matrix<bool> D(3,3);
16
17      std::cout << C(2,4) << "\n";
18      std::cout << D(0,0) << "\n";
19
20      C(0,12);
21
```

```
22      return 0;
23  }
```

Outputs:

```
3.14
-6.28
-6.28
0
Invalid column index, 12.
Valid column indices are: 0-11.
```

7.2.1 Overloading `operator*`

Let us overload `operator*` to implement matrix multiplication. It would also make sense to overload `operator*` with other call signatures to implement scalar multiplication or vector broadcasting, but for now we will just stick with matrix multiplication.

Add the following two non-friend, non-member functions. They should be defined just after the class definition, but they should remain in the body of namespace, `mtx`.

```
 1  template<class T>
 2  matrix<T> operator*(const matrix<T> & L, const matrix<T> & R) {
 3      #ifndef NDEBUG
 4      check_sizes_for_matmul(L, R);
 5      #endif
 6
 7      matrix<T> P(L.n_rows(), R.n_cols());
 8      for(int i=0; i < L.n_rows(); i++) {
 9          for(int j=0; j < R.n_cols(); j++) {
10              // Inner product between row ‘i‘ of matrix, ‘L‘
11              // and column ‘j‘ of matrix, ‘R‘.
12              for(int k=0; k < L.n_cols(); k++) {
13                  P(i,j) += L(i,k)*R(k,j);
14              }
15          }
16      }
17
18      return P;
19  }
```

```
20
21  template<class T>
22  void check_sizes_for_matmul(const matrix<T> & L, const matrix<T> & R) {
23      if (L.n_cols() != R.n_rows()) {
24          std::cerr << "Size mismatch for matrix multiplication.\n"
25                    << "Left matrix has " << L.n_cols() << " cols, but\n"
26                    << "right matrix has " << R.n_rows() << " rows.\n";
27          exit(1);
28      }
29  }
```

Note that because we've chosen to implement these functions as non-member and non-friend, we must prefix the function definitions with `template<class T>`. This isn't necessary when the functions were declared inside of the class body (i.e., member or friend functions) since the class declaration already begins with `template<class T>`.

We've added a simple matrix multiplcation function above and a debugger functions to ensure that if two matrices are multiplied, that it is well-defined to do so. Otherwise, we print an error message and abort, as with the element accessor function. We've also wrapped the debugger function with compiler macros, so that it can be left out if we choose to do so.

Here is some example usage:

```
 1  #include <iostream>
 2  #include "matrix.h"
 3
 4  int main() {
 5      mtx::matrix<double> A(2,2);
 6      mtx::matrix<double> B(2,3);
 7
 8      A(0,0) = A(0,1) = A(1,0) = 1; A(1,1) = -1;
 9
10      B(0,0) = B(1,0) = 1;
11      B(0,1) = B(1,1) = -1;
12      B(0,2) = B(1,2) = 3;
13
14
15      mtx::matrix<double> C = A*B;
16
17      for(int i=0; i < C.n_rows(); i++) {
```

```
18          for(int j=0; j < C.n_cols(); j++) {
19              std::cout << C(i,j);
20              if (j < C.n_cols()-1)
21                  std::cout << ", ";
22          }
23          std::cout << "\n";
24      }
25
26      B*A;
27
28      return 0;
29  }
```

Output:

```
2, -2, 6
0, 0, 0
Size mismatch for matrix multiplication.
Left matrix has 3 cols, but
right matrix has 2 rows.
```

7.3 Inheritance

Often classes have a natural, hierarchical structure. For example, a row or column vector could be seen as natural "subclass" of the matrix class. So, supposing we had only written the matrix class, it would be nice if we could write a row or column vector class that inherited many of the properties of the parent matrix class, but with additional properties, unique to the vector classes. This is what inheritance may allow us to do.

Let's create another couple of vector classes, called `rowvec` and `colvec` in a file called `vectors.h`:

```
1  #ifndef VECTORS_H
2  #define VECTORS_H
3
4  #include "matrix.h"
5
6  namespace vec {
7      // Row vector class inherits from the matrix class.
8      template<class T>
9      class rowvec : public mtx::matrix<T> {
```

```
10      public:
11          // Constructor.
12          rowvec(int n) : mtx::matrix<T>(1, n) {}
13
14          // A length function.
15          int len() const { return this->n_cols(); }
16
17          // Accessors.
18          T & operator()(int i) const { return mtx::matrix<T>::operator()(0, i); }
19          // Add [] as a possibility, too.
20          T & operator[](int i) const { return (*this)(i); }
21      };
22
23      // Column vector class also inherits from the matrix class.
24      template<class T>
25      class colvec : public mtx::matrix<T> {
26      public:
27          // Constructor.
28          colvec(int n) : mtx::matrix<T>(n, 1) {}
29
30          // A length function.
31          int len() const { return this->n_rows(); }
32
33          // Accessor.
34          T & operator()(int i) const { return mtx::matrix<T>::operator()(i, 0); }
35          // Add [] as a possibility, too.
36          T & operator[](int i) const { return (*this)(i); }
37      };
38  }
39
40  #endif
```

We have defined two classes in the `vec` namespace, `rowvec` and `colvec`. Notice that the declaration of the `rowvec` class is slightly augmented,

```
1  class rowvec : public mtx::matrix<T> {
2  // ...
3  };
```

This says that the `rowvec` class inherits all the methods and variables defined for the `matrix` class. The keyword `protected`, rather than `private`

in the `matrix` class allows derived classes access to these variables and functions. The keyword, `public`, in the class declaration above signals that every `rowvec<T>` instance may be cast as a `matrix<T>` instance. This means that functions (such as our `operator*` in the `mtx` namespace) which expect `matrix` types in argument, can also now accept `rowvec` types: `rowvec` instances will be cast as `matrix` instances when possible.

Next, observe that the constructor simply calls the parent class constructor, with an argument of `1` for the number of rows. We also add a `len()` function which returns '$n_{cols}()$', calling the parent class function. `len()` is more natural for a vector class.

Finally, `operator()` is overwritten so that only a single argument is necessary. In the body, we simply call the parent class method. We also add `operator[]`, which duplicates the functionality of `operator()`.

Notice that any other methods that exist for the parent class, `matrix`, have been inherited. For example, it is not necessary to write a new destructor function since this already exists for the parent and the functionality is the same.

In the same, `vec`, namespace we have declared a `colvec` class which is analogous to the `rowvec` class except with the dimensions swapped.

Let's check out an example:

```
 1  #include <iostream>
 2  #include "matrix.h"
 3  #include "vectors.h"
 4
 5  int main() {
 6      vec::rowvec<float> x(5);
 7      vec::colvec<float> y(5);
 8
 9      for(int i=0; i < x.len(); i++) {
10          x(i) = i;
11          y[i] = -i; // elements may be accessed with either operator.
12      }
13
14      // ‘rowvec‘ and ‘colvec‘ inherit from ‘matrix‘,
15      // so ‘operator*‘ is well-defined if the shapes
16      // align, which always do for ‘rowvec‘ * ‘colvec‘.
17      // However, the result is a ‘matrix‘ object.
18      mtx::matrix<float> a = x*y;
19
```

```
20      std::cout << a.n_rows() << ", " << a.n_cols()
21                << " ... " << a(0,0) << "\n";
22
23      return 0;
24  }
```

This prints:

```
1, 1 ... -30
```

So `operator*` works by casting both the `rowvec` and `colvec` arguments as `matrix` types. The dimensions of these matrices match for matrix multiplication and the resulting 1x1 matrix is returned. Let's overwrite this behavior to return a scalar instead of a matrix when the operation, `rowvec*colvec` is performed. Add the inside the `vec` namespace but outside the class declarations:

```
 1  // Overwrite the matrix inherited `operator*`
 2  // so that a scalar is returned when a `rowvec` and
 3  // `colvec` are multiplied.
 4  template<class T>
 5  T operator*(const rowvec<T> & x, const colvec<T> & y) {
 6      // Use the inherited matrix operator defined in the `mtx` namespace.
 7      mtx::matrix<T> a = mtx::operator*(x,y);
 8      // Grab the only entry.
 9      T z = a(0,0);
10      // And return it.
11      return z;
12  }
```

Observe that we can call the `operator*` function in the `mtx` namespace, just like any other function. Now, the product between a `rowvec` and a `colvec` is more natural:

```
 1  #include <iostream>
 2  #include "matrix.h"
 3  #include "vectors.h"
 4
 5  int main() {
 6      vec::rowvec<float> x(5);
 7      vec::colvec<float> y(5);
```

```
 8
 9      for(int i=0; i < x.len(); i++) {
10          x(i) = i;
11          y[i] = -i; // elements may be accessed with either operator.
12      }
13
14      float a = x*y;
15      std::cout << a << "\n";
16
17      return 0;
18  }
```

Output:

```
-30
```