Contents

Chapter 1

C and C++ Introduction

1.1 Where can I get it from?

- **Web Version:** If you don't want to install any software, then you can use a free C++ compiler on a server at

  `https://www.tutorialspoint.com/compile_cpp_online.php`

- **Microsoft Windows:** Visual Studio

  The Microsoft C++ compiler is available for free from the Microsoft Store. Search for "visual studio" and choose "Visual Studio Community".

  (You might need to sign in with your Trinity email address and password.)

  - After you run the installer, you only need to select one workload to install:

    "**Desktop development with C++**" (in the Desktop & Mobile section)

- **Linux:** Use g++ in a "Terminal (or Ubuntu Shell)" (see below for instructions).

1.2 Keyboard Settings for Microsoft Windows

Make sure that your keyboard is set to US (not England) so that you can type single quotes ' and double quotes ".

- On your keyboard, the single quotes key ['] should be on the same key as double quotes ["]

  Use the [SHIFT] key to toggle between the two options.

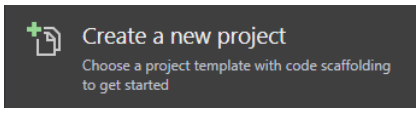
- If you get @ when you press the double quotes key, then your keyboard is set for England(UK). To fix this, open "Settings" and search for "keyboard". Then install "English(Australia)" or "English(United States)".

- You can quickly change your keyboard version by clicking on, e.g. [ENG US], in the bottom right corner of your screen.

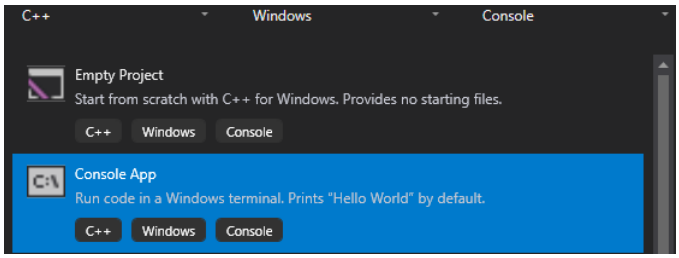
1.3 Your First Program

1.3.1 Visual Studio "Hello World"

- In the Windows start menu  , click on Visual Studio
- Click on



- Click on "Console App"



  and then click on "Next".

- Type **first** as your project name
- Click on "Create" ... the project will be created:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

- Use `Ctrl`+"mouse wheel" to change font size (or "Tools" → "Options").
- Run your program by clicking on "Local Windows Debugger"

- A terminal window will appear on your screen with the words "Hello World".

1.3.2 Linux "Hello World"

- Open a **Terminal (or Ubuntu Shell)**
- Type `g++ <Enter>` to see if the g++ compiler is installed on your computer.
  If g++ is not installed then your computer will give you instructions on how to install it.
- Type `nano first.cpp <Enter>` to create the source code file
- Carefully type the following:

```
#include <iostream>
int main()
{
    std::cout << "Hello World!\n";
}
```

- You can Cut/Paste in **nano** by selecting text with SHIFT Arrow Keys and then CTRL-k to cut and CTRL-u to paste. To copy text use ALT-6
- Type `<ctrl x> <y> <enter>` to exit and save
- Type `g++ -o first first.cpp <Enter>` to compile
- Type `ls -all <Enter>` to see your files
- Type `./first <Enter>` to run the program called "first"
- Feel free to use a different editor like "Visual Studio Code" to edit the source file.

1.4 Let's Start Programming :)

Example

```
#include <iostream>

int main()
{
  std::cout << "Hello World!\n";
}
```

Every program **must** include a `main()` function. This function tells the program where to start.

C++ is **case-sensitive** and so `main()` is different from `Main()`

Each C++ statement must end in a semicolon "`;`" and we can put multiple statements on one line!

Example

```
#include <iostream>

int main()
{
  std::cout << "Hello World!\n";  std::cout << "Hello\n";
}
```

The sequence "`\n`" tells the output to start a new line.

The command `std::cout` tells the compiler to **print** the following string `"Hello World!\n"`

We pronounce `cout` as "C" - "out", which is basically telling the **C** compiler to **out**put.

The **standard** prefix `std::` tells the compiler to look in the standard-library namespace.

The command `#include <iostream>` tells the compiler to use the standard library functions for **input** and **output**.

Note. You need to type in the programs **carefully**. Spaces do not matter, but `Cout` is different from `cout` and so `Cout` will generate a compiler error.

1.5 Variables

Variables are used to store information, and are a lot like the variables we use in mathematics. Note that variable names are **case sensitive** and so 'X' is **different** from 'x'. The main 'types' of variables are **Integer**, **Real** (floating point), **Boolean** (True/False), **Character** and **String**.

1.5.1 Integer Type

Example

```
#include <iostream>

int main()
{
   int i;   // we are declaring the variable i to be an integer ...,-2,-1,0,1,2,...
   i = 32;
   std::cout << "i = " << i;
}
```

The output of this program is

```
i = 32
```

Note

The text

```
// we are declaring the variable i to be an integer ...,-2,-1,0,1,2,...
```

is called a **comment**, which is ignored by the compiler. We use comments to help us understand programs.

A multiline comment is begun with `/*` and ended with `*/`

```
 /*  first comment
    second comment */
```

There is nothing special about the string `"i = "` in `std::cout << "i = " << i;`

We can change the program to

Example

```
#include <iostream>

int main()
{
   int i;
   i = 32;
   std::cout << "The value is " << i;
}
```

The output of this program is

```
The value is 32
```

Unlike Python, indentation does not matter, but you must have a semicolon ";" at the end of each line. If your program does not compile, then check for missing semicolons!

1.5.2 Real (floating point) Type

Example

```
#include <iostream>
int main()
{
  double y;
  y = 1.0/3;
  std::cout.precision(20);
  std::cout <<  "double y      = " << y << "\n";
}
```

The last few digits might be incorrect in a floating point number. Do not Panic. This is normal. We normally deal with this by only printing out, say, 7 decimal places.

Example

```
#include <iostream>
int main()
{
double y;
y = 1.0 / 3;
std::cout.precision(7);
std::cout << "double y      = " << y << "\n";
}
```

Also note that `2.7048138294215165e+100` means

$$2.7048138294215165 \times 10^{100}$$

1.5.3 Character Type

A character is a *single* letter, and we enter characters with single quotes ''

Example

```
#include <iostream>
int main()
{
   char c;
   c = 'r';
   std::cout << c;
}
```

Note

On your keyboard, the single quotes key ['] should be on the same key as double quotes ["]

Use the [SHIFT] key to toggle between the two options.

1.5.4 String Type

A string is any sequence of characters and we enter strings with double quotes `" "`

Example

```
#include <iostream>
#include <string>
int main()
{
  std::string mystring;
  mystring = "hello";
  std::cout << mystring;
}
```

Notice that we are accessing a new library called `string` by entering `#include <string>`

`std::string` is a new type that lets us store strings.

Identifiers and Literals

The name we give to a variable is called an **identifier** and its value is called a **literal**.

In the line

```
mystring = "hello";
// mystring is called the identifier
// "hello" is the literal
```

1.6 Changing the Value of Variables

C++ has many ways to change the value of a variable. We will look at the simplest methods.

Example

```
#include <iostream>
int main()
{
   int i;
   i = 1;
   std::cout << "i = " << i << "\n";
   i = i + 1;
   std::cout << "i = " << i << "\n";
}
```

Notice that `i = i + 1;` is not a maths equation!

It adds one to `i` and puts the result back in the variable `i`

C++ has quicker commands to do the same operation: `i++` and `i+=1`

Example

```
#include <iostream>
int main()
{
  int i;
  i = 1;
  i++;
  std::cout << "i = " << i << "\n";
  i += 1;
  std::cout << "i = " << i;
}
```

Note

`i--` is the same as `i=i-1`

If we divide two integers, we will always get an integer (truncated to zero decimal places).

Example

```
#include <iostream>
int main()
{
  int x,y,z;
  y = 10;
  x = 3;
  z = y/x;
  std::cout << "z = y/x = " << z;
}
```

To do maths, we should use the `double` type.

Note

Be careful: if `x,y` are of type `int` then `y/x` is an integer, even if we put `z = y/x` where `z` is `double`.

To force `double` calculations, we can use `z = 1.0*y/x`

Example

```
#include <iostream>
int main()
{
  double z;
  int x, y;
  y = 10;
  x = 3;
  z = 1.0*y/x;
  std::cout << "z = y/x = " << z;
}
```

1.7 Entering data while the program is running

Sometimes we want the user to enter data while the program is running. The `cin >> x` command will pause the program while the user enters a value for `x`. After the user presses `<Enter>` the program will start running again.

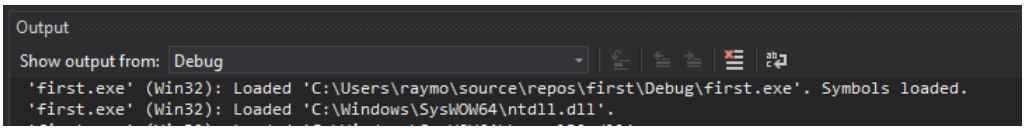
Example

```
#include <iostream>
int main()
{
  double z, x, y;
  std::cout << "x = ";
  std::cin >> x;
  std::cout << "y = ";
  std::cin >> y;
  z = y / x;
  std::cout << "z = y/x = " << z;
}
```

1.8 The Windows executable file .exe

Note that, unlike Python, Visual Studio C++ generates an executable file called **first.exe** in the debug folder in our top folder (first). We can give this executable to our friends and it should run on their computer without installing C++.

The location of **first.exe** is shown in the first line of the compile window.



If you open the containing folder with file explorer, and then double-click on **first.exe**, the program will run, but the console window will immediately close after the program finishes.
To stop the console window from closing, see the next section.

1.8.1 Console Window

The output of the program will be displayed in a console window. Make sure that you close the window after the program finishes, otherwise further console windows will automatically close and you won't be able to see the output!

Note that you can force the program to pause (so that you can always see the output) by using `std::cin` to read the keyboard for input.

Example

```
#include <iostream>
int main()
{
  char c;
  std::cout << "Hello World! \n";
  std::cout << "Press x <Enter> to end this program";
  std::cin >> c;
}
```

Note. You should also be able to pause the program by writing
`std::cin.ignore(); std::cin.get();`

1.9 using namespace std

The names `std::cout` and `std::cin` are in the standard-library namespace, and this is why we put `std::` in front of `cout` and `cin`.

We can avoid writing `std::` in front of every input and output command by writing

```
using namespace std;
```

on the line before `int main()`

Example

```
#include <iostream>
using namespace std;
int main()
{
  char c;
  cout << "Hello World! \n";
  cout << "Press x <Enter> to end this program";
  cin >> c;
}
```

The Advantage. We only have to write `cin` instead of the longer `std::cin`.

The Disadvantage. Now we are **not** allowed to create another variable called `cin` since that would conflict with `cin` in the standard-library namespace. It is safest to use variable names like `x`, `y`, `z` to avoid conflicts, but the compiler will tell you if you accidentially use a variable name that it is already using.

1.10 Summary of C++

Variables (store information)

```
// means comment (ignored by compiler)
  bool B;     // B = true;  B = false; // B is called a Boolean variable
  int i;      // i is an integer ... -3, -2, -1, 0, 1, 2, 3, ...
  double x;   // x is a real number (to about 16 significant figures)
  char c;     // single letter of alphabet, for example  c = 'a';
  int M[10];  // M is an array of integers with elements M[0], ..., M[9]


// strings (for storing sequences of letters in a variable)
  #include <string>
  std::string mystring;
  mystring = "hello there";

  struct {double x; double y;} mypoint;
    // mypoint is a structure (container) with two variables x and y
    // for example, we can write
       mypoint.x = 1;
       mypoint.y = 2;
```

Commands (tell the computer what to do)

```
// means comment (ignored by compiler)
// input and output
  std::cout << "Hello World"; // output
  std::cin >> x;              // input

// control flow (direction) of executed code
  if (condition) {do this;} else {do that;}  // if-then-else

// repeating steps (iteration)
  while (condition) {do this;}       // repeat with while
  do {this;} while (condition);      // repeat with do-while
  for (i=0; i < 10; i++){do this;}   // repeat with for

// functions
  double square(double x) {return x*x;} // square is a function that outputs x*x
```

Chapter 2

Changing Program Flow

In C++, the computer normally reads the commands in order from **top** to **bottom**

Example

```
#include <iostream>
int main()
{
 STEP 1 (top)
 STEP 2
 STEP 3
 STEP 4 (bottom)
}
```

We use `if (condition) {do this;} else {do that;}` to tell the computer to 'branch out' to a different set of steps, depending on whether `condition` is `true` or `false`.

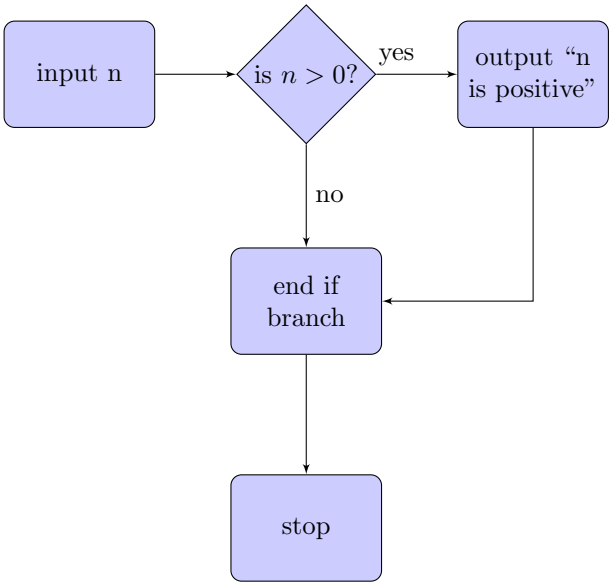
if-else

```
if (condition) {do this;}
if (condition) {do this;} else {do that;}
if (condition) {do 1;}
   else if (condition2) {do 2;}
   else if (condition3) {do 3;}
   else {do 4;}  // otherwise, if all other conditions fail do this part

 Use == to test if two expressions are equal
 Use != to test if two expressions are NOT equal
 Use && for AND         Use || for OR            Use ! for NOT
 All if conditions are enclosed with brackets ()

// other commands to control program flow
switch (expression)  // expression must evaluate to an integer
{
  case myint1:
       do this1; //  if expression == myint1;
       break;
  case myint2:
       do this2;  //  if expression == myint2;
       break;
  default:
       do this3; // for all other values of expression
}
// Advanced coders sometimes use the conditional expression
   expr1 ? expr2 : expr3
// if expr1 is true then output expr2 else output expr3
```

2.1 if (condition) {do this;}

The following program branches to output "n is positive" if a positive number is entered. If the number entered is not positive then there are no further commands to read and so the program stops (outputting nothing).

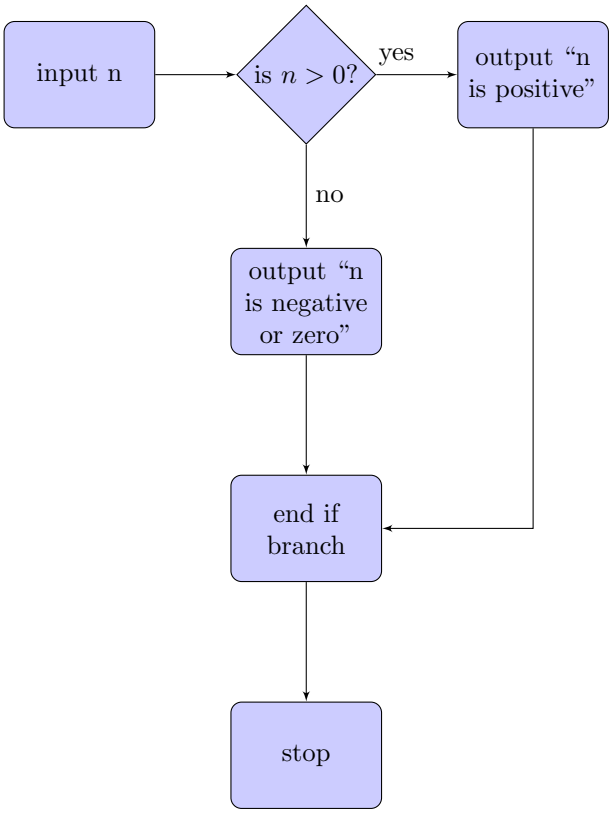
```
#include <iostream>
using namespace std;
int main()
{
   double n;
   cout << "n = ";
   cin >> n;  // input n
   if (n > 0) { cout << "n is > 0 \n"; }
}
```



2.2 if (condition) {do this;} else {do that;}

The `else` command tells the computer what to do if `condition` is `false`.

```
#include <iostream>
using namespace std;
int main()
{
  double n;
  cout << "n = ";
  cin >> n;  // input n
  if (n > 0)
  {
    cout << "n is positive";
  }
  else
  {
    cout << "n is negative or zero";
  }
}
```



2.3 if (condition) {do 1;} else if (condition2) {do 2;} else {do 3;}

We can use `else if` to handle more cases.

Example

```
#include <iostream>
using namespace std;
int main()
{
    double n;
    cout << "n = ";
    cin >> n;  // input n
    if (n > 0)
    {
        cout << "n is positive";
    }
    else if (n < 0)
    {
        cout << "n is negative";
    }
    else
    {
        cout << "n is zero";
    }
}
```

Note. You can write many `else if` commands in the same "multi-decision". Be carefull to make sure that the `conditions` are mutually exclusive (cannot both occur at the same time); then the final `else` handles the "none of the above" case.

2.4 AND, OR, NOT, and ==

C++ has the following conventions:

Logic

- Use `==` to test if two expressions are equal
- Use `!=` to test if two expressions are NOT equal
- Use `&&` for AND
- Use `||` for OR
- Use `!` for NOT
- All if statements are enclosed with brackets `()`

Example

`if (!(n==0))` is the same as `if (n != 0)`

but

`if !(n==0)` generates an error since we need brackets `()` around the `condition` for an `if` command.

Example

`if ((n > 4) || (n < -4))` means $n > 4$ or $n < -4$

but

`if (n > 4) || (n < -4)` generates an error (missing brackets `()`).

2.5 Integer Reminder after division

The command `m % n` outputs the remainder when `m` is divided by `n`. For example, `3 % 2` outputs `1`.

Example

An integer n is called **even** if it has a remainder of 0 when it is divided by 2, that is `n % 2 == 0`

An integer n is called **odd** if it has a remainder of 1 when it is divided by 2, that is `n % 2 == 1`

The following program determines if a number is odd or even.

```
#include <iostream>
using namespace std;
int main()
{ int n;
  cout << "Enter an integer: ";
  cin >> n;
  if (n % 2 == 0)
  {
    cout << n << " is even";
  }
  if ((n % 2 == 1) || (n % 2 == -1))  // we could just use  else  here
  {
    cout << n << " is odd";
  }
}
```

2.6 switch ... case

C++ programs often change execution depending on the value of a variable, and we often use the `switch...case` construct for this, even though we could just use `if...else`

Example

```
#include <iostream>
using namespace std;
int main()
{
  int r;
  cout << "Enter an integer: ";
  cin >> r;
  cout << "r = " << r << " \n";
  switch (r)
  {
   case 0:
     cout << "r is zero";
     break;
   case 1:
     cout << "r is 1";
     break;
   default: // r is any other value
     cout << "r is bigger than 1, or negative";
  }
}
```

If we omit `break` then **all** following cases will be executed until `break` is found.

Note

- `switch(r) ... case` is used when the variable `r` can have only finitely many values, normally `0, 1, 2, 3, ..., n`. In particular, it is not used for real numbers with inequalities.
- `switch(r) ... case` runs **faster** than `if...else` because the program jumps directly (by using `r`) to the location of each case.

Chapter 3

Loops (repeating steps)

In C++, the computer normally executes the commands in order from **top** to **bottom**

Example

```
#include <iostream>

int main()
{
 STEP 1 (top)
 STEP 2
 STEP 3
 STEP 4 (bottom)
}
```

We use loops to repeat the same steps many times.

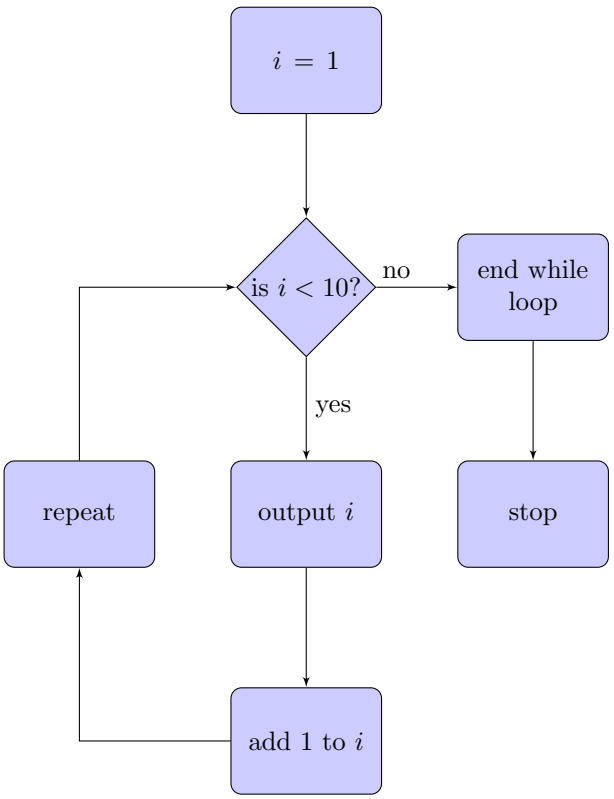
Repeating steps (iteration)

```
while (condition) {do this;}        // "if (condition) is true" then  {do this;}
                                    // "if (condition) is still true" then do it again.

do {this;} while (condition);       // do {this;}
                                    // "if (condition) is still true" then do it again.

for (i=0; i < 10; i++){do this;}    // {do this;} for each i = 0, ... , 9
```

3.1 While Loop

The following program uses a "while loop" to output the numbers $1, 2, 3, 4, 5, 6, 7, 8, 9$

```
#include <iostream>

int main()
{
 int i;
 i = 1;
 while (i < 10)
 {
   std::cout << i << "\n";
   i = i + 1;
 }
}
```



3.2 Code Blocks

Notice that the above code uses the *code block*

```
{
   std::cout << i << "\n";
   i = i + 1;
}
```

Everything in the braces `{}` gets repeated in the while loop.

Note

If we change the code to

```
#include <iostream>

int main()
{
 int i;
 i = 1;
 while (i < 10)
 {
   i = i + 1;
 }
 std::cout << i << "\n";
}
```

then `std::cout << i << "\n";` is not repeated in the while loop, and so `i` is only printed once!

3.3 User Input

We can also check for user input to decide when to stop the loop.

Example

```
#include <iostream>

int main()
{ int i;
  char c;
  i = 1;
  c = 'y';
  while (c == 'y')
  { std::cout << i << "\n";
    i = i + 1;
    std::cout << "Continue? (y/n): ";
    std::cin >> c;
  }
}
```

3.4 Do While Loop

Sometimes it is better to test for the "repeat condition" at the end of the loop.

Example

```
#include <iostream>

int main()
{
  int i;
  char c;
  i = 1;
  do
  { std::cout << i << "\n";
    i = i + 1;
    std::cout << "Continue? (y/n): ";
    std::cin >> c;
  } while (c == 'y');
}
```

3.5 Infinite While Loop

To stop the following infinite loop, you can close the output window, or type `CTRL C`

Example

```
#include <iostream>

int main()
{
  int i;
  i = 1;
  while (true)
  { std::cout << i << "\n";
    i = i + 1;
  }
}
```

3.6 For Loop

A `for` loop is normally the quickest way to write a loop.

Example

```
#include <iostream>

int main()
{
  int i;
  for (i=0; i < 10; i++)
     std::cout << i << "\n";
}
```

We can put the integer declaration inside the `for` loop

Example

```
#include <iostream>

int main()
{
   for (int i=0; i < 10; i++)
     std::cout << i << "\n";
}
```

and even write all code on one line.

Example

```
#include <iostream>

int main()
{
   for (int i=0; i < 10; i++)  std::cout << i << "\n";
}
```

We can increase `i` in steps of 2 by changing `i++` with `i=i+2`

Example

```
#include <iostream>

int main()
{
   for (int i=0; i < 10; i=i+2)  std::cout << i << "\n";
}
```

Note

The variable `i` is now a local variable (inside the `for` loop) and so cannot be accessed outside the loop. The following code will generate an error when we try to compile.

```
#include <iostream>

int main()
{
   for (int i=0; i < 10; i=i+2)
     std::cout << i << "\n";
   std:cout << i;
}
```

To fix this, we put `int i` outside the `for` loop

```
#include <iostream>

int main()
{
  int i;
  for (i = 0; i < 10; i=i+2) std::cout << i << "\n";
 std::cout << "The final value of i is " << i;
}
```

Notice that the final value of `i` is 10, which is two more than the last value of `i` inside the loop.

3.6.1 Brackets

Remember that we use brackets `{}` to determine what gets repeated in the loop.

Example

```
#include <iostream>

int main()
{
  int i;
  for (i = 1; i <= 10; i=i+1)
   {
    std::cout << i << "\n";
    std::cout << "hi \n";
   }
}
```

Notice that **hi** is printed 10 times.

If we put `std::cout << "hi \n";` outside the brackets

```
#include <iostream>

int main()
{
  int i;
  for (i = 1; i <= 10; i=i+1)
   {
    std::cout << i << "\n";
   }
  std::cout << "hi \n";
}
```

then **hi** is only printed once (at the end).

3.6.2 Order of Execution

Statements are executed in order from top to bottom.

Compare

```
#include <iostream>

int main()
{ int i;
  i = 1;
  while (i < 10)
  { std::cout << i << "\n";
    i++;
  }
}
```

with

name ...

```
#include <iostream>

int main()
{ int i;
  i = 1;
  while (i < 10)
  { i++;
    std::cout << i << "\n";
  }
}
```

Chapter 4

Time

In this chapter, we are going to measure the time it takes for a compiled C++ program to run, which will essentially measure the speed of our computer's CPU.

To measure time, we are going to use a library called `ctime`. This library has a function called `clock()` that returns the number of milliseconds since our program began.

Note

`clock()` returns a new type of variable called `clock_t`.

The type `clock_t` is similar to the `int` type.

Example

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

int main()
{
    clock_t mytime;   // mytime is a variable of type clock_t,
                      // which is similar to an integer
    mytime = clock(); // set mytime to be the number of milliseconds
                      // since the program began
    cout << "This program began = " << mytime << " milliseconds ago\n";
}
```

Before we do a speed test of our computer, let's look at a simple example for measuring elapsed time.

Example

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

int main()
{
    clock_t starttime, currenttime;
    starttime = clock();
    cout << "Program began = " << starttime << " milliseconds ago\n";
    cout << "startime = " << starttime << "\n";
    for (int i = 0; i < 10;i++)
    {
        cout << "i = " << i << "\n";
        currenttime = clock();
        cout << "clock = " << currenttime << "\n";
        cout << "Elapsed time in for-loop = currenttime-starttime = "
             << currenttime - starttime << " ms \n";
    }
}
```

4.1 Speed Test

Example

```
#include <iostream>
#include <ctime>

int main()
{
 clock_t starttime;
 int count = 0;
 int x2, y2, z2;
 z2 = 0;
 starttime = clock();
 while (count < 20000000) // 20 million
  {
    x2 = 47586 + count;
    y2 = 86869 + count;
    z2 = x2 * y2;
    count++;
   }
 std::cout << z2 << "\n"; // need this to remove optimizations
                          // (the compiler ignores unnecessary code with Compile->Release)
 std::cout << "time = " << (clock() - starttime) / 1000.0 << " seconds \n";
 // time = 0.005 seconds on 2022 Surface Pro (compile for release (not debugging))
 // time = 2.860 seconds with Python on 2022 Surface Pro
 // C++ is (2.860/0.005 = 572) times faster!!!
```

This is about 500 times faster than Python (choose "release" instead of "debug" when compiling).

Chapter 5

Integer Division and Remainder after Division

When we divide an integer a by a positive integer b, we can write the answer as an integer quotient q and an integer remainder r

$$\frac{a}{b} = q + \frac{r}{b} \qquad \text{where } 0 \leq r < b$$

You may have used the process called **long division** in school to calculate q and r.

C++ has the operation `/` to calculate the quotient q, and the operation `%` to calculate the remainder r.

These operations are often used in programs that use computer graphics (pictures), and for finding prime numbers.

Example

```
#include <iostream>

int main()
{
 int q,r;
 q = 10 / 3;
 r = 10 % 3;
 std::cout << "10.0/3 = " << 10.0 / 3 << "\n";
 std::cout << "10/3 = " << q << " + " << r << "/3" << "\n";
 std::cout << "in C++, 10/3 = " << 10 / 3 << "\n";
}
```

Unlike Python, C++ allows negative remainders

Example

```
#include <iostream>

int main()
{
 int q,r;
 q = -10 / 3;
 r = -10 % 3;
 std::cout << "-10.0/3 = " << -10.0 / 3 << "\n";
 std::cout << "-10/3 = " << q << " + " << r << "/3" << "\n";
 std::cout << "in C++, -10/3 = " << -10 / 3 << "\n";
}
```

Chapter 6

Random Numbers

The `rand()` function randomly chooses a number between 0 and `RAND_MAX`

Example

```
#include <iostream>
#include <ctime>

int main()
{
 int r;
 srand(time(NULL));
 r = rand();
 std::cout << "Random number: " << r << " \n";
 std::cout << "RAND_MAX = "<< RAND_MAX;
}
```

The function `srand(time(NULL))` sets the *seed* (start) of the random number generator to the computer's clock time so that the random numbers are different each time we run the program. Notice that the random number is (slightly) different each time we run this program. The function `time(NULL)` returns the number of seconds since the start of 1 Jan 1970, and so `srand(time(NULL))` only sets a different seed each second.

If we generate 4 subsequent random numbers then the 2nd, 3rd and 4th will be more different each time we run the program, but the first random number does not change much.

Example

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

int main()
{
 int r;
 srand(time(NULL));
 for (int i = 0; i < 4; i++)
 {
   r = rand();
   std::cout << "Random Number : " << r << "\n";
 }
}

OUPUT:
Random Number : 15375
Random Number : 9880
Random Number : 13600
Random Number : 24723
```

Sometimes (in simulations, and e.g. Minecraft) we want the random numbers to be the same each time we run the program. In this case we set the seed of the random number generator to the same value each time we run the program.

If we start with the same seed each time then the random numbers will start the same way.

Example

```
#include <iostream>
#include <ctime>

int main()
{
 int r;
 srand(7);
 for (int i = 0; i < 10; i++)
 {
   r = rand();
   std::cout << "Random number: " << r << " \n";
 }
}
```

Warning

We should only use `srand(time(NULL))` **once** in our program.

If we write

```
#include <iostream>
#include <ctime>

int main()
{
 int r1, r2;
 srand(time(NULL));
 r1 = rand();
 srand(time(NULL));
 r2 = rand();
 std::cout << "Random number r1: " << r1 << " \n";
 std::cout << "Random number r2: " << r1 << " \n";
}
```

Then the random numbers `r1` and `r2` will be very similar, or even the same!! since `time(NULL)` does not change by much (or at all) in the program (because the program runs very fast).

6.1 Random numbers between 0 and n

To get a random number between 0 and n, we just find the remainder when the random number is divided by $n+1$.

Example

To find 10 random numbers between 0 and 3, we can use the following code:

```
#include <iostream>
#include <ctime>

int main()
{
 int r;
 srand(7);
 for (int i = 0; i < 10; i++)
 {
   r = rand() % 4; // this is a comment: r = 0,1,2,3
   std::cout << "Random number: " << r << " \n";
 }
}
```

Note that `//` starts a "comment" in our code (not executed), and that `%` finds the remainder after division.

6.2 Random numbers between m and n

To find random numbers between m and n we just find the remainder when the random number is divided by $n-m+1$ and then add m.

Example

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

// find random numbers between m and n (including m and n)
int main()
{
    int r, m, n;
    m = 12;
    n = 17;
    srand(time(NULL));
    for (int i = 0; i < 100; i++)
    {
        r = rand() % (n - m + 1);  // r = 0, 1, ..., n-m
        r = r + m;  // r = m, m+1, ..., n
        std::cout << "Random number: " << r << "\n";
    }
}
```

6.3 Larger random numbers

In the first example, we saw that `RAND_MAX = 32767` which means that the largest random number that we can make is 32767. If we want larger random numbers then we could just multiply random numbers together.

Example

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

int main()
{
    int r1, r2;
    srand(time(NULL));
    for (int i = 0; i < 20; i++)
    {
      r1 = rand();
      r2 = rand();
      std::cout << "Big random number = " << r1 * r2 << "\n";
    }
}
```

The obvious question is "Are these numbers still random?". In most cases they are "random enough". See the next section for a better algorithm.

The random numbers generated by a computer are called pseudo-random because they are never truly random. To get a truly random number we would use a "random-device" like a Geiger counter measuring radioactive decay, or even throwing a real-life $n-$sided die.

6.4 Algorithm for creating random numbers

There are many algorithms for creating random numbers, but the most popular one is

$$x_{i+1} = (ax_i + c) \bmod m \quad \text{where } i = 0, 1, 2, \ldots$$

where

- "mod m" means remainder after division by m.
- $x_0 = 1$ (seed)
- x_1 = first random number, x_2 = next random number, etc.
- m is the computer's word size which is typically 2 bytes (16 bits), and so $m = 2^{16} = 65536$. If possible, we choose m to be as large as possible (so that we can generate more random numbers without repetition). Computers can now handle at least 32 bit numbers, and so a good choice in modern algorithms is $m = 2^{31} - 1 = 2147483647$.
- a is a natural number between $0.01m$ and $0.99m$. A good choice for a is 314159262.
- When a is chosen well, we can have $c = 1$.
- At most $\frac{m}{1000}$ random numbers should be generated (to avoid repetition).

See page 184 of "The Art of Computer Programming, Volume 2" by Donald E. Knuth for more details. Half of this book is devoted to random numbers!

Example

Using the above algorithm, we can generate larger random numbers as follows.

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

int main()
{
    unsigned long long x = 1;
    cout << "size of x is " << sizeof(x) << " bytes \n";
     // x is 8 bytes (64 bits) and so max x = 2^64-1 =  18446744073709551615;
    unsigned long long m = 2147483647;
    unsigned long long a =  314159262;
    unsigned long long c = 1;
    for (int i = 0; i < 10; i++)
    {
        cout << "random number = " << x << "\n";
        x = (a * x + c) % m;
    }
}
```

The first C book "The C Programming Language" by the creators Brian Kernighan and Dennis Ritchie actually has the C code for the `rand()` function on page 45, which is essentially given in this example:

Example

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

int main()
{
    unsigned long int myseed = 1;

    for (int i = 0; i < 10; i++) // make 10 random numbers
    {
      myseed = myseed * 1103515245 + 12345;
      myseed = (myseed / 65536) % 32768;
      std::cout << myseed <<  "\n";
    }
}
```

6.5 Modern random number generators

The C++ compiler has a library called `<random>` which generates better random numbers. To use it, we just write `#include <random>` at the top of our code.

Example

```
#include <iostream>
#include <ctime>
#include <random>

using namespace std;

int main()
{   // generate 10 random integers between 1 and 6 (including 1 and 6)
    default_random_engine myrandomengine{};
    uniform_int_distribution<> myrand{ 1,6 };
    myrandomengine.seed(time(NULL));
    for (int i=0; i < 10; i++)
    {
        cout << myrand(myrandomengine) << "\n";
    }
}
```

In the above example, we generate 10 random integers between 1 and 6. This gives better random numbers than `1+rand() % 6`, and now we can choose random numbers up to 2^{32}.

Note

This new random number generator comes in 2 parts:

1. The engine (we should always use the `default_random_engine`, unless you know what you are doing).
2. The distribution. We typically use `uniform_int_distribution` since that makes all the integers that are generated equally likely.

It is possible to choose random numbers from a different distribution, for example the Normal Distribtion.

Example

```
#include <iostream>
#include <ctime>
#include <random>

using namespace std;

int main()
{
    default_random_engine myrandomengine{};
    normal_distribution < double > myrand{ 8,2 };  // mean = 8, std dev = 2
    myrandomengine.seed(time(NULL));
    for (int i=0; i < 10; i++)
    {
        cout << myrand(myrandomengine) << "\n";
    }
}
```

We can also generate random real numbers in an interval $[a, b)$.

Example

In this example, `myrand` is a real number in $[2, 7)$, with all outcomes in the interval equally likely.

```
#include <iostream>
#include <ctime>
#include <random>

using namespace std;

int main()
{
    default_random_engine myrandomengine{};
    uniform_real_distribution < double > myrand{ 2,7 };  // 2 <= myrand < 7
    myrandomengine.seed(time(NULL));
    for (int i=0; i < 100; i++)
    {
        cout << myrand(myrandomengine) << "\n";
    }
}
```

There are more distributions that we can use. For example, we can sample from the binomial distribution.

Example

In this example, `myrand` is a random number from the binomial distribution with $n = 10$ and $p = 0.2$. Since $np = 2$, we expect most of the numbers to be close to 2 .

```
#include <iostream>
#include <ctime>
#include <random>

using namespace std;

int main()
{
    default_random_engine myrandomengine{};
    binomial_distribution <> myrand{ 10, 0.2 };  // n = 10, p = 0.2
    myrandomengine.seed(time(NULL));
    for (int i=0; i < 100; i++)
    {
        cout << myrand(myrandomengine) << "\n";
    }
}
```

Chapter 7

Functions

We can make the function $f(x) = x^3$ as follows

Example

```
#include <iostream>

double cube(double x)    // the name of the function is cube
{                        // input x        with type double
  return x*x*x;          // output x*x*x   with type double
}

int main()
{
  std::cout << cube(2);
}
```

Notice that `main()` is also a function. This makes C++ different from other languages: `main()` is always executed first.

Functions would normally return a value, and so `main()` would often have `return 0;` at the end. (The return value of 0 gets sent to the operating system.)

We can choose any name for the function, and we can even call the function `f`

Example

```
#include <iostream>

double f(double x)
{
  return x*x*x;
}

int main()
{
  std::cout << f(2);
}
```

The parameter `x` can be changed to any other variable

Example

```
#include <iostream>

double f(double a)
{
  return a*a*a;
}

int main()
{
  std::cout << f(2);
}
```

All parameters (and variables) inside the function are **local** and cannot be accessed outside the function. We get an error when we run the following program:

```
#include <iostream>

double f(double a)
{
  return a*a*a;
}

int main()
{
  std::cout << f(2);
  std::cout << a;  // error: a is not defined outside the function
}
```

7.1 Function Parameters

Functions can have as many parameters as we like

Example

```
#include <iostream>

double f(double a, double b, double c, double x)
{
  return a*x*x+b*x+c;
}

int main()
{
  std::cout << f(1,2,3,1.1);  // evaluate 1*x*x+2*x+3 at x = 1.1
}
```

7.2 Function Return Types

We need to specify the return type of each function before the name of the function.

Example

```
#include <iostream>

int daysinweeks(int numweeks) // this function outputs an integer
{
  return 7*numweeks;
}

int main()
{
  std::cout << daysinweeks(2);
}
```

If we don't want to return a value, then use `void`

Example

```
#include <iostream>

void sayhello()
{ std::cout << "hello"; }

int main()
{
  sayhello();
}
```

Now let's write a function that determines if a number is odd or even (returning a string).

Example

```
#include <iostream>
#include <string>

std::string oddeven(int n)   // string is part of the std library
{
  if (n % 2 == 0)
  {
    return "even";
  }
  else
  {
    return "odd";
  }
}

int main()
{
   int i;
   for (i = -30; i < 31; i++)
   {
     std::cout << i << " is " << oddeven(i) << "\n";
   }
}
```

Finally, let's just print the even numbers.

Example

```
#include <iostream>
#include <string>

std::string oddeven(int n)
{
  if (n % 2 == 0) {return "even";}
  else {return "odd";}
}

int main()
{
  int i;
  for (i = -30; i < 31; i++)
  {
    if (oddeven(i) == "even")
    {std::cout << i << "\n";}
  }
}
```

7.3 Changing the value of a variable in a function

Variables passed into a function are copied into local variables, and so normally, we cannot change the value of a variable passed into a function.

In the following example, the value of `y` is not changed. This is the normal behaviour of a function.

Example

```
#include <iostream>

void add1(double x)
{
    x++;
    std::cout << "Inside add1, x = " << x << "\n";
}

int main()
{
    double y = 1;
    std::cout << "y = " << y << "\n";
    add1(y);
    std::cout << "After add1, y = " << y << "\n";
}
```

We can pass parameters (variables) *by reference* to a function. This enables the function to change the variable outside of the function.

Example

```
#include <iostream>

void add1(double &x)   // &x means we are passing the memory location of x
{                      // into the function
  x++;
  std::cout << "Inside add1, x = " << x << "\n";
}

int main()
{
   double y = 1;
   std::cout << "y = " << y << "\n";
   add1(y);
   std::cout << "After add1, y = " << y << "\n";
}
```

Functions work in two ways:

(a) The standard way is that variables passed into the function are copied into local variables and so, outside the function, the variables cannot be changed. This is the `void add1(double  x){}` form

(b) The other way a function can work is to work directly with external variables (the memory location of the variable is passed into the function, and we say that the variables are passed "by reference"). When we do this, we can change variables passed into a function. This is unsafe in the sense that we might accidentally change a variable in a function when we don't want to, but sometimes it is the best way to deal with a problem. This is the `void add1(double  &x){}` form which means "pass `x` by reference" into the function.

Note. A C++ function can only return *one* value (unlike Python). If you need to return more than one value then could do so by changing variables passed into the function.

Chapter 8

Global Variables

Global variables are accessible throughout the entire program. Unlike Python, there is no `global` command. We just put the definition of such a variable outside the `main()` function; it then automatically becomes global (and so can be changed inside *any* function).

Example

```
#include <iostream>

int score = 0;   // score is a global variable

void addbonus()
{
  score = score + 10;
}

int main()
{
  std::cout << score << "\n";
  addbonus();
  std::cout << score << "\n";
}
```

Chapter 9

Example: Acey Ducey Game

Acey Ducey is a card game where two cards $\boxed{a}$ and $\boxed{b}$ are drawn from a deck of cards. The cards are ordered

$$\boxed{2}, \boxed{3}, \ldots, \boxed{10}, \boxed{\text{Jack}}, \boxed{\text{Queen}}, \boxed{\text{King}}, \boxed{\text{Ace}}$$

and we display the two drawn cards on the table as

$$\boxed{a} \qquad \boxed{b}$$

where $\boxed{a} < \boxed{b}$.

We then bet if the next card $\boxed{c}$ drawn is between $\boxed{a}$ and $\boxed{b}$, that is, if $\boxed{a} < \boxed{c} < \boxed{b}$.

This game was originally written in BASIC by Bill Palmy.
Source: BASIC Computer Games (1973)

Website: Basic Computer Games

Let's first write a function to output the card names.

```
#include <iostream>
#include <string>
std::string card(int i)
{
  if (i <= 10) { return std::to_string(i); }
  if (i == 11) { return "Jack"; }
  if (i == 12) { return "Queen"; }
  if (i == 13) { return "King"; }
  if (i == 14) { return "Ace"; }
  return "Not a card";  // all other values for i
}

int main()
{ std::cout << card(2) << " ";
  std::cout << card(11); }
```

Now let's randomly choose two cards $a < b$

```
#include <iostream>
#include <string>
#include <ctime>   // need this to randomise random!!

std::string card(int i)
{
  if (i <= 10) { return std::to_string(i); }
  if (i == 11) { return "Jack"; }
  if (i == 12) { return "Queen"; }
  if (i == 13) { return "King"; }
  if (i == 14) { return "Ace"; }
  return "Not a card";  // all other values for i
}

int main()
{
  int a, b;
  srand(time(NULL));
  a = 0; b = 0;
  while (b <= a + 1)
  {
     a = rand() % 13 + 2;
     b = rand() % 13 + 2;
  }
  std::cout << card(a) << " " << card(b);
}
```

Now let's choose the next card.

```
#include <iostream>
#include <string>
#include <ctime>

std::string card(int i)
{
  if (i <= 10) { return std::to_string(i); }
  if (i == 11) { return "Jack"; }
  if (i == 12) { return "Queen"; }
  if (i == 13) { return "King"; }
  if (i == 14) { return "Ace"; }
  return "Not a card";  // all other values for i
}

int main()
{
  int a, b, c ;
  srand(time(NULL));
  a = 0; b = 0;
  while (b <= a + 1)
  {
     a = rand() % 13 + 2;
     b = rand() % 13 + 2;
  }
  std::cout << card(a) << " " << card(b) << "\n";
  c = rand() % 13 + 2;
  std::cout << "The next card is " << card(c) << "\n";
  if ((a < c) && (c < b))
  {
     std::cout << "You Win!!";
  }
  else
  {
     std::cout << "Sorry, you lose";
  }
}
```

Finally, we add money and while loops so that we can play the game again ...

```
#include <iostream>
#include <ctime>
#include <string>

int money;  // gobal variable to store money

std::string card(int i)
{
  if (i <= 10) { return std::to_string(i); }
  if (i == 11) { return "Jack"; }
  if (i == 12) { return "Queen"; }
  if (i == 13) { return "King"; }
  if (i == 14) { return "Ace"; }
  return "Not a card";  // all other values for i
}
```

Continued on next page....

Continued ...

```
int main()
{
  int bet, a, b, c;
  bet = 0;
  srand(time(NULL));
  std::cout << "Acey Ducey\n";
  money = 100;
  while (money > 0)
  {
        std::cout << "You have " << money << " dollars\n";
        std::cout << "Here are your next two cards \n";
        a = 0; b = 0;
        while (b <= a + 1)
        {
          a = rand() % 13 + 2;   // 2 <= a <= 14
          b = rand() % 13 + 2;   // 2 <= b <= 14
        }
        std::cout << card(a) << "\n";
        std::cout << card(b) << "\n";
        std::cout << "What is your bet (in dollars): ";
        std::cin >> bet;
        if (bet <= 0)
        { std::cout << "Chicken!\n";}
        if (bet > money)
        {
          std::cout << "Sorry, my friend but you have bet too much.\n";
          std::cout << "You have only " << money << " dollars to bet\n";
        }
        if ((bet > 0) && (bet <= money))
        {
          c = rand() % 13 + 2;
          std::cout << "The next card is " << card(c) << "\n";
          if ((a < c) && (c < b))
          {
            for (int i=0;i < 10;i++)
                std::cout << "You Win!!!";
            std::cout << "\n";
            money = money + bet;
          }
          else
          {
             std::cout << "Sorry, you lose.\n";
             money = money - bet;
          }
          std::cout << "\n\n";
        }
  }
  std::cout << "Sorry, my friend but you have no money left.\n";
}
```

Play with your friends. Try to be the first to get past $500.

9.1 Pausing (slowing) the program

Note. We can slow the printing like this:

Example

```
#include <iostream>
#include <string>
#include <chrono>
#include <thread>

void pause(int t)
{
  std::this_thread::sleep_for(std::chrono::milliseconds(t));
}

void charprint(std::string s)
{
  for (int i = 0; i < s.length(); i++)
  { std::cout << s[i]; pause(900); }
}

int main()
{
  cout << "hello ";
  pause(300);
  charprint("SLOW printing");
}
```

Chapter 10

Strings

A string is a sequence of characters.

Example

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

int main()
{
   string mystring = "This is my first string";
   cout << mystring;
}
```

Note

Remember to add `#include <string>` and `using namespace std;` to the start of your program.

We can access the i^{th} `char` of the string `mystring` by writing `mystring[i]`

Example

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

int main()
{
  char c;
  string mystring = "This is my first string";
  c = mystring[0];
  cout << "first char = " << c;
}
```

Unlike Python, we can use the same notation to *change* the value of the i^{th} `char`

Example

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

int main()
{
char c;
string mystring = "This is my first string";
mystring[0] = 'X';
cout << "mystring = " << mystring;
}
```

10.1 Example: Caesar Cipher

A cipher is a set of rules that converts normal text (called *plaintext*) into *ciphertext* (a encoded string of symbols that has no English meaning).

The Caesar Cipher is one of the simplest ciphers. We just shift each letter in the message to a new letter further in the alphabet. For example, starting with CAESAR, we could replace every A with B, and every B with C, etc, to get

$$\text{CAESAR } (\textit{plaintext}) \longrightarrow \text{DBFTBS } (\textit{ciphertext})$$

In C++, we can change each character by using the fact that each character is represented by a number (its ASCII value).

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string plaintext;
    char c;
    plaintext = "CAESAR";
    c = plaintext[0];
    cout << c << " has ASCII value " << (int) c << "\n";
    c = c + 1;
    cout << c << " has ASCII value " << (int) c << "\n";
}
```

We now do this for each character (with a for-loop) and we introduce variable `shift` to tell the program how much to shift each character.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string plaintext, ciphertext;
    int shift, i;
    plaintext = "CAESAR";
    shift = 1;
    ciphertext = plaintext;
    for (i = 0; i < plaintext.length(); i++)
    {
        ciphertext[i] = plaintext[i] + shift;
        cout << plaintext[i] << " shifts to " << ciphertext[i] << "\n";
    }
    cout << "plaintext = " << plaintext << "\n";
    cout << "ciphertext = " << ciphertext << "\n";
}
```

In the above example, `plaintext.length()` is the number of characters in `plaintext`.

To decipher the message, we just subtract `shift` from each character.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string plaintext, ciphertext;
    int shift, i;
    ciphertext = "DBFTBS";
    shift = 1;
    plaintext = ciphertext;
    for (i = 0; i < plaintext.length(); i++)
    {   plaintext[i] = ciphertext[i] - shift;}
    cout << "ciphertext = " << ciphertext << "\n";
    cout << "plaintext = " << plaintext << "\n";
}
```

10.2 Searching within a string

In the previous examples, we added `shift` to a character to obtain a new character for the ciphertext. Unfortunately, the new character could be very strange. We want to limit the characters available to

```
letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 .,?!;$";
```

To do this, we want `A` to correspond to 0, `B` to correspond to 1, and in general letter ℓ to correspond to the location of ℓ in `letters`. We can use the method `mystring.find()` to do this.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring;
    int location;
    mystring = "Hello There!";
    location = mystring.find("e");
    cout << "location of e in mystring is " << location << "\n";

    location = mystring.find("e",2); // start looking at position 2
    cout << "location of next e in mystring is " << location << "\n";
}
```

If e cannot be found in mystring then `mystring.find("e")` returns `string::npos`

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string A, B;
  int i;
  A = "Hello There";
  B = "Th";
  i = A.find(B);
  cout << "Found  " << B << "  at location " << i << "\n";
  i = A.find("zz");
  if (i == string::npos)
   {cout << "Cannot find zz in " << A; }
}
```

We can now use `letters.find()` to convert letters to numbers, and numbers to letters.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string letters;
  int letternum;
  letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 .,?!;$";
  letternum = letters.find("A");
  cout << "A" << " is letter number " << letternum << "\n";
  letternum = letters.find("$");
  cout << "$" << " is letter number " << letternum << "\n";
  // find 10th letter
  cout << "10th letter = " << letters[10];
}
```

Now we can use `letters` to encode our `plaintext`

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string letters, plaintext, ciphertext;
  int i, shift, letternum, newletternum, maxsymbols;
  shift = 10; // shift each char 10 letters to right
  letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 .,?!;$";
  maxsymbols = letters.length();
  // all variables have to be the same type for comparison,
  // otherwise -8 > 1 for unsigned int
  plaintext = "CAESAR";
  ciphertext = plaintext;
  for (i=0; i < plaintext.length(); i++)
  {
  letternum = letters.find(plaintext[i]);
  cout << plaintext[i] << " is letter number " << letternum << " which gets shifted to ";
  newletternum = letternum + shift;
  if (newletternum > maxsymbols)
          { newletternum = newletternum - maxsymbols; } // wrap around to 0 .. 68
  if (newletternum < 0)
          { newletternum = newletternum + maxsymbols;} // wrap around to 0 .. 68
  cout << newletternum << " (" << letters[newletternum] << ") \n";
  ciphertext[i] = letters[newletternum];
  }
  cout << "plain message = " << plaintext << "\n";
  cout << "encryptedmessage  = " << ciphertext << "\n";
}
```

10.3 String size and maximum size

There are two methods we can use to get the number of characters in a string, namely `mystring.size()` and `mystring.length()`. Both give the same result.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring;
    mystring = "hello there";
    cout << "mystring.size() = " << mystring.size() << "\n";
    cout << "mystring.length() = " << mystring.length() << "\n";
}
```

There is a maximum size that a string can have, but that is well beyond the storage size of our computer.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    long long terabyte = (long long) 1024 * 1024 * 1024 * 1024;
    string mystring;
    mystring = "hello there";
    cout << "mystring.size() = " << mystring.size() << "\n";
    cout << "mystring.max_size() = " << mystring.max_size() << " bytes \n";
    cout << "One Terabyte        = " << terabyte << " bytes \n";
```

Our program allocates memory for the string which is at least the size of the string. We can use the method `mystring.capacity()` to see how much memory has been allocated for `mystring`. When we add characters to `mystring` that would make `mystring` bigger than its capacity, the program reallocates memory for `mystring`.

If we are going to add characters to `mystring` many times while our program is running, it is more efficient (faster run-time) to allocate memory at the start with `mystring.reserve(n)` where `n` is the maximum number of characters that `mystring` will have.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring;
    mystring = "hello there";
    cout << "mystring.size() = " << mystring.size() << "\n";
    cout << "mystring.capacity() = " << mystring.capacity() << "\n";
    mystring.reserve(1000);
    cout << "new mystring.capacity() = " << mystring.capacity() << "\n";
}
```

Note

`mystring.reserve(1000)` does not change the size `mystring.size()` of the string.

10.4 String comparisons

There are various methods that we can use to compare strings.

To see if two strings are equal, we can just use `==` in a `if` statement.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1, mystring2;
    mystring1 = "hello";
    mystring2 = "Hello";
    if (mystring1 == mystring2)
    {
        cout << "The strings are equal";
    }
    else
    {
        cout << "The strings are NOT equal";
    }
}
```

Notice that the comparison is **case sensitive** and so `hello` is **not** the same as `Hello`.

We can also use `!=`, `<`, `<=`, `>` and `>=` to compare strings.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1, mystring2;
    mystring1 = "heal";
    mystring2 = "hello";
    if (mystring1 != mystring2)
    {
        cout << "heal != hello \n";
    }
    if (mystring1 < mystring2)
    {
        cout << "heal < hello \n";
    }
}
```

Notice that `<` is the usual "alphabetical" ordering for words.

10.5 Adding (Concatenating) strings

Adding strings together is easy since we can just use +

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1, mystring2, mystring3;
    mystring1 = "Hello ";
    mystring2 = "there";
    mystring3 = mystring1 + mystring2;
    cout << mystring3;
}
```

10.6 Reading individual Characters

We have already seen that `mystring[i]` returns the i^{th} character of `mystring`. There is a method `mystring.at(i)` that does the same thing. The difference is that `mystring.at(i)` is safer to use since it reports an error if `i >= mystring.size()`.

Example

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

int main()
{
    string mystring;
    mystring = "Hello there";
    cout << "mystring[10] = " << mystring[10] << " \n"; // last letter
    cout << "mystring[11] = " << mystring[11] << " \n"; // past last letter, but no error

    cout << "mystring.at(10) = " << mystring.at(10) << " \n"; // last letter
    cout << "mystring.at(11) = " << mystring.at(11) << " \n";
     // past last letter, generates error code
}
```

One more example:

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string A;
   int i;
   char c;
   A = "Hello There";
   for (i=0; i < A.length(); i++)
   {
     c = A.at(i);
     cout << c << ":";
   }
}
```

10.7 Returning part of a string (substrings)

We can use the method `mystring.substr(start, length)` to return a new string that is the substring of `mystring` that begins with `mystring[start]` and ends with `mystring[start+length-1]`.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string A, B, C;
  A = "Hello There";
  B = A.substr(6, 3);
  C = A.substr(6); // all following characters after A[5]
  cout << "3 characters starting at location 6: " << B << "\n";
  cout << "C = " << C;
}
```

10.8 Replacing part of a string

We can use the method `mystring1.replace(start, n, mystring2)` to erase n characters from `mystring1` starting at `mystring1[start]` and to replace those characters with `mystring2`.

Example

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

int main()
{
    string mystring1, mystring2;

    mystring1 = "Hello there you";
    mystring2 = "Trinity College";
    mystring1.replace(6, 5, mystring2);
        // start at mystring1[6]=t and replace next 5 chars with  mystring2
        // remember that mystring1[0]=H
    cout << mystring1;
}
```

Note

In the above example, `mystring1` is changed into the new string (with replacement). That is `mystring1.replace(6, 5, mystring2)` modifies `mystring1`. This is worth remembering because Python does the opposite (`mystring1` is not changed, and we must read the return value of `mystring1.replace()` to get the new string).

10.9 String insertion

We can use the method `mystring1.insert(pos, mystring2)` to insert `mystring2` just before `mystring1[pos]`.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1, mystring2;

    mystring1 = "Hello there you";
    mystring2 = "Trinity College";
    mystring1.insert(6, mystring2);
        // insert mystring2 before mystring1[6]=t
        // remember that mystring1[0]=H
    cout << mystring1;
}
```

10.10 String erase

We can use the method `mystring1.erase(pos,n)` to remove the next n characters starting at `mystring1[pos]`.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1;

    mystring1 = "Hello there you";
    mystring1.erase(6, 5);
        // erase next 5 characters starting at mystring1[6]=t
        // remember that mystring1[0]=H
    cout << mystring1;
}
```

We can use the method `mystring1.erase(pos)` to remove all characters after `mystring1[pos-1]`.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring1;

    mystring1 = "Hello_there you";
    mystring1.erase(6);
        // erase all characters after mystring1[5]=_
        // remember that mystring1[0]=H
    cout << mystring1;
}
```

10.11 Converting Strings to Numbers

The following commands can be used to convert strings into numbers.

```
stoi(A) // convert A to integer
stol(A) // convert A to long integer  (normally the same as integer, 32 bit)
stoll(A) // convert A to long long (64 bit)
stoull(A) // convert A to unsigned long long (64 bit)
stof(A) // convert A to float
stod(A) // convert A to double
```

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string A, B, C, D;
  int a;
  long b;
  float c;
  double d;
  A = "42";
  B = "47585685";
  C = "3.145";
  D = "3.123456789";
  a = stoi(A);
  b = stol(B);
  c = stof(C);
  d = stod(D);
  cout.precision(17);
  cout << a << "\n";
  cout << b << "\n";
  cout << c << "\n";
  cout << d << "\n";
}
```

10.12 Converting Numbers to Strings

We can use the command `to_string()` to convert a number (of any type) to a string.

Example

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

int main()
{
  string A;
  double a;
  a = 124.123456789;
  A = to_string(a);
  cout << "String = " << A;
}
```

Notice that `to_string()` only outputs to 6 decimal places. We can improve on this by using the C function `sprintf()` and arrays of characters; see the following chapter on arrays for an example.

10.13 Reading in strings with getline

When we read user input from the console, the command `cin` will only read one word at a time (with each word separated by spaces).

To read a whole line of text, we use `getline()`

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string mystring;
   cout << "input a line of text:";
   getline(cin, mystring);
   cout << "The line is " << mystring;
}
```

The default operation is to read the string until `\n` (Enter) is found, but we can change this by stopping at any character of our choice.

Example

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string mystring;
    cout << "input a line of text:";
    getline(cin, mystring, ',');   // stop reading mystring when ',' is found
    cout << "The line is " << mystring;
}
// input : first statement, second statement
// output: first statement
```

Notice that the ',' is removed from `cin` before the text is copied into `mystring`.

10.14 Wide Strings

Wide strings are normally used for different languages that require more than 256 symbols (characters). Each element of a wide-string is a wide-character. The size of a wide-character is typically 2 bytes (and so can reference $256 \times 256 = 65536$ symbols). Some C++ compilers use 4 bytes for each wide-character so that all "Unicode" characters are available.

Example

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

int main()
{
char c;
string mystring = "This is my first string";
c = mystring[0];
cout << "The char " << c << " has size " << sizeof(c) << " byte(s) \n";

wchar_t wc;
wstring mywstring = L"This is my first string";    // need L in front for wide-strings
wc = mywstring[0];
cout << "The wide char " << wc << " has size " << sizeof(wc) << " byte(s) \n";
}
```

Note

Console programs will not print more than 256 symbols. If you want to see the new symbols, then use Python:

```
for i in range(32,1000):
    print("chr(",i,")= ",chr(i))
```

For a list of Unicode chars, look at `https://www.unicode.org/charts/`

Chapter 11

Arrays

An array stores a sequence of variables, all accessible by just specifying an integer location (zero based).

Example

```
#include <iostream>

int main()
{
  int mynumbers[4];
  int i;

  mynumbers[0] = 10;
  mynumbers[1] = 20;
  mynumbers[2] = 30;
  mynumbers[3] = 40;
  i = 0;
  while (i < 4)
  {
    std::cout << mynumbers[i] << "\n";
    i++;
  }
}
```

Note that we should not try to read `mynumbers[4]` since we have not allocated space for it. The memory pointed to by `mynumbers[4]` does not belong to our program!!! (C++ does *not* check).

11.1 Default Values

Use curly braces `{}` to initialize the array with default values: `int mynumbers[4]={10,20,30,40}`

Example

```
#include <iostream>


int main()
{
   int mynumbers[4] = { 10,20,30,40 };
   int i;

  i = 0;
  while (i < 4)
  {
     std::cout << mynumbers[i] << "\n";
     i++;
  }
}
```

11.2 Example: Variance of Numbers

Example

```
#include <iostream>

int main()
{
  int n;
  double numbers[] = { 1,2,46,6.7,3.2,-1.2,9 };
  double sum, sumsq, v;

  n = sizeof(numbers)/sizeof(double);
  std::cout << "Variance of "<< n << " numbers\n";
  std::cout << "The numbers are \n";

  sum = 0;
  sumsq = 0;
  for (int i = 0; i < n; i++)
   {
     std::cout << numbers[i] << "\n";
     sum = sum + numbers[i];
     sumsq = sumsq + numbers[i]*numbers[i];
   }

  std::cout << "The variance is " << (n*sumsq-sum*sum)/(n*(n-1));
}
```

11.3 Example: Converting double to string

We can use the C function `sprintf_s()` to convert a real number (double) to an array of chars (and then to a string). This is better than `to_string()` since we can specify the number of decimal places.

Example

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

int main()
{
    double x;

    x = 0.123456789123456789;
    // use C function sprintf() to print
    printf("x = %4.16f", x);  // minimum width = 4 characters
                              // 16 decimal places

    // now use the C function sprintf_s() to convert output to
    // array of chars (C-version of string)

    char arraytext[100]; // C uses arrays of characters instead of strings
    int size;  // number of characters written when converting to string

    size  = sprintf_s(arraytext, "%4.16f", x);
        // convert x to array of chars, 16 decimal places
    cout << "\n";
    cout << "char array = "<< arraytext;

    // now convert char array to a c++ string

    string mystring = "";
    mystring.append(arraytext, size);

    cout << "\n";
    cout << "mystring = " << mystring;
}
```

11.4 2 Dimensional Arrays

A two dimensional array is what we call a *matrix* in Maths.

Example

```
#include <iostream>

int main()
{
  int A[2][3] = { {1,2,3},{4,5,6} }; // [rows][columns]
  int i,j;

  for (i=0; i < 2; i++)
  {
     for (j=0; j < 3; j++)
     {
      std::cout << A[i][j] << " ";}
      std::cout << "\n";
     }
}
```

Chapter 12

Example: The Bisection Method for Solving Equations

Let's try to solve $x^2 + 3x + 1 = 0$

```
#include <iostream>

double f(double x)
{ return  x*x + 3*x + 1; }

int main()
{
  double x;
  x = 1;
  std::cout << "f(" << x << ") = " << f(x) << "\n";
  x = 0;
  std::cout << "f(" << x << ") = " << f(x) << "\n";
  x = -1;
  std::cout << "f(" << x << ") = " << f(x) << "\n";
}
```

Notice that there is a solution between $x = 0$ and $x = -1$

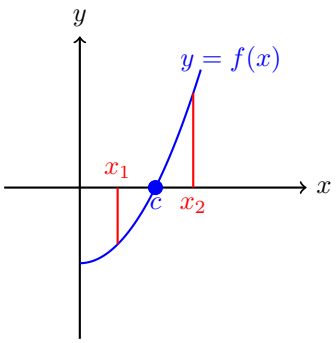
Our next guess is the average $x = \frac{0+(-1)}{2} = -0.5$

First let's make it easier to input the numbers

```
#include <iostream>

double f(double x)
{ return  x * x + 3 * x + 1; }

int main()
{
  double x;
  while (true)
  {
    std::cout << "x = ";
    std::cin >> x;
    std::cout << "f(" << x << ") = " << f(x) << "\n";
  }
}
```

Now let's write a program to find the solution.

If f is a continuous function with $f(x_1) < 0$ and $f(x_2) > 0$ then there is a number c such that $f(c) = 0$.



We first guess the value of c by taking the average of x_1 and x_2, namely $x_a = \frac{x_1+x_2}{2}$.

- if $f(x_a) > 0$ then we repeat the above process with $f(x_1) < 0$ and $f(x_a) > 0$
- if $f(x_a) < 0$ then we repeat the above process with $f(x_a) < 0$ and $f(x_2) > 0$

Let's try to solve $x^2 + 3x + 1 = 0$ again.

First print some values of $f(x)$ to see where it is positive and where it is negative.

```
#include <iostream>

double f(double x)
{ return  x*x + 3*x + 1; }

int main()
{
  int i;
  for (i = -10; i <= 10; i++)
  {
    std::cout << "f(" << i << ") = " << f(i) << "\n";
  }
}
```

We see that $f(-1) < 0$ and $f(0) > 0$

Let $x_1 = -1$ and $x_2 = 0$ and $x_a = \frac{x_1+x_2}{2} = -0.5$. Then x_a is a better approximation to $f(x) = 0$.

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

int main()
{
  double x1, x2, xa;
  x1 = -1;
  x2 = 0;
  xa = (x1 + x2) / 2;
  std::cout << "f(" << xa << ") = " << f(xa) << "\n";
}
```

Since $f(x_a) = -0.25 < 0$ and $f(0) > 0$, we repeat the above with

$$x_1 = x_a = -0.5 \quad \text{and} \quad x_2 = 0$$

Bascially, we put the above in a loop making sure that

```
f(x1) < 0
f(x2) > 0
```

To get started, let's use an infinite loop

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

int main()
{
  double x1, x2, xa;
  x1 = -1;
  x2 = 0;
  xa = (x1 + x2) / 2;
  std::cout.precision(17); // 17 decimal places
  while (true)  // repeat forever
  {
    std::cout << "f(" << xa << ") = " << f(xa) << "\n";
  if (f(xa) > 0)    // always have f(x1) < 0 and f(x2) > 0
     { x1 = x1;
       x2 = xa;}
     if (f(xa) < 0)
     { x1 = xa;
       x2 = x2;}
     xa = (x1 + x2) / 2;
  }
}
```

Now lets tidy up the program, so that it stops!

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

int main()
{
  double x1, x2, xa;
  x1 = -1;
  x2 = 0;
  xa = (x1 + x2) / 2;
  std::cout.precision(17); // 17 decimal places
  while (f(xa) != 0)  // stop when f(xa)==0
  {
    std::cout << "f(" << xa << ") = " << f(xa) << "\n";
    if (f(xa) > 0)   // always have f(x1) < 0 and f(x2) > 0
    {
       x1 = x1;
       x2 = xa;
    }
    if (f(xa) < 0)
    {
       x1 = xa;
       x2 = x2;
    }
    xa = (x1 + x2) / 2;
  }
  std::cout << "f(" << xa << ") = " << f(xa) << "\n";
}
```

We can limit the error in our solution by checking $|x_1 - x_2|$.

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

int main()
{
  double x1, x2, xa;
  x1 = -1;
  x2 = 0;
  xa = (x1 + x2) / 2;
  std::cout.precision(17); // 17 decimal places
  while (std::abs(x1-x2) > 0.00000000001)   // correct to 10 d.p.
  {
     std::cout << "f(" << xa << ") = " << f(xa) << "\n";
     if (f(xa) > 0)   // always have f(x1) < 0 and f(x2) > 0
     {
        x1 = x1;
        x2 = xa;
     }
     if (f(xa) < 0)
     {
        x1 = xa;
        x2 = x2;
     }
     if (f(xa) == 0)
     {
        x1 = xa;
        x2 = xa;
     }
     xa = (x1 + x2) / 2;
  }
  std::cout << "f(" << xa << ") = " << f(xa) << "\n";
}
```

Finally, lets get the program to guess the initial values of x_1 and x_2, trying all integers between -10 and 10. Then we have a good chance of finding all solutions.

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

double solve(double x1, double x2) // need f(x1) < 0 and f(x2) > 0
{
  double xa;
  while (std::abs(x1 - x2) > 0.00000000001)
  {
     xa = (x1 + x2) / 2;
     if (f(xa) > 0)
     {
        x1 = x1;
        x2 = xa;
     }
     if (f(xa) < 0)
     {
        x1 = xa;
        x2 = x2;
     }
     if (f(xa) == 0)
     {
        return xa;
     }
  }
  return xa;
}

int main()
{
  std::cout.precision(15);
  double x1, x2, xa, xs;
  for (x1 = -10; x1 < 10; x1++)
      for (x2 = -10; x2 < 10; x2++)
          if ((f(x1) < 0) && (f(x2) > 0))
          {
           xs = solve(x1, x2);
           std::cout << "f(" << xs << ") = " << f(xs) << "\n";
          }
}
```

Chapter 13

More on Functions

13.1 Function Return Value

We can use `NAN` to report an error in the function's evaluation

Example

```
#include <iostream>

double f(double x)
{
  if (x != 0)
  {
     return 1 / x;
  }
  else
  {
     return NAN;
  }
}


int main()
{
    std::cout << f(0);
}
```

After the program gets to `return` the function stops executing and returns to the main program.

Notice the difference between

(this will print hello)

```
#include <iostream>
double f(double x)
{
  std::cout << "hello \n";
  return x;
}

int main()
{
  std::cout << f(3);
}
```

and

(this will not print hello)

```
#include <iostream>
double f(double x)
{
  return x;
  std::cout << "hello \n";
}

int main()
{
  std::cout << f(3);
}
```

13.2 Function Default Parameters

Sometimes we don't want to enter all the parameters in a function; we can use **default** values in that case.

Example

```
#include <iostream>
void count(int len=10)
{
  int i;
  for (i = 0; i < len; i++)
  std::cout << i << " ";
}

int main()
{
  count();
  std::cout << "\n";
  count(20);
}
```

13.3 Passing an Array into a Function

Example

```
#include <iostream>

double sum(double A[], int len)
{
  int i;
  double total = 0;
  for (i = 0; i < len; i++)
    total = total + A[i];
  return total;
}


int main()
{
  double B[4] = { 0.5,1.1,2.2,3.3 };
  std::cout << sum(B,4);
}
```

Arrays are always passed by *reference* into a function and so we don't need to include `&` to modify the array inside a function.

Example

```
#include <iostream>

void change(int A[])
{
A[0] = 0;
}

int main()
{
int B[4] = { 1,2,3,4 };
std::cout << B[0] << "\n";
change(B);
std::cout << B[0] << "\n";
}
```

13.4 Passing a 2D Array into a Function

We can pass a two dimensional array into a function, but we cannot use the more general notation `void change(int A[][])` and so just use `void change(int A[2][3])`

Example

```
#include <iostream>

void change(int A[2][3])
{
A[0][0] = 0;
}

int main()
{
int B[2][3] = { {1,2,3},{4,5,6} };
std::cout << B[0][0] << "\n";
change(B);
std::cout << B[0][0] << "\n";
}
```

Chapter 14

Example: Prime Numbers

A **prime** number is an integer $n > 1$ whose only (postive integer) factors are 1 and n. If n is a prime number and $n = a \times b$ where a, b are positive integers then $\{a, b\} = \{1, n\}$.

Example

The number 2 is prime. The set of factors of 2 is $\{1, 2\}$. There are no other positive integers a and b such that $2 = a \times b$.

Example

The number 6 is **not** prime. Note that $6 = 2 \times 3$ and so 3 is a factor of 6. Thus 6 has factors that are *different* from 1 and 6.

Let's write a program to test for prime numbers.

First print the possible divisors of n, i.e., start at 2 and end at $n - 1$

```
#include <iostream>

int main()
{
   int i, n;
   n = 10;
   for (i = 2; i < n; i++)
   {
       std::cout << i << "\n";
   }
}
```

Now check if the values of i are divisors of n.

```
int main()
{
   bool prime;
   int i, n;
   n = 10;
   prime = true;
   for (i = 2; i < n; i++)
   {
       if ((n % i) == 0) // remainder is 0 when n is divided by i
        {
          prime = false;
          std::cout << i << " is a divisor of " << n << "\n";
        }
   }
   if (prime == true)
   { std::cout << n << " is prime \n";}
   else
   { std::cout << n << " is NOT prime \n";}
}
```

Now write a function to see if a number is prime.

```
#include <iostream>

bool testprime(int n)
{
  int i;
  for (i = 2; i < n; i++)
  {
     if (n % i == 0)
     {
        return false;
     }
  }
  return true;
}

int main()
{
   std::cout << testprime(10);
}
```

Now let's print all prime numbers up to `limit`

```
#include <iostream>

bool testprime(int n)
{
  int i;
  for (i = 2; i < n; i++)
  {
    if (n % i == 0)
    {
       return false;
    }
  }
  return true;
}


int main()
{
  int i, limit;
  limit = 1000;
  for ( i=2; i <= limit; i++ )
    if (testprime(i) == true)
    {
      std::cout << i << " ";
    }
}
```

Chapter 15

Objects for Storing Data

15.1 vector

A vector is like an array, but its size can grow and it is an object with methods (functions) like `vector.find()`

We need to have `#include <vector>` at the start

Example

```
#include <iostream>
#include <vector>

int main()
{
   std::vector <int> A = { 7,8,9,0 };
   for (int i = 0;i<4;i++)
   {
       std::cout << A[i] << " ";
   }
}
```

Notice that `std::vector` is a member of the standard template library (like `std::cout`)
We can avoid writing `std::` by writing `using namespace std;` after `#include`

Example

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

int main()
{
   vector <int> A = { 7,8,9,0 };
   for (int i = 0;i<4;i++)
   {
      cout << A[i] << " ";
   }
}
```

15.1.1 vector.push_back() and vector.size()

Let's insert two more integers at the end with `vector.push_back()`

Example

```
using namespace std;

int main()
{
  vector <int> A = { 7,8,9,0 };
  A.push_back(1);
  A.push_back(2);
  for (int i = 0; i < A.size(); i++)
  {
      cout << A[i] << " ";
  }
}
```

Notice that `vector.size()` gives the number of elements in the vector.

15.1.2 Initial size

We can also set the initial size (10) and initial values (0) with `vector <int> A(10,0)`

Example

```
using namespace std;

int main()
{
  vector <int> A(10,0);
  A.push_back(1);
  A.push_back(2);
  for (int i = 0; i < A.size(); i++)
  {
      cout << A[i] << " ";
  }
}
```

15.1.3 vector of strings

Vectors can also contain strings.

Example

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
   vector <string> A = {"car", "boat"};
   for (int i = 0; i < A.size(); i++)
   {
       cout << A[i] << " ";
   }
}
```

15.1.4 vector.begin() and vector.end()

`vector.end()` points to the end of the vector (just *past* the last element) and `vector.begin()` points to the first element. To access an element, we need to *dereference* the pointer with `*vector.begin()`

Example

```
using namespace std;

int main()
{
  vector <string> A = {"car", "boat"};
  for (int i = 0; i < A.size(); i++)
  { cout << A[i] << " "; }
  cout << "\n first=" << *A.begin();
  cout << "\n next=" << *(A.begin()+1);
}
```

15.1.5 vector.insert()

We can insert elements at any position in a vector by using `vector.insert()`

Example

```
using namespace std;

int main()
{
  vector <string> A = {"car", "boat"};
  A.insert(A.begin(), "Ace");
  A.insert(A.end(), "Tree");
  A.insert(A.begin()+2, "Middle");
  for (int i = 0; i < A.size(); i++)
   { cout << A[i] << " "; }
}
```

Example

We can even insert elements from another list `B` into our first list `A`.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector <int> A(10, 0);
  vector <int> B = { 1,2,3,4 };
  A.insert(A.begin() + 1, B.begin(), B.end());
  for (int i = 0; i < A.size(); i++)
  { cout << A[i] << " "; }
}
```

Note

Each vector object is stored in one block of memory, that is, all of its elements are located side-by-side in memory. This enables quick retrieval of a specific element (e.g. `A[i]`). If we insert something into the vector, then the vector needs to request another contiguous block of memory from the operating system (which is *not* fast). Don't put `insert()` in a large `for` loop!

If you need to do a lot of inserting, then use `std::list` since a list does not store its data in one block (each element has its own block).

15.1.6 Passing a vector into a function

We can pass vectors into a function as follows.

Example

```
void printv(vector <int> &A)  // function to print vector
{
  for (int i = 0; i < A.size(); i++)
   { cout << A[i] << " "; }
  cout << "\n";
}

int main()
{
  vector <int> A(10, 0);
  vector <int> B = { 1,2,3,4 };
  printv(A);
  printv(B);
}
```

15.1.7 vector.pop_back()

We can remove the last element with `vector.pop_back()`

Example

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector <int> A = { 1,2,3,4 };
  A.pop_back();
  for (int i = 0; i < A.size(); i++)
   {cout << A[i] << " ";}
}
```

15.1.8 find()

We need to add `#include <algorithm>` to use `find()`

We can find an element in a vector with `find()`

If the element is not in the vector, then `find()` returns `vector.end()`

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  vector <int> A = { 1,1,1,0,7,4 };
  auto loc = find(A.begin(), A.end(), 7);
  if (loc != A.end())
  {
    cout << *loc << " found at position " << distance(A.begin(), loc);
  }
  else
  {
    cout << "Not found !";
  }
}
```

15.1.9 count()

We need to add `#include <algorithm>` to use `count()`

This is like `find()` but it also counts the number of matches.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  int total;
  vector <int> A = { 1,1,1,0,7,4 };
  total = count(A.begin(), A.end(), 1);
  cout << "Number of 1s found = " << total;
}
```

15.1.10 vector.clear()

Use `vector.clear()` to remove all of the vector's elements.

Example

```
int main()
{
  int total;
  vector <int> A = { 1,1,1,0,7,4 };
  total = count(A.begin(), A.end(), 1);
  cout << "Number of 1s found = " << total;
  cout << "\n Size  = " << A.size();
  A.clear();
  cout << "\n Size  = " << A.size();
}
```

Note

`vector.clear()` will also call the destructor of all objects in the vector, and so all memory used by the objects in the vector should be deallocated.

Note

Once the vector goes "outside the scope" of a function (that is, after the function finishes) all memory used by the vector should automatically be freed, and so normally, it is not necessary to call `vector.clear()`

15.1.11 copy()

Requires `#include <algorithm>`

We can copy from one vector to another.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  vector <int> A = { 1,1,1,0,7,4 };
  vector <int> B = { 9,9,9,9,9 };
  copy(B.begin(), B.end(), A.begin() + 1);
   // copy B into A at position 1
  for (int i = 0; i < A.size(); i++)
   cout << A[i];
}
```

Note

`copy()` will overwrite the elements of `A`. If want to insert then use `insert()`

15.1.12 vector.erase()

We can erase (delete) elements of a vector by using the erase() function, which is a member of the vector class.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector <int> A = { 1,1,1,0,7,4 };
    A.erase(A.begin()+4); // remove 7
    for (int i = 0; i < A.size(); i++)
        cout << A[i];
}
```

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector <int> A = { 1,1,1,0,7,4 };
    A.erase(A.begin()+3, A.end()-1); // remove 0,7
    for (int i = 0; i < A.size(); i++)
        cout << A[i];
}
```

15.1.13 remove()

Requires `#include <algorithm>`

We can remove elements in a vector with a specific value with the command `remove(vector.begin(), vector.end(), value)`

Note

The `remove()` function does not change the size of the vector, it only moves the other elements to overwrite the ones we do not want anymore. This leaves 'garbage' at the end of the vector (it then returns the location (called an iterator) to the first element of the 'garbage'.
Thus, must also use `vector.erase()` after we call the `remove()` function.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector <int> A = { 1,2,3,1,9, 1,0,7,4 };
    auto newend = remove(A.begin(), A.end(), 1); // remove all 1s
    for (int i = 0; i < A.size(); i++)  // garbage at end
        cout << A[i];
    cout << "\n";
    A.erase(newend, A.end()); // remove garbage
    for (int i = 0; i < A.size(); i++)
        cout << A[i];
}
```

15.1.14 sort()

Requires `#include <algorithm>`

Finally, we can sort the elements in the vector.

Example

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  vector <int> A = { 1,1,1,0,7,4 };
  sort(A.begin(), A.end());
  for (int i = 0; i < A.size(); i++)
    cout << A[i];
}
```

15.2 Example: Deck of Cards

Let's make a deck of cards.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

int main()
{
  vector<string> A = { };
  for (int i=2;i < 11; i++)
     {A.push_back(to_string(i));}
  for (int i = 0; i < A.size(); i++)
    { cout << A[i] << " ";}
}
```

Now add Jack, Queen, King, Ace

```
int main()
{
   vector<string> A = { };
   for (int i = 2; i < 11; i++)
      {A.push_back(to_string(i));}
   A.push_back("Jack");
   A.push_back("Queen");
   A.push_back("King");
   A.push_back("Ace");
   for (int i = 0; i < A.size(); i++)
   {
       cout << A[i] << " ";
   }
}
```

We also need the suits Hearts, Diamonds, Spades and Clubs

```
vector<string> Suits = { "Hearts", "Diamonds", "Spades", "Clubs" };
```

We now can join `A` to Suits to make our `Cards` vector

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

int main()
{
    vector<string> A = { };
    vector<string> Suits = { "Hearts", "Diamonds", "Spades", "Clubs" };
    vector<string> Cards = {};
    for (int i = 2; i < 11; i++)
    { A.push_back(to_string(i)); }
    A.push_back("Jack");
    A.push_back("Queen");
    A.push_back("King");
    A.push_back("Ace");
    for (int i = 0; i < Suits.size(); i++)
        for (int j = 0; j < A.size(); j++)
        {
            Cards.push_back(A[j]+" of "+Suits[i]);
        }
    for (int i = 0; i < Cards.size(); i++)
    { cout << Cards[i] << "\n"; }
    cout << "Number of cards = " << Cards.size();
}
```

Now lets shuffle the cards.

First make `Cards` a global variable by placing its declaration outside `main()` and make a `swap(i,j)` function to swap cards `i` and `j` in the deck.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

vector<string> Cards = {};

void swap(int i, int j)
{
  string temp;
  temp = Cards[i];
  Cards[i] = Cards[j];
  Cards[j] = temp;
}

int main()
{
  vector<string> A = { };
  vector<string> Suits = { "Hearts", "Diamonds", "Spades", "Clubs" };
  for (int i = 2; i < 11; i++)
  { A.push_back(to_string(i)); }
  A.push_back("Jack");
  A.push_back("Queen");
  A.push_back("King");
  A.push_back("Ace");
  for (int i = 0; i < Suits.size(); i++)
      for (int j = 0; j < A.size(); j++)
       { Cards.push_back(A[j] + " of " + Suits[i]); }
  for (int i = 0; i < Cards.size(); i++)
  { cout << Cards[i] << "\n"; }
  cout << "top card = " << Cards[0] << "\n";
  swap(0, 51);
  cout << "top card = " << Cards[0] << "\n";
}
```

To shuffle the cards, we perform 100 random swaps

```
  #include <ctime>

  cout << "Shuffling.......... \n\n\n\n";
  srand(time(NULL));
  for (int i = 0; i < 100; i++)
      { swap(rand() % 52, rand() % 52); }
```

Finally, we can deal 5 cards from the deck by making a `deal()` function.

```
string deal()
{
  string card;
  card = Cards.back();
  Cards.pop_back();
  return card;
}


  cout << "Dealing.......... \n\n\n\n";
  cout << deal() << "\n";
  cout << deal() << "\n";
  cout << deal() << "\n";
  cout << deal() << "\n";
  cout << deal() << "\n";
```

15.3 Lists

Lists are like sequences $a_0, a_1, a_2, \ldots$ where a_i can be any object (even another list). Lists are *not* stored in one contiguous block of memory, and so we cannot use array notation to access the elements. Each element is stored in its own block of memory, together with a pointer pointing to the next element.

Lists are fast when we need to do a lot of inserting.

Most of methods of `list` are the same as `vector` but printing the list is a little harder.

We need to have `#include <list>` at the start

Example

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

void printlist(list<int> &L)
{
  for (auto e = L.begin(); e != L.end(); e++)
  {
      cout << *e;
  }
  cout << "\n";
}

int main()
{
  list<int> L = { 1,1,1,0,7,4 };
  printlist(L);
  L.sort();
  printlist(L);
  L.push_back(9);
  printlist(L);
  auto loc = find(L.begin(), L.end(), 7);
  if (loc != L.end())
  {
     cout << *loc << " found at position " << distance(L.begin(), loc);
  }
  else
  {
     cout << "Not found!";
  }
}
```

15.3.1 list.pop_back()

We can use a list as a stack by using `list.pop_back()` and `list.back()`

Example

```
int main()
{
  list<int> L = { 1,1,1,0,7,4 };
  L.push_back(8);
  cout << L.back();
  L.pop_back();
  cout << L.back();
}
```

15.3.2 list.remove()

`list.remove(myitem)` will search for and then remove all `myitems` from the list. The size of the list will be reduced by the number of times `myitem` appeared.

Example

```
std::list<int> L = { 1,2,3,3,3,4,5,6 };
std::cout << "List size = " <<  L.size() << "\n";
L.remove(3); // remove all 3s
std::cout << "List size after remove = " << L.size() << "\n";
```

15.3.3 iterators

We use a list iterator to step through the list, but we need to `#include <iterator>` first.

Example

```
  std::list<int> L = { 1,2,3,3,3,4,5,6 };
  std::list<int>::iterator e;  // e is a pointer that steps through
                               // the list and is called an iterator
                            // *e is the value located at e
  e = L.begin();
  std::cout << "First element = "<< *e << "\n";
  e++;
  std::cout << "Second element = "<< *e << "\n";
```

Chapter 16

Files

16.1 Writing data to a file

To read and write files, we include `#include <fstream>`

Example

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

using namespace std;

int main()
{
  fstream myfile;
  myfile.open("test.txt", ios_base::out);
  if (myfile.is_open())
  {
    myfile << "This is a test \n";
    myfile << "Second line \n";
    cout << "File test.txt has been written to SSD";
  }
  myfile.close();
}
```

Note

- `ios_base::out` means "write out to file". The file will be created if it does not exist. If the file does exist then it will be overwritten.
- `\n` means "start a new line"
- The file `test.txt` will be located in the same directory as our C++ source file (or very close) when we run the program from Visual Studio. If we just double click on the executable (.exe), then the file will be located in the same directory as the executable.

Hopefully, you can find the file by choosing "File" → "Open" → "File"

16.2 Specifying a path

You can also specify the full path:

Example

```
int main()
{
  fstream myfile;
  myfile.open("c:\\users\\raymo\\desktop\\test.txt", ios_base::out);
  if (myfile.is_open())
  {
     myfile << "This is a test \n";
     myfile << "Second line \n";
     cout << "File test.txt has been written to SSD";
  }
  myfile.close();
}
```

Note

If you want to see what your path looks like in *your* operating system then use **Python**. Load IDLE and type

```
import os
os.getcwd()
```

16.3 Reading data from a file

To read data from the file, we use `ios_base::in` as follows

Example

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;


int main()
{
  fstream myfile;
  string s;
  myfile.open("test.txt", ios_base::in);
  if (myfile.is_open())
  {
     while (myfile.good())
     { getline(myfile, s);
       cout << s << "\n"; }
  }
myfile.close();
}
```

Note that getline only reads one line, and removes the "\n" from each line. If you want to read a whole text file and not add an extra "\n" to the last line then use this code

```
std::string FileName = "savedtext.txt";
std::fstream myfile;
std::string s, stotal = "";
int linecount = 0;
myfile.open(FileName, std::ios_base::in);
if (myfile.is_open())
    {
        while (myfile.good())
        {
            if (linecount > 0) { stotal = stotal + "\n"; }
            getline(myfile, s);  // getline removes \n
            stotal = stotal + s; // do not add \n to last line
            linecount++;
        }
    }
myfile.close();  //  stotal contains the entire text file
```

Chapter 17

Math Library

The `cmath` library provides a set of maths functions including

```
cos()
sin()
tan()
acos()
asin()
atan()
exp()
log()
sqrt()
pow()
round()
```

Example

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    double x = 3;
    cout << "The square root of " << x << " is " << sqrt(3);
```

Note

In Visual Studio, we do not need to write `#include <cmath>` since that library is (apparently) automatically included in all projects.

17.1 Absolute Value abs(x)

$$\text{abs}(x) = |x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Example: $|-1|$

```
#include <iostream>

using namespace std;

int main()
{
   cout << abs(-1) << "\n";
}
```

Normally, `abs()` outputs an integer, but this function has been overloaded many times to include differents types like `float` or `double`.

If you try

```
#include <iostream>

int main()
{
   std::cout << abs(-1.1) << "\n";
}
```

you might get a compiler error because the compiler cannot work out which version of `abs()` to use, or you might get the result `1`

It best to use `std::abs()` or `fabs()` to be safe.
To use `fabs()` you should `#include "math"`

17.2 Powers

Example: 2^{10}

```
#include <iostream>

using namespace std;

int main()
{
  cout << pow(2, 10) << "\n";
}
```

17.2.1 Euler's number e

Example: $e = e^1 = \exp(1)$ and $e^2 = \exp(2)$

```
#include <iostream>

using namespace std;

int main()
{
```

```
  cout << exp(1) << "\n";
  cout << exp(2) << "\n";
}
```

17.3 Logs

$$\log_e x \quad \text{and} \quad \log_{10} x$$

```
#include <iostream>

using namespace std;

int main()
{
  cout.precision(16);
  cout << log(2) << "\n";  // log to base e
  cout << log10(100) << "\n";
}
```

17.4 Trigonometry

$$\sin x \quad \text{and} \quad \cos x \quad \text{and} \quad \tan x$$

All angles are in **radians**

```
#include <iostream>

using namespace std;


int main()
{
  cout.precision(16);
  cout << sin(3) << "\n";
  cout << cos(3) << "\n";
  cout << tan(3) << "\n";
  cout << sin(3)*sin(3)+cos(3)*cos(3) << "\n";
}
```

17.5 Inverse Trigonometry and Pi

$$\sin^{-1} x \quad \text{and} \quad \cos^{-1} x \quad \text{and} \quad \tan^{-1} x$$

All angles are in **radians**

```
#include <iostream>

using namespace std;

int main()
{
  cout.precision(16);
  cout << asin(0.5) << "\n";
  cout << acos(0.5) << "\n";
  cout << "pi=" << 4*atan(1) << "\n";
}
```

17.6 Solving Equations Part 2

We can now improve our program which solves equations.

```
#include <iostream>

double f(double x)
{ return x * x + 3 * x + 1; }

double solve(double x1, double x2) // need f(x1) < 0 and f(x2) > 0
{
  double xa;
  while (std::abs(x1 - x2) > 0.00000000000000001)
  {
     xa = (x1 + x2) / 2;
     if (f(xa) > 0)
     {
        x1 = x1;
        x2 = xa;
     }
     if (f(xa) < 0)
     {
        x1 = xa;
        x2 = x2;
     }
     if (f(xa) == 0)
     {
        return xa;
     }
  }
  return xa;
}

int main()
{
  std::cout.precision(15);
  double x1, x2, xa, xs;
  for (x1 = -10; x1 < 10;  x1=x1+0.5)
      for (x2 = -10; x2 < 10; x2=x2+0.5)
          if ((f(x1) < 0) && (f(x2) > 0))
          {
           xs = solve(x1, x2);
           std::cout << "f(" << xs << ") = " << f(xs) << "\n";
          }
}
```

Now let's use a vector to store the solutions and avoid repeated solutions.

```
#include <iostream>

double f(double x)
{
   return x * x + 3 * x + 1;
}

double solve(double x1, double x2) // need f(x1) < 0 and f(x2) > 0
{
  double xa;
  int count = 0;
  while ((std::abs(x1 - x2) > 0.000000000000001) and (count < 1000))
  {
     xa = (x1 + x2) / 2;
     if (f(xa) > 0)
     {
       x1 = x1;
       x2 = xa;
     }
     if (f(xa) < 0)
     {
       x1 = xa;
       x2 = x2;
     }
     if (f(xa) == 0)
     {
       return xa;
     }
     count++;
  }
  return xa;
}
```

```
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<double> A = {};
  std::cout.precision(15);
  double x1, x2, xa, xs;
  for (x1 = -10; x1 < 10; x1 = x1 + 0.5)
     for (x2 = -10; x2 < 10; x2 = x2 + 0.5)
        if ((f(x1) < 0) && (f(x2) > 0))
        {
         xs = round(solve(x1, x2)*10000000000000)/10000000000000;
         auto loc = find(A.begin(), A.end(), xs);
         if (loc == A.end())
         {
           A.push_back(xs);
         }
        }
```

```
  cout << "The Solutions are \n";
  for (int i = 0; i < A.size(); i++)
     cout << "x = " << A[i] << "\n";
}
```

Exercise. Use the above program to solve the following Maths 1 equations

1. $x^4 + 10x^3 + 35x^2 + 50x + 24 = 0$
2. $|2x + 1| = |2 + x|$
3. $2\cos\left(2x + \frac{\pi}{2}\right) - 1 = 0 \quad$ where $\quad x \in [0, 2\pi]$

Note. This program will not work if the graph has asymptotes, or just touches the $x-$axis without passing through.

Chapter 18

Print Format

C++ supports the C method of printing (and formating) strings.

```
#include <iostream>

int main()
{
  printf("This %s computer has %d cores \n", "windows", 4);
}
```

where

```
%s stands for a string to be inserted
%d stands for a number to be inserted
%f stands for a float  to be inserted
```

We can use extra arguments to make sure that our numbers fit in columns. Notice that the following numbers do not line up:

```
#include <iostream>

int main()
{
  printf("Three numbers: %f %f %f \n" , 1.0, -42.0, 3.0);
  printf("Three numbers: %f %f %f" ,45.0, -42.0, 300.0);
}
```

Now let's print the numbers to 2 decimal places, with a minimum width of 10 characters.

```
#include <iostream>

int main()
{
  printf("Three numbers: %10.2f %10.2f %10.2f \n" , 1.0, -42.0, 3.0);
  printf("Three numbers: %10.2f %10.2f %10.2f" ,45.0, -42.0, 300.0);
}
```

We can use this to print out a matrix with correctly aligned columns.

18.1 Printing a Matrix

Now we can make a function to print a matrix.

```
#include <iostream>
```

```
void printmatrix(double M[2][2])
{
  for (int i = 0; i < 2; i++)
  {
    printf("[");
    for (int j = 0; j < 2; j++)
    printf("%10.2f", M[i][j]);
    printf("]\n");
  }
}

int main()
{
  double M[2][2] = { {-7.03,800},{0.01,-1} };
  printmatrix(M);
}
```

Chapter 19

Breaking Out of a Loop

19.1 break

The `break` command breaks out of a loop. This command is not needed in most programs and is even considered "bad programming" by some programmers, but many people still use it. It replaces some of the `goto` statements used in BASIC programs many years ago.

```
#include <iostream>

int main()
{
  int i;
  for (i = 0; i < 100; i++)
  {
    if (i == 20)
    {
      break;
    }
    std::cout << i << "\n";
  }
}
```

Note that the `break` command only breaks out of the current loop. If it is in a 'double loop' then the break command breaks into the outermost loop.

```
#include <iostream>

int main()
{
  int i;
  while (true)
  {
    for (i = 0; i < 100; i++)
    {
      if (i == 20)
      {
         break;
      }
      std::cout << i << "\n";
    }
    std::cout << "In while loop \n";
  }
}
```

The `break` command will also break out of a `while` loop.

19.2 continue

The command `continue` jumps to the top of the enclosing loop without running the following commands in the loop, and so `continue` also replaces some of the `goto` statements used in BASIC programs many years ago.

The following program will print the numbers 1 to 9 excluding 3.

```
#include <iostream>

int main()
{
   for (int i = 1; i < 10; i++)
   {
     if (i == 3)
     {
       continue;
     }
     std::cout << i;
   }
}
```

Chapter 20

Try and Catch (for exceptions)

We can avoid runtime errors by using `try` and `catch`

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  int i=0;
  string s;
  try
  {
    cout << "input integer: ";
    cin >> s;
    i = stoi(s); // could be a runtime error here
    cout << "your number is: " << i;
  }
  catch (...)
  {
  std::cout << "Integers only!!!";
  }
}
```

Chapter 21

Structures

We can group together related variables to form a structure.

Example.

```
struct Tlocation
{
   int x;
   int y;
};
```

We have created a new type of variable (structure) and we have given it the name `Tlocation`

We can use structures as follows

```
struct Tlocation
{
   int x;
   int y;
};

Tlocation myloc;
myloc.x = 1;
myloc.y = 2;
std::cout << "("  <<  myloc.x <<  "," << myloc.y << ")";
```

The structure `myloc` now contains two fields `x` and `y`

21.1 Arrays of Structures

We can create an array of structures in the usual way:

```
Tlocation myloc[10];
```

and we use structures and arrays as usual

```
struct Tlocation
  {
    int x;
    int y;
  };

Tlocation myloc[10];
 for (int i = 0; i < 10; i++)
   {
     myloc[i].x = i*10;
     myloc[i].y = 20;
   }

 for (int i = 0; i < 10; i++)
   {
     std::cout << "(" << myloc[i].x << "," << myloc[i].y << ")\n";
   }
```

21.2 Functions and Structures

Functions can return structures. The structure is copied from the local variable inside the function to the variable assigned to the returned structure. This can be slow if the structure is very large. We can use pointers to avoid copying (see next the chapter).

```
struct Tlocation
{ int x;
  int y;
};

Tlocation myloc;

Tlocation getNewLoc(int mx, int my)
{
  Tlocation loc;
  loc.x = mx;
  loc.y = my;
  return loc;
}

int main()
{
  myloc = getNewLoc(10,11);
  std::cout << "(" << myloc.x << "," << myloc.y << ")\n";
}
```

We can also pass structures into functions, but note that the structure is copied into a local variable. If we want to change the inputed structure then we can return the new structure at the end of the function (or we can use pointers).

```
struct Tlocation
{
  int x;
  int y;
};

Tlocation myloc;

Tlocation MoveRight(Tlocation loc)
{
  loc.x = loc.x+1;
  return loc;
}


int main()
{
  myloc.x = 1; myloc.y = 1;
  myloc = MoveRight(myloc);
  std::cout << "(" << myloc.x << "," << myloc.y << ")\n";
}
```

Chapter 22

Pointers

A pointer stores a memory location.

We declare a pointer with * as follows

```
<type>  *<name>
```

where <type> can be any type including structures. The compiler needs to know what type of data the pointer is pointing to, so that it knows how to retrieve the data (which is called dereferencing the pointer).

```
int main()
{
  int *pMyint, *pMyint2; //pointers which point at integers
  int x=2;
  pMyint = &x;  //pMyint points to memory location of x
  pMyint2 = &x;
  std::cout << "dereferenced pointer = " << *pMyint << "\n";
  std::cout << "pointer 1 = " << pMyint << "\n";
  std::cout << "pointer 2 = " << pMyint2 << "\n";
  *pMyint2 = 3; // change integer at memory location pMyint2
  std::cout << " x = " << x << "\n";
}
```

22.1 NULL

Pointers start off with random data and so to avoid errors, it is best to initialise pointers with NULL

```
int *pInt = NULL;
```

If we try to use this pointer we will get a runtime error; we need to correctly set it to a valid memory location before using it.

22.2 Pointers and Functions

We can pass pointers into functions. This gives us another way of changing external variables inside a function.

```
#include <iostream>

void inc(int *pMyint)
{
```

```
   *pMyint = *pMyint + 1;
}

int main()
{
   int x=2;
   std::cout << "x = " << x << "\n";
   inc(&x);
   std::cout << "after inc, x =  " << x << "\n";
}
```

Notice that `inc(&x)` passes the memory location of `x` into `inc()` and so we are passing a pointer into `inc()`.

We have already seen another way of doing this (passing parameters by reference). For example

```
#include <iostream>

void inc(double &x)   // note  &x  we are passing the memory location of x
{                     // into the function
  x++;
}

int main()
{
   double y = 1;
   std::cout << y << "\n";
   inc(y);
   std::cout << y << "\n";
}
```

The second way is common because it avoids the use of pointers. Notice that `inc(y)` does **not** require `&` !!!!!!

22.3 Allocating Memory for Pointers

We can get extra memory with the `new` command. The `new` command returns a pointer to the newly allocated memory. This is done while the program is running (and not at compile time). When we are finished with the allocated memory we should free the memory with the `delete` command.

```
#include <iostream>

int main()
{
   int *pmyint=NULL;
   pmyint = new int;
   *pmyint = 3;
   std::cout << "valued stored at pmyint is " << *pmyint << "\n";
   delete pmyint;
   pmyint = NULL;
}
```

Chapter 23

Objects

An **object** is a structure that contains variables and functions. The functions inside the object are called **methods**.

We use a **class** to build a template for an object.

Then we create (many) instances of the object from the class.

23.1 Classes

We can define a class `Quadratic` with three instance variables a, b and c as follows.

```
#include <iostream>

class Quadratic
{
 public:
   Quadratic(int mya, int myb, int myc); // constructor
 private:
   double a, b, c;
};

Quadratic::Quadratic(int mya, int myb, int myc) // constructor
                                                // no return type
{
   a = mya;
   b = myb;
   c = myc;
}
```

The method `Quadratic::Quadratic(int mya, int myb, int myc)` is called the **constructor** of the class `Quadratic`. It sets the initial values of a, b, and c.

Notice that `a, b, c` are declared as **private**. This means that we can only access `a, b, c` inside the class.

23.2 Object Instances

We can make an object instance `q` of the class by just writing `Quadratic q(1,2,3)`

Then the object `q` will be created with variables $a = 1$, $b = 2$ and $c = 3$.

23.3 Methods

We can attach a method (function) `void display()` to the object, as follows:

```
#include <iostream>

class Quadratic
{
 public:
   Quadratic(int mya, int myb, int myc); // constructor
   void display();
 private:
   double a, b, c;
};

Quadratic::Quadratic(int mya, int myb, int myc) // constructor
                                                // no return type
{
   a = mya;
   b = myb;
   c = myc;
}

void Quadratic::display()
{
  std::cout << a << "x^2+" <<  b << "x+" << c;
}

int main()
{
  Quadratic q(1,2,3);
  q.display();
}
```

Objects are automatically destroyed (and allocated memory freed) after the object goes out of scope (that is, once the program gets past the braces `{}` that the object is created in). In particular, if we create an object inside a function, then the object is destroyed when the function ends. If the object contains other objects then they are also automatically destroyed.

Note. If we specially allocate memory (and use a pointer access the memory) then we are responsible to free the allocated memory. We should do this in the **destructor** method of the object.

23.4 Destructors

A object's destructor is called automatically just before the object is destroyed (freed from memory). We use the destructor to tidy up before the object disappears.
The compiler destroys an object when the object goes out of scope (outside `{}`) and when we delete a pointer to the object.
Every object has a default destructor that is automatically created in hidden code when we create an object. So most of the time we don't need to worry about writing a destructor for an object.

The **constructor** method always has the same name as the class.

```
class Quadratic
{
 public:
   Quadratic(int mya, int myb, int myc); // constructor
```

The **destructor** method always starts with ~ and is followed by same name as the class.

```
class Quadratic
{
 public:
   Quadratic(int mya, int myb, int myc); // constructor
   ~Quadratic(); // destructor
```

The destructor has no return value (just like a constructor) and no arguments (input).

Example.

```
int i;
 for (i = 0; i < 10; i++)
   {
     Quadratic q(1, 2, 3);
     q.display();
   }
```

In this example q is created, displayed, and destroyed 10 ten times. The object is destroyed before it is created again. Note that

```
     Quadratic q(1, 2, 3);
     q.display();
     Quadratic q(1, 2, 3);
     q.display();
```

will **not compile** because we cannot create two objects with the same variable name, but

```
    { Quadratic q(1, 2, 3);
     q.display(); }
     Quadratic q(1, 2, 3);
     q.display();
```

will **compile** because q is destroyed after }

Chapter 24

std::cin and std::cout

We can format the width of output of `cout` by using `setw()` but you must include `<iomanip>`

To output in columns of width 7 use

```
#include <iostream>
#include <iomanip>

int main()
{
std::cout << std::setw(7) << 1 << std::setw(7) << 10 << std::setw(7) << 100;
}
```

Notes.

1. The code

   ```
   std::cin.get();
   ```

   will pause the program and wait for the user to press Enter.

2. After entering a value with `std::cin`, there is always a `\n` char in the keyboard buffer, and so `std::cin.get()` will not stop the program anymore. To fix this, we can remove that `\n` with `std::cin.ignore()`.

   For example, we can write

   ```
   std::cin >> y;
   std::cout << "y = " << y << "\n";
   std::cin.ignore();  // remove \n from previous cin
   std::cin.get();
   ```

   This will pause the program and wait for the user to press Enter.

3. If you enter a character (say 'c') when `std::cin` expects an integer, the integer will be set to 0, and **all further** `std::cin`s will be **ignored.**

Chapter 25

Variable Type Conversion

We can convert from char `c` to int `i` by writing `i=(int)c`

```
#include <iostream>

int main()
{
  int i;
  char c;
  c = 'A';
  i = (int)c;
  std::cout << i << " - " << c;
}
```

We can convert from int `i` to char `c` by writing `c=(char)i`

```
#include <iostream>

int main()
{
  int i;
  char c;
  i = 67;
  c = (char)i;
  std::cout << i << " - " << c;
}
```

Chapter 26

Size of Integer

An integer is typically stored in 4 bytes (32 bits). Even thought this allows a maximum of $2^{32} - 1 = 4294967295$, the first bit is normally used for $\pm$, and so the maximum number that an integer can hold is $2^{31} - 1 = 2147483647$. We subtract 1 from 2^{31} because the positive numbers start at 0, but the negative numbers can start at -1 and so the minimum number is $-2^{31} = -2147483648$.

```
#include <iostream>
#include <bitset>
int main()
{
   int i;
   i = 2147483647;
   std::cout << "Size of i is " << sizeof(i) << " bytes \n";
   std::cout << i << " = " << std::bitset<32>(i) << "\n";
   i++;
   std::cout << i << " = " << std::bitset<32>(i) << "\n";
   i++;
   std::cout << i << " = " << std::bitset<32>(i) << "\n";

   std::cout << -1 << " = " << std::bitset<32>(-1) << "\n";
   std::cout << 1 << " = " << std::bitset<32>(1) << "\n";
   std::cout << "Note that  -1 + 1 = 0";
}
```

Note that `std::bitset<32>(i)` outputs `i` in **binary** form and that you need `#include <bitset>` to do this.

Note. The type `unsigned int` will start at 0 and end at a maximum of $2^{32} - 1 = 4294967295$.

26.1 long long type

The type `long int` is normally the same size as `int` (32 bit).

To use a 64 bit integer we use the type `long long`

```
#include <iostream>
#include <bitset>
int main()
{
  long long k = 1;
  for (int j = 1; j < 63; j++)
    {
       k = k * 2;
       std::cout << "2^" << j << " = " << k << "\n";
```

```
    }
    std::cout << k << " = " << std::bitset<64>(k) << "\n";
    k = k * 2;
    std::cout << k << " = " << std::bitset<64>(k) << "\n";
}
```

You can also use other type names for integer:

```
  int16_t                          -32,768 to +32,767
  int32_t (same as int)            -2,147,483,648 to +2,147,483,647
  int64_t (same as long long)    -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
```

26.2 Other types of integer

```
#include <iostream>

int main()
{
   char c;  // characters are 8 bit integers with first bit for +-
   c = 127;  // max c = 2^7-1 = 127, min c = -128
   std::cout << (int)c << "\n";
   c++; // c wraps around to bottom c= -128
   std::cout << (int)c << "\n";

   short si; // short type integers are 16 bit integers with first bit for +-
   si = 32767; // max s = 2^15-1 = 32767, min s = -32768
   std::cout << si << "\n";
   si++; // si wraps around to bottom c = -32768
   std::cout << si << "\n";
}
```

Another small type of integer (in Visual Studio) is `__int8` which only holds values from -128 to 127.

Example

```
__int8 s=0;
for(int i = 1;i < 200; i++)
{
    s++;
    std::cout << (int) s << "\n";
}
```

if we keep on adding 1 to `s` then we get

```
...
126
127
-128
-127
...
```

Notice that $127 + 1 = -128$, and so the number wraps around back to negative numbers.

26.3 Unsigned integers

We can force our integers to be non-negative by using the type `unsigned int`.
If we try to assign a negative number to an `unsigned int` then the number "warps around" back to postive values.

Example

```
unsigned int  t =-1;
std::cout << t ; // t = maximum int value - 1  =  4294967295
```

26.4 Warning: Unsigned integers and inequalities

We need to be very careful when working with unsigned integers and inequalilties since negative numbers are actually very **large** numbers!!!!

Example

```
unsigned int k = 100;
 if (-1 > k) { std::cout << "-1 > k"; } // (-1 > k) is TRUE !!!
  // -1> 100 since -1 = maximum int value- 1;

 // to fix this type of problem, convert all variables to integer;

 int myint = k;

 if (-1 > myint) { std::cout << "-1 > myint"; } // (-1 > myint) is FALSE
  // -1 is NOT bigger than myint since both -1 and myint are integers
```

Note that some functions return unsigned integers, like mystring.length(), and so it is best to convert all variables to integers.

Example

```
std::string mystring = "hello there";
if (-1 > mystring.length()) { std::cout << "-1 > mystring.length()"; }
        // -1 > mystring.length() is TRUE !!!!
       // because mystring.length() is an unsigned integer
// to fix this convert to integer;
int stringsize = mystring.length();
if (-1 > stringsize) { std::cout << "-1 > stringsize"; }
   // (-1 > stringsize) is FALSE, as we would expect.
```

Chapter 27

What does ++i do?

`++i` adds 1 to `i`, but it also returns the new value immediately.

```
#include <iostream>

int main()
{
  int i,j;
  i = 1;
  std::cout << ++i << "\n";   // this will output 2
  j = 1;
  std::cout << j++ << "\n";   // this will output 1 then j=j+1
  std::cout << j << "\n";   // j = 2
}
```

Chapter 28

Parallel Processing (Using Multicore CPUs)

```
#include <iostream>
#include <chrono>
#include <thread>


void sum(int top)
{
  int total = 0;
  for (int j = 0; j < 40; j++)
  {
    for (int i = 0; i < top; i++)
     {
        total = total + i;
     }
  }
  std::cout << total<< "\n";
}

int main()
{
  std::thread t1(sum, 30000000);
  std::thread t2(sum, 30000000);
  std::thread t3(sum, 30000000);
  std::thread t4(sum, 30000000);
  t1.join();
  t2.join();
  t3.join();
  t4.join(); // wait until all threads finished, otherwise abort() error
             // program ends before threads!!!!!
}
```