Hands-On Introduction to C++ Programming

The supporting material for this course is made up of three main components:

- Sparse notes whose main purpose is to establish the order in which topics should be covered and to provide some broad motivation for covering them.
- Detailed examples and explanations demonstrating the language features covered above.
- As our example program grows, we will record its evolution in a Mercurial repository, which will be made available to you at the end of the course.

1 The first steps

1.1 Checking the tool set

Our first task is to ensure that we have a working tool set: that we are able to compile and run C++ programs.

Save this program source code on your local filesystem, in a file whose name ends in `.cc`, `.cpp`, `.C`, `.CPP`, `.cxx`, `.c++` or `.cp`. In what follows, I assume that you have named the file `hello.cc`.

We avoid using sophisticated IDEs to start with. Compile the code with `g++`, by issuing the command

```
g++ -ansi -pedantic -Wall hello.cc
```

Now run (execute) the program with the command

```
./a.out
```

The `-ansi -pedantic` options help you write portable programs.

The `-Wall` option instructs the compiler to warn you about certain features of the code which, while perfectly acceptable by the C++ standard, are *usually* signs of a mistake.

Try using clang (NB: to compile C++ you invoke clang++) to help you understand the compile-time problems you encounter.

1.2 Understanding the basic structure of a C++ program

The *hello world* program exposes us to a number of features of the language. We should understand them before proceeding.

1.2.1 Types

The `int` appearing just before `main` states that the information being returned by `main` has integer type.

1.2.2 `#include` directives.

Note: what is written below is not strictly true. (The truth is that both of the include styles discussed below are *implementation-defined*. True, but not helpful.) What follows, while not strictly true, is more useful from a pragmatic point of view.

The `#include <iostream>` appearing in our program is a *preprocessor directive*. It states that the whole contents of the file `iostream` should be inserted into the source code at the given point.

The angled brackets in the directive indicate that `iostream` is a system file. In order to include files which are not supplied along with the compiler, you should write their names in double quotes. For example

```
#include <systemstuff> // Looks in system directories
```

```
#include "MyStuff.h"    // Looks in user-specified directories
```

The system directories are specified by the compiler; these are directories where the header files for system libraries are kept.

The user-specified directories are, in the gcc toolchain, specified with the `-I` command line flag (`I` for **I** nclude). Other compilers and IDEs may have different mechanisms for specifying the directories to be searched.

For example, to inform the compiler that it should look for headers in the directories `/usr/local/include/readline` and `~/MyProject/include`, you would compile your program like this

```
g++ MyProg.cc -ansi -pedantic -Wall -I /usr/local/include/readline -I ~/MyProject/include
```

In other words, there is one `-I` flag for **each** directory you wish to specify.

[Hint: if you find yourself `#include` ing a `.cc` file, you are doing it wrong!]

The major features of the preprocessor are described here. If you are not already familiar with the preprocessor, you should inspect this file carefully.

1.2.3 `main`

The entry point of any C++ program is the function called `main`. Look here for more information.

1.2.4 Namespaces

The `std::` appearing before `cout` and `endl` demonstrates *namespace qualification*. `cout` and `endl` are components which were declared in `<iostream>`. Being standard features of the language, they reside in the *standard* namespace: `std`.

More information about namespaces can be found here.

1.2.5 Streams

`std::cout` is an *output stream*. Output streams are consumers or sinks of data. `cout` is connected to standard output.

<< is a streaming operator, specifically it is the *insertion operator* (as it inserts data into a stream).

`std::cin` is the input stream tied to *standard input*. Input streams are sources of data, which can be extracted with the *extraction operator* >>.

Note that < < and >> have a completely unrelated meaning in C (arithmetic shift operators). This meaning is also present in C++. How can you distinguish between the two? By the *type* of the operands:

- The arithmetic shift operators must have a number as their left operand and an integer as their right operand,
- Streaming operators must have a stream as their left operand.

Having distinct meanings and/or behaviours for a single operator is known as *operator overloading*. It is a special case of function overloading: having a number of different functions to share a single name.

2 A bigger goal

For the rest of the course, we will explore C++ by writing a very simple simulation of particles bouncing around in a 1-dimensional box. The emphasis is on the discovery and understanding of C++ features. As such, no knowledge of physics (beyond distance = speed × time) is required to write these programs.

2.1 Version Control

We will manage the code we develop with the Mercurial Distributed Version Control System (DVCS).

Mercurial should have been installed for you on the training machines. Create a new repository for your code

```
hg init BounceParticles
```

This will create a new directory called `BounceParticles`, and prepare it to act as a version controlled Mercurial repository.

Save this code in the directory you just created. Ask Mercurial about the status of the directory

```
hg st
```

Mercurial should inform you that it has noticed the presence of a file called `bounce.cc` (or whatever you decided to call your file). It should also show you that it is not tracking the file, by marking it with a question mark.

```
? bounce.cc
```

Tell mercurial to start tracking the file

```
hg add bounce.cc
```

Check the status of the directory once more

```
hg st
```

Mercurial now reports that it has been instructed to add this file to the repository: the `?` has turned into an `A`.

```
A bounce.cc
```

Tell mercurial that you wish to *commit* or *check in* (`ci`) this change of status to the repository

```
hg ci -m "Added simple bouncing particle code"
```

The `-m` option is a *commit message* which will appear in the repository log.

Mercurial is likely to force you to identify yourself at the time you commit. You can do this either through a configuration file, or through the `-u` command line option:

```
hg ci -m "Added simple bouncing particle code" -u "Ada Lovelace"
```

After you successfully commit the change, a further `hg st` should show no output: by default it reports only differences between the state of the repository and the working directory.

2.1.1 TODO Write reference section about Mercurial, VCS and DVCS

2.2 Understanding our task

Compile and run the program

```
g++ -ansi -pedantic -Wall bounce.cc
./a.out
```

Observe that there is a bug: the particle should be bouncing off the left and right boundaries of its box. Instead, when it reaches the right boundary, it reappears at the left boundary.

It is a worthwhile skill, to be able to put yourself in the mind of the computer, to perform a dry run: Pretend that you are the computer and follow the instructions given in the code, in order to understand how, why and when the data change. Try this approach now.

Try to figure out where the mistake is, and fix it.

2.3 Debuggers

Sometimes, the code is too complicated to treat in this way. Sometimes it is better to use a tool to help you understand the flow of the program.

A debugger is a tool which helps you to analyse and understand how a program behaves at runtime by giving you the means to stop and resume execution at arbitrary points in the program, and to inspect the data stored in the program.

Assisted debugging is especially useful in situations where the error is in some code *over there*, but its consequences only become apparent while executing some code *over here*. Unsafe languages such as C++ offer greater scope for such errors: for example, an out-of-bounds array access *over here*, may destroy some data used *over there*. Spotting this without the help of a debugger is extremely difficult.

In our case, the bug is caused by a pretty obvious error in our code and you should be able to spot it without the help of a debugger, but it would still be a worthwhile exercise to get familiar with how debuggers work.

2.3.1 gdb and Emacs

Debuggers are not standardized. For the purposes of this course, I will show you how to use `gdb` via `Emacs`. `gdb` only provides a command line interface, which requires quite some effort to get used to. By talking to `gdb` through `Emacs` you can get a much more user-friendly experience.

In order for gdb to be able to hook in to your running program, we must provide it with some extra information about the program sturcture. This is known as *compiling with debugging symbols*. In the gcc toolchain, you use the `-g` flag to instruct the compiler to generate debugging symbols.

```
g++ -ansi -pedantic -Wall bounce.cc -g
```

Start up Emacs (by typing `emacs &` on the command line), and once you are inside Emacs, issue the command `M-x gdb`. (`M-x` is pronounced "Meta ex". On most modern keyboards this means that you should hold down the `Alt` key while pressing the `x` key. If that does not work, try pressing `Escape` once *before* presing `x`.)

At this point, Emacs (for Emacs versions which predate Emacs 24) should prompt you with the following message

```
Run gdb (like this): gdb --annotate=1 a.out
```

Attention: You **must** change the annotation level from 1 to 3, if you want to have what follows work properly. (The main symptom is `gdb` being copmletely unresponsive.)

If Emacs hasn't correctly guessed the name of the executable you wish to debug (in the above examples `a.out`), then change the name to suit your neeeds, and press RETURN.

If you are using Emacs 24 or newer, you should be prompted with

```
Run gdb (like this): gdb -i=mi a.out
```

Make sure the name of the executable is correct and press RETURN.

Now switch to a richer Emacs-gdb interface with

```
M-x gdb-many-windows
```

Note that the command `gdb-many-windows` **does not exist in Emacs 21 or earlier**.

Set breakpoints by clicking in the left margin of the window displaying the source code. Control execution by typing gdb commands in the command window, or by clicking on the icons in the toolbar.

2.4 Committing changes in Mercurial

Once you have fixed the bug (and removed the comment highlighting the presence of the bug), commit the change to the repository. `hg st` should confirm that the program has been modified, by marking the relevant file with an `m`.

```
M bounce.cc
```

Commit it

```
hg ci -m "Bugfix: particle bounces off right boundary rather than jumping to the left." -u "Alyssa P. Hacker"
```

Get into the habit of recording the changes you make in your projects **frequently**.

2.5 Magic numbers

What is the meaning of the `0` which appears on lines 23 and 24 of our code?

Such numbers which appear in code without any obvious clue to their meaning or role, are referred to as *magic numbers*. They make code

- more difficult to understand,
- more difficult to maintain.

Remove this magic number from our code, by replacing it with a sensibly named variable.

Once you have done this, the code will be

- easier to understand: the meaning of the number is immediately obvious,
- easier to maintain: if we want to change the position of the left boundary, we will only have to change the value in one place.

Check that the code continues to work as expected. Don't forget to commit your change to the repository.

2.6 Abstraction

The clarity of the program could benefit if its components were clearly separated. Our program contains some code which is responsible for displaying the particle on the screen, and some independent code which is responsible for moving the particle through its simulated universe.

Rewrite the program so that the former component is contained in a function called `draw`, and the latter in a function called `move`. Do not give the functions any parameters (yet). Notice that the code fails to compile, and try to understand why.

2.7 Local vs. global variables

The variables used in the program were declared inside the `main` function: they are *local* variables, and cannot be accessed from outside of the function. As standard C++ does not allow you to define functions inside other funcions, our `draw` and `move` functions cannot access the data.

One way around the problem is to make the variables *global*. This is done by moving their declarations out of the main functions. Variables declared outside functions are global; those declared inside functions are local.

Note: This is not a very good solution, we will discuss better ones soon!

Make the variables global.

2.8 Parameters and arguments

Check that the program now compiles and the behaviour has not changed. Don't forget to commit your changes to the repository.

2.9 Parameters and arguments

A better solution to the problem is to let the function receive the data it needs to manipulate, as its input. Inside the function, such inputs are called *parameters*. From the perspective of the caller, they are called *arguments*.

Rewrite your functions so that they accept the required data as inputs. Specifically, change the signature of the `draw` function from

```
void draw()
```

to

```
void draw(double position, char symbol)
```

`position` and `symbol` are parameters of the function.

After this modification, the call site must be changed from

```
draw();
```

to

```
draw(particlePosition, particleSymbol);
```

`particlePosition` and `particleSymbol` are the arguments in this context.

Make the equivalent change to `move` and observe that `move` now fails to work correctly.

The problem is that the function receives a **copy** of its argument. `move` moves the particle by modifying its position, but the value it now modifies is a copy of the value used in the simulation, rather than the value itself, so the modification has no effect outside the function.

Our first solution to this problem will be to instruct the compiler to create a version of `move` which does not receive a copy of the particle's position, but gets to work on the original.

2.10 References

2.10.1 The solution

The signature of the `move` function should now look something like this

```
void move(double position, double speed)
```

change it to

```
void move(double& position, double speed)
```

The only difference is the addition of an ampersand (`&`) between the type and the name of the `position` parameter.

Recompile, run, and check that this has solved part of the problem: before, the particle didn't move; now it does move. Solving the remaining part of the problem is left as an exercise.

Fix the bug: make sure the particle bounces correctly.

Leave `maxColumn` and `minColumn` as global variables for now.

Make all the others local variables of `main` once more.

2.10.2 How references work

The ampersand indicates that the variable being declared should not be allocated any memory of its own. Instead, it must *refer* to some memory which has been allocated previously: it is a *reference*. (With the advent of C++ 11, you should get used to calling this an *lvalue-reference*. Further discussion of this subtlety is **way** beyond the scope of this course.)

In real code, reference variables are almost always function parameters. It is possible to have non-parameter references, but they must be initialized. For example,

```
int  v; // Uninitialized value variable: fine
int& r; // Uninitialized reference: compilation error
```

produces a compilation error, while

```
int  v;
int& r=v; // Initialized reference: fine
```

is ok.

Reference parameters are unavoidably initialized by the arguments passed to the function, therefore it makes no sense to initialize them in them using the syntax shown above.

The two main uses of references are

- allowing a function to modify its input,
- saving time and space by avoiding unnecessary copying of large objects when they are used as function arguments.

2.11 Constness

It is possible to declare variables to be `const`. The purpose of this is

- to indicate to readers of the code that this variable's value cannot and should not be changed,
- to ask the compiler to reject any code which tries to modify the variable's value.

You should get into the habit of adding `const` declarations wherever they are sensible.

Add const declarations to your code wherever they make sense. Check that your code still compiles and behaves as expected. Don't forget to commit your changes.

[Hint: the `const` keyword may be placed before or after the type name.]

2.12 Pointers

References are a feature of C++ which is not present in C. In C, a more general, more powerful, and therefore more difficult to use and more dangerous feature was available: pointers.

Pointers are also available in C++: **if references are sufficent for your purposes, you should prefer references over pointers**.

In order to start getting to grips with poniters, we will look at how we could have used pointers to solve the bug in `move` (where `move` was failing to move the particle).

2.12.1 What are they?

Pointers are data which specify memory locations: they tell us where other data reside. Pointers provide an indirect, and hence more flexible, way of accessing other data. We will discuss this in greater depth, later in the course.

Pointers are numbers which represent memory locations. We refer to these numbers as *addresses*. In order to find the memory location of some datum, you use the *address-of* operator, `&`, as shown below.

2.12.2 Address-of vs. reference to

Note that the same symbol is used to declare reference variables. Take care not to confuse the two uses:

```
int i;
int &x = i; // Make x a new variable equivalent to the existing variable i
p = &x;     // Store the address of x in the variable p
```

In the first case, `&` is an unary prefix operator; in the second case it is part of a type declaration. (The whitespace is not significant. You could remove any of the whitespace in the above two lines, or even add whitespace between the `&` and the `x`, without changing the meaning of the code.)

2.12.3 Pointer dereference

If you have a pointer, you can use it to access the datum it is pointing to by using the *pointer dereference operator*, `*`. The same symbol is also used in the syntax for specifying pointer types. Just like in the case of `&`, you must be careful not to confuse the two meanings.

```
int a;
int* pi; // pi is a variable of type "pointer to integer"
a = *pi; // copy the integer at address pi into the variable a
*pi = a; // copy the integer in variable a into the memory location pi
```

In other words, if `pi` is a pointer to `i` then

- both `*pi` and `i` evaluate to the contents of i,
- both `pi` and `&i` evaluate to the address of i.

Draw a memory map representing the state imposed by the following four lines. (You will have to play the role of the compiler in deciding where to store each variable.)

```
int a = 10;
int* b = &a;
int& c = a;
int*& d = b;
```

Now work out the values of the following expressions.

```
&a; a; *a;
&b; b; *b;
&c; c; *c;
&d; d; *d;
```

Write them in a table like this

	&		*
a			
b			
c			
d			

Some of these expressions will result in an error.

Remember, references were good enough to fix our bug in `move`, so in real life we would use references. For didactic purposes, we will see how the problem could have been solved with pointers.

Rewrite your `move` function to use a pointer parameter, rather than a reference. Note, you will have to change the call site too! Once it works, commit the change.

2.13 Shared mutable state

Global variables are very obvious and wide-reaching form of *shared mutable state*. The presence of shared mutable state makes it difficult to understand the behaviour of programs and to write correct programs. One of the main reasons for this is that it opens up the possibility of someone changing some state on which you depend, when you don't expect it.

Modifiable parameters are a less wide-reaching form of mutable shared state.

The out-of-bounds array access mentioned above, is an example of unintended shared mutable state.

While shared mutable state often looks very convenient at the time you write programs, you often end up paying heavily for it later in the project.

2.13.1 Functional programming

Functional programming is a style of programming which eschews shared mutable state. While C++ doesn't lend itself particularly well to such a style of programming (and the object-oriented style practised in C++ is largely at odds with the functional style), you can still benefit from making portions of your C++ programs functional. Put another way: it is often worth while to avoid functions which modify their inupts.

We could rewrite the program in a functional style: The functions should not modify their inputs; the new values should be returned as results of the functions. However, `move` would then need to return *two* results: this is not something which can be done very conveniently in C++03. (Returning an arbitrary number of values from a function is easier in C++11, with the advent of the *tuple* type.)

Nonetheless, it would be worthwhile to get a feeling for how a functional approach would look.

Rewrite `move` so that it returns the particle's new position, rather than mutating it. (It will still have to mutate the speed.)

Before you start, remove the pointers from our program: references are perfectly up to the task, so they should be preferred in this case.

Don't forget to commit your change, once done.

Notice that it is now clear at the `move` **call site** that `particlePosition` is being changed; but in order to see that `particleSpeed` is being changed, we would have to look at the **definition** of `move`.

2.14 Screen buffer

We want to enhance our program to be able to simulate more than one particle at a time. Look at our `draw` function and observe that the algorithm we are using will not allow us to draw more than one particle in a single time slice, correctly.

To get around this problem, we will use a screen buffer: some memory which will store the data we wish to display on the screen. Once all the required data are stored in the buffer, the buffer's contents can be printed on the physical screen.

2.15 Arrays

We will use an *array* to implement the screen buffer. An array is a contiguous region of memory used to store a number of elements of the same type.

The syntax for declaring arrays is

```
int a[10];
```

This instructs the compiler to reserve a chunk of memory big enough to store 10 integers, and to allow us to refer to that memory using the name `a`.

There is a related syntax for accessing individual elements in this array:

```
a[0];       // Read the first element in the array
a[1] = 6;   // Overwrite the second element of the array with 6
```

Note that array indices are zero-based. The first element in the array is element number 0, the second element is element number 1, and so on. The last element of an array of size `N` is element number `N-1`.

2.16 Fundamental and user-defined types

There are two distinct kinds of types in C++

- fundamental types
- user-defined types

2.16.1 Fundamental types

The fundamental types are very basic types provided by the language. The fundamental types we have met so far are `char`, `int`, `double`, pointers and arrays. (Note that the pointers are always pointers *to some other type*, and that the arrays are always arrays *of some other type*.)

2.16.2 User-defined types

The user-defined types are made by grouping together other types along with some operations (functions which act on those data), using a mechanism called *classes*. These will be discussed in great depth later in the course.

The name *user-defined* may be a little misleading, as some user-defined types are provided by the language itself. For example, `std::cout` has a user-defined type, even if it is provided by the language standard.

2.16.3 Instantiation is initialization

An important difference between fundamental types and user-defined types is that the former are not ititialized at declaration time, unless the code explicitly dictates it. The user-defined types are initialized as soon as they are declared.

One consequence of this is that the array declaration shown above will result in the array containing arbitrary data. At the beginning of a simple program, it is quite likely to contain zeros, but do not be fooled into believing that you can rely on this. In general, the array will contain whatever ones and zeros happened to be lying around in those memory locations: garbage.

2.17 Implementing the screen buffer

Create a *global* screen buffer.

```
char screen[80];
```

Rewrite the `draw` function, so that it writes information into the screen buffer. Add some code which prints the contents of the buffer onto the physical screen, at the end of each time step.

Don't forget to clear the screen at the beginning of each time step.

2.18 Casting

If you did not change the signature of `draw`, you are likely to have bumped into a problem such as

```
g++ -ansi -pedantic -Wall bounce.cc -g
bounce.cc: In function 'void draw(double, char)':
bounce.cc:8: error: invalid types 'char [80][const double]' for array subscript
```

This problem would arise from a line such as

```
screen[position] = symbol;
```

The declared type of `position` is `double`, but `position` is being used as an array substript or index. It makes no sense ask for element 6.3 of an array. In general it makes no sense to use a non-integer as an array subscript: this type (`double`) does not make sense in this context.

2.18.1 Explicit casts

One way around the problem is to tell the compiler explicitly that you want a type conversion. As is so often the case in C++, we have a legacy C-ish way of doing it, and a modern C++-ish approach. Let's look at the former first. In C (and therefore C++) you can ask the compiler to convert some value into an integer like this: `(int)value`. So, in our context

```
screen[(int)position] = symbol;
```

This is known as *casting*.

A C++-style cast looks like this: `static_cast<int>(value)`. In our context

```
screen[static_cast<int>(position)] = symbol;
```

Check that both of these styles solve this problem.

Note that this cast results in loss of data: we started off with a number like 6.3, and we are left with just 6. In this case, we don't mind that loss. The compiler made us aware of the loss of data by requiring an explicit cast. But that doesn't always happen, sometimes the compiler silently accepts such loss of information.

2.18.2 Implicit casts

Remove the cast, recompile and check that the compiler rejects the code. Change the type of `draw`'s `position` parameter from `double` to `int`. Recompile.

Observe that we now have the same data loss as before, but the compiler lets it pass silently. (Depending on the version of your compiler, you may actually observe different behaviour.)

2.19 Encapsulate the screen manipulations

If you have screen clearing and screen drawing code floating around `main`, abstract or *encapsulate* this functionality by tucking its details away inside a couple of functions. I suggest names like `clear_screen` and `display_screen`.

Note that I am using the word *encapsulate* in its broad sense. The C++ and Java Object-Oriented Programming traditions have hijacked this word and use it in a far more restrictive sense. Be aware that this word may mean different things to different people and keep your mind open. Don't get too hung up on one particular definition.

Before you did the last exercise, the details of how the screen is implemented, and how other code interacts with it, were spread all over your code. Changing some of these details would involve tracking them down all over your code: in big projects, this could prove very expensive. The client code may, in many places, rely on the details in ways which would break if those details were to change.

If you hide these details inside functions, the screen's client code ceases to depend on these details directly: it just needs to know that you interact with the screen by using the functions `clear_screen` and `display_screen`. Changing these details becomes cheap: as long as you don't change the interface (the signatures of the functions) the client code should continue to work.

This is one of the most important concepts in software development: promote loose coupling between different components of your program by ensuring that they do not depend on eachother's implementation details.

2.20 Second particle

It's time to make our model a bit more interesting, by introducing a second particle:

Change the program so that there are *two* particles bouncing around simultaneously, but not interacting with eachother in any way.

Don't try to do anything remotely clever yet, do it in the naivest way possible: by adding three new variables representing the new particle's position, speed and symbol, and using the draw and move functions with those new variables.

2.21 Arbitrary number of particles

Adding more particles in this way would be trivial, but tedious.

Add a third particle, but, this time, rather than adding another set of 3 variables, use arrays to store the positions, speeds and symbols of all the particles.

Write the calls to `draw` and `move` only once: use a loop to ensure that these calls are made as many times as necessary.

Did you use a magic number to represent the number of particles? If so, in how many places would you need to change this number, each time you add an extra particle? In how many places would you need to change it if you did *not* have a magic number?

2.22 Array initializers

You may like to use the following array initialization syntax

```
int a[5] = {3,8,7,2,1};
```

Investigate what happens when the number of data you specify does not match the length of the array.

Notice that, when using array initializers, you may omit the array size.

2.23 Array sizes must be known at compile time

Obsevre that the array size must be a compile time constant. This means that it must either be a literal integer, or an integral variable which has been declared `const` (or a macro, but we try to avoid macros in C++).

Ensure that your code contains no magic numbers referring to the screen size.

2.24 Always compile with warnings enabled

If you are using g++ and you are *not* compiling with the `-Wall` flag, you will not notice any problem with dynamically allocated arrays. gcc has a non-standard extension, which allows you to specify arras sizes dynamically (i.e. at run time). With the warnings disabled, the compiler happily accepts such non-standard code, without giving you any hints about the fact that your code is likely to be rejected by other compilers.

Would you rather discover this as soon as you introduce such non-standard code into your project, or later, after you have written many lines of code which depend on it, and you are trying to port your project to a different environment?

Always compile with all warnings enabled.

2.25 Loosen the coupling to the screen

At the moment we have one, global, hard-wired screen buffer in our program.

Make this more flexible by making it possible to choose among any number of screen buffers, through a parameter of the functions which interact with the screen buffer.

Looser coupling makes programs more flexible, reusable, extensible, maintainable (imagine a very long list of warm and fuzzy joy inspiring adjectives at this point).

When you wish to declare a parameter of type array-of-something, you may leave out the array size, just like you can when you use array initializers.

Now that there is no more need for having a global screen buffer, make it local.

2.26 No bounds checking

I have deliberately been trying to lead us into a trap. Whether you have fallen into the trap by now, and the exact symptoms of falling into the trap, may depend on many factors including apparently insignificant choices you made in your code (such as the order of declarations of variables) and the version of the compiler that you are running.

Therefore it is impossible, in these notes, to give any specific details of how the problem might manifest itself in your case. This is something that must be looked at in the context of a given program being compiled on on a given compiler. If you have not fallen into the trap so far, please attract my attention so that we can demonstrate some appropriately nasty behaviour in your code.

The big picture is this: I deliberately hinted that the screen size should be 80, or `maxColumn - minColumn`. This is subtly wrong. Our program allows the particle to be drawn at positions ranging from 0 to 80 **inclusive**. That's 81 different positions, not 80! When the particle's position is 80, that 80 is used as an array index whose meaning is "the array's 81st element": it is placed beyond the end of the screen.

What lies beyond the end of the screen? The language standard doesn't tell us; in practice, it is usually one of the variables what was declared somewhere near the array.

Notice that neither the C++ compiler nor the run-time offer any hints about this error. Welcome to one of the joys of C and C++ programming: out of bounds array access.

Investigate with a debugger. Watchpoints could be very useful here.

Remove this bug from your program.

2.27 Arrays are (not) pointers

(Language lawyers will tell you that arrays are **not** pointers. However, in practice, for novice (and not-quite-so-novice) C and C++ programmers, treating them as if they were the same is **far** more useful trying to understand the ugly truth: the similarities are significant and obvious; the differences are subtle and rarely important. At the beginning, treat arrays and pointers as equivalent, but do bear in mind that there are some subtle differences which we are ignoring at this juncture.)

Arrays are little more than syntax sugar on top of constant pointers.

- The name of an array is a constant pointer to the beginning of the array
- `a[b]` ≡ `*(a+b)`

2.27.1 Pointer arithmetic

To fully understand the second point, we must understand *pointer arithmetic*. Consider the following code

```
double* pd = 0;
pd + 1; // Not 1
```

The value of the expression on the second line depends on the amount of memory your compiler uses to represent a single *double*.

In general, when adding an integer `i` to a pointer `p`, the result will be `p + i` × `sizeof(target type)`. This means that adding 1 to a pointer, will result in a pointer which points to the end of the object pointed to by the original pointer (yes, you'd better read that *slowly* a few times; ask me in person if it's still confusing), even if this means that several memory locations must be traversed. If a number of similar objects are stored in contiguous memory, pointer arithmetic allows us to navigate between them easily. The array is the same idea dressed up in different clothing.

Convince yourself that any occurence of something of the form `a[i]` in your code, can indeed be replaced by `*(a+i)`, and vice-versa.

Similarly, convince yourself that any function parameters of the form `sometype var[]` can be replaced with `sometype* var`.

2.27.2 Array declarations reserve memory

Non-parameter array declarations can **not** be replaced with pointer declarations. Declaring non-parameter arrays results in the compiler reserving new memory for the contents of the array. In the case of parameter arrays, the compiler merely reserves memory for the pointer to the beginning of the array: the contents are not copied. In some sense, arrays are always passed by reference, even though you don't ask for it.

Leave some examples of pointer arithmetic and some examples of array syntax in your code. Similarly, declare the `screen` parameter as an array in some functions, as a pointer in others. There is no good reason to do this in production code, but we'll leave all styles in our code for a while, so that we keep being reminded of the equivalences. We will revent to a sensible, uniform style later on.

2.28 Dynamic memory allocation

All the memory allocation in our prorgam so far, has been static: it has been done at compile time. (This is not *quite* true because of stack frames).

If we wanted to read in some configuration of particles, including their number, at runtime, our current program would not be able to deal with it, because the size of the arrays we use to represent the particles must be known at compile time.

One popular way of dealing with this problem is to pick some arbitrarily large number, and hope that the program will never be asked to store more than that number of particles. Sooner or later, this will have disastrous results. Don't do that.

A better approach, is to allocate exactly the amount of memory we need, dynamically.

This is yet another thing that could be done in the old-fashioned C way (`malloc` and `free`), or in the modern C++ way. In this course we will ignore the old-fashioned way.

2.28.1 `new`

At this stage, you should be aware of two distinct, modern ways of dynamically allocating memory:

- allocate memory to store just one object
- allocate memory to store lots of objects

```
int* one  = new int;      // Allocate memory for a single integer
int* many = new int[10]; // Allocate memory for 10 intgerers
```

In both cases, the `new` operator returns a pointer to the start of the newly allocated memory.

It is our responsibility to deallocate the memory (make it available for reuse) when we no longer need it. In nontrivial programs, deallocating memory before it is too late, but not too soon, can place considerable burden on the programmer.

2.28.2 `delete`

There are two different ways to deallocate dynamically allocated memory corresponding to the two styles shown above:

```
delete one;
delete [] many;
```

If you use the wrong one, something nasty is likely to happen in your program sooner or later. If you are lucky, the program will crash immediately; if you are unlucky, it may silently destroy some of your data and continue running.

2.28.3 In practice

Let's put this into practice by dynamically allocating our screen buffer. (We will leave the dynamic determination of the required array size, until later: for now we'll demonstrate the dynamism by not having to use a compile time constant to specify the size.)

Make the screen buffer dynamically allocated. Don't forget to deallocate the memory once we're done with it.

2.29 Structs

There are three particles in our model, each of which has three data associated with it: a position, a speed and a symbol. Nine data altogether.

Currently the positions are grouped together, as are the speeds and the symbols. It might be more convenient to group these data together by particle.

We can achieve this with *structs*. Structs are a means of creating user-defined types.

Our first look at structs will not take us beyond what was available in C.

```
struct Particle {
  char symbol;
  double position;
  double speed;
};
```

This creates a new type, `Particle`. Being a type, its name can be used to declare variables:

```
Particle p;
```

Instances of this type have 3 *member* data: `symbol`, `position` and `speed`, which can be accessed using the following syntax.

```
p.symbol;        // Read the symbol member of p
p.position = 12 // Set the position member of p
```

Rewrite your program in terms of the `Particle` struct shown above. For didactic purposes, your first version of `move` should mutate the particle, rather than returning a modified copy.

2.30 Revising pointers and references

In order to enable `move` to mutate the particle it receives, you had a choice of parameter type

- `Particle& p`
- `Particle* const p`

As references are up to the task, you should prefer them over pointers, in this case. However, for didactic purposes:

modify your `move` and `draw` functions to accept pointers rather than references to particles.

When pointers and constness meet for the first time, things get a little confusing. You can read about it in detail here.

Add as many `const`-qualifications to your pointer paramenters as possible. This means that the `Particle*` parameters of `move` and `draw` will have different `const`-qualification.

2.31 `a->b ≡ (*a).b`

Your code should now be peppered with expressions of the form `(*p).x`, and you should have been annoyed by how tedious it is to write.

C (and hence C++) provides a shorthand for accessing a member of a structure via a pointer to the structure. The compiler will generate exactly the same code for both of the following lines.

```
(*pointer_to_struct).member_name;
pointer_to_struct->member_name;
```

Rewrite your code to take advantage of this syntax.

2.32 Struct initialization legacy syntax

While its use in C++ is rare (though C++11 has given it new life through `std::initializer_list`), you should be aware that C++ inherits C's syntax for initializing structs;

```
Particle p = {'x', 0, 6.3};
```

The intention is that C++'s structs should be able to do exactly what C's structs did, while also being able to do more.

2.33 Intializing particles

The process of initializing our particles is quite tedious at the moment. In my sample code, it looks something like this:

```
particles[0].symbol = 'x';
particles[0].position = 0;
particles[0].speed = 6.3;
```

Write a function called `initialize`, to make this process more convenient. It should be used like this

```
initialize(&particles[0], 'x', 0, 6.3);
```

2.34 Beyond C structs

All the features of structs that we have met so far, were already present in C. We are about to use a C++ feature which is not present in C.

Our `Particle` struct has 3 member data (`symbol`, `position`, `speed`). In C++, structs may also have member functions.

Just like member data, member functions are invoked through some instance of the type of which they are a member, either using a `->` (for pointers) or a `.` (for non-pointers).

```
SomeType  v;       // value
SomeType& r = v;   // reference
SomeType* p = &v;  // pointer

v.member_datum;
v.member_function();
r.member_datum;
r.member_function();
```

```
p->member_datum;
p->member_function();
```

2.35 `this` is an implicit parameter

All member functions (unless declared `static`), have a hidden parameter: it is not visible in the surface syntax of the language, but the compiler inserts it into the code it generates.

- It is always called `this`,
- you cannot *see* it in the parameter list, but you can refer to it in the function's body,
- its type is always *const pointer to the struct (or class) of which the function is a member*.

The `move`, `draw` and `initialize` functions are obvious candidates for `Particle` member functions.

Make `move`, `draw` and `initialize` member functions of the `Particle` struct.

2.36 `this` is implicit in lookup

You may omit `this` when accessing members from within the body of a member function. When the compiler finds a mention of an unqualified name in the body of a member function, and that name matches the name of some member of the struct (or class), it will act as if there were a `this->` just before the name.

Remove explicit use of `this` from your code.

At first glance, the last two sections suggest that you could simply ignore the existence of `this` in C++ programming. If you do so, sooner or later you will fail to understand subtle but important features of your programs. It is important to understand `this` thoroughly.

2.36.1 Experienced programmers insist on `this->`

There are two commonly used (alternative) coding conventions relating to the optional implicitness of `this` when referring to members.

- Whenever you may use `this->` you **must** use `this->`.
- **Every** member of **every** user-defined type **must** have a name starting with `m_`.

When you see an unqualified name appearing in the body of a member function, you cannot tell whether that name refers to a member or to something in some other visible scope. Many experienced programmers have concluded that this is more trouble than it is worth, and insist on one of the above coding conventions in their projects.

2.37 constness of `this`

Before you moved `draw` into the `Particle` struct, it should have had a parameter with type `Particle const * const`. When turning `draw` into a member function, this parameter is replaced by `this`. Where do we get to write `const` now that the parameter has disappeared from the source code? (Remember, it is still there in the generated code!)

`this` is always `const`: it keeps a record of the particle through which the function was called, which cannot change during a single execution of the function. That takes care of the rightmost `const` above: we don't have to write it because the compiler *always* puts it in.

How about the leftmost `const`? It informs the reader (and the compiler) that `draw` will not change the particle at the end of the pointer. Contrast this with `move` where this second const is absent: `move` **may** change the particle, by changing its position and speed.

If you want to `const`-qualify the target of `this`, you do so by writing `const` *after* the closing parenthesis of the parameter list of the function.

```
struct T {
  void member_fn_with_const_this_target() const {}
};
```

2.38 `const` member functions

The last section describes in low level detail exactly what is going on. On a higher level, you can think of member functions with a `const` after the parameter list, as functions which do not modify the instance through which they were accessed.

Make `draw` a `const` member function. See what happens when you make `move` const too.

2.39 `mutable`

You are allowed to declare member data to be mutable: `const` member functions are allowed to modify member data which have been declared `mutable`.

2.40 TODO Why bother with constness?

- clarity
- safety
- more code compiles

2.41 Abstract types and formal types

An *abstract data type* (a broad concept in computer science, closely related to the concept of *abstract class* in C++) is a data type defined by the operations that can be performed on it: its *interface*.

Our particle was an abstract data type defined by the operations `initialize`, `draw` and `move`. This abstract data type was implemented in our program through the three functions with those names, used in conjunction with the `Particle` struct. There was no knowledge at the C++ level tying those four components together: the programmer had to know, from some external source, that they go hand in hand.

By promoting the three functions into the struct, we made the relationship formal, and clearly visible at the C++ level.

The *interface* of the type—the ways we can interact with it—is still an extremely importent concept, and by grouping all the components together, we make the language formally aware of it.

2.42 Create a screen type

Notice that our screen buffer is also an abstract data type.

Turn the screen buffer into a formal C++ type:

- Create a struct called `Screen`.
- Give it a `char* screen` member.
- Give it two member functions `display` and `clear`.
- Give it an `initialize` function (exactly the same name as the `initialize` belonging to `Particle`).

Now that we have two different functions called `initalize`, how does C++ know which one should be called?

2.43 More encapsulation

Does your code directly accesses the `char*` member of `Screen`, outside the bodies of the member functions of `Screen`?

If it does, then `Screen`'s client code is dependent on an implementation detail of `Screen`. You could, in principle, change the way you store the data in the buffer: Maybe your screens are mostly empty, and you conclude that it would be more efficient to use sparse arrays. (This is the subject of an extra exercise). In order to take advantage of this observation, you would have to track down all uses of this buffer in your code, and change them accordingly.

If you provide the screen buffer as a utility in a library, and contemplate changing some implementation detail, then you would have to check all the code written by all the clients of the library, if you didn't want to risk breaking the client code. This is an intractable problem.

A better apporach would be to make it clear to clients of `Screen`, that the `char*` member is not part of the interface and that they should therefore not use it directly. You will need to provide enough functionality in the interface to enable your clients to do anything reasonable with the screen.

This idea of separating code you write into details which should be ignored by clients, and an interface which should be used by clients, is the essence of *encapsulation*. (In C++ circles, however, the word tends to be used in a narrower sense.)

You retain the freedom to change anything that is not part of the interface, but you commit yourself to leaving the interface as it is (though you may *add* more features to it).

2.44 Privacy

C++ provides a mechanism for indicating whether any member of a user-defined type is part of the interface or not, and for preventing the use of non-interface members by clients: the keywords `private` and `public`. (There is a third keyword in this family: `protected`, but it is too early to discuss it at this point.)

They are used as follows

```
struct MyType {
public:
  // part of the interface
private:
  // implementation details
public:
  // more bits of interface
private:
  // more details
};
```

though, typically, you would only use each of these keywords once per class:

```
struct MyType {
public:
  // all the public bits go here
private:
  // all the private bits go here
};
```

Make the screen's `char*` member `private`. See how the compiler reacts to attempts to access it directly from outside of the bodies of `Screen`'s member functions.

Rewrite the code so that `Screen`'s clients only use its interface. You will need to extend the interface. I suggest the signature

```
void put(int position, char symbol);
```

In C++ circles, the meaning of the word *encapsulate* is strongly tied to the use of the `private` keyword.

The rule of thumb, which has been elevated to the level of dogma, is that **all data members must be private**.

2.44.1 All member data must be private

While the language imposes no such requirement, C++ programmers religiously follow the rule that all data members of a class must be private. Why?

In brief: in C++ you can make member data look like member functions (by writing getters and setters: trivial functions which read or modify the member datum), but not the other way around. So the safest thing to do is only to allow functions in the interface: if a public function turns into a datum, you can still pretend that it's a function (with getters and setters); if an public datum turns into a function you can't pretend that it's a datum, so you will have to break the interface.

Looking at it in more depth, there may be many different ways to solve a particular problem. Which one you choose may depend on many factors. New factors may come into play with time. Sometimes you may discover a better solution to a problem in the future. The key point to understand is that software developers frequently change their mind about the implementation details of their components.

As a toy example demonstrating this idea consider how we might implement a class representing rectangles. Here are 4 alternative implementations.

```
struct Rectangle {
public:
  double width;
  double height;
  double area() { return width * height; }
};

struct Rectangle {
public:
  double width;
  double height() { return area / width; }
  double area;
};

struct Rectangle {
public:
  double width() { return area / height; }
  double height;
  double area;
};

struct Rectangle {
public:
  double width;
  double height;
  double area;
};
```

Note that each of these implementations, though functionally equivalent to the others, has a different interface. As soon as clients start using the implementation we have chosen initially, we lose the freedom to change our mind.

Here are the same four ideas packaged up behind *identical* interfaces:

```
struct Rectangle {
public:
  double width()  { return width_; }
  double height() { return height_ ;}
  double area()   { return width_ * height_; }
private:
  double width_, height_;
};

struct Rectangle {
public:
  double width()  { return width_; }
  double height() { return area_ / width_; }
  double area()   { return area_; }
private:
  double width_, area_;
};

struct Rectangle {
public:
  double width()  { return area_ / height_; }
  double height() { return height_; }
  double area()   { return area_; }
private:
  double height_, area_;
};

struct Rectangle {
public:
  double width()  { return width_; }
  double height() { return width_; }
  double area()   { return area_ ; }
private:
  double width_, height_, area_;
};
```

Whatever choice we make initially, we retain the freedom to switch to an alternative without breaking clients' code.

The problem with the first set of 4 examples is that the internal organization of the class is explicitly visible in its interface. The solution relies on hiding this internal organization behind a uniform interface.

There is no performance penalty associated with this trick: the compiler will optimize away trivial accessor functions and generate code that accesses the members directly. Effectively, there *are* public member data, but, to the clients, they look like member functions, which means that the data can be replaced by member functions in the future without breaking the clients' code.

While this may seem pointless in the context of such a trivial example, the idea becomes really important in real world scenarios. Consider our `Screen`. To represent a screen of size 10000 with a `*` in position 1675 and a `x` in position 7629, our original implementation would require 10000 bytes of storage: an array containing 9998 spaces, one `*` and one `x`. A much more efficient implementation choice exists: for each particle to be drawn on the screen, store one `char` representing its symbol, and one `int` representing its position, dynamically allocating an appropriate amount of memory depending on the number of items to be drawn. For large, sparsely populated screens, this could lead to significant memory savings. By preventing the client from accessing our internal data structures directly, we maintain the freedom to switch to better implementations without breaking clients' code.

2.44.2 Bounds checking in `Screen::put`

Notice that going through a higher-level interface, rather than accessing `Screen::screen` directly, offers a further advantage. Previously the client was burdened with the responsibility of making sure not to write beyond the boundaries of the buffer, and the author of the class had no means of protecting against out-of bounds writes. Our `put` functions gives us the opportunity of dealing with this problem. We will deal with this later.

2.45 Classes

You are not obliged to use `private`, `public` or `protected` in your structs. In structs, anything appearing before the first occurrence of one of these keywords is public by default.

In addition to `struct` C++ also gives us the possibilty to create user defined types with the `class` keyword. User-defined types created with `class` are **identical** to those crated with `struct`; the **only** difference is that the default accessibility of class members is `private`.

In practice, most C++ programmers use the `struct` keyword when they are writing something which only uses the features provided by C's structs; the `class` keyword is usually used to implement code which uses the extra features provided by C++.

Change your structs to classes. Convince yourself that the default accessibility of structs is public, while that of classes is private.

It is a widespread practice to put the superfluous `private` keyword at the top of class definitions.

We will use classes rather than structs from now on.

2.46 Constructors

Recall that fundamental types are not initialized automatically.

User-defined types, in contrast, are initialized as soon as they are created. C++ allows the authors of user-defined types to tailor the process to their needs, by writing *constructors*.

Constructors are functions which are run as soon as an instance of a user-defined type is created, in order to initialize it. Constructors are just like member functions except that

1. Their names **must** match the name of the type to which they belong.
2. They have no return type (not even `void`): they work by mutating the newly created object.
3. They are called automatically when objects are created, rather than by explicit invocation.
4. They may have an initializer list (explained later).

Turn `Screen::initialize` into a constructor.

2.46.1 Ctors

In the literature (though rarely in speech) you may find constructors referred to by the abbreviation *ctor*

2.46.2 Default constructor

For user-defined types, a constructor **must** be invoked (automatically, the user has no control over this) **as soon as** the object is created.

Specifically, a constructor must run in situations such as this

```
UserDefinedType u;
```

As there is no information provided about *how* the object should be constructed, some *default* construction process must take place. The constructor that runs in such a situation receives no information, therefore in has no parameters (other than it's hidden prameter, `this`: it needs that, to know which object in is to initialize.) Constructors with no (explicit) arguments are called *default constructors*.

2.46.3 Compiler generated default constructor

Repeating what was said in the last section: For user-defined types, a constructor **must** be invoked automatically **as soon as** the object is created.

Specifically, a constructor must run in situations such as this

```
Screen screen;
```

This implies that a constructor was called on our `Screen` instance, even before we wrote the constructor. How is that possible?

If a user-defined type contains no constructors, the compiler will generate a (trivial) default constructor for it.

2.46.4 Non-default constructors

It is possible to write constructors that do have parameters.

Write a constructor for `Screen` with a parameter for specifying the screen size.

In order to use such a non-default constructor, you will need to provide some extra information in the declaration. Here is the syntax:

```
Screen screen(screenWidth);
```

Addtionally, in the case of one-parameter constructors this syntax is also available:

```
Screen screen = screenWidth;
```

The compiler should generate exactly the same code in both cases.

It is possible to have more than one constructor in a single user-defined type, because of function overloading.

2.47 Destructors

Before making `Screen::screen` private, you should have had something equivalent to `delete [] screen;` in your code, and making `screen` private should have caused the compiler to complain about it. A simple solution would have been simply to remove the `delete`: Your program runs fine without it. But now it contains a memory leak: the memory you reserved is never freed, even after you stop using it. (The operating system will reclaim it all once your program stops running, but it will be unavailable to you as long as your program runs. This doesn't matter in out toy program, but can cause serious problems in real world scenarios.)

We have already seen that authors of user-defined types can specify what happens when their types are instantiated, by writing constructors. A similar mechanism exists for specifying what should happen when instances of user-defined types are destroyed: *destructors*.

Destructors are similar to constructors:

1. Their names **must** match the name of the type to which they belong, with a prepended tilde (~).
2. They have no return type (not even `void`): they work purely by side-effect.
3. They are called automatically when objects are deallocated, rather than by direct invocation.
4. They may **not** have an initializer list.

Write a destructor for `Screen` which frees the memory which was dynamically allocated in the constructor.

[At this point it is very likely that your code will crash at runtime: It is likely that your code contains two bugs which which have been cancelling eachother out so far. We will address this shortly when we discuss copy constructors.]

A destructor **must** be called (automatically, the programmer has no control over this) whenever an instance of a user-defined type is destroyed. Just like in the case of the default constructor, the compiler generates a (trivial) destructor for any user-defined type which does not have one.

2.47.1 Dtors

In the literature (though rarely in speech) you may find destructors referred to by the abbreviation *dtor*

2.48 Automatic memory management

C++ does have some limited automatic memory management.

While the programmer is responsible for deciding when to free dynamically allocated memory, and for ensuring that it is done in the appropriate way, *statically* allocated memory is automatically released when its associated variable goes out of scope.

2.48.1 RAII

The guarantee that destructors of static local objects will be called on scope exit, has been put to work to create robust techniques of resource management.

The idea is to create user-defined types whose purpose is to manage resources: constructors to acquire them, destructors to release them. It goes by the name *Resource Acquisition Is Initialization* (*RAII*).

The importance of RAII hinges on the fact that, in C++, the only code that is *guaranteed* to run after an exception is thrown, are the destructors of statically allocated objects going out of scope.

2.49 Pass by value

You should have a `Particle::draw(Screen)` function. Investigate how the program's behaviour depends on whether you accept the `Screen` parameter by reference or by value?

`Particle::draw` modifies the screen. If this modification is to have any lasting effect, it must take place on the original screen, rather than a copy of it. This suggests that we need to accept the screen by reference. The program seems to work even if the screen is passed by value.

2.49.1 Copy constructor

What happens when an object is passed by value to a function? Consider

```
void put(Screen s) { ... }
```

`s` is a local variable inside `put`. As it is *not* a reference, a new instance of `Screen` must be created when the function is entered, in order to populate this variable: this implies that a constructor must be called. This new instance will be created by copying the instance which is being passed as an argument to the function.

Such a constructor goes by the name *copy constructor*.

1. Signature of the copy constructor

   The copy constructor **must** accept an object of its own type **by reference**.

   If it accepted the object by value, the copy constructor would have to be called. But this is the copy constructor of the type in question, so it would have to call itself in, order to be able to receive its own argument, resulting in an infinite loop.

   Copying an object should not modify the original, therefore copy constructors *should* accept the original by **const** reference.

2. Compiler generated copy constructor

Given that we have been passing user-defined type instances by value without having written a single copy constructor, you should be able to deduce that the copy constructor is yet another member that might be automatically provided by the compiler.

The following trivial class seems to contain no constructors at all

```
1: struct Foo {
2:
3: };
4:
5: int main() {
6:   Foo f(1);
7: }
```

but the compiler reports the presence of both a default constructor and a copy constructor on line 1:

```
g++ -ansi -pedantic -Wall automatic_members.cc
automatic_members.cc: In function 'int main()':
automatic_members.cc:6: error: no matching function for call to 'Foo::Foo(int)'
automatic_members.cc:1: note: candidates are: Foo::Foo()
automatic_members.cc:1: note:                 Foo::Foo(const Foo&)
```

These are the compiler-generated versions.

Write a noisy copy constructor (one that prints a message whenever it runs) for `Screen`, and confirm that it is called when screens are passed by value, but not when they are passed by reference.

But why does the program work correctly when our screen-mutating `draw` function receives a *copy* of the screen it is supposed to be mutating?

The compiler-generated copy constructor copies all of the object's member data. This means that the copy of the screen contains a copy of the internal `char*` member. The value of the pointer is exactly the same as that inside the original: both the original screen and its copy share the same chunk of memory to represent the screen of the state. The compiler-generated copy constructor copied the outer shell of the object, but not the internals.

So, our code contains two serious bugs, which cancelled eachother out in this limited case!

Write a copy constructor for `Screen` which does **not** result in the sharing of buffers between the original and the copy. Confirm that `put` now needs to accept the screen by reference (and that your runtime crash (if you had one) disappears).

2.50 Initializer lists

[Note: we are talking about *mem-initializer-lists* rather than C++11 `std::initializer_list` and related uses of the term.]

Repeating, once again, what was said before: For user-defined types, a constructor **must** be invoked **as soon as** the object is created.

This is also true for user-defined members of user-defined types. Specifically, this means that constructors of such members will run **before** the body of the constructor of their enclosing type.

If these member constructors are to run before the body of the enclosing constructor, how can we pass any information to the member constructors?

C++ provides a syntax for this purpose, called the *initializer list*. Here is an example

```
class MyClass {

private:
  int i;
  HisClass h;  // A class with a 2-arg constructor
  YourClass y; // A class with a 1-arg constructor

public:
  MyClass(int a, int b, int c, int d)
    : i(a), h(b,c), y(d) {
    // Body of the constructor
  }
};
```

The initializer list (if present) appears between the parameter list and the body of the constructor. It consists of a colon (`:`) followed by the names of (a non-trivial subset of) the members, with each member name followed by the (parenthesized) constructor argument list, separated from each other by commas.

Rewrite `Screen`'s constructors so that *as much work as possible* is done in the initializer list. [One of the constructors will now have an empty body.]

Because of the initializer list, many constructors end up having empty bodies. (You still need the braces!)

2.50.1 TODO The order of the initalizer list is irrelevant

Find a way of generating a spectacular bug, to get the point across.

2.51 Constructor puzzle

```
struct Inner {
   Inner(int){}
};

struct Outer {
   Inner i;
};

int main() {
  Outer o;
}
```

1. Explain the compilation error generated by the above code.
2. Get the code to compile by changing the definition of `Inner` only (3 different ways).
3. Get the code to compile by changing the definition of `Outer` only (2 different ways).

2.52 Encapsulating `Particle`

Make `Particle`'s data members private. Turn `Particle::initialize` into a constructor with an initializer list. (Your program will fail to compile at the end of this exercise: we'll fix it in the next.)

`Particle` now has a 3-arg constructor, which has suppressed the compiler-generated default constructor. When you try to create an array of `Particle`

```
Particle particles[3];
```

that array is immediately populated with instances of `Particle`. As there is no way of saying *how* each element of the array should be initalized, the array is going to be populated with default-constructed ones. But there is no `Particle` default constructor. So we get a compilation error.

[Note: C++11 *does* provide a way of initializing elements of arrays at construction time.]

2.53 Temporaries

You can create *temporary* instances like this

```
 1: class Circle {
 2:   double x, y, radius;
 3: public:
 4:   Circle(double X, double Y, double R);
 5:   bool intersects(const Circle& other);
 6:   }
 7: };
 8:
 9: // ...
10:
11: Circle(3,4,2).intersects(Circle(1,2,3));
```

On line 11, two *temporary* instances of `Circle` are created. They are temporary because they are created

- neither as part of a variable declaration,
- nor by dynamic allocation through `new`.

They will be destroyed as soon as their values are used.

Write a default constructor for `Particle` (in addition to the one it already has). This will allow the creation of an array of (default constructed) particles. Use temporaries to overwrite the elements of the array.

Note that our solution is wasteful: First we fill the whole array with nonsense, then we discard the nonsense. It would be better to fill the container with the values we want, in the first place. We cannot do this with arrays (before C++11), but we will look at other approaches later in the course.

2.54 TODO Operators

Introduce operators. Write Screen::operator [] as synonym for Screen:put.

2.55 TODO Assignment operator

Unless a better context, containing a genuine need for writing an assginment operator, arises later, do this:

Earlier we arranged for the dummy elements of our array of particles to be overwritten by the by temporaries containing our real data.

This was done by the assignment operator, yet another thing that the compiler might write for us.

Write a noisy assignment operator for `Particle`, and confirm where it is being called.

In this case, the compiler-generated version is good enough for our needs. But it's not good enough for `Screen`.

Write an assignment operator for `Screen`.

2.56 Compiler generated member functions

[Note: C++11 gives you more direct control over when the compiler generates these functions.]

You have to be aware of, and thoroughly understand, the four class members that we have met so far which might be written automatically by the compiler:

1. Default constructor
2. Destructor
3. Copy constructor
4. Assignment operator

The compiler-generated default constructor and destructor are trivial: their bodies contain no code. Their sole purpose is to exist so that they may be called in situations that require it. For example

```
{
  MyType t; // The default constructor is called at this point.
} // Leaving scope: the destructor is called at this point.
```

A default constructor and a destructor for `MyType` must exist if this code is to compile, even if they don't do anything when called.

The compiler-generated copy constructor and assignment operator perform copies/assignments of the member data. This may be enough in the case of very simple classes, but it doesn't take much complication before more sophisticated implementations become necessary. You will almost certainly need to write a custom copy constructor and assignment operator as soon as the object contains dynamically allocated memory.

2.57 The rule of three

The copy constructor, assignment operator and the destructor are coupled. In most cases, when the compiler-generated version of *any* of the three is not good enough for your class, the other two have to be hand-written too. As soon as you write one of them, you have to write all three of them.

2.58 TODO What should assignment operators return?

This is possibly beyond the scope of the course, but someone is bound to ask.

The compiler doesn't care, but users expect the following to work

```
a = b = c = d;
```

To make it work you would need to return the value which has been assigned. By value? By reference?

2.59 TODO Take care with self-assignment

This should probably be an extra exercise.

Consider the following

```
Screen  a;
Screen& r=a;
a = r; // Self-assignment
```

Would the copy-and-swap idiom be too much for this course?

2.60 TODO Check bounds in `Screen::put`

Discuss the 3 options

- Fail silently
- Return status code
- Exceptions

  No time for exceptions. Cover them in the reference.

2.61 Separate compilation

2.61.1 Out of class definitions

So far, we have been writing the definitions of member functions inside the class. It is possible merely to declare the members inside the class, providing the definitions outside. For example, the code

```
class SomeClass {
  int i;
public:
  SomeClass(int j) : i(j) {}
  int twice() const { return 2*i; }
};
```

Is equivalent to

```
class SomeClass {
  int i;
public:
  SomeClass(int);
  int twice() const;
};

SomeClass::SomeClass(int j) : i(j) {}

int SomeClass::twice() const { return 2*i; }
```

Move all the member function definitions out of `Screen` and `Particle`.

Note that the public parts of the class block now contain the interface only. All implementation details are either in the private parts, or outside of the class block.

2.61.2 Separate files

We have been writing all our code in a single file. This works well for our small program whose main purpose is didactic: we need to see the code at once.

In real projects, the code is usually spread out among many separate files. Each class normally occupies two files:

- A *header* containing the class block containing only declarations of the members. (File extensions: `.h` or `.hh`, or no extension, in the case of standard headers.)
- An *implementation* file, containing the definitions of the member functions. (File extensions `.cc`, `.cpp`, etc.)

Compiling C++ progams is a very costly task. If a large project's source code were contained in a single file, even the smallest change would require a recompilation of the whole lot. By breaking it up into smaller components, we make it possible to recompile only those which have changed.

Break the code up into 5 separate files:

1. `Particle.hh`
2. `Particle.cc`
3. `Screen.hh`
4. `Screen.cc`
5. `main.cc`

Hints below.

Initially, instruct the compiler to compile all the implementation files and link them together, in a single instruction:

```
g++ -ansi -pedantic -Wall Particle.cc Screen.cc main.cc
```

We will learn how to compile them separately once we have correctly split the program.

Any files which refer to names which they do not declare themselves, must `#include` a header which provides those declarations.

- Implementation files will `#include` their corresponding header files
- Clients `#include` their providers' headers.

As an example of the latter, consider `void Particle::draw(Screen&)`. `Screen` appears in the signature of this function, therefore the compiler must have seen a declaration of `Screen` before it sees `draw`'s declaration. Therefore, `Screen.hh` must be `#include`d in `Particle.hh`.

2.61.3 Header guards

In the previous exercise, you should have stumbled over circular imports: File A includes file B which includes file A which includes file B which includes file A which includes file B which includes file A which includes file B which includes file A which …

The technique used to protect against this infinite recursion, goes by the name of *header guard*.

```
// Inside file MyClass.hh

#ifndef MyClass_hh
#define MyClass_hh

#include "This.hh"
#include "That.hh"

class MyClass {
  // ...
};
#endif
```

1. The first time the compiler includes `MyClass.hh`, the macro `MyClass_hh` is **not** defined.
2. It therefore enters the `#ifndef` block,
3. defines `MyClass_hh`, and
4. processes the contents of the file.
5. On subsequent inclusions of `MyClass.hh`, the macro `MyClass_hh` **is** defined,
6. therefore the `#ifndef MyClass_hh` causes the contents of the file to be ignored.

Protect your headers with header guards.

2.61.4 Class forward declarations

Consider this code.

```
class A {
  void foo(B);
};

class B {
  void bar(A);
};
```

The compiler must have already seen a declaration of `B` at the time it parses `A`, but it must already have seen a declaration of `A` at the time it parses `B`.

We have a chicken and egg problem, which can be resolved by providing a *forward declaration* of the second class before the first. It looks like this

```
class B;
```

2.61.5 Translation units and linkage

We have some global variables lying around our code (`minColumn`, `maxColumn` and `screenWidth`). In production code we would almost certainly replace them with a better mechanism, but here they serve the purpose of briefly attracting our attention to a couple of details. For completeness, translation units, linkage and the ODR are briefly explained in the reference section.

1. TODO Sweep it under the rug

   For now, we sweep it all under the rug by sticking these global variables into `Particle.hh`.

2.61.6 Separate compilation

The full compilation process consists of 3 stages

1. Preprocessing
2. Compilation
3. Linking

Just like the `-E` flag instructs the compiler to stop after the preprocessing stage but before compilation, the `-c` flag instructs it to stop after compilation but before linking.

Linking is the process of combining incomplete compiled programs into a single, executable, whole.

Each translation unit can be compiled on its own, but it does not contain enough information to form a complete executable, therefore the linking stage would fail, so it must be suppressed.

```
g++ -ansi -pedantic -Wall Particle.cc -c
g++ -ansi -pedantic -Wall Screen.cc -c
g++ -ansi -pedantic -Wall main.cc -c
```

You could specify the names of the generated output files with `-o`. By default the output file has the same name as the source file, with the extension changed to `.o`

If you feed object files to the compiler, it will figure out for itself that only the linking stage is left, or you could invoke just the linker explicitly: it is called `ld`. (Remember that all the details of command names and options described here, are specific to the gcc toolset: other compilers will be invoked in different ways, but all of them will have a preprocessor, a compiler and a linker.)

Link the object files like this

```
g++ Particle.o Screen.o main.o
```

2.61.7 Makefiles

In large projects, you can save significant compilation time by only compiling those portions of the project which have changed since the last compilation. Makefiles and the `make` program, are a means automating this. IDEs have such mechanisms built in (sometimes based on makefiles).

Here are some makefiles you may use and adapt for your projects.

- Makefile: To use, follow the instructions in the comments in the file.
- Makefile\$_{minimal}$: minimalistic, shows essential components without any explanation.
- Makefile\$_{explicit}$: more sophisticated, with copious comments explaining how it works.

Adapt one of the above makefiles for use with your code and add it to your repository.

2.62 Streams

Our particle data are currently hard-wired into our code. This is very inflexible: we have to recompile every time we want to change our dataset. It would be better to read the particle configuration in from a file.

2.62.1 Output streams

To write an `int` and a `double` to standard output, you might write

```
#include <iostream>
//...
int i=1;
double d=2.3;
//...
std::cout << i << " " << d << std::endl;
```

<< is called the *stream insertion operator*

2.62.2 Input streams

To read an `int` and a `double` from standard input, you might write

```
#include <iostream>
//...
int i;
double d;
//...
std::cin >> i >> d;
```

To read an `int` and a `double` from a file, you might write

```
#include <iostream>
#include <fstream>
//...
int i;
double d;
//...
std::ifstream in("MyInputFile");
if (!in)
  std::cerr << "Could not open file." << std::endl;
else
  in >> i >> d;
```

>> is called the *stream extraction operator*

Note that invalid streams are false in boolean contexts.

2.62.3 Configuration file

Write a configuration file of the form

```
3
x 0 6.3
+ 20 3.1
o 60 -2.1
```

where the first line states how many particles are to be read in and each subsequentree specifies the symbol, position and speed of a particle you want to include in your simulation.

Populate the array of particles with the data contained in this file.

2.62.4 Streaming operators

This should probably be an extra exercise

Write an extraction operator for `Particle`, and read the configuration directly into the array, without going through an intermediate.

2.62.5 TODO Friends

Discuss how friends extend the interface, and why they might unavoidable in the case of streaming operators.

2.63 Dynamic array

Now that we use a configuration file, we no longer have to recompile the program every time we want to change the configuration of particles in our simulation.

But changing the number of particles in the configuration, would still require recompiling, because the size of the array we use to store the particles is a compile time constant.

We can get around this problem, by using `new` to dynamically allocate the memory that we need. Rather than doing this in an ad-hoc fashion, it would be better to encapsulate it, and provide it as a packaged utility.

Write a class which will act as an array of particles, whose size can be specified at run time, via an argument to the constructor.

Make sure that the elements stored in the class can be read and written using the usual `array[index]` syntax.

Specify the number of particles to be read from the configuration file, at its beginning and use it to set the array size at run time.

2.64 Templates

Our dynamic array class is quite useful, but it has some serious shortcomings.

Notice that arrays are *polymorphic*: an array can hold elements of any type you choose; our class can only store `Particle` s. If we wanted to store `int` s, we would have to write a separate class; if we wanted to store `Banana` s, that would be another class. What's more, all of these classes would have essentially the same implementation: the only difference would be in the name of the contained type, peppered throughout the implementation.

C++ provides a means of writing this code only once: *templates*.

Here is a simple class, and an example of its usage.

```
class Store {
  int thing;
public:
  Store(int t) : thing(t) {}
  int gimme() { return thing; }
  void replace(int t) { thing = t; }
};

Store s(12);
s.gimme();
s.replace(24);
```

And here is a template from which an equivalent class can be *instantiated*.

```
template<typename T> //
class Store {
  T thing;
public:
  Store(T t) : thing(t) {}
  T gimme() { return thing; }
  void replace(T t) { thing = t; }
};

Store<int> s(12);
s.gimme();
s.replace(24);
```

In this context `typename` may be replaced with `class`. (In all other contexts, `class` and `typename` are not interchangeable.)

Notice how the template is instantiated on line 10. In the first example, `Store` is a class; in the second example, `Store` is a template, while `Store<int>` and `Store<Particle>` are classes instantiated from that template.

The syntax for defining member function templates outside of the class block is

```
template<class T>
T Store<T>::gimme() { return thing; }
```

Turn your dynamic particle array class into a template and check that it now works for arbitrary element types.

You can't get it to work? That's probably because you are not including the whole implementation in the header.

2.65 STL containers

You may wish to make further improvements to the dynamic array as as an extra exercise, but we will now switch to using the standard-provided facilities.

The C++ standard library provides a number of robust and dynamic container types. (For historical reasons, you may often hear this portion of the standard library referred to as the *STL*: *Standard Template Library*.) The most commonly used is almost certainly `std::vector`:

```
#include <vector>
// ...

std::vector<int> v;
v.push_back(1);
v.push_back(4);
v.push_back(9);
v.push_back(15);

v[0];
v[3] = 16;
```

You should not use C arrays in C++, except for legacy reasons: use `std::vector` instead (or one of the other C++ standard container types, if it is better suited to your needs). In extreme cases, you may benefit from writing your own.

Use `std::vector` to store the particles in your simulation.

2.66 Inheritance

Let's make our simulated universe a bit more interesting by introducing a new kind of particle.

We will do this by introducing a `MagicParticle` class, which differs from `Particle` only in that rather than bouncing off the boundaries of the universe, it reappears at the opposite boundary.

We could do this by copy-paste code reuse:

- Copy `Particle.hh` and `Particle.cc` into `MagicParticle.hh` and `MagicParticle.cc`,
- Search-and-replace `Particle` to `MagicParticle` in the new files,
- Change the body of `MagicParticle::move` to implement the new behaviour.

This would leave us with two major problems

1. There is a lot of duplicated code: if we want to change some of it, we would have to track down all copies and to ensure that the change is universal. This is a maintenance nightmare.
2. We store the particles in a container of type `std::vector<Particle>`. This container cannot store `MagicParticle` s. We would need a separate vector of type `std::vector<MagicParticle>` to store the new kind. Another separate vector would be required for every new kind of particle we add in this way.

Inheritance provides a way of solving both of these problems.

> Write `MagicParticle` as a subclass of `Particle`: When a `MagicParticle` reaches the boundary, it should be transported instantly to the opposite boundary, without a change in speed.
>
> Extend the syntax of the configuration file to specify the particle type.

Hints:

- Any nontrivial C++ program using dynamic dispatch will probably use containers of pointers.
- As soon as the keyword `virtual` appaears in your class, you should give it a virtual destructor.
- Did you release all dynamically allocated memory? Did you use the correct version of `delete`?

2.67 TODO Overloading on constness

Need to find a place where this emerges naturally. Mention C++11's `override` feature.

2.68 TODO Better hierarchies, Abstract classes

Discussion of well thought out class hierarchies belongs in the second course, but we should probably cover the mechanism of abstract classes.

Make `Particle` abstract, with two concrete subclasses, `BouncingParticle` and `MagicParticle`.

2.69 TODO Static methods and members

Count of all particles?

3 Reference

3.1 Types and binary representation of data

All the information stored by computers is, at a fundamental level, stored in binary code, a code that consists of entirely of ones and zeros. These ones and zeros may be represented by two different voltages, two different directions or levels of magnetization, or some other physical characteristic, but in all cases there are only two distinct states that can be represented.

And yet, computers give the impression of dealing with a vast range of different data: numbers, text, photographs, sound, video, models of aircraft or LHC detectors, etc. This is achieved by devising clever schemes for interpreting sequences of ones and zeros as

For example, if we are to treat our binary data as integers, we might use the following interpretation

```
...000 -> 0
...001 -> 1
...010 -> 2
...011 -> 3
...100 -> 4
...101 -> 5
...110 -> 6
...111 -> 7
```

while the same data might be interpreted like this

```
...000 -> a
...001 -> b
...010 -> c
```

```
...011 -> d
...100 -> e
...101 -> f
...110 -> g
...111 -> h
```

if we are to think of them as characters.

In order for the computer to know how to interpret some raw data that it is managing, it must know the *type* of those data.

While there is much more to the concept of types in computer science, as a first step we can think of types as different ways of interpreting sequences of ones and zeros in meaningful ways.

The declaration

```
int i;
```

has a meaning roughly equivalent to

> Dear compiler, please set aside some memory for my use. I want to store integers there, so please interpret any ones and zeros you find there as an integer. I would like to refer to the contents of this memory, by name i.

3.2 Compilers

3.2.1 Avoid using an IDE to start with

Later on in the course, you may use whichever compiler/IDE/OS you prefer out of those that have been installed on the training machines.

We will start on GNU/Linux with g++ and a plain text editor. The IDEs are likely to perform some magic on your behalf and also complicate the writing of simple programs. Once you are familiar with the basics of the language, you can move on to an IDE if you like.

3.2.2 Portability

While the C++ language is defined by an internanional standard, most compilers include extensions to the language. While these extensions may appear attractive and useful, they should almost never be used in serious projects. Using such extensions may make it very difficult to compile your program with different compilers and/or on different platforms.

The `-ansi` and `-pedantic` options instruct C++ to reject programs which are not standards conformant.

Get in the habit of always compiling your programs with these options switched on. Other compilers should have options with similar functionality, even if they are switched on differently.

3.2.3 Warnings

You should strive to have your programs compile without the compiler issuing any warnings, even though all the warning flags are on: This means that you should rewrite your program until the compiler no longer feels the need to issue any warnings.

Yes, you could just switch the warnings off, or be satisfied with code that was generated with warnings, but this usually leads to more serious problems later on.

3.2.4 Where should the output go

Notice that the executable produced by g++ is, by default, called `a.out`. Other default names will be chosen in other circumstances (such as production of object files), but the default name for executables is `a.out`.

In order to specify the name of the file the compiler should produce, you can use the `-o` option. The following are all equivalent ways of telling g++ to perform the same compilation, while making it leave the resulting executable in a file called `hello`.

```
g++ -ansi -pedantic -Wall hello.cc -o hello
g++ -ansi -pedantic -Wall -o hello hello.cc
g++ -o hello -ansi -pedantic -Wall hello.cc
```

3.2.5 Clang

Clang is a new C++ compiler built on top of LLVM. Even if you do not have the freedom to use clang for creating release binaries in your project, I still thoroughly recommend using it in order to help you understand the problems you encounter during compilation, because of Clang's vastly superior diagnostic messages.

3.3 C and C++

C++ is (almost) a superset of C. Valid C programs are, for the most part, also valid C++ programs, but **not** vice-versa.

In C++ you often have the choice to use a feature which C++ has inherited from C, or to use an alternative, new feature which is not present it C. Usually such features are considered (by the designers of C++) to be superior to those that C++ inherited from C. Programmers frequently disagree.

When faced with the possibility of using a C-feature and a C++-feature, this course will generally emphasize the C++ approach.

3.4 The preprocessor

The preprocessor transforms the input file **before** the compiler gets to see the source code.

Normally, the preprocessor is invoked automatically when you compile code. In the gcc toolchain, you may explicitly invoke the preprocessor with the `-E` flag.

Here is a file which demonstrates the major features of the preprocessor. Run the preprocessor on this file (or smaller portions of it) and see how the text is transformed.

Invoke the preprocessor like this:

```
g++ -E preprocessor.cc
```

Remember that the preprocessor transforms the code **before** the compiler gets to see it.

The preprocessor was inherited by C++ from C.

3.5 `main`

In order to run a program, the program's starting point must be known. In C++ this is defined by the function with the name `main`. There must be exactly one such function in any executable.

Library code can be compiled without a `main`, but it will not produce an executable until it is *linked* with some code which does contain a `main`.

In C++, the return type of `main` **must** be `int`. This means that, when the program terminates, it produces an integer as a result. This integer is interpreted as a status code by the operating system.

The C++ standard says that your compiler **must** accept `main` with the following signatures.

```
int main()
int main(int argc, char* argv[])
```

The compiler is *allowed* to accept other signatures too.

3.6 Signatures

The *signature* of a function, is the information making up the basic interface of the function, consisting of the function's

- return type
- name
- parameter types

The signature is closely related to the idea of a function prototype.

3.7 Prototypes

The *protoype* of a function is used to inform the compiler about the existence of a function before it has seen its definition.

A prototype looks just like a function definition, with the whole function body replaced by a semicolon. For example the prototype of the function

```
int add_three_ints(int a, int b, int c) {
   return a + b + c;
}
```

is

```
int add_three_ints(int a, int b, int c);
```

Parameter names may be omitted in prototypes. For example

```
int add_three_ints(int, int, int);
```

Fuction prototypes are used to declare functions before their definition is seen by the compiler.

3.8 Declarations

A *declaration* is a statement informing the compiler about the existence of program components such as variables, functions or classes. A declaration specifies the name and full type or interface of the component.

In the case of functions a declaration can come in the form of

- a definition: signature + body,
- a prototype: signature + semicolon.

3.9 Namespaces

Namespaces are a C++ feature not present in C. The purpose of namespaces is to reduce problems related to name conflicts.

You can create your own namespaces by enclosing your code in a namespace block, like this

```
namespace Mine {
    int cube(int n) { return n*n*n; }
}
```

Code which has seen the above declaration may now access the function `cube` in three different ways

1. Explicit scope resolution

   ```
   Mine::cube(2);
   ```

2. `using` the `cube` from `Mine`

   ```
   using Mine::cube;
   cube(2);
   ```

3. `using` the **whole** `Mine` namespace

   ```
   using namespace Mine;
   cube(2);
   ```

   I strongly recommend that you **do not** use the last approach, without thinking about it very carefully. It is fine for quick and dirty trials, but best avoided in production code, in most cases.

Here is some sample code demonstrating the use of namespaces.

3.10 Function overloading

Function overloading is the creation of more than one function with the same name. In order to be able distinguish between them, the functions must have different parameter types. (Namespaces provide another, unrelated, mechanism for having different functions with the same name.)

In C++, function overloading is a form of static polymorphism. The names are resolved at compile time: the compiler creates functions with different names, through a process called *name mangling*, by mixing information about the function's parameter types into its name. The mangled names are used in the binary produced by the compiler, but completely invisible in the surface syntax of the language.

To see name mangling in action compile this code without linking:

```
g++ -ansi -pedantic -Wall mangling.cc -c
```

and then look at the names contained in the generated object file:

```
nm mangling.o
```

You should see different names based on the name of the function which appears in the source code.

The name mangling algorithm is **not** standard: different compilers may mangle names differently. Name mangling can make interoperation between C++ and different languages difficult, as it is not obvious by which name to refer to compiled C++ functions, from other languages.

When the function is an operator, then the process is called operator overloading.

It is possible to have functions with the same name which reside in different namespaces. This is not overloading, as the fully qualified names of the functions are different.

3.10.1 Overloading on constness

From the perspective of function overload resolution, `int` and `const int` are different types. Similarly `Foo* this` and `Foo* const this` are different types.

It is therefore possible to have different functions in the same scope whose signatures differ only by the presence/absence of a `const`-qualifier.

This can lead to surprising behaviour and subtle bugs, when some function other than the one you were expecting, is called.

3.11 TODO Header files

3.12 TODO The function call stack

3.13 Pointers and constness

When dealing with pointers, we deal with two distinct bits of data

1. the pointer itself
2. the data indicated by the pointer

When mixing the concepts of pointers and constness, there a four possibilities

1. Both the pointer and the target location are `const`.
2. The pointer is `const` but the contents of the target location may be changed through the pointer.
3. The pointer itself may be changed (made to point to some other location), but the contents of the target location may not be changed through the pointer.
4. Both the pointer and the contents of the target location may be changed.

All four possibilities may be expressed in a type declaration, but there are more than 4 ways of expressing them, and many people find the semantics to be "the wrong way around".

I'll start off with an extensive set of examples, and later suggest a scheme for helping you to work out which is which.

In the following examples, the numbers in parentheses refer to the numbers in the list above.

```
int * a;                // mutable pointer, mutable target (4)
const int * b;          // mutable pointer,   const target (3)
int const * c;          // mutable pointer,   const target (3)
int * const d;          //   const pointer, mutable target (2)
const int * const e;    //   const pointer,   const target (1)
int const * const e;    //   const pointer,   const target (1)
```

Whitespace between `*` and any of *typename*, *variable-name* or `const` is optional.

Notice that a `const` appearing before the `*`, refers to the target location; a `const` appearing after the `*` refers to the pointer.

Many people feel that this is the wrong way around. I'll present you with three ways of thinking about it, which might help.

3.13.1 Eliminate redundancy

Given that `const int` has the same meaning as `int const` we can reduce the six different cases to four:

```
T       *       a;
T const *       b;
T       * const c;
T const * const d;
```

3.13.2 Eliminate the obvious

The meanings of the cases where `const` appears twice and `const` appears zero times are obvious, leaving us with two cases to ponder over:

```
T const *       a;
T       * const b;
```

3.13.3 What is const? Look to the right of the `const`

The two interesting cases boil down to

```
const *a; // *a is const (that's the value)
const  b; //  b is const (that's the pointer)
```

3.13.4 What type is const? Look to the left of the `const`

The two interesting cases boil down to

```
T  const // constant value
T* const // constant pointer
```

3.13.5 Read the declarations backwards!

```
T       *       a; // a is a pointer to a T
T const *       b; // b is a pointer to a const T
T       * const c; // c is a const pointer to a T
T const * const d; // d is a const pointer to a const T
```

3.14 Standard header naming

C++ inherits the C standard library. C's standard headers have names like `stdio.h`, `math.h` and `stdlib.h`. In the early days, C++ followed this style with names such as `iostream.h`.

When namespaces were added to C++, all the components provided by the standard library were placed in the `std` namespace. For the sake of backward compatibility, the old headers were left as they were, while the new namespace-enabled versions of the headers were provided under the same name, with the `.h` extension suppressed.

So `iostream.h` was deprecated in favour of `iostream`: the former without namespaces; the latter providing all its contents inside the `std` namespace.

For those headers which were inherited from C, the renaming is slightly different: in addition to suppressing the `.h`, an extra `c` appears at the beginning of the name.

Thus, `stdio.h` and `stdlib.h` are deprecated in favour of `cstdio` and `cstdlib`.

3.15 Translation units

Here is what the C++ standard has to say about the term *translation unit*

> The text of the program is kept in units called source files in this International Standard. A source file together with all the headers (17.6.1.2) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a translation unit.

3.16 Linkage

Linkage refers to the accessibility of a symbol (the name of a function or global variable) across translation units.

Internal linkage
: symbol only accessible in one translation unit

External linkage
: symbol accessible across translation units

The `extern` keyword is used to indicate that a symbol should have external linkage; `static` is used to indicate that in should have internal linkage.

`const` symbols, by default, have internal linkage; non- `const` symbols, by default, have external linkage

In summary

```
// For names at global scope (or in a namespace)

int                 implicitly_extern_v = 1; // Accessible in all TUs
const int           implicitly_intern_c = 2; // Accessible in this TU only
extern const int explicitly_extern_c = 3; // Accessible in all TUs
static int          explicitly_intern_v = 4; // Accessible in this TU only

// There are no const functions, therefore they are extern by default
       int implicitly_extern_f(); // Accessible in all TUs
static int explicitly_intern_f(); // Accessible in this TU only
```

3.17 TODO ODR

3.18 TODO Template implementations must go in headers

Otherwise, how would the compiler be able to make new instantiations of the template, during separate compilation?

The 1998 standard included the `export` keyword, which would allow template definitions not to be included in headers. It took about 7 years for the first compiler implementor (Comeau) to implement this functionality. Only one other vendor (Borland) managed it, about a year later.

The `export` keyword has been removed from the 2011 standard.

You must include the *implementation* of templates in their header file.

3.19 Inheritance

In C++ inheritance makes the following possible.

- Storing objects of differing types in the same variable.
- Changing the behaviour of a piece of code at runtime, on the basis of the type of the data present at the time of the call.
- Sharing code by allowing user-defined types to *inherit* features from other user-defined types.

3.19.1 Basic syntax

Here is sample code demonstrating the syntax and basic features of inheritance in C++. There should be nothing new to you in the first class:

```
struct Base {
  int datum;
  Base(int i) : datum(i) {}
  int read() { return i; }
  virtual void late() { say("Base::late"); }     //
  void early()        { say("Base::early"); }    //
  void method()       { say("Base::inherit"); } //
};
```

except for the `virtual` keyword. `virtual` switches on *dynamic dispatch* (also known as *late binding*). More on this later.

The next class is a *subclass* (also known as a *derived* class) of the above. Notice the syntax for expressing this relation.

```
class Sub : public Base { //
public:
  Sub(int j) : Base(j) {} //
  void late()  { say("Sub::late"); }  //
  void early() { say("Sub::early"); } //
};
```

In this context `public` may be omitted or replaced by `private` or `protected`. It states that any public or protected members inherited from Base should be private in Sub. By default, classes inherit privately, structs inherit publically.

Notice that there is a definition of late in the superclass and another in the subclass. Similarly, there is a definition of early in the superclass, and a second one in the subclass. late was declared virtual, early was not. There is a single definition of method, in the superclass. We'll investigate the consequences of all this below.

Superclasses may appear in the initializer list. The arguments are passed to the constructor of the superclass.

3.19.2 Reduced type strictness

Inheritance gives us the ability to store objects in variables whose type does not match that of the object. The example code shows objects of type Sub being stored in varibles of type Base, in three varieties: value, reference and pointer.

The reverse, storing superclass objects in subclass variables, is not possible (even through references or pointers).

```
  // Subclass objects may be stored in superclass variables
  Base  base_v_base = Base(10);
  Base  base_v_sub  = Sub(10);      //
  Sub    sub_v_sub  = Sub(10);
//Sub    sub_v_base = Base(10);     //

  Base& base_r_base = base_v_base;
  Base& base_r_sub  =  sub_v_sub;   //
//Sub&   sub_r_base = base_v_base;  //

  Base* base_p_base = &base_v_base;
  Base* base_p_sub  = & sub_v_sub;  //
//Sub*   sub_p_base = &base_v_base; //
```

3.19.3 Members are inherited

The subclass inherits the members of the superclass: Even though neither datum nor void method() were defined in the subclass, we can access them through subclass instances:

```
// Subclasses inherit members of superclasses
sub_v_sub.method(); // (inherit_method)
sub_v_sub.datum;    // (inherit_datum)
```

3.19.4 Dynamic dispatch, late binding

The next section shows that, when calling virtual functions through a base class variable, the result depends on the actual type of the object present. Note that this only works through pointers and references: it does not work for value variables.

```
// virtual methods are late-bound via pointers and references only
base_r_base.late();
base_r_sub.late();
base_p_base->late();
base_p_sub->late();
base_v_base.late();
base_v_sub.late();  // No dynamic dispatch! (vals_bind_early)
```

Virtual functions are a means of providing different implementations of some operation, and delaying the choice of which of those implementations to use, until run-time. This process goes by the names *dynamic dispatch*, *late binding* or *dynamic binding*. (Contrasted with *early binding* or *static binding*.)

3.19.5 Early binding

There is no dynamic dispatch for non-virtual methods:

```
// non-virtual methods are always bound early
base_r_sub.early();
base_p_sub->early();
base_v_sub.early();
```

Even though these variables contain objects of type `Sub`, the version of `early` defined in `Base` is called. The decision about which method should be called was made at compile time.

3.19.6 Static/dynamic, formal/run-time type

The type of the variable, as indicated by declarations in the source code is called the *static type* or the *formal type*; the type of the object residing in the memory tied to the variable is called the *dynamic type* or the *run-time type*.

3.19.7 Dynamic dispatch can be elusive

Remember that 3 conditions must be satisfied for dynamic dispatch to work:

1. The different implementations must reside in classes which are related by inheritance.
2. The operation must be declared `virtual` in the formal type of the call site.
3. The operation must be invoked through a pointer or a reference.

3.19.8 `protected`

You are familiar with the keywords `private` and `public`. `protected` has an intermediate meaning, which only becomes relevant in the context of inheritance. It means public if accessed by my subclasses, but private if accessed from anywhere else.

3.20 TODO Containers of pointers

3.21 TODO Virtual destructor

Talk about how not doing so risks undefined behaviour in your programs.

3.22 TODO Undefined behaviour

Talk about implementation-defined behaviour, and about how evil undefined behavior is.

4 TODO C++11

4.1 TODO Intro

Waffle a bit about the new standard

4.2 TODO auto, decltype

```
#include <vector>
std::vector<int> v;
std::vector<int>::iterator it_old = v.begin();
                      auto it_new = v.begin();
```

4.3 TODO tuples

```
#include <tuple>
auto t1 std::make_tuple(a,b,c);
std::tuple<A,B,C> t2{a,b,c};
auto t3 std::tie(a,b,c); // creates a tuple of lvalue-references
std::get<0> t1;
```

4.4 TODO `std::initializer_list`

std::initializer$_{list}$

4.5 TODO uniform initialization

4.6 TODO range-based for loops

```
for ( auto item : container) {
   use_it(item);
}
for ( auto& item : container) {
   mutate_it(item);
}
```

4.7 TODO operator deletion and defaulting

Avoid suppression of compiler generated default constructor

```
struct A {
    A() = default; // Default constructor provided by compiler, even though other constructor is present
    A(B b);
    A& A(const A&) = default; // Redundant, but stronger statement than a comment
};
```

Suppress compiler generated function

```
struct A {
    A(const A&) = delete;
    A& operator=(const A&) = delete;
};
```

Suppress ability to call a given signature

```
struct A {
    void f(double d);
    void f(int) = delete;
};
```

(otherwise calling A::f(1) would perform a silent conversion to double).

This idea can be generalized with templates: suppress ability to call `A::f` with *any* argtype other than `double`.

```
struct A {
    void f(double d);
    template<class T> void f(T) = delete;
};
```

4.8 TODO initialization of inherited members

4.9 TODO constructor delegation

4.10 TODO base constructor inheritance

An in-class `using` declaration stating that a (a whole set of overloaded) base class' member functions should be 'lifted' into a derved class.

```
struct B : A {
    using A::fn; // Worked in C++98
    using A::A;  // Didn't work until C++11
};
```

so now you can avoid writing lots of annoying, trivial constructors which do nothing other that initializing the base class.

4.11 TODO In-class member initialization

```
struct C {
   int x = 3;
};
```

All C's constructors will initialize x to 3, unless they specifically override it.

4.12 TODO Virtual function override safety

In C++03, it is possible to accidentally create a new virtual function, when one intended to override a base class function. For example:

```
struct A {
    virtual void meth(float);
};

struct B : A {
    virtual void meth(int); // overload NOT override.
};
```

This is a common mistake. To help avoid it, you can explicitly state that you mean to override:

```
struct A {
    virtual void meth(float);
};

struct B : A {
    virtual void meth(int) override; // Error, if exacts signature not present in superclass
};
```

C++11 also adds the ability to prevent inheriting from classes or simply preventing overriding methods in derived classes. This is done with the special identifier final. For example:

```
struct A final { };

struct B : A { }; // Error: A is final
```

```
struct A {
    virtual void f() final;
};

struct B : A {
    void f(); // Error: A::f is final
};
```

`final` and `override` are NOT language keywords: they can be used as identifiers in other locations.

4.13 TODO template1<template2<T>> parse fixed

5 Extra exercises

Here is a collection of exercises which are not considered to belong to the core of the course.

You may like to try them if you find that you spend significant time during the course waiting for the rest of the class to catch up with you.

Some of them may be suggested to the whole class, if I find that we are working through the material quickly.

You may like to try them in your own time, for extra practice.

5.1 Sparse screen buffer

Change the internal representation of the screen buffer to use some kind of sparse data structure. The client code should continue to work, unchanged.

5.2 Improve the dynamic array class

The dynamic array class you are asked to write during the course is minimalistic in the extreme. Improve it.

5.2.1 TODO Suggestions for improval of dynamic array class

5.3 TODO Linked list class

It should have the same interface as the dynamic array

Validate